Reversibility for Efficient Computing

(Manuscript, Based on Ph.D. Thesis)

Michael P. Frank

CISE Department
University of Florida
http://www.cise.ufl.edu/~mpf

Working version of
Dec. 20, 1999

Abstract

Today's computers are based on *irreversible* logic devices, which have been known to be fundamentally energy-inefficient for several decades. Recently, alternative *reversible* logic technologies have improved rapidly, and are now becoming practical.

In traditional models of computation, pure reversibility seems to decrease overall computational efficiency; I provide a proof to this effect. However, traditional models ignore important physical constraints on information processing.

This thesis gives the first analysis demonstrating that in a realistic model of computation that accounts for thermodynamic issues, as well as other physical constraints, the judicious use of reversible computing can strictly *increase* asymptotic computational efficiency, as machine sizes increase. I project real benefits for supercomputing at a large (but achievable) scale in the fairly near term. And with proposed future computing technologies, I show that reversibility will benefit computing at all scales.

Next, the thesis demonstrates that reversible computing techniques do not make computer design much more difficult. I describe how to design asymptotically efficient processors using an "adiabatic" reversible electronic logic technology that can be built with today's microprocessor fabrication processes. I describe a simple universal reversible parallel processor chip that our group recently fabricated, and a reversible instruction set for a more traditional RISC-style uniprocessor.

Finally, I describe techniques for programming reversible computers. I present a high-level language and a compiler suitable for coding efficient reversible algorithms, and I describe a variety of example algorithms, including efficient reversible sorting, searching, arithmetic, matrix, and graph algorithms. As an example application, I present a linear-time, constant-space reversible program for simulating the Schrödinger wave equation of quantum mechanics.

Acknowledgments

First and foremost, I am enduringly grateful to my advisor Tom Knight and also to Norm Margolus, for taking me into their fold, for their extensive and wise guidance in all areas of this research, for the many things I learned from them, and for providing an exceptionally supportive and stimulating work environment. Thanks very much also to the other readers Gill Pratt and Gerry Sussman, for kindly agreeing to serve on my thesis committee and lend their expertise to the evaluation of this work.

I would also like to thank the Pendulum group's head graduate student, Carlin Vieri, for inventing the Pendulum architecture, which provided the context for much of this work; for taking it upon himself to do most of the project's administrative chores; and for the fun we had working together on the nuts and bolts of various aspects of the project such as the Pendulum ISA and the FLATTOP chip. (Also, thanks for all the Nolios!)

I should also thank a number of other MIT students: Matt DeBergalis created many helpful software tools, such as the Pendulum assembler and emulator. Josie Ammer assisted with chip design, and made important contributions to the theory and algorithms work. Scott Rixner and Nicole Love were my primary design partners on the Tick and FLATTOP processors, respectively. Matt Becker often popped into my office for a friendly greeting and an interesting conversation.

Prof. Michael Sipser deserves thanks for his informal but very helpful guidance during the development of the proof in §3.4. I am also grateful to Alain Tapp and Pierre McKenzie of the University of Montreal, for providing much helpful feedback on that result, and for inviting me to visit their lab. Thanks also to Prof. Tom Leighton for communicating the reversible shortest-path algorithm, and to both him and Prof. Charles Leiserson for feedback on the scaling results of chapter 6.

Finally, I would like to thank Carl R. Witty and Warren D. Smith for helpful editorial comments on drafts of this document, my family and my wife Lori for their love, and all my friends throughout my years at MIT for their companionship and encouragement.

This research was supported by DARPA (the Defense Advanced Research Projects Agency of the United States of America) as part of the Scalable Computing Systems research program [43], under contract number DABT63-95-C-0130.

to my father, Patrick Gene Frank

Contents

List of Figures

List of Tables

Chapter 1

Introduction and background

In this chapter, we describe (§1.1) and motivate (§1.2) the topic of this thesis, outline some of the history of the body of research upon which this work builds (§1.3), summarize the major contributions of this thesis (§1.4), and give a brief overview of the contents of the later chapters (§1.5).

1.1 What this thesis is about

This thesis is a detailed study of the advantages (and disadvantages) of the use of *reversibility* in computing. What do we mean by reversible computing? For our purposes, there are two important meanings:

Logical reversibility. First, a computational operation can be *logically reversible*, meaning that the logical state of the computational device just prior to the operation (its *input* state) is uniquely determined by its state just after the operation (its *output* state). Computing in a logically reversible fashion implies that no information about the computational state of the system can ever be lost; any earlier state can always be recovered by computing backwards from a given point. Another way to understand logical reversibility is that the system is deterministic looking *backwards* in time.

In chapter 10 we will see that logical reversibility, in itself, has some interesting computational applications. Chapter 9 will discuss how to program logically reversible computers. But the larger emphasis of this dissertation will not be on logical reversibility by itself, but on the benefits to be gained from using logical reversibility to enable another important kind of reversibility, namely, *physical reversibility*.

Physical reversibility. A *physically reversible* process is a process that dissipates no energy to heat, and produces no *entropy*. It seems that absolutely *perfect* physical reversibility is technically unattainable in practice in a complex, controlled dynamical system, simply because there will always be some nonzero probability for a random event to occur (*e.g.*, the impact of a cosmic ray, or an asteroid) that is sufficiently energetic that it will interfere with even the most carefully-controlled and well-isolated of systems. Nevertheless, physical reversibility is a useful concept, because (as we will see in ch. 7) even with present-day electronic technology, we can already make logic devices that are *almost* physically reversible, and we do not yet know of any fundamental limits to how close we can get to perfect reversibility, as technology improves.

As we will see in §2.5, logical reversibility is necessary in order to approach complete physical reversibility. Chapters 6 and 7 will focus on the study of computing devices that are *both* logically and physically reversible, and on their resulting implications for the potential efficiency of computation. Usually when we speak of reversibility in this thesis, we will be referring to this combination of logical and physical reversibility, rather than to just logical reversibility by itself.

1.2 Motivation

Why study reversible computing? Aside from pure academic interest, we feel that the study of reversible computing can be motivated in a fairly strong way, in terms of the long-term goals of society in general and the field of computer science in particular.

For the productivity of society, and the growth of the economy, efficient information processing is critical. A relatively small improvement in the speed and power of computers facilitates progress in virtually every industry. The great value of information processing has motivated the enormous technological investments fueling Moore's law, the trend of exponential improvement in computer speed and cost-efficiency that has been maintained over the last half-century.

It is in society's interest that computer technology continue to improve rapidly for as long as possible. Therefore, it is important to identify various potential obstacles to further improvements far enough in advance so that the research community has time to develop solutions before such an obstacle has a chance to stall the rate of progress. Or, if some truly insurmountable barrier to further improvement can be identified early on, at least society will have time to prepare for the consequences.

The semiconductor electronics industry is well aware of a variety of potential obstacles to further improvements of its technology over the next 10 to 15 years [145]. Even if these obstacles are overcome, we can expect that eventually a point will be reached where it is technically or economically impossible to refine semiconductor technology further. At that point, perhaps alternative computing technologies will eventually emerge and supersede semiconductors. (We discuss several potential alternatives in ch. 8.)

However, in the longer term, we can foresee a variety of more fundamental limits to the improvement of computer technology, limits that are qualitatively independent of the particular technology used (such as semiconductors), and whose existence depends only on well-established fundamental laws of physics. These fundamental limits will become increasingly important as computer technology improves, whatever path it takes. If we can, right now, identify some techniques that will allow technology to perform as well as possible given these ultimate physical limits, then we will be well prepared to cope with these limits once they become dominant concerns in computer engineering. (We discuss this research philosophy in more detail in ch. 5.)

Not to keep the reader in suspense, one fundamental physical limit, known since at least 1961 (Landauer, [97]), is that for every bit's worth of computational information that is discarded within a computer, at least one bit's worth of new physical entropy must be generated. Moreover, due to basic thermodynamic principles, this entropy cannot simply be destroyed, but must instead be physically moved out of the computer, if one is to keep the machine from eventually overheating. (We will explain these constraints in more detail in chapter 2.)

In chapter 6 of this thesis, we establish that in order for a scalable computer architecture to be as efficient as possible in the face of these constraints, the machine must contain the capability to perform computations in a logically and physically reversible manner, which minimizes the production of unnecessary entropy, and the overhead of its removal from a densely-packed machine. This suggests a framework for algorithm design in which information is considered as a *conserved* material-like thing, embedded in 3-D space. As we will see, this is what information really is like. The expert programmer should not mind expanding his expertise to working with such a model, because it allows designing the best algorithms that are physically possible.

The capability of reversibility is completely lacking from today's processor designs. But the technology now exists to remedy this situation, and Part II of this thesis discusses how to design and program machines that use reversibility to achieve asymptotically optimal efficiency. The high-level concepts of reversible circuits in chapter 7 are described in terms of existing semiconductor technology, but are not dependent on it: they can be applied equally well to a wide range of future logic-device technologies that might emerge.

Near-term benefits. Present-day technology is far from the fundamental physical limits of computation, but reversibility offers some of the same benefits today that it will offer in the limiting technology. We now know how to build approximately physically reversible computers using today's electronic technology. These techniques may have benefits in the near-term, in applications where energy dissipation is of paramount importance (see §7.10). There may even be some near-term uses for logical reversibility by itself, regardless of physical reversibility, as discussed in chapter 10.

But one must be careful: chapter 3 reveals some of the theoretical inefficiencies incurred when using logical reversibility by itself, in situations where saving energy and minimizing entropy production are unnecessary.

In summary, motivations for studying reversible computing include both short and long-term applications; the long-term ones being more fundamental. The major motivation lies in the economic value of making computers more efficient through reversibility, under any of a variety of measures of efficiency that are influenced by energy dissipation. This thesis focuses on exploring how this can be done.

1.3 Brief history of reversible computing

In this section we briefly summarize some of the history of reversible computing research. This is not intended to be a complete account. Some additional historical information about particular sub-areas will be provided in later chapters. A more comprehensive review of the early history of part of the field is provided in Bennett 1988 [18].

1.3.1 Early thermodynamics of computation

The study of thermodynamically and logically reversible computational processes has historically been motivated by concerns in fundamental physics. For example, the proper resolution of the famous "Maxwell's Demon" paradox of thermodynamics (see the papers in [100]) required understanding that the means of disposal of unwanted information can be important when considering the thermodynamics of a system.

The first connection between computation and fundamental thermodynamics was apparently made by John von Neumann ([182], p. 66). In a December 1949 lecture at the University of Illinois, he reportedly performed a calculation of the thermodynamical minimum energy that is dissipated "per elementary act of information, that is, per elementary decision of a two-way alternative and per elementary transmittal of 1 unit of information." He quantified this energy as $k_{\mathrm{B}}T\ln\mathcal{N}$, where k_{B} is Boltzmann's constant, T is the temperature, and $\mathcal{N} = 2$ is the number of alternatives to be decided between. Unfortunately, there is apparently no existing complete record of this lecture, or of any corresponding written analysis by von Neumann, so it is difficult to determine exactly how he explained this analysis, how seriously he took it, and whether it was actually original to him.

Rolf Landauer (1961, [97], §4) was apparently the first person to explicitly state the argument establishing that the irreversible erasure of a bit of computational information inevitably requires the generation of a corresponding amount of physical entropy (namely $1\,\mathrm{bit} = \ln 2$ "nats" $= k_{\mathrm{B}}\ln 2 \approx 9.57\times10^{-24}\,\mathrm{J/K}$). In that paper, Landauer also recognized that reversible operations need not incur such dissipation, and that any irreversible computation can be performed via a sequence of reversible operations by saving a history of all the information that would otherwise be irreversibly dissipated. However, Landauer then proceeded upon the mistaken assumption that the space occupied by this history record would have to be irreversibly cleared in order to be reused, and concluded that therefore, reversible operations could not avoid the fundamental unit dissipation incurred by each computational step, but could only postpone it until the memory needed to be reused. To his credit, Landauer realized that this argument was not rigorous, and did not present it as such.

1.3.2 Development of reversible models of computation

Landauer's error was not caught until Charles Bennett (1973, [16]) discovered that the reversibly-recorded history of an irreversible computation could also be cleared in a logically reversible way, leaving only the input and the desired computational output in memory. This refuted Landauer's argument that each useful computational step must incur, in the long run, at least about $k_{\mathrm{B}}T$ energy dissipation. With Bennett's trick, the amount of memory that would need to be irreversibly cleared between runs could be smaller, by an arbitrarily large factor, than the number of useful irreversible computation steps that are reversibly simulated during the course of the computation.

Bennett described his technique using a formal Turing machine model, but later researchers showed that Landauer's trick of recording a history could also be applied to permit other models such as cellular automata (Toffoli 1977 [160]) and logic circuits (Toffoli 1980 [161], Fredkin & Toffoli 1982 [74]) to operate reversibly as well. Indeed, the Laundauer/Bennett techniques seem to apply generally to "reversiblize" any model of computation.

1.3.3 Development of physically reversible logic devices

However, showing that logically irreversible operations can be avoided in useful computations is only part of the problem of demonstrating that reversible computing can save energy. The other part requires showing that physically reversible primitive logic devices can actually be built. Bennett's 1973 paper [16] suggested the possibility of an enzymatic reversible computer using biomolecules, and in later papers such as (1982, [17]) he described a clockwork mechanical Turing machine powered by Brownian motion. Meanwhile, Fredkin and Toffoli had described an electronic implementation (1978, [73]), and an idealized model based on the ballistic motion of rigid spheres (1982, [74]), which we will describe in more detail in §7.7.1, p. 201. Konstantin Likharev showed in 1982 [108] that superconducting Josephson junction circuits could be used to compute in a reversible fashion.

Later reversible device proposals (see ch. 8) include various mechanical and electronic proposals by the pioneering molecular nanotechnologists Drexler and Merkle (Drexler 1992 [51], ch. 12; Merkle 1993 [123, 124]; Merkle & Drexler 1996 [126]), and a single-electron system analyzed by Likharev and Korotkov (1996, [111]).

So at present, there is no shortage of reversible device ideas. Moreover, in the years since Fredkin & Toffoli's 1978 proposal [73] it has become quite feasible and economical to build reversible devices using conventional VLSI electronic fabrication techniques (*cf.* Athas *et al.* 1994 [5], Younis & Knight 1994 [193]); we will review those developments in more detail in chapter 7.

1.3.4 Previous reversible computing theory

Independently of the type of reversible devices that are used, there are algorithmic issues involved in performing large computations using logically reversible primitives. For example, Bennett's original reversible simulation technique is limited by the fact that the algorithm requires an amount of temporary storage space that is proportional to its run-time. In contexts where digital storage is expensive and energy is cheap, one might do better by just discarding the bits instead.

So, in 1989, Bennett developed a more space-efficient version of his algorithm [19]. Unfortunately, it incurs a polynomial slowdown factor that cannot be made arbitrarily close to linear without making the space usage exponentially large (Levine and Sherman 1990 [103]). Similarly, in 1997, Lange, McKenzie, and Tapp [98] gave a general algorithm for reversible simulation of irreversible computations using *no* extra space, but with exponentially inflated run-times. It remains an important open problem to prove whether or not there is a *single* reversible simulation technique that incurs overheads in *neither* space nor time, but, as we will prove in §3.4, any such technique cannot be totally general, in the sense of applying to any conceivable model of computation.

1.3.5 Optimal scaling of physical machines

This thesis takes the study of reversible computing beyond the traditional focus on devices and classical complexity theory; chapter 6 introduces a new area of study, namely of how reversibility affects the scaling behavior of the most powerful physically possible computers, based on fundamental physical arguments.

The optimal scaling of computation within physically realistic constraints is an issue that has been studied previously (*cf.* Vitányi 1988 [180], Bilardi & Preparata 1993 [24], Smith 1995 [152]), but never before with particular attention to how the reversibility of physics allows reversible computation to improve physical scaling behavior. The research reported in this thesis is, to our knowledge, the first work that explores this new angle.

1.3.6 Programming reversible machines

We will save our review of the history of this area until chapter 9.

1.4 Major contributions of this thesis

The primary novel, original contributions of this thesis are the following:

- **Chapter 2** gathers together and presents in an organized form a variety of known fundamental physical constraints on information processing, that are expected to apply to any physically possible computing technology, at least in the non-relativistic regime. We conjecture that this is the first such listing that is sufficiently complete that it encompasses all the fundamental physical constraints (within that regime) that determine the maximum asymptotic scalability of computers and algorithms.

- **Chapter 3, section 3.4** (work done with Josie Ammer) underscores the overheads for reversibility in traditional models of computation by proving, for the first time, that any completely general transformation of irreversible machines to reversible ones must sometimes increase either the asymptotic computational time or space requirements for solving some problems. It gives lower bounds on the amount of increase required. The proof applies to cases where there is reversible access to an external black-box ROM or oracle. It is conjectured to also be true for pure models with no external black box. The proof might be extensible to that realm if it assumes that one-way functions exist, as is frequently assumed in cryptography.

- **Chapter 5** presents a novel physically-realistic model of computation (the R3M or "reversible 3-D mesh") and conjectures a "tight Church's thesis" claiming that this model is asymptotically as powerful as is physically possible given the constraints from ch. 2, within a constant factor.

- **Chapter 6** proves that the proposed R3M model is asymptotically strictly more powerful than any irreversible model of computation, by small polynomial factors in the machine size. Specifically, reversible machines of physical diameter D are shown to be asymptotically faster than diameter-D irreversible machines, by a factor of $\mathbf{\Theta}(\sqrt{D}\,)$. Also, reversible machines of mass M are both faster and more hardware-efficient than mass-M irreversible machines by a factor of $\mathbf{\Theta}(\sqrt[18]{M}\,)$. These bounds are shown to apply to a wide class of parallel computations that require sufficiently tight communication, but that need not be inherently reversible.

I consider the previous item to be the central, most important contribution of the thesis.

- **Chapter 8** uses the scaling results from ch. 6 together with parameters of present-day and proposed future technologies to show that with present-day technology, reversibility becomes advantageous at a reasonable scale, and in future technologies, it will be advantageous at just about any scale.
- **Chapter 7, section 7.6** does some novel analysis showing how to choose speeds, voltages and temperatures so as to minimize energy dissipation in one form of reversible electronics.
- **Chapter 7, section 7.7** and **appendix A** present the design (for which I was primarily responsible) of the world's first ever fabricated reversible parallel processor, which in principle obeys the scaling results of chapter 6 and thus is asymptotically faster than all previous parallel processing architectures, which are irreversible.
- **Chapter 9** and **appendices B through E** present examples of reversible instruction sets, programming languages, and algorithms. Similar efforts have been undertaken before by other researchers, so this area of contribution is not completely novel. However, much of my work proceeded independently of the earlier efforts. This reinvention helps underscore my point that reversible programming concepts are not difficult to master.

That completes our summary of the major contributions of the thesis. We will revisit this list once again in chapter 11.

1.5 Overview of thesis chapters

Here we summarize the contents of the various chapters of this thesis.

Chapter 2 surveys what is currently known about the fundamental constraints that known physics places on the potential capabilities of computing systems. We describe limits on the speed at which information can travel, the density at which it can be stored, and the rate at which it can cross a surface. We also review recent fundamental limits from Margolus and Levitin (1996, [118]) on the rate at which a computer can change state. We discuss the meaning and the computational implications of physical reversibility and the second law of thermodynamics.

Chapter 3 examines various formal theoretical models of reversible computing, and describes all the known results in the area. Then the chapter focuses on proving an important new conjecture in the theory of reversible computing: namely that in ordinary, nonphysical models of computation, imposing reversibility on the model must cause either the space or time complexity of some problems to increase. We prove that the conjecture is indeed true in a model of computation that invokes a contrived (but computable) oracle, and we establish lower bounds on the resulting increase in complexity. This proof implies that if there is an algorithm for simulating irreversible machines on reversible ones with perfect efficiency, then that algorithm cannot be totally general (relativizable to all oracles), in contrast to all the reversible simulation algorithms that are known currently.

Chapter 4 reviews the possibility of computation using large coherent superpositions of states (quantum computation). Quantum computers are inherently reversible.

Chapter 5 introduces the concept of "ultimate" physical models of computing, which are designed to accurately capture the true asymptotic complexity of all computational problems under the laws of physics. Then the chapter outlines the form that we will argue such models must take—namely, some sort of three-dimensional mesh of potentially reversible processors.

Chapter 6 discusses how the use of reversibility affects the scaling behavior of computers in several important respects. Due to their unavoidable generation of entropy which must be removed, irreversible computers turn out to ultimately be limited to processing rates that are only proportional to their surface area. In contrast, if a computer uses devices that are reversible, even in a limited sense that takes frictional effects into account, then it can perform $\Theta(\sqrt{d})$ times more operations per second within a physical space of diameter d. Even if we do not constrain the physical area of the computer, but only its mass (number of processors), reversible computers are still faster at some problems by a factor that grows as $\Theta(\sqrt[18]{n})$ where n is the number of processors.

Chapter 7 describes and analyzes in detail some known reversible circuit technologies, how they perform as various parameters are scaled, how they compare to traditional circuits, and how to design processors based on these techniques that realize the scaling benefits described in the previous chapter. We describe a very simple example of such a processor that we designed.

Chapter 8 reviews a variety of advanced logic technologies that have been proposed for use when the limits of traditional VLSI are reached. Then, we use our scaling results from chapter 6, together with parameters of the proposed technologies, to show that if we assume reasonable limits on future cooling systems, then any computers of macroscopic size that are built using these future technologies will be considerably faster if their logic elements are operated reversibly.

Chapter 9 illustrates in detail how to program reversible computers.

Instruction sets. We start with a description of some properties that a good reversible microprocessor machine instruction set needs to have, and how we achieved these properties in our group's Pendulum instruction set architecture.

High-level languages. Next we describe important issues in the design of high-level programming languages for reversible processors. Special programming languages are required in order to permit optimum efficiency on reversible processors. We describe the simple reversible programming language "R" which we designed and wrote a compiler for.

Algorithms. Finally, we describe some good reversible algorithms for a number of problems, including sorting, searching, arithmetic, matrix operations, graph problems, and simulations of physical systems.

Chapter 10 briefly discusses some potential alternative applications for reversible computing, aside from the energy dissipation issues. These include applications in hardware error detection, protecting against accidental or malicious data destruction, program debugging, transaction processing and database rollback, and speculative execution in multiprocessors.

Chapter 11 summarizes the progress in reversible computing achieved in the thesis, and points out the main areas where future work is needed.

Appendix A shows circuit schematics and VLSI layouts for the proof-of-concept parallel reversible processing element we describe in chapter 7.

Appendix B gives program-level specifications for PISA, the instruction set architecture for PENDULUM, our group's reversible RISC processor design.

Appendix C gives a complete account of "R," the simple C-like reversible programming language we developed.

Appendix D describes our compiler, written in Common Lisp, which translates R source programs into reasonably efficient PISA assembly code.

Appendix E gives the detailed derivation and code for our reversible program for simulating the Schrödinger wave equation of quantum mechanics (our illustration of an efficient reversible physical simulation).

Appendix F gives tables of mathematical units, constants, and notations used in the text, for easy reference.

1.6 Overall message of thesis

The overall message of this thesis is that (1) reversible computing techniques are not very different from or more difficult than ordinary computing techniques, and (2) they will definitely be a necessary part of the long-term future of computing.

It is hoped that this thesis will help to convince the larger computing community of these very important points, and thus help to spur further research in this field.

Part I

Foundations of reversible computing

Chapter 2

Physical constraints on computation

In this chapter we briefly review some of the important fundamental constraints that physical law places on computational capabilities. These constraints will serve as the basis for the arguments in chapter 6, which will establish that reversible models are necessary for permitting the maximum possible computational power in the limiting technology.

Most of existing computer science theory today deals not with physics, but with abstract realms of pure mathematics, exploring a plethora of different models of computation having wildly varying capabilities. Sometimes these theoretical models have capabilities substantially different from those of physics as we know it.

But real-world computers are physical devices, and their ultimate potential capabilities are defined not by some arbitrarily-chosen model, but rather by the hard facts of physical law. Unfortunately, physics is not yet completely understood (witness the lack of an accepted unification of quantum mechanics with general relativity), and even those parts that *are* well understood are not usually described in terms that facilitate the use of physics itself as a model of computation.

However, physics does constrain information processing in a number of important ways that can already be identified with fairly high confidence.

2.1 Propagation speed limits

The most obvious physical limit important to information processing is the lightspeed bound for the speed at which information may propagate through space.

Physical dynamics, as currently understood, proceeds purely through *local* interactions; there is no "action at a distance." Even gravity, thought by Newton to be an instantaneous force, is now understood, in the context of general relativity, to propagate through space at only the speed of light, $c \approx 3 \times 10^8$ m/s.

Even the quantum-mechanical systems that are sometimes interpreted as demonstrating "spooky action at a distance" (such as separated EPR pairs), can be explained instead in terms of local interactions. As we will review in chapter 4, quantum dynamics is based on an "amplitude function" which is a function of the global state of a system (the whole universe if you like). This leads to a statistical behavior that may at first *appear* to require nonlocal interactions, but the wavefunction actually evolves over time through a transformation (the Hamiltonian) that can be expressed as a composition of interaction terms that are entirely spatially local.

In general, due to the locality of underlying physical law, *all* influences are restricted to traveling, at most, at the speed of light. Thus, the physical transmission of information in a computer is limited to this speed as well.

This bound is "tight" in the sense that it is, of course, already achieved in practice in our ubiquitous telecommunication systems, and in optical interconnection networks in some computers. Signals in typical electrical transmission lines travel a bit slower, about half the speed of light. But propagation times are still linear in the distance traveled.

One important exception is that signals in low-inductance, resistive wires (such as the wires on integrated circuit chips) do not actually travel at constant speed, but rather, for long wires, require propagation time that is proportional to the *square* of the length ℓ of the wire, in accordance with the diffusion equation. This unfavorable scaling presents problems in integrated circuit design today. However, even with current technology, this ℓ^2 scaling is not inevitable, but can be avoided through simple schemes such as periodic re-buffering of the signal.

2.2 Information density limits

Another important constraint for computation results from physical limits on the amount of information that can be stored within a given volume of space (such as memory in a computer). We can say with confidence that some such bounds do exist, but unfortunately their exact value is hard to determine. However, these bounds will be very important in our later arguments about the advantages of reversibility, so we will now take some time to look at the various possible answers in some detail.

Fundamental quantum mechanics appears to dictate a particular finite upper bound on the total amount of information (including entropy) that can be contained in any system, as a function of the system's physical volume and the amount of energy it contains. By the amount of information in a system, we mean simply the logarithm of the number of states that the system could occupy, given some definition of what constitutes “the system.” (See §2.5.2.) According to Margolus (1996, [118]),

> [The question of the number of states] is really a very old question: the correct counting of physical states is the problem that led to the introduction of Planck’s constant into physics [137], and is the basis of all of quantum statistical mechanics. The question can be answered by a detailed quantum mechanical counting of distinct (mutually orthogonal) states. It can also be well approximated in the macroscopic limit [87, 184] by simply calculating the volume of phase space accessible to the system, in units where Planck’s constant is 1.

Let us look at some particular information density bounds in more detail.

2.2.1 Entropy bounds from black hole physics.

Some particular upper bounds on information content as a function of system size and energy are given by Bekenstein (1984, [15]) and by Joos and Qadir (1992, [88]). Bekenstein’s bounds, which originally came out of his studies of the entropy of black holes (*e.g.*, [14]), are fairly loose, in the sense that his bounds may conceivably be much higher than the maximum information content for systems *other* than black holes. One bound Bekenstein gives ([15], eq. 1) is:

$$S < 2\pi ER/\hbar c, \tag{2.1}$$

where S is the capacity for entropy or information (in natural log units or *nats*), E is the total energy (including rest mass-energy) in a system, and R is the radius of the system.

The maximum mass-energy for a system of given radius is of course achieved only in the case of black holes, since anything with a black hole’s mass within a black hole’s radius has such a high surface gravity that it *is* a black hole. The radius of a black hole is proportional to its mass M according to $R = 2GM/c^2$ (Wald 1984 [183], p. 124, eq. 6.1.45), and so the energy of any system of that radius is bounded by the black hole rest energy,

$$E \le \frac{c^4}{2G} R \tag{2.2}$$

where G is Newton’s gravitational constant, $G = 6.67259 \times 10^{-11}\,\mathrm{N\,m^2/kg^2}$.

Combining (2.1) and (2.2), we have

$$S < \frac{\pi c^3}{\hbar G} R^2. \tag{2.3}$$

In other words the entropy of a system is ultimately bounded in proportion to its minimal surface area, rather than to its volume! This is somewhat counter-intuitive. Perhaps one way to understand this result is the following: Imagine growing a black hole up from a tiny size by throwing objects into it. Due to a gravitational time dialation that stretches to infinity as objects approach the horizon, items we throw at the hole never actually enter it from our point of view; the objects just keep getting closer and closer to the event horizon "surface" of the hole (and the light from them gets more and more red-shifted). From that point of view, all the information about everything we throw into the hole is held at the hole's surface. So it is perhaps understandable that the horizon area should scale up in proportion to the amount of information held there.

If this is indeed the case, the information density at the event horizon given by Bekenstein's bound is truly enormous: 1/4 nat of entropy for each square of area that is 1 Planck length, or $\ell_{\mathrm{P}} = \sqrt{G\hbar/c^3} = 1.62 \times 10^{-35}$ m, on a side. That is, an astounding 2.21×10^{70} bits per square meter, or 2.21×10^{50} bits per square Ångstrom (roughly atom-size) area. (It's probably safe to say that DRAM densities won't reach *that* level for a while!)

In any case, black holes are certainly not a very good place to store information that we might want to retrieve later, although they might conceivably be a good place to dump unwanted entropy. Macroscopic black holes have intrinsic temperatures near absolute zero, and in contrast to most systems, they get *cooler* as you dump more energy and entropy into them! (*Cf.* eq. 26 in Smith's paper [152], and his references to Hawking, his source.) So a black hole would be a sort of natural heat sink, cooler even than the cosmic microwave background which is at ~3 K. But for the foreseeable future, black holes will remain rather hard to come by, so it behooves us to also consider where we stand without them.

2.2.2 Entropy bounds for a photon gas.

Much tighter bounds can be given for the entropy of normal (non black-hole) systems, given additional assumptions about their composition. This is done in Bekenstein's paper [15], as well as in papers by Joos and Qadir [88] and Smith (1995 [152]) and the related literature. Smith argues that for high-temperature systems (above 1000 K or so, roughly the melting point of ordinary solids), the maximum entropy density for a *given mass density* is approximately achieved (within a small constant factor) by a thermal photon gas, in which the entropy density (entropy per unit volume) is ([152], eq. 22)

$$\frac{S}{\mathcal{V}} = \frac{16\sqrt{\pi}}{3 \cdot 60^{1/4}} \left(\frac{c}{\hbar} \cdot \frac{M}{\mathcal{V}}\right)^{3/4} \tag{2.4}$$

where $M/\mathcal{V}$ is the energy density of the photon gas, in mass units.

This equation would appear to allow arbitrarily high entropy densities to be achieved by raising the temperature and mass-energy density, except that actually of course the energy density is itself limited as a function of a system's size, since beyond a certain point the system would form a black hole.

A full general-relativistic analysis of the situation would of course be very complex, but as a simplifying first approximation, let's derive the maximum entropy density of a sphere of photons without taking GR into account, except in the sense of allowing it to set a maximum energy for a system of given radius.

Using eq. (2.2) we find that a sphere of photons of (nonrelativistic) volume $\mathcal{V} = \frac{4}{3}\pi R^3$ must have energy density

$$\frac{E}{\mathcal{V}} \leq \frac{3\,c^4}{8\pi\,G\,R^2} \tag{2.5}$$

i.e., mass density

$$\frac{M}{\mathcal{V}} \leq \frac{3\,c^2}{8\pi\,G\,R^2} \tag{2.6}$$

to avoid gravitational collapse. Substituting this into eq. (2.4), we find that non-black-hole entropy is bounded by

$$\begin{aligned} S &\leq \frac{64}{9 \cdot 60^{1/4}} \left(\frac{3\pi}{8}\right)^{3/4} \frac{c^{9/4}}{(\hbar G)^{3/4}} R^{3/2} && (2.7) \\ &\approx 2.889 \left(\frac{R}{\ell_{\mathrm{P}}}\right)^{3/2} && (2.8) \end{aligned}$$

where ℓ_{P} is again the Planck length. This bound, interestingly, scales with increasing radius even less rapidly than in the black hole case, where we had $S = \pi(R/\ell_{\mathrm{P}})^2$. Incidentally, the ratio $S_{\mathrm{BH}}/S_{\mathrm{PG}}$ between black-hole entropy and maximum photon-gas entropy in this (admittedly simplistic) analysis is

$$\frac{S_{\mathrm{BH}}}{S_{\mathrm{PG}}} \approx 1.087\sqrt{\frac{R}{\ell_{\mathrm{P}}}}. \tag{2.9}$$

This ratio is required to be ≥ 1 by Bekenstein's argument that black holes always maximize entropy, which implies $R \gtrsim 0.846\,\ell_{\mathrm{P}}$, perhaps suggesting that $R \approx 0.846\,\ell_{\mathrm{P}}$, or thereabouts, is a minimum physical length in some sense. The entropy of either a black hole or maximum-energy photon gas sphere having that radius is ~ 3.24 bits, suggesting perhaps there is an absolute upper limit to entropy density for objects of *any* size, on the order of ~ 1.28 bits/ℓ_{P}^3, or $\sim 3.03 \times 10^{98}$ bits/cm^3. Of course this density would not be achievable for any object greater than about a Planck length in size.

But even for larger objects, our new bound (2.8), though lower than before, is still extremely high; for example, an object as large as the Earth could still have an average entropy density of 117 Gigabytes of information per cubic Ångstrom without exceeding this limit.

Given this, it is probably necessary to back a little further away from fundamental theoretical arguments, if we want to achieve any sort of meaningful bound.

2.2.3 More reasonable mass densities.

One observation is that if we rule out burying our computers inside star-sized or at least planet-sized masses of gravitating material, then there is probably no way to apply enough pressure to get their average mass density to be much greater than in ordinary solids. At a mass density of 10 g/cm^3 (about that of lead), eq. 2.4 gives only 36 kilobytes per cubic Ångstrom.

However, achieving even this much more reasonable entropy density using photons requires extremely high temperatures. The temperature of a photon gas of energy density ρ_{E} is (solving Smith's [152] eq. 13 for T):

$$T = \sqrt[4]{\frac{\rho_{\mathrm{E}} c}{4\sigma_{\mathrm{SB}}}} \tag{2.10}$$

where σ_{SB} is the Stefan-Boltzmann constant,

$$\sigma_{\mathrm{SB}} = \frac{\pi^2}{60} \frac{k_{\mathrm{B}}^4}{c^2 \hbar^3} \tag{2.11}$$

Achieving a mass density of 10 g/cm^3 (That's one heavy field of light!) thus requires a temperature of roughly 10^9 Kelvins. It is difficult to see how such high temperatures can possibly be maintained at ordinary pressures without completely destroying any structure the computer might have. Thus we need to move to still more conservative estimates.

2.2.4 More reasonable temperatures.

For example, at a more feasible temperature such as the melting point of copper, 1356 K, the energy density of light is only 2.56×10^{-9} J/cm^3, so it is essentially weightless, at 2.85×10^{-23} g/cm^3. Moreover, the *entropy* density is then only 0.74 bits per cubic micron. (This makes sense since ordinary visible light, emitted by glowing but still-solid blackbodies, has wavelengths on the order of a micron.) This clearly is much less than the entropy density of the hot copper atoms themselves, which exist at a density of roughly 0.08 atoms per cubic Angstrom. So at temperatures where any useful solid structure can exist, the energy density of light is very low, and it is also far from maximizing the possible entropy density.

What is the entropy density in an ordinary solid material like lead? It may conceivably be on the order of kilobytes per atom, as we calculated above for light of the same density, if information about the nuclear structure is included in the count—after all, Smith's argument tells us to count the total mass-energy of the system in computing his bound.

However, without a reliable way to probe the structure of nuclei, most of this information, even if it is there, will be inaccessible as a place to store information for later retrieval. The nucleus may nevertheless be capable of absorbing some amount of thermal information (heat entropy), but I have not researched whether a figure as high as 36 kilobytes of entropy at normal temperatures is consistent with what is known about nuclear structure and the heat characteristics of atomic materials. If variability in the nuclear structure does not make a large contribution to total atomic entropy, then the actual maximum entropy per atom in normal solids is probably much lower than the 36 kilobyte figure.

However, at room temperatures, entropy density is probably not much lower than on the order of 1 bit per atom (or per cubic Ångstrom), since at those temperatures atoms have enough energy to jiggle around a little, and so will have on average 1 nat (k_{B}) of entropy ($k_{\mathrm{B}}T$ energy) per vibrational degree of freedom. For three-dimensional vibrations, there are six degrees of freedom, three of position and three of momentum, so this gives 6 nats$/\ln 2 \approx 8.66$ bits per atom. There should also be entropy contributions from variability in the nuclear spin orientation, and from electrons that are free to roam in molecular orbitals or in conduction bands. But most atoms are somewhat larger than 1 $\mathring{\mathrm{A}}^3$ in volume, so 1–10 bit/Å^3 is still probably the right overall order of magnitude for entropy density in normal materials.

A more detailed (but still fairly crude) analysis based on actual thermochemical data from the CRC handbook [107] suggests that experimentally, at atmospheric pressure, the entropy density for copper is indeed found to be in the rather narrow range 0.5-1.5 bits/Å^3 for a wide range of temperatures from room temperature up to its boiling point, and moreover that the entropy densities in a variety of other pure elemental materials are also close to this level. For mixtures, we would expect the entropy density to be potentially greater, due to the additional degree of freedom implicit in choosing what species of atom resides at any given location.

Table 2.1 summarizes the above results by giving the average entropy density of a sphere of radius 1 meter that contains the maximum entropy according to the various

Material	Upper bound on entropy density	Caveats
Black hole	4.14×10^{39} b/Å^3	Need mass ≈ Saturn; can't get info. out
Non-black hole	1.53×10^{22} b/Å^3	Requires nearly as much mass
Normal density	$\sim 3\times10^5$ b/Å^3	May require billion-degree temperatures
Atomic matter	$\sim$ 1–10 b/Å^3	Hand-waving estimate.

Table 2.1: Theoretical limits on entropy or information density for a 1-meter-radius sphere, in various scenarios. The radius is important because in the high-gravity regime, the maximum average entropy density decreases with increasing size. It is difficult to know which of these limits, if any, might someday be approachable in real computational systems.

bounds. (Keep in mind that entropy actually scales less rapidly than volume for the systems near black-hole mass.)

This concludes our discussion of information density limits. Although we were unable to determine precisely the maximum density that was possible, we saw that entropy density does appear to ultimately be limited by some function of energy density, such as in eq. (2.4). Furthermore, much of a system's rest mass-energy may not count for purposes of this calculation, if it is energy that is tied up in an inaccessible nucleus, for example. At this stage I believe it would be premature to predict that a density greater than say $\sim$ 10 bits per cubic Ångstrom could ever actually be achieved for stable, retrievable storage of information. I would need more information before I could make a similar statement regarding thermal entropy densities.

2.3 Information flux rate limits

Another physical quantity of importance in computation is the maximum flux (rate of flow per unit area) of information or entropy through any surface in the computer. We should point out that one class of bounds on this quantity immediately follows from the bounds of sections 2.1 & 2.2, as follows.

Suppose a material having entropy density ρ_{S} passes through of surface at velocity v. Then the entropy in that material is crossing the surface with exactly the flux $F_{\mathrm{S}} = \rho_{\mathrm{S}} v$. Section 2.2 gave us bounds on the maximum value of ρ_{S}, and the maximum v is of course c, so this leads immediately to corresponding bounds on F_{S}.

One caveat is that in normal materials traveling at near the speed of light, the relativistic length contraction of the material should increase its effective entropy density, according to

$$\rho'_{\mathrm{S}} = \gamma\rho_{\mathrm{S}} \tag{2.12}$$

where

$$\gamma = \frac{1}{\sqrt{1 - v^2/c^2}} \tag{2.13}$$

is the normal relativistic correction factor (*cf.* [52]). We assume that a given chunk of material maintains the same entropy at high speed, but is compacted into a shorter space. This would seem to allow arbitrarily high fluxes to be attained.

However, in addition to this compression into a smaller volume, the chunk will also have its mass-energy increased (in the reference frame of the stationary surface through which the material is passing) by another factor of γ, so that the mass-energy density of the moving material will actually scale as

$$\rho'_{\mathrm{M}} = \gamma^2\rho_{\mathrm{M}}. \tag{2.14}$$

and so, solving (2.14) for γ and substituting into (2.12), the entropy density actually scales as

$$\frac{\rho'_{\mathrm{S}}}{\rho_{\mathrm{S}}} = \sqrt{\frac{\rho'_{\mathrm{M}}}{\rho_{\mathrm{M}}}}; \tag{2.15}$$

i.e., the increase in entropy density only scales as the *square root* of the energy density. So asymptotically, we could thereby do no better than with light, which already travels at lightspeed and where the entropy density scales with the energy density to the 3/4 power, according to eq. (2.4). So the maximum entropy flux we derive from our entropy density bounds is *not* exceeded in materials traveling at relativistic speeds, if the energy invested in accelerating the material to that speed is taken into account.

Smith 1995 [152], p. 6, eq. 7 gives an explicit formula for the maximum entropy flux F_{S} using light, given an energy flux F_{E}:

$$F_{\mathrm{S}} \leq \frac{4}{3}\sigma_{\mathrm{SB}}^{1/4}F_{\mathrm{E}}^{3/4} \tag{2.16}$$

This is the formula for the entropy flux emitted by a blackbody that is at the appropriate temperature to emit energy flux F_{E}. As Smith points out, there is a simple proof that this is the maximum entropy flux that can be transmitted with photons given that energy flux. Imagine using photons to continuously transmit energy and entropy through a small aperture into an insulated box (a perfect blackbody). The interior of the box will heat up, and, at equilibrium, will radiate energy out of the aperture exactly as fast as it is coming in (since energy is conserved), and will also radiate entropy out *at least* as fast as it comes in (since global entropy cannot decrease). Therefore the entropy flux of the thermal blackbody radiation coming out of the box upper-bounds the achievable entropy flux of the light coming in, which may be of any form (coherent, *etc.*).

At this point we could go on to calculate upper bounds on information flux at *any* energy based on the black hole limits to entropy density that we discussed in §2.2. For example, for a postulated minimum-size (Planck-length scale) black hole moving at near the speed of light, we estimate entropy flux would be around 10^{109} bit/s-cm^2. However, this sort of bound is rather far from anything meaningful, since it does not represent a sustainable rate, or a rate achievable over an area much larger than a Planck length—black holes placed near each other would rapidly conglomerate into a larger black hole with lower entropy density. Even if we were so bold as to allow for the use of such exotic objects as black holes as computer components, properly accounting for gravitational effects in such systems would make our scaling analysis of chapter 6 much more complex. So instead, for the rest of the thesis, we will ignore high-gravity situations, and instead focus only on the bounds obtained for normal matter.

2.4 Computation rate limits

In chapter 6 we will examine in detail how certain kinds of limits on computation rates for irreversible and imperfectly-reversible computers can be derived from the limits on information flux we saw in §2.3. However, there are other limits on processing rates that apply even to perfectly reversible computers.

In particular, there is the result of Margolus and Levitin (1996, [118]) that the fundamental laws of quantum mechanics imply that the maximum rate $\nu_\perp$ at which a system at an average energy E (above some minimum energy E_0) can transition between distinguishable (*i.e.*, orthogonal) states is

$$\nu_\perp \leq 4(E - E_0)/h. \tag{2.17}$$

This bound is derived in a totally general way, and applies even for systems traveling at relativistic velocities. Insofar as any computational operation requires that some part of a system change from one distinct state to another, Margolus and Levitin's bound is an absolute upper limit on the rate at which operations can be performed within a computer.

Further, Margolus suggests [personal communication] that for systems in which not all the system's energy is accessible for computational purposes (for example, if some of it is in the form of heat, or tied up in rest mass), it is the *free energy* of the system, rather than its total energy, that determines the maximum rate at which the system can transition between useful computational states according to eq. (2.17).

As a simple example, a single electron excited to a potential of 1 Volt above its ground state contains 1 eV of accessible energy and thus can never perform computational steps (or any state change) more rapidly than at a rate of 4 eV/h = 9.67×10^{14} Hz, or about once per femtosecond.

2.5 Reversibility of physics

Another physical constraint of great importance for computation is that all physical dynamics is reversible (invertible), that is, it is deterministic looking backwards in time. (See figure 2.1.)

Quantum mechanics is sometimes described in nondeterministic terms, but it is actually perfectly deterministic (and reversible) at the level of the evolution of the quantum wave function. The apparent nondeterminism of quantum events can be interpreted as merely a subjective, emergent phenomenon that is predicted perfectly well by the underlying deterministic theory [57].

One possible exception to reversibility may be black holes, which, in some theoretical arguments, are found to destroy information (see Preskill 1992 [138] for a review of the situation). However, there is currently no accepted, complete theory of black hole physics from which we could draw indisputable theoretical conclusions, and there is no experimental evidence that supports information loss. The truth of the issue is still being actively debated (*e.g.*, [55, 120]). Moreover, it appears that some recent developments in string theory would allow reversibility to be maintained, if the theory is correct (Myers 1997, [130]).

In any case, it seems to be the general consensus among physicists that reversibility is certainly maintained in at least all areas of mechanics that do not involve extreme situations such as black holes. So regardless of the black hole situation, physics remains reversible for all practical purposes.

2.5.1 Physical reversibility and information erasure

Another way of characterizing physical reversibility is that two states (of a classical system, or of a quantum wavefunction) that are initially distinct can never evolve to become the *same* state at some later time. (In the language of functions, the system's transition function over any time period is one-to-one/bijective/invertible.) Consequently, the number of possible states of a system is irreducible over time; we say that the system's state space is incompressible. The invertibility is a simple consequence

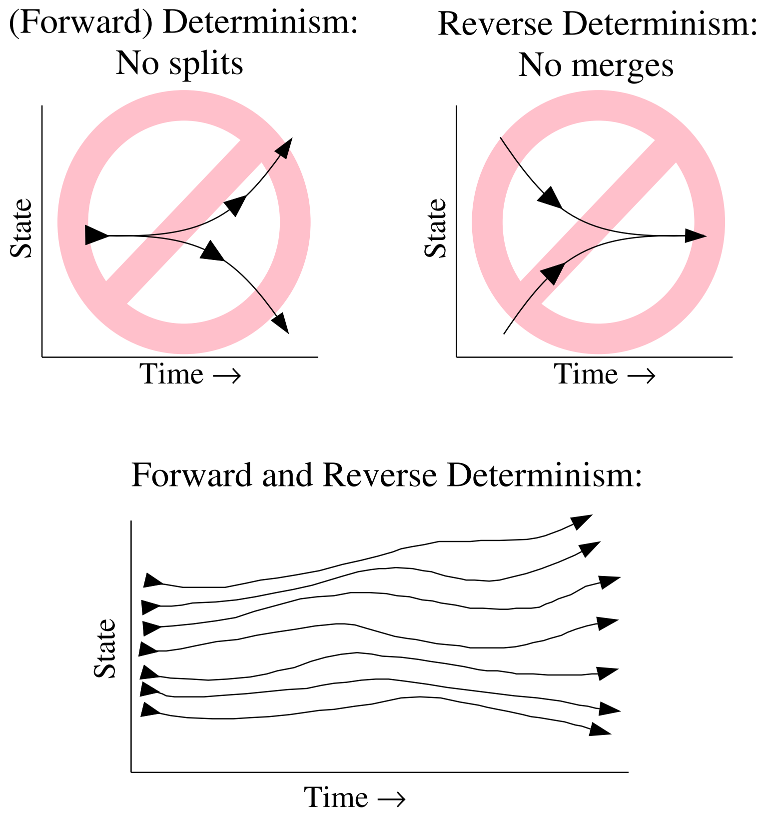


Figure 2.1: Forward and reverse determinism in physics. Normal forward determinism means that a single state cannot evolve to become one of two different states at any single later time, and similarly, reverse determinism or just *reversibility* means that two initially-distinct states cannot evolve to become the *same* state at some later time. Physics is both forward and reverse deterministic, and so the possible trajectories of a system through configuration space-time never intersect.

of the fact that in any standard description of physics, such as Hamiltonian dynamics, an isolated physical system evolves according to a time-differential equation. The differentials apply equally well in either time direction.

[Note: Our "incompressibility of state space" concept is closely related to, but not the same thing as, another property that is shared by all systems with Hamiltonian dynamics, namely that their "phase space *volume*" is incompressible. However, we will not delve into those rather subtle distinctions here, as they are not important for our immediate purposes.]

In any event, the incompressibility of state space has an important consequence for information erasure within a computer, first described explicitly by Landauer [97]. Whenever we attempt to irreversibly erase a piece of information from a computer, that information is not truly destroyed, but instead is simply *transferred* to another part of the system, typically to the uncontrolled thermal state of the computer and its environment. We explain in more detail with reference to figure 2.2.

The figure illustrates a 1-bit piece of computational state within a computer. We wish to perform an "erasure" operation, which we may characterize as an operation that transforms that bit to a zero regardless of whether it was originally a 0 or a 1.

In addition to the bit in question, the computer also contains some amount of other information in the form of other bits in memory, together with the entropy of its thermal state. Let $\mathcal{N}$ denote the number of possible states of the system, apart from the bit in question.

We want our "erase" operation to operate correctly, independently of which of the $2\mathcal{N}$ possible states the combined system is in. Due to physical reversibility, each of these $2\mathcal{N}$ states must be mapped to a distinct state after the erase operation—but all of those states have value 0 in the erased bit. Thus there must be $2\mathcal{N}$ possible states of the rest of the system, after the operation. The amount of information in the rest of the system has therefore increased by $\lg 2\mathcal{N} - \lg \mathcal{N} = 1$ bit.

So the presumably erased information has not really been destroyed, but is still present somewhere, either in some other part of the computational state or in the thermal state. The original value of the bit could *in principle* be retrieved by, for example, running the laws of physics backwards.

However, if the information has been lost in a sea of thermal chaos, then *in practice* there is *no* way to reconstruct the original value of the bit.

2.5.2 Reversibility, entropy, and the second law

We now see how physical reversibility can be understood to imply the second law of thermodynamics.

The second law of thermodynamics states that the total entropy of any closed system cannot decrease. What is entropy? Quantitatively, it is the logarithm of the

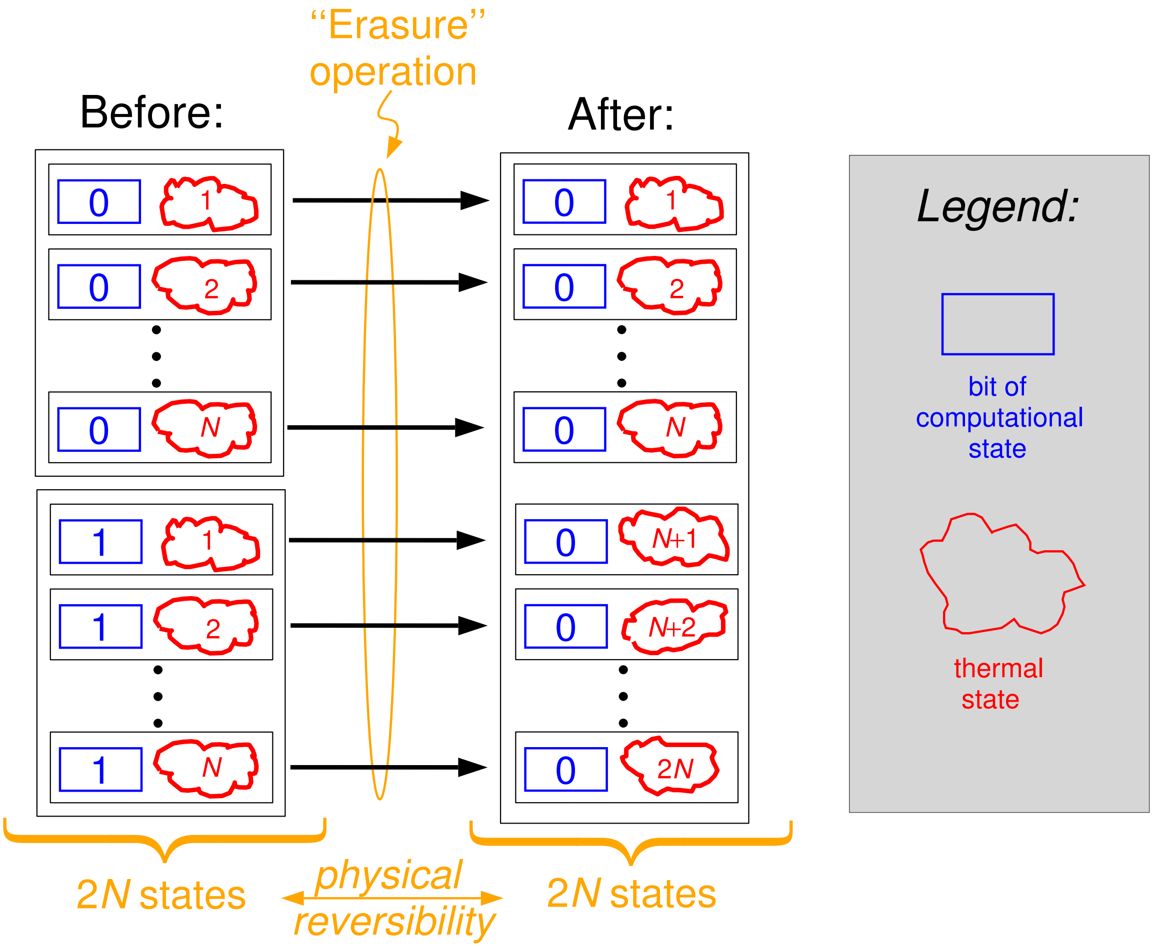


Figure 2.2: Information "erasure" under reversible physics. In order to erase an unknown bit and thereby reduce the number of possible digital states of the computer by a factor of 2, one has to make up for this by increasing the number of possible thermal states of the rest of the system by a factor of 2.

number $\mathcal{N}$ of possible states of a system. The base of the logarithm determines the unit of entropy: if the base is $e \approx 2.718\ldots$, the base of natural logarithms, then we might call the corresponding unit of entropy 1 *nat*, equal to Boltzmann's constant k_B. If the base is 2, then the unit of entropy is called 1 *bit*. Thus, 1 bit = $(\ln 2)$ nat $\approx$ 0.693 nat.

If entropy is the log of the number of possible states, what, then, do we mean by a "possible state?" This depends entirely on the context, specifically on how we define what constitutes a legal example of "the system" in question.

However, even with this broad definition of entropy, we can already make some meaningful statements about entropy in connection with reversibility. First, it is clear that if the entropy of a system were to decrease over time, then the system would not be reversible, because we would have an example of multiple possible initial states evolving to become a smaller number of resulting final states, violating the incompressibility of state space that is implied by reversibility. Therefore, the reversibility of a system immediately implies that its entropy can never decrease over time.

There is a similar connection between entropy increase and determinism. In a deterministic system, state space is "inexpandable" since a given state can not evolve to more than one possible new state in a given amount of time. Thus the number of possible states cannot increase, in this strict sense, and so deterministic systems undergo no "true" increases in entropy.

However, even if a system is deterministic, we may find it convenient to label more and more states as "possible" over a system's time evolution, simply because, given an incomplete model of a system's initial state, we may lose track of the exact trajectories of the initially possible states over time, and so many additional states may become possible over time from the point of view of the model. In such circumstances, it is convenient to say that entropy increases. An example is the situation in figure 2.2 (p. 44). Suppose we have constructed an initial condition in which only the "1" value of the bit is possible, so the entropy before the "erase" operation is just $\ln \mathcal{N}$. But since we fail to model what becomes of the information *that* the bit is 1 after the "erase" operation is performed, the entropy of the system under the model increases to $\ln 2\mathcal{N}$. This increase will happen whenever a non-entropy bit turns into thermal form, because the evolution of the micro-state of a thermal system is, by definition, un-trackable by us.

We thus can state the following principle: Total entropy increases (permanently) by at least 1 bit's worth any time a bit that is originally non-entropic moves to reside in a thermal system. Furthermore, this happens whenever a digital bit is erased, unless (a) the bit was *already* entropy, in which case moving it to thermal form does not necessarily increase total entropy, or (b) the bit is canceled out instead, by uncomputing it from other bits of state with which it is correlated.

2.5.3 Entropy and energy

As we just saw, the second law of thermodynamics states that the entropy of a closed system cannot decrease over time. We saw that it also cannot increase, except in the sense that our incomplete model of the system may lose track of what happens to a state over time, so that more states become "possible" from the point of view of the model. However, entropy *can* be moved from one subsystem to another.

Correspondingly, the *first* law of thermodynamics states that the *energy* of a closed system can neither increase nor decrease, but can only be moved from one subsystem to another.

How are these conservation laws for energy and entropy related? We find empirically that in order to increase or decrease the entropy of any subsystem (not counting increases due to deficiencies in our model), we generally must also increase or decrease its energy (given a closed, constant-volume system). For sufficiently small changes in entropy, we find that the change in energy required is proportional to the change in entropy. The constant of proportionality is called the *temperature* T of the subsystem. Formally,

$$T = \partial E / \partial S. \tag{2.18}$$

This is a perfectly valid definition of temperature, in terms of the relation between a system's energy and its number of states.

Under this definition, 1 Kelvin of absolute temperature is definable as a requirement of $\approx 1.38 \times 10^{-23}$ J of energy per 1-nat increase in entropy. A nat of entropy can therefore also be expressed in units of energy per unit temperature, such as 1.38×10^{-23} J/K. In such form, 1 nat of entropy is often referred to as *Boltzmann's constant* k_{B}.

From all this, it follows immediately that the amount of energy E that must be added to system in order to double its number of possible states is just

$$E = k_{\mathrm{B}} T \ln 2 \tag{2.19}$$

since $k_{\mathrm{B}} \ln 2$ is just a 1-bit increase in entropy, and multiplying by the system's temperature just converts this entropy increase to the required change in energy, by the definition of temperature.

2.5.4 Logical irreversibility and energy dissipation

We saw in section 2.2 that in any system with particular size and energy there is a consequent upper bound on the entropy that system can contain. If a given system is found to contain less entropy than the maximum given the amount of energy in the

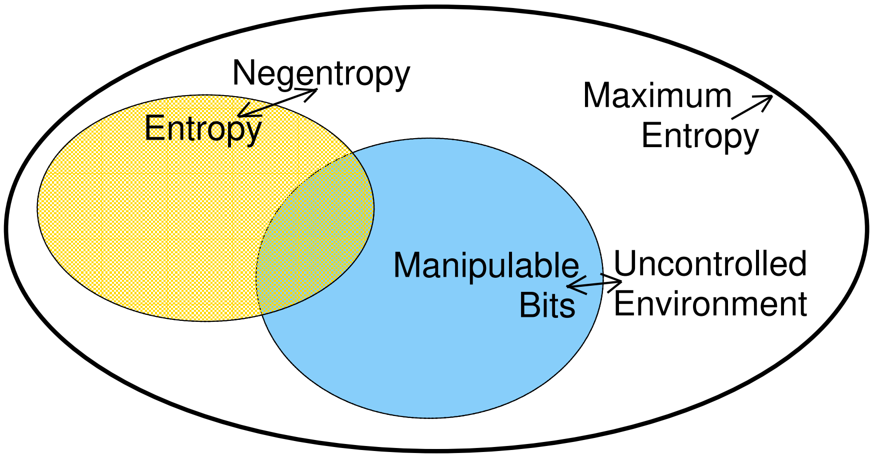


Figure 2.3: Venn diagram of entropy and information. Any system of finite size and energy has a finite maximum entropy; however, if the system expands without bound, the maximum entropy may also. The maximum entropy may be considered the total amount of information of all kinds in the system. However, much of it may be redundant and cross-correlated. Bits of information that are uncorrelated, or whose correlations have become lost beyond all hope of recovery, are entropy, which can only increase in a reversible universe. Some portion of the system, and the information in it, is within our ability to manipulate and control, such as bits within a computer. These bits, too, may either be entropy or not, depending on our ability to know their correlations.

---

system, then that must mean our model of the system is imposing further structure on the system, ruling out some of the states that would otherwise be possible.

For such a system with non-maximal entropy, only a portion of the energy of the system is actually needed for permitting the entropy that is actually present. This portion of the total energy will be referred to as the amount of *dissipated energy* in the system. The rest of the system's energy will be referred to as its *free energy*. The difference between the entropy of the system and its maximum entropy will be termed the *negentropy* or *information capacity* of the system. Some of this information capacity may become allocated for storing computational information. (See fig. 2.3.)

As we discussed in §2.5.2, even in the context of a perfectly deterministic underlying physics, the entropy of a system can be seen to increase, through a failure to completely model the determinism inherent in the system's physical evolution. When this happens, the amount of the system's energy that is needed to support this entropy will increase by some amount, and the free energy will decrease by the same amount. We say this amount of energy has been *dissipated*.

We are now in a position to accurately state and explain the central statement on which the field of reversible computing is based:

Landauer's principle. *The irreversible loss of 1 bit of computational information requires the dissipation of* $k_\mathrm{B}T \ln 2$ *energy*, where T is the temperature of the subsystem in which the lost bit finally ends up. By the "irreversible loss," we mean that some bit of *computational information* (not a bit that is already entropy!) becomes transformed in such a way that our computational models can not track it, for example by becoming mixed up with parts of the system whose state is already *thermal*, or unknown. Thus by definition the bit has become entropy, and the entropy of the system as a whole is increased by 1 bit. This increase is eventually reflected in some subsystem at temperature T, and by definition of temperature, the energy of this subsystem must be increased by $k_\mathrm{B}T \ln 2$. The energy invested in the entropy increase is heat. If T is the lowest available temperature, then this energy must come out of the free energy, because all the dissipated energy in the system is already fully occupied with containing the pre-existing entropy. Thus the free energy is decreased by $k_\mathrm{B}T \ln 2$.

The above principle was first explicitly conjectured by Landauer [97].

Note that since T is the temperature of the system where the entropy finally ends up, not the temperature of the device that held the entropy originally, cooling a computer cannot in the long run decrease the total energy dissipation required to erase bits, if the dissipation in the cooling system is taken into account. The entropy that is generated can not build up indefinitely in the cooling system, or else it would not stay cool. Instead, it ultimately ends up in some natural thermal reservoir in the environment. The coolest thermal reservoir of effectively unlimited capacity that might be available in the foreseeable future is the interstellar microwave background, at a temperature of ~2.73 K. Thus, no process that generates entropy can, in the long run, sustain an energy dissipation cost less than $k_\mathrm{B}(2.73\,\mathrm{K}) \ln 2 \approx 2.6 \times 10^{-23}$ J per bit generated, and this can only be attained if the entropy can be transmitted directly into space. For terrestrial systems that use the atmosphere as their thermal reservoir, the relevant temperature is in the neighborhood of room temperature or 300 K, for a minimum energy dissipation of $\sim 3 \times 10^{-21}$ J/b.

2.6 Quantum computation

One area in which physics may actually constrain computation *less* than might be expected is in the possibility of quantum computation (*cf.* [60, 47, 23, 22, 21, 149, 146]), that is, computation using large, complex, coherent superpositions of states. If it can be implemented successfully, quantum computation seems likely to be strictly asymptotically faster than classical computation on certain problems, by as much as an exponential factor. But it is not yet known if quantum computation would be beneficial for purposes other than obsoleting the RSA cryptosystem, or simulating physical quantum systems. Still, if the recent progress on implementing quantum computers [56, 168, 35, 39, 76, 40, 158, 49, 150, 10, 34] eventually culminates in success, then we would certainly like to consider quantum computation as a physically possible means of computation. But even a quantum computer would still need to obey the fundamental constraints discussed above affecting the maximum density and propagation speed of information.

We will review quantum computing in more detail in chapter 4.

2.7 Physical constraints—conclusion

This concludes our discussion of fundamental physical limits on computation. Table 2.2 summarizes the limits we discussed, and the presumed effect on the form of a physically-realistic model of computation, which we will discuss further in ch. 5.

In chapter 6 we will see how these limits affect the scaling of computation speeds in reversible and irreversible computers. But first, in the next chapter, we review the non-physical theoretical underpinnings of reversible computing, and show that in an imagined non-physical computational framework, reversibility leads to unfavorable scaling. The contrast between that result and the results of chapter 6 underscores that traditional non-physical theoretical frameworks for computation are inadequate for realistically modeling the advantages of reversibility, and thus, more sophisticated models of computation that take the above-described physical constraints into account are required for a correct analysis. Such models will be discussed in chapter 5.

Fundamental principle	**Constrained quantity**	**Symbol**	**Quantitative constraint**	**Impact on our model**
Quantum mechanics	Entropy density	ρ_S	$\lesssim$ 1–10 b/Å^3?	Finite state/ processor
	Entropy flux	F_S	$\leq \rho_\mathrm{S} v$	Finite info. flux
	Rate of state change	$\nu_\perp$	$\leq 4(E - E_0)/h$	Finite oper. frequency
Locality	Info. prop. velocity	v	$\leq c \approx 3\times 10^8$ m/s	Mesh arch. (Vitányi '88)
3-dimensionality of space	Connectivity		$\mathcal{O}(t^3)$	3-D mesh
Micro-reversibility, thermodynamics	Entropy change	ΔS	≥ 0 always, ≥ 1 bit/bit erasure	Logical reversibility, entropy accounting
	Energy dissipation	ΔE	≥ 0 always, $\geq k_\mathrm{B} T \ln 2$/eras.	
Frictional effects	Entropy coefficient	k_S	> 0 b/Hz?	Time-prop. reversibility

Table 2.2: Fundamental physical constraints on computation, and their effects on the form of a physically-realistic model of computation. The value of the bound on ρ_S is very uncertain, but the assertion that some such bound exists is not. For any particular computing technology through the foreseeable future, there will generally be much stricter limits than the above on most of these quantities.

Chapter 3

Reversible computing theory

In the previous chapter, we set the stage for our research by reviewing the known physical limits on computation, including the entropic cost of logically irreversible information loss. We saw that avoiding this cost requires the use of computational primitives that possess the special property of logical reversibility. This observation leads naturally to the question: What implications would logical reversibility have in the context of the traditional theory of computation?

This chapter addresses that question, while also introducing the related question of how the traditional measures of complexity and models of computation will need to be adjusted to more effectively cope with thermodynamic issues and other important physical considerations. That line of study is continued in chapters 5 and 6.

One reason that computer designers have not yet rushed to adopt reversible computing principles is that purely reversible operation is not necessarily optimal in all circumstances. For many applications of computer technology today and in the future, energy dissipation may not be a limiting factor. In such circumstances, purely reversible operation appears to incur significant computational overheads compared to irreversible operation. If energy dissipation is modeled as costing exactly nothing, then it seems that the total cost overhead factor for pure reversible computing becomes unboundedly large as problem sizes increase.

In §3.4 of this chapter, we will rigorously prove a technical theorem in computational complexity theory which suggests that such overheads are inevitable, and that no amount of clever improvements of reversible algorithms can avoid these overheads on all problems. This result indicates that if we wish to be able to perform asymptotically optimally *even under cost models in which the energy cost is zero*, then our computer models should at least include the *option* of not being completely reversible.

However, in chapter 6, we will show that if energy dissipation has *any* non-zero cost, then our physical model of computation must also include the option to have an arbitrarily *high* degree of reversibility, if it is to achieve asymptotically optimal speed and cost-efficiency on all problems.

But first, in §3.1 and §3.2, we will review general concepts of models of computation, computability, and complexity, and introduce a few new measures of complexity that attempt to better capture important physical considerations. Then in §3.3 we review the major results of existing reversible computing theory, leading up to our own contributions in §3.4. Finally, §3.5 sums up the comparison of traditional reversible and irreversible computing models.

3.1 Models of computation

Discussions of the theory of computation often start with the definition of a particular model of computation to work with, such as, for example, Turing machines. However, in this thesis, we do not wish to pick a particular model, since our interest is in comparing the relative efficiency of different models. If we wished to pursue a completely formal mathematical approach, we would need to give a precise definition of what a model of computation *is* in general, describe various particular models in terms of that general framework, define what it means to compare two models under that framework, and then prove various theorems comparing the different models. This would be straightforward but tedious, and it is unclear whether we would learn anything important from that highly formal approach that is not already sufficiently clear using our more informal understanding of the situation.

Therefore, in this thesis we will refrain from presenting a detailed formal explication of the concept of a "model of computation," and instead we will rest our discussion on the intuitive understanding of the phrase that the reader will be expected to have, given a general background in computer science. To refresh the reader's memory, a partial list of existing models of computation may be helpful (table 3.1).

Informally speaking, a model of computation merely delineates a space of abstract computing machines, and the computations that run on them. Most models were originally introduced as an attempt to approximate some class of physical machines; however, the existing models unusually end up ignoring one or another of the important realities of physical law that we saw in chapter 2. Sections 5.2 and 5.3 of ch. 5 review some of the problems with the existing models, and discusses candidates for a new model (which we might call PM, the "physical machine") intended to exactly represent the computing capabilities of physics.

Physically realistic or not, any abstract model of computation needs to be reduced to a physical implementation in order to actually run. In chapter 6 we will compare the power of two fairly realistic classes of models of physically-implemented machines: the FIA (fully irreversible architectures) and the TPRA (time-proportionally reversible architectures), and we show that the TPRAs are strictly more efficient, in several

Notation	Model name	Example references
PRF	Primitive recursive functions	Rogers 1987 [141], §1.2, pp. 5–9
RF	Recursive functions	[141], ch. 1
FA	Finite automata	Hopcroft & Ullman 1979 [86], ch. 2
RFA	Reversible finite automata	Pin 1987 [136]
TM	Turing machines	[169]; [86], ch. 7
RTM	Reversible Turing machines	[99, 16, 98]
NTM	Nondeterministic Turing machines	[86], §7.5
CA	Cellular automata	von Neumann 1966 [182], Toffoli & Margolus 1987 [164]
BBM	Billiard-ball model	Fredkin [74]
RAM	Random access machines	Papadimitriou 1994 [133], §2.6
PRAM	Parallel random access machines	Papadimitriou 1994 [133], §15.2, pp. 371–375
BLC	Boolean logic circuit	Papadimitriou 1994 [133], §4.3
3dM	3-d mesh	Leighton 1992 [101], ch. 1

Table 3.1: Some existing theoretical models of computation.

physically-relevant senses.

3.1.1 Computability

For any model of computation, an obvious first question is “What computations can it possibly perform?” (Given unlimited resources.) This question was the subject of much early research on computation, but eventually it was realized that a large variety of physically reasonable models of computation can all compute exactly the same set of functions, namely the *recursive* (now just called *computable*) functions (cf. [141]), and so the issue became less interesting. The famous “Church’s thesis” is the conjecture that the recursive functions *are* indeed exactly the functions that real physically-realizable machines can compute; the conjecture is true as far as anyone knows, and it would be extremely surprising if physical machines were to turn out to be able to compute non-recursive functions.

Of course, there also exist weaker models of computation that cannot even compute all recursive functions, such as finite automaton (FA) models.

With computability turning out to be mostly a non-issue, the next natural issue in computing theory is to discover how difficult or *complex* one finds various computational tasks to be under a given model of computation, or (in complementary terms), how *efficiently* the model can perform on various tasks.

3.2 Computational complexity and efficiency

Now we review some of the basic concepts used in traditional computational complexity theory, and extend them to capture some new, more general measures of computational complexity and cost-efficiency that will help us better address real-world concerns in later sections.

3.2.1 Computational efficiency vs. computational complexity

The focus of this thesis is on how to achieve maximum *computational efficiency*, which can mean several things, but most often we will use it to mean *cost efficiency*, defined as follows.

Given some way ¥ of characterizing the *cost* of a computation (or any process), one very general notion of efficiency is the fraction of the cost that is actually well-spent. In other words, if the minimum *possible* cost to perform some task is $\$_{\text{min}}$, and the actual costs incurred by a particular computation that performs that task are $\$$, then we can say that the cost-efficiency $\%_{\$}$ of the computation (under the cost measure ¥) is

$$\%_{\$} = \frac{\$_{\text{min}}}{\$} \tag{3.1}$$

because only $\$_{\text{min}}$ out of the total cost $\$$ was really warranted; the remainder $\$ - \$_{\text{min}}$ was wasted.

Thus, whatever the minimum cost $\$_{\text{min}}$ for a task, in order to maximize the efficiency $\%_{\$}$, one should try to minimize the actual cost $\$$. This leads to the frequent emphasis in computer science on characterizing and studying various abstract measures of cost, which are often referred to in theoretical computer science as measures of *computational complexity*.

3.2.2 Characterizing computational complexity

In this section we examine how measures of computational complexity are traditionally characterized, and propose the use of some new complexity measures that may allow different computational models to be compared in a more realistic way.

3.2.2.1 Scaling with problem size

When comparing the cost-efficiency of two algorithms or two models of computation, it is sometimes difficult to make a definitive distinction as to which candidate is better,

Our Notation for the Cost Measure	Meaning
Computational cost measures.	
N_{ops}	Number of primitive operations.
T	Number of computational clock "ticks" (called "time" in traditional complexity theory).
S	Maximum memory used at any time (called "space" in traditional complexity theory).
(S, T)	Computational "space" paired with "time" (p. 57).
ST	Computational "space" times "time" (p. 57).
Physical cost measures.	
t_{phys}	Physical time taken.
$\mathcal{V}_{\max}$	Maximum physical volume of space used.
S_{tot}	Total entropy generated.
$\$_{\text{c}}$	Comprehensive physical cost complexity (p. 58).
$\$_{\text{s}}$	Simplified physical cost complexity (p. 58).

Table 3.2: Some measures of cost or complexity. We distinguish the non-physical, "computational" cost measures from the physical cost measures. The physical measures can be accurately determined only for models of computation that realistically take into account physical constraints on computation such as we discussed in chapter 2.

if one of them is more efficient on some problems, and the other one is more efficient at others. Even within a particular class of problems, one machine may be better at small problems and the other at large ones.

However, if we look at how the performance of the two machines scales as the problem size increases, it may often be the case that one machine performs better than the other at problems of *all* sizes above a certain size, and the ratio between the efficiency of the two machines may even grow unboundedly large as problem sizes increase. Asymptotic order-of-growth analysis (see table F.4, p. 409) is the traditional tool for determining if such relationships hold, because it allows ignoring the many details of algorithm design that cause constant-factor differences in complexity, which often end up being irrelevant in an asymptotic determination of which machine is better.

Table 3.2.2.1 lists several measures of complexity which we will now discuss.

3.2.2.2 Traditional measures of complexity

Traditionally in computer science, theoreticians study only very simple measures of complexity, in order to make their analysis easier. Two of the most popular measures are time complexity and space complexity.

Time complexity. The "time complexity" of a computation can be characterized simply as the amount of physical time t_{phys} that the computation takes (from its start to its end), or as the number N_{ops} of computational "operations" (at whatever level of interest) that are performed, which is proportionally equivalent to real time if, for example, operations are performed serially and take $\Theta(1)$ (*i.e.*, constant) time each. If operations are performed in parallel, a better approximation to time would be the number of "ticks" T of some (real or imagined) computational "clock" that is thought of as synchronizing the operations of all the processing elements.

The problem with using time complexity alone as a cost measure is that it ignores the cost of the computer that is needed to solve a problem with the minimum time complexity. The minimum time complexity might only be achieved by a computer that is unfeasibly expensive.

One may reply that the machine cost is negligible because it may be amortized over arbitrarily many uses of the machine into the future, but one can counter with the point that whenever the computer is fully occupied with solving the given problem, its components can not meanwhile be used for another problem, so there is an opportunity cost inherent in using a large machine that must be considered as well.

Thus, minimizing only time complexity may completely miss the solution that minimizes cost in the real world.

Space complexity. Another measure of computational complexity which attempts to take the machine cost into account is space complexity, that is, the maximum amount of digital storage (in bits, say) that is in use at any point during the computation (we will denote this as S).

Given fixed lower bounds to the physical size and mass-energy required for a bit's worth of storage, space complexity can also be equated (within a constant factor) to the amount of physical volume ($\mathcal{V}_{\max}$) or mass in the computer, assuming there are no cost advantages in storing bits with an asymptotically increasing mass-per-bit or volume-per-bit. We conjecture that asymptotically, this assumption is true.

Of course, like time complexity, space complexity by itself is also inaccurate for real-world situations. Most significantly, it ignores the impact of the length of time during which the given amount of storage needs to be used. If the storage requirements for a computation are large, but the computation is rather short, or even if just the time during which the bulk of the storage is in use is short, then the computation may actually be less costly, in real terms, than a computation that has a smaller formal space complexity but which occupies that space for an extremely long time.

3.2.2.3 Some new measures of complexity

Given the inadequacies of the most popular traditional measures of complexity, we now describe some new alternative measures which attempt to more closely approximate the real-world economics of computing.

Joint space-time complexity. We saw earlier that both space complexity and time complexity, although they each took important cost factors into account, were individually incomplete. We can try to improve on the situation by combining both space and time complexity into a single measure of complexity.

One way to combine a space complexity measure s and a time complexity measure t is to simply group them into a pair (s, t). We can define a partial order $\succsim$ between pairs (s_1, t_1) and (s_2, t_2) by saying, for example, that $(s_1, t_1) \succsim (s_2, t_2)$ iff $s_1 \succsim s_2$ and $t_1 \succsim t_2$. (The $\succsim$ notation is defined in table F.4, p. 409.) However, this approach suffers from the problems that two complexity measurements may be incomparable (for example if $s_1 \prec s_2$ but $t_1 \succ t_2$), and that it is difficult to define a numerical measure of overall efficiency in this system. However, this simple complexity measure still suffices for some purposes, such as for our proof in §3.4.

Space-time product complexity. One interesting, improved measure of complexity is the product of space and time complexity. This comes closer to a true measure of cost because it increases monotonically with both space and time and allows comparisons between any two instances. It can be viewed as a measure of *rental cost*, the cost of renting a computer having storage capacity s for a period of time t; we might expect such a cost to be roughly linear in both storage capacity and time. Another way to look at the product is as a measure of the total volume of spacetime (as in the theory of relativity) that is dedicated to the computation.

However, even the space-time product is still somewhat inaccurate, since it does not take into account that a particular algorithm may not have constant space usage over time, and that the resources that are unused by the algorithm during a particular period of time can (in an appropriate machine architecture) be used for solving other problems during that time, thus reducing the effective cost of the program whose complexity we are measuring.

Another point is that besides spacetime volume, there is another resource that a computation uses up: namely, free energy. Energy that is dissipated by the computer is forever unavailable for use in other computations, because it is in a disorganized, maximum-entropy form that cannot do useful work. (We discussed these issues in much more detail in §2.5.) So this dissipation has a cost. In fact, in contexts such as battery-powered portable computers, the energy costs may be fairly high because the readily-available supply of energy is so limited. So a comprehensive model ought to take energy costs into account. One way to characterize free energy loss is by the total amount of entropy that is generated during the computation (S_{tot}).

Finally, there is the point that the storage space itself can be separated into several constituent entities that separately contribute to the total rental costs: the mass-energy of the computation/storage medium, the volume of physical space it occupies, and perhaps even its surface area (real estate it occupies). Mass-energy can be further broken down into free energy and rest mass, which can be further decomposed into the cost of various types of constituent components and the raw materials that they are made of; but we will not go this far in our modeling.

These observations lead to the following new complexity measures.

Comprehensive physical cost complexity. For a computation (or really, any) process that increases total entropy by S, takes total real time t, and that at times $0 \leq \tau \leq t$ (between the start and end of the computation) occupies spatial volume $\mathcal{V}(\tau)$, contains free energy $E(\tau)$, rest mass $M(\tau)$, and has a minimum surface area of $A(\tau)$, we define the *comprehensive physical cost* $\$_{\mathrm{c}}$ of the process as

$$\$_{\mathrm{c}} \;\equiv\; \pounds_{\mathrm{S}} S + \int_0^t \Big[\pounds_{\mathrm{V}} \mathcal{V}(\tau) + \pounds_{\mathrm{E}} E(\tau) + \pounds_{\mathrm{M}} M(\tau) + \pounds_{\mathrm{A}} A(\tau) \Big] \, d\tau \tag{3.2}$$

where the various $\pounds_X \geq 0$ are *cost coefficient* constants whose values are parameters of the cost model. The $\pounds_X$ convert all cost elements to some canonical cost unit, perhaps even a monetary unit.

This cost model is very comprehensive, probably more so than needed. In our explorations of the efficiency of reversible and irreversible machines in chapter 6, we have found that not all of the above terms need to be included in the cost model in order to find the optimal machine configurations for the kinds of computational tasks we have considered so far. So we also suggest a simplified version of this model.

Simplified physical cost complexity. For a computation process that generates entropy S, takes total real time t, and that at times $0 \leq \tau \leq t$ requires a free energy allocation of $E(\tau)$, we define the *simplified physical cost* $\$_{\mathrm{s}}$ of the process as

$$\$_{\mathrm{s}} \;\equiv\; \pounds_{\mathrm{S}} S + \int_0^t \pounds_{\mathrm{E}} E \, d\tau \tag{3.3}$$

where the $\pounds \geq 0$ constants are parameters of the cost model. Given the Margolus-Levitin bound on computation rate from §2.4, the second term in this cost measure can be considered a measure of the maximum number of states that could be traversed using the given energy profile over the given time.

We propose that cost models like the above are appropriate for exploring the asymptotic physical limits of computation.

3.2.3 Complexity classes

A complexity measure tells us how to assign a cost to a particular instantiation of a computation process. In chapter 5 we will discuss a variety of models of computation processes. Given a complexity measure and a model of computation, we can characterize the complexity of any program written for that model, as a function of the length n_{in} of its input (in bits, say). The program complexity for length n_{in} is often defined as the worst-case complexity of the program over all the inputs of length n_{in}.

Further, we can define the complexity of a given *task* under a model of computation as the complexity of the program that performs that task with the lowest program complexity, on that model.

A *complexity class* is the set of all problems that can be solved under a given model of computation within given bounds on asymptotic complexity, according to a given complexity measure.

3.3 Review of existing reversible computing theory

In this section we review the past developments in reversible computing theory. Much of our predecessors' work can be interpreted as an attempt to compare the computational efficiency of reversible and irreversible machines under various complexity measures and models of computation. In this section we will show how each of the existing results can be interpreted in this way, and then in §3.4 and ch. 6 we will carry this effort onward to the new complexity measures that we proposed in §3.2.2.3.

3.3.1 Reversible models of computation

Reversible models of computation can be easily defined in general as models of computation in which the transition function between machine configurations has a single-valued inverse. In other words, the directed graph showing allowed transitions between states has in-degree 1. In this thesis we will always deal with machines that are deterministic, so that the configuration graph always has out-degree one as well. See figure 3.1, p. 60.

3.3.2 Computability in reversible models

As we already noted in §3.1.1, one of the most important questions to answer for any new kind of computation is "What functions it can compute at all?" This comes before efficiency questions, since obviously a machine's efficiency at a task is meaningless if the machine cannot even perform the task.

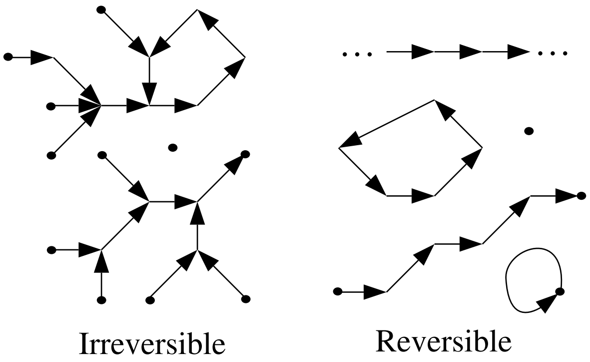


Figure 3.1: Machine configuration graphs in (deterministic) reversible and irreversible models of computation.

In the configuration graphs of irreversible machines, configurations may have many different predecessor configurations. In reversible models of computation, each configuration may have at most one predecessor. The configuration graph therefore consists of disjoint loops and chains, which may be infinite. In both reversible and irreversible models we may, if we wish, permit configurations having 0 predecessors (initial states) and/or 0 successors (final states).

3.3.2.1 Unbounded-space reversible machines are Turing-universal

In his 1961 paper [97], Landauer had already pointed out that arbitrary irreversible computations could be embedded into reversible ones by simply saving a record of all the information that would otherwise be thrown away (*cf.* §3 of [97]). This observation makes it obvious that reversible machines with unbounded memory can certainly compute all the Turing-computable functions.

We will call this idea, of embedding an irreversible computation into a reversible one by saving a history of garbage, a "Landauer embedding," since Landauer seems to have been the first to suggest it.

3.3.2.2 Reversible finite automata are especially weak

In contrast, in 1987 Pin [136] investigated reversible *finite* automata, which he defined as machines with fixed memory reading an unbounded-length one-way stream of data, and found that they cannot even decide all the regular languages, which means that technically they are strictly less powerful than normal irreversible finite automata, which are in turn strictly less powerful than unbounded-space Turing machines.

So there are functions computable by an irreversible machine with fixed memory that no purely reversible machine with fixed memory can compute, given an external one-way stream of input. We should note, however, that this incapacity might be due solely to the non-reversible nature of the input flow, rather than to the finiteness of the automaton memory itself. Conceivably, if a finite reversible machine was permitted to read backwards as well as forwards through its read-only input, and perform some sort of "unread" operations, it might then be able to recognize any regular language. But we have not investigated that possibility in detail.

That issue aside, in the rest of this thesis we consider models of computation that permit access to arbitrarily large amounts of memory as input sizes increase. For such machines, pure computability is no longer an issue, and we turn to questions of computational efficiency.

3.3.3 Time complexity in reversible models

One of the most common simple measures of computational cost we have seen is "time complexity," which in a theoretical computer science context often means the number of primitive operations performed. Landauer's suggestion (*cf.* §3 of [97]) of embedding each irreversible operation into a reversible one makes it clear that the number of such operations in a reversible machine need not be larger than the number for an irreversible machine, as was demonstrated more explicitly by many later embeddings *e.g.*, [99, 16]. So under the time complexity measure by itself, reversibility does not hurt.

Can a reversible machine perform a task using *fewer* computational operations than an irreversible one? Obviously not, if we take reversible operations to just be a special case of irreversible operations. However, physically speaking, actually it is the converse that is true: so-called "irreversible" operations, implemented physically, are really just a special case of reversible operations, since physics is *always* reversible at a low level. We will see the implications of this for *physical* time complexity in ch. 6. But, using the usual computer-science definition of time as the number of *computational* operations required, clearly reversible machines can be no more "time"-efficient than irreversible ones.

Although Lecerf and Bennett explicitly discussed their time-efficient reversible simulations only in the context of Turing machines, the approach is easily generalized to any model of computation in which we can give each processing element access to an unbounded amount of auxiliary unit-access-time stack storage. For example, based on Toffoli's embedding [160], one could use essentially the same trick to create a time-efficient simulation of irreversible cellular automata on reversible ones, by using an extra dimension in the cell array to serve as a garbage stack for each cell of the original machine. (To actually recycle the garbage in a CA, we would also need a boundary condition that applies globally after an appropriate amount of time in order to reverse the simulation.)

3.3.4 Reversible entropic complexity

The original point of reversibility was not to reduce time but to reduce energy dissipation, or in other words entropy production. Can this be done by reversible machines? In 1961 Landauer [97] argued that it could not, since if we cannot get rid of the "garbage" bits that are accumulated in memory, they just constitute another form on entropy, no better in the long term than the kind produced if we just irreversibly dissipated those bits into physical entropy right away.

3.3.4.1 Lecerf reversal

However, in 1963, Lecerf [99] formally described a construction in which an irreversible machine was embedded into a reversible one that first simulated the irreversible machine running forwards, then turned around and simulated the irreversible machine in reverse, uncomputing all of the history information and returning to a state corresponding to the starting state. If anyone familiar with Landauer's work had noticed Lecerf's paper in the 1960's, it would have seemed tantalizing, because here was Lecerf showing how to reversibly get rid of the garbage information that was accumulated in Landauer's reversible machine in lieu of entropy. So maybe the entropy production can be avoided after all!

Unfortunately, Lecerf was apparently unaware of the thermodynamic implications of reversibility; he was concerned only with determining whether certain questions about reversible transformations were decidable. Lecerf's paper did not address the issue of how to get useful results out of a reversible computation. In Lecerf's embedding, by the time the reversible machine finishes its simulation of the irreversible machine, any outputs from the computation have been uncomputed, just like the garbage. This is not very useful!

3.3.4.2 The Bennett trick

Fortunately, in 1973, Charles Bennett [16], who was unaware of Lecerf's work but knew of Landauer's, independently rediscovered Lecerf reversal, and moreover added the ability to retain useful output. The basic idea was simple: one can just reversibly copy the desired output into available memory before performing the Lecerf reversal! As far as we can tell, this trick had not previously occurred to anyone.

Bennett's idea suddenly implied that reversible computers could in principle be *more* efficient than irreversible machines under at least one cost measure, namely entropy production. To compute an output on an irreversible machine, one must produce an amount of entropy roughly equal to the number of (irreversible) operations performed; whereas the reversible machine in principle can get by with *no* new entropy production, and with an accumulation of only the desired output in memory.

3.3.4.3 Entropy proportional to speed

Unfortunately, absolutely zero entropy generation per operation is achievable in principle only in the ideal limit of a perfectly-isolated ballistic (frictionless) system, or in a Brownian-motion-based system that makes zero progress forwards through the computation on average, and takes $\Theta(n^2)$ expected time before visiting the nth computational step. In useful systems that progress forwards at a positive constant speed, the entropy generation per operation appears to be, at minimum, proportional to the speed. (We do not yet know how necessary this relationship is, but it appears to be the case empirically.) A cost analysis that takes both speed and entropy into account will need to recognize this tradeoff. We do this is chapter 6.

3.3.5 Reversible space complexity

In addition to the number of computational operations performed and the entropy produced, another important element of cost is the number S of memory cells that are required to perform a computation.

3.3.5.1 Initial estimates of space complexity.

As Landauer pointed out [97], his simple strategy of saving all the garbage information appears to suffer from the drawback that the amount of garbage that must be stored in digital form is as large as the amount of entropy that would otherwise have been generated. If the computation performs on average a constant number of irreversible bit-erasures per computational operation, then this means that the memory usage becomes proportional to the number of operations. This means a large asymptotic increase in memory usage for many problems; up to exponentially large. Even if the garbage is uncomputed using Lecerf reversal, this much space will still be needed temporarily during the computation.

3.3.5.2 Bennett's pebbling algorithm

In 1989, Bennett [19] introduced a new, more space-efficient reversible simulation for Turing machines. This new algorithm involved doing and undoing various-sized portions of the computation in a recursive, hierarchical fashion. Figure 3.2 is a schematic illustration of this process. We call this the "pebbling" algorithm because the algorithm can be seen as a solution to a sort of "pebble game" or puzzle played on a one-dimensional chain of nodes, as described in detail by Li and Vitányi '96 [105]. (Compare figure 3.2(a) with fig. 3.7 on page 79.) We will discuss the pebble game interpretation and its implications in more detail in §3.4.2.

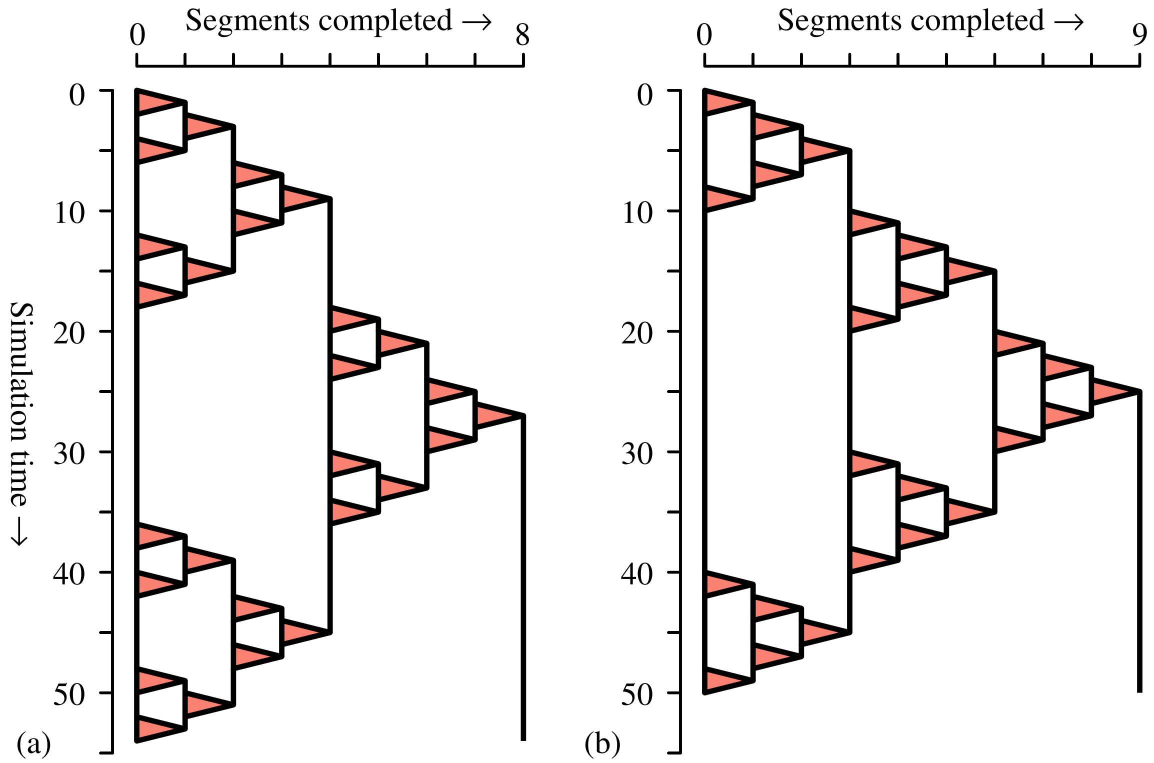


Figure 3.2: Illustration of two versions of Bennett's 1989 algorithm for reversible simulation of irreversible machines. Diagram (a) illustrates the version with $k = 2$, diagram (b) the version with $k = 3$. (See text for explanation of k.)

In both diagrams, the horizontal axis indicates which segment of the original irreversible computation is being simulated, whereas the vertical axis tracks time taken by the simulation in terms of the time required to simulate one segment. The black vertical lines represent times during which memory is occupied by an image of the irreversible machine state at the indicated stage of the irreversible computation, whereas the shaded areas within the triangles represent memory occupied by the storage of garbage data for a particular segment of the irreversible computation being simulated.

Note that in (b), where $k = 3$, the 9th stage is reached after only 25 time units, whereas in (a) 27 time units are required to only reach stage 8. But note also that in (b), at time 25, five checkpoints (after the initial state) are stored simultaneously, whereas in (a) at most four are stored at any given time. This illustrates the general point that higher-k versions of the Bennett algorithm run faster, but consume more memory.

The overall operation of the algorithm is as follows. The irreversible computation to be simulated is broken into fixed-size segments, whose run time is proportional to the memory required by the irreversible machine. The first segment is reversibly simulated using a Landauer embedding (§3.3.2.1). Then the state of the irreversible machine being simulated is checkpointed using the Bennett trick of reversibly copying it to free memory. Then, we do a Lecerf reversal (§3.3.4.1) to clean up the garbage from simulating the first segment.

We proceed the same way through the second segment, starting from the first checkpoint, to produce another checkpoint. After some number k of repetitions of this procedure, all the previous checkpoints are then removed by reversing everything done so far except the production of the final checkpoint. Now we have only a single checkpoint which is k segments along in the computation. We repeat the above procedure to create another checkpoint located another k segments farther along, and then again, and again k times, then reverse everything again at the higher level to proceed to a point where we only have checkpoint number k^2 in memory. The procedure can be applied indefinitely at higher and higher levels.

In general, for any number n of recursive higher-level applications of this procedure, k^n segments of irreversible computation are be simulated by $(2k-1)^n$ reversible simulations of a single segment, while having at most $n(k-1)$ intermediate checkpoints in memory at any given time [19].

The upshot is that if the original irreversible computation takes time T and space S, then the reversible simulation via this algorithm takes time $O(\mathsf{T}^{1+\epsilon})$ and space $O(\mathsf{S}\log\mathsf{T}) = O(\mathsf{S}^2)$. As k increases, the ϵ approaches 0 (very gradually), but unfortunately the constant factor in the space usage increases at the same time [103].

Li and Vitányi '96 [105] proved that Bennett's algorithm (with $k = 2$) is the most space-efficient possible pebble-game strategy for reversible simulation of irreversible machines.

Crescenzi and Papadimitriou '95 [42] later extended Bennett's technique to provide space-efficient reversible simulation of *nondeterministic* Turing machines as well.

3.3.5.3 Achieving linear space complexity

Bennett's results stood for almost a decade as the most space-efficient reversible simulation technique known, but in 1997, Lange, McKenzie, and Tapp [98] showed how to simulate Turing machines reversibly in linear space—but using worst-case exponential time. Their technique is very clever, but simple in concept: Given a configuration of an irreversible machine, they show that one can reversibly enumerate its possible predecessors. Given this, starting with the initial state of the irreversible machine, the reversible machine can traverse the edge of the irreversible machine's tree of possible configurations in a reversible "Euler tour." (See figure 3.3.) This is

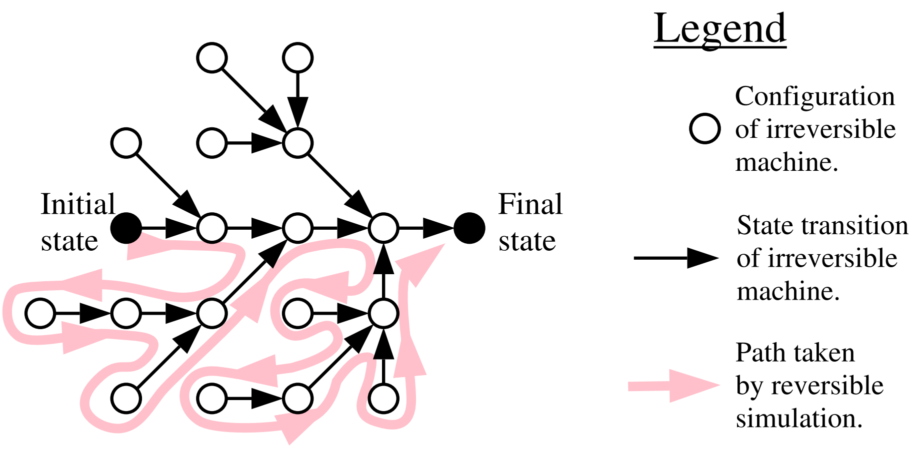


Figure 3.3: Illustration of an Euler tour of an irreversible machine's computation tree. Although the tree has branches, the Euler tour is itself both forward- and reverse-deterministic, and so can be traversed in purely reversible fashion, using no more space than is needed to keep track of the current irreversible machine configuration [98].

---

analogous to using the "right-hand rule" technique (move forward while keeping your right hand on the wall) to find the exit of a planar non-cyclical maze. The search for the final state is kept finite, and the space usage is kept small, by cutting off exploration whenever the configuration size exceeds some limit. Unfortunately, the size of the pruned tree, and thus the time required for the search, is still, in the worst case, exponential in the space bound.

Lange *et al.* originally thought that a limit on the size of the final state was required to be known in advance of the computation in order to guarantee finding the final state, but after seeing a draft of their paper, I pointed out to them (in personal discussions) that in fact, one could determine the appropriate limit dynamically by simply traversing repeatedly around and around the tree, advancing to a successively higher size limit each time the initial state is re-encountered, until the size limit is made large enough that the final state is found. This approach does not increase the worst-case asymptotic run-time, because that time is dominated anyway by the final traversal around the tree, due to the exponential nature of the worst-case branching.

As with Bennett's techniques, the Lange-McKenzie-Tapp technique was defined explicitly only in terms of Turing machines, but it is easily generalized to many different models of computation.

The above time and space complexity results for reversible simulation (§3.3.3 & §3.3.5) are very interesting in themselves, but to our knowledge, no one has yet directly addressed the question of whether a single reversible simulation can run in linear time like Bennett's 1973 technique *and* in linear space like the new Lange *et al.* technique. Li and Vitányi's analysis [105] of Bennett's 1989 algorithm [19] leads to our proof in §3.4 that if such an ideal simulation exists, it would not relativize to oracles, or work in cases where the space bound is much less than the input length.

3.3.6 Miscellaneous developments

Here, we mention in passing a few more miscellaneous developments in reversible computing theory, but we do not go into them in detail.

Coppersmith and Grossman (1975, [37]) proved a result in group theory which implies that reversible boolean circuits only 1 bit wider than a fixed-length input can compute arbitrary boolean functions of that input. (Thanks to Alain Tapp for bringing this paper to our attention.)

Toffoli (1977, [160]) showed that reversible cellular automata can simulate irreversible ones in linear time using an extra spatial dimension. Fredkin and Toffoli also developed much reversible circuit theory (1980–1982, [161, 162, 74]).

As we already mentioned in §3.3.2.2, Pin (1987 [136]) showed that reversible finite automata (defined in a certain way) cannot decide all regular languages.

3.4 Reversible vs. irreversible space-time complexity

In this section we prove that reversible machine models require higher asymptotic complexity on some problems than corresponding irreversible models, if a certain new reversible black-box operation is made available to both models. Thus, no *completely* general technique can exist for simulating irreversible machines on reversible ones with no asymptotic overhead.

However, the new primitive operation that we defined in order to make this proof go through is not itself physically realistic. The operation implements a computable function, but the operation is modeled as taking constant ($\Theta(1)$) time to perform independent of the size of its input, which violates physical locality (ref. §2.1) and the asymptotically very large number of steps that it would take to compute the operation using the algorithm that corresponds directly to the operation's definition.

Therefore, technically, even given our proof, it is still an open question whether a perfect simulation technique might still exist that works in the case of reversible machines simulating irreversible machines that are composed only of primitives that are physically realistic in the sense of obeying locality. However, if one wishes to progress to *complete* physical realism, then irreversible machines are themselves already reversible at the micro-level (§2.5), and therefore are efficiently implementable on reversible machines, as we will see in ch. 6.

Nevertheless, we conjecture that if the constraint of physical reversibility is ignored, then reversible machines are strictly less efficient on some problems than irreversible machines, even if the machines are constrained to be physically realistic in all other respects. If this conjecture is true, then in combination with our results of ch. 6, it would follow that the constraint of physical reversibility is not independent of other physical constraints from a computational complexity perspective, and that it *must* be taken into account in order to have a realistic physical model of computational complexity, as we will discuss in ch. 5.

If our conjecture were false, and irreversible models can be simulated with no overhead on reversible machines, then one would not necessarily have to explicitly incorporate reversibility in a model of computation in order for it to qualify as an accurate model for predicting problem complexity, such as we advocate in ch. 5. But as a matter of opinion, we consider that possibility *a priori* to be very unlikely.

In this section, we will prove our results in both oracle-relativized and non-oracle forms for serial (uniprocessor) machines. The oracle results cover a large family of possible asymptotic bounds on the joint space and time requirements of machines. For all bounding functions within this family, we show that there exist an oracle and a language such that the language is decidable within the given bounds by serial machines that can query the oracle only if the machines are *ir*reversible. This result is non-trivial (compared to Pin's, for example) because the individual oracle calls are themselves reversible and easy to undo.

A similar result, not involving an oracle, covers cases where the space bound is much smaller than the length of the randomly (and reversibly) accessible input. Corollaries to both the oracle and non-oracle results give loose lower bounds on the amount of extra space required for a reversible machine to decide the language within the time bounds.

Another contribution of our proof is to illustrate ways to use incompressibility arguments in analyzing reversible machines. It is conceivable that similar techniques might increase the range of reversible and irreversible space-time complexity classes that we can separate without resorting to the oracle.

Acknowledgment. Some ideas in the proof below originated with M. Josephine Ammer, who was an undergraduate research assistant in our group at the time this work was done. Ms. Ammer also assisted with the writing of the original manuscript [69] from which this section is derived. That manuscript has not yet been formally published, but some version of it may be in the future.

3.4.1 General definitions

Space-time complexity classes. Given any reversible model of computation (*e.g.*, reversible Turing machines), and given any computational space and time bounding functions $\mathsf{S}(n_{\text{in}}), \mathsf{T}(n_{\text{in}})$, we define the *reversible space-time* S, T *complexity class*, abbreviated $\mathbf{RST}(\mathsf{S}, \mathsf{T})$, to be the set of languages that are accepted by reversible machines that take worst-case space of $\mathcal{O}(\mathsf{S}(n_{\text{in}}))$ memory bits and worst-case time $\mathcal{O}(\mathsf{T}(n_{\text{in}}))$ ticks, where n_{in} is the length of the input. Similarly, we define the unrestricted *space-time* S, T *complexity class*, abbreviated $\mathbf{ST}(\mathsf{S}, \mathsf{T})$, to be the set of languages accepted in that same order of space and time on the corresponding normal machine model, without the restriction on the in-degree of the transition graph. For oracle-relativized complexity classes, we use the notation C^O, as is standard in complexity theory, to indicate the class of problems that can be solved by the machines that define the class C if they are allowed to query oracle O.

We want to know whether $\mathbf{RST}(\mathsf{S}, \mathsf{T}) \stackrel{?}{=} \mathbf{ST}(\mathsf{S}, \mathsf{T})$, for all S, T, in normal sorts of serial computational models such as multi-tape Turing machines or RAM machines.

Unfortunately, we have found this question, in its purest form, very difficult to definitively resolve. We do not see any general way to simulate normal machines on reversible machines without suffering asymptotic increases in either the time or space required. But neither do we know of a language that can be proven to require extra space or time to recognize reversibly in ordinary machine models. The difficulty is in constructing a proof that rules out all reversible algorithms, no matter how subtle or clever.

But is the $\mathbf{RST}(\mathsf{S}, \mathsf{T}) \stackrel{?}{=} \mathbf{ST}(\mathsf{S}, \mathsf{T})$ question truly difficult to resolve, or have we just been unlucky in our search for a proof? Often in computational complexity theory, we find ourselves unable to prove whether or not two complexity classes (for example, **P** and **NP**) are equivalent. Traditionally (as in [9]), one way to indicate that such an equivalence might really be difficult to prove is to show that if the machine model defining each class is augmented with the ability to perform a new type of operation (a query to a so-called "oracle"), then the classes may be proven either equal or unequal, depending on the behavior of the particular oracle. This shows that any proof equating or separating the two classes must make use of the fact that normal machine models are only capable of performing a particular limited set of primitive operations. Otherwise, we could just add the appropriate oracle call as a new primitive operation, and invalidate the supposed proof. In complexity theory, it is said that any proof of the equivalence or inequivalence of the two classes must not "relativize," that is, it does not remain valid relative to models that are augmented with oracles. Reputedly, this rules out a large number of proof techniques from recursion theory, and means that resolving the question will be more difficult.

In this section we will demonstrate, for any given S, T in a large class, an oracle

A relative to which we prove $\mathbf{RST}(\mathsf{S},\mathsf{T})^A \neq \mathbf{ST}(\mathsf{S},\mathsf{T})^A$, for the case of serial machine models with a certain kind of oracle interface. For these same S,T we have not yet found an alternative oracle B for which $\mathbf{RST}(\mathsf{S},\mathsf{T})^B = \mathbf{ST}(\mathsf{S},\mathsf{T})^B$. It may be that none exists, but this is uncertain.

Reversible oracle interface. First, we define an oracle interface that allows a reversible machine to call an oracle. Ordinarily, oracle queries are irreversible, and thus impossible in reversible machines. For example, a bit of the oracle's answer cannot just overwrite some storage location, because regardless of whether the location contained `0` or `1` before the oracle call, after the call it would contain the oracle's answer. The resulting configuration would thus have two predecessors, and the machine would be irreversible.

Our reversible oracle-calling protocol is as follows. Machines will have reversible read and write access to a special *oracle tape* which has a definite start, unbounded length, and is initially clear. At any time, the machine is allowed to perform an *oracle call*, a special primitive operation which in a single step replaces the entire contents of the oracle tape with new contents, according to some fixed invertible mapping $A : \mathcal{C} \to \mathcal{C}$ over the space $\mathcal{C}$ of possible tape contents. The function A is called a *permutation oracle*. Further, if A is its own inverse, $A = A^{-1}$, it will be called *self-reversible*. Presented more formally:

Definition 3.1. A *permutation oracle* A is an invertible (bijective) function $A : \mathcal{C} \to \mathcal{C}$, where $\mathcal{C}$ is the space of possible contents of a semi-infinite *oracle tape*.

Definition 3.2. A *self-reversible (permutation) oracle* is a permutation oracle A such that $A = A^{-1}$.

In the below, we will deal only with self-reversible oracles. Self-reversibility ensures that machines can easily undo oracle operations, just as they can easily undo their own internal reversible primitives. If oracle calls were hard to undo, then the oracle model would be unlikely to teach us anything meaningful about ordinary machines.

ST-constructibility. In order for our proof to go through, we will need to restrict our attention to space and time functions $\mathsf{S}(n_{\mathrm{in}}), \mathsf{T}(n_{\mathrm{in}})$ which are *ST-constructible*, meaning that given any input of length n_{in}, an irreversible machine can construct binary representations of the numbers $\mathsf{S}(n_{\mathrm{in}})$ and $\mathsf{T}(n_{\mathrm{in}})$ using only space $\mathcal{O}(\mathsf{S}(n_{\mathrm{in}}))$ and time $\mathcal{O}(\mathsf{T}(n_{\mathrm{in}}))$. We state here without proof that many reasonable pairs of functions are indeed ST-constructible. For example, $\mathsf{S} = n_{\mathrm{in}}^2$, $\mathsf{T} = n_{\mathrm{in}}^3$ can both be computed in time $\mathcal{O}(\log^2 n_{\mathrm{in}})$ plus $\mathcal{O}(n_{\mathrm{in}})$ to count the input bits, and space $\mathcal{O}(\log n_{\mathrm{in}})$ plus $\mathcal{O}(n_{\mathrm{in}})$ if we include the input.

Next, we need some basic definitions to support the notion of incompressibility that will be crucial to the proof of our theorem. The following definition and lemma follow

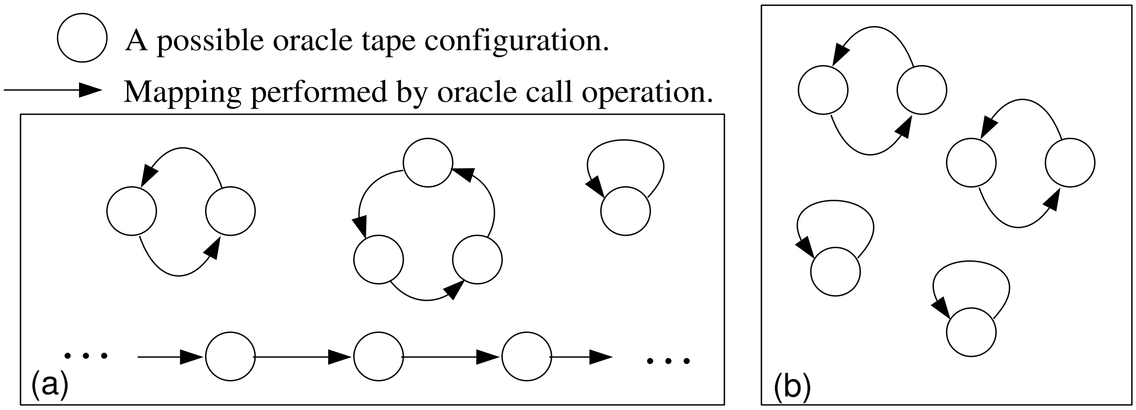


Figure 3.4: Illustration of the structure of (a) a permutation oracle, and (b) a self-reversible permutation oracle.

In either case, the oracle call operation replaces the old contents of the oracle tape with new contents according to a transition function $A : \mathcal{C} \to \mathcal{C}$ that is a permutation mapping—a bijective function—over the space $\mathcal{C}$ of possible tape contents. The bijectivity of this function means that a call to a permutation oracle is always a reversible operation. After an oracle call, the previous oracle tape contents can be uniquely determined by applying the inverse mapping A^{-1}. In self-reversible oracles, $A = A^{-1}$.

the spirit of the discussions of incompressibility in Li and Vitányi's excellent book on Kolmogorov complexity [104].

Description systems and compressibility. A *description system* s is any function $s\colon \{\mathtt{0},\mathtt{1}\}^* \to \{\mathtt{0},\mathtt{1}\}^*$ from bit-strings to bit-strings, that is, from *descriptions* to the bit-strings they describe. We say that a bit-string d *describes* bit-string x in description system s if $s(d) = x$. We say that a bit-string x is *compressible* in description system s if there is a shorter bit-string that describes it; *i.e.* if there exists a string d such that $s(d) = x$ and $|d| < |x|$, where the notation $|b|$ denotes the number of bits in bit-string b.

Lemma 1. *Existence of incompressible strings.* For any description system s, and any string length ℓ, there is at least one bit-string x of length ℓ that is not compressible in s.

Proof. (Trivial counting argument.) There are 2^ℓ bit-strings of length ℓ, but there are only $\sum_{i=0}^{\ell-1} 2^i = 2^\ell - 1$ descriptions that are shorter than ℓ bits long. Each description d can describe at most one bit string of length ℓ, namely the string $s(d)$ if that string's length happens to be ℓ. Therefore there must be at least one remaining bit-string of length ℓ that is not described by any shorter description. □

In our main proof, we will be selecting incompressible strings from a series of computable description systems.

Notational conventions. In the following, we will often abbreviate the space and time function values $\mathsf{S}(n_{\text{in}})$ and $\mathsf{T}(n_{\text{in}})$ by just S and T, respectively; likewise for other functions of n_{in}. For comparing orders of growth, we will use both the standard $\boldsymbol{\Theta}$, $\mathcal{O}$, $\boldsymbol{\Omega}$, $\mathbf{o}$, $\boldsymbol{\omega}$ notations, and our mnemonic custom $\sim$, $\precsim$, $\succsim$, $\prec$, $\succ$ notation, defined in table F.4 on p. 409.

3.4.2 Oracle results

Theorem 3.1. Relative separation of reversible and irreversible space-time complexity classes. Let S, T be any two non-decreasing functions over the non-negative integers. Then the following are true:

(a) If $\mathsf{S} \succsim \mathsf{T}$ or $\mathsf{T} \succsim 2^{\mathsf{S}}$, then $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{S},\mathsf{T})^O$ for any self-reversible oracle O.

(b) If $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$, and if S, T are ST-constructible, then there exists a computable, self-reversible oracle A such that $\mathbf{RST}(\mathsf{S},\mathsf{T})^A \neq \mathbf{ST}(\mathsf{S},\mathsf{T})^A$.

Proof.

Part (a). (Cases $\mathsf{S} \succsim \mathsf{T}$ and $\mathsf{T} \succsim 2^{\mathsf{S}}$.) First, if $\mathsf{S} \succ \mathsf{T}$, then obviously we have both $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{RST}(\mathsf{T},\mathsf{T})^O$ and $\mathbf{ST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{T},\mathsf{T})^O$ simply because in time T no more than $\mathsf{S} \sim \mathsf{T}$ memory cells can be accessed on a machine that performs $\boldsymbol{\Theta}(1)$

operations per time step. Similarly, if $\mathsf{T} \succ 2^{\mathsf{S}}$, then $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{RST}(\mathsf{S},2^{\mathsf{S}})^O$ and $\mathbf{ST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{S},2^{\mathsf{S}})^O$, because no computation using only S bits of memory can run for more than 2^{S} steps without repeating. So part (a) reduces to proving $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{S},\mathsf{T})^O$ only for the case where $\mathsf{S} \sim \mathsf{T}$ or $\mathsf{T} \sim 2^{\mathsf{S}}$.

From here, the result follows due to the existing relativizable simulations. When $\mathsf{S} \sim \mathsf{T}$, Bennett's simple reversible simulation technique [16] can be applied because it takes time $\mathcal{O}(\mathsf{T})$ and space $\mathcal{O}(\mathsf{T})$. Similarly, when $\mathsf{T} \sim 2^{\mathsf{S}}$ the simulation of Lange *et al.* [98] can be used because it takes time $\mathcal{O}(2^{\mathsf{S}})$ and space $\mathcal{O}(\mathsf{S})$. Both techniques can be easily seen to relativize to any self-reversible oracle O. Thus, in both cases, any irreversible machine can be simulated reversibly in $\mathcal{O}(\mathsf{T})$ and space $\mathcal{O}(\mathsf{S})$, and therefore $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{S},\mathsf{T})^O$.

Part (b). (Case $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$.) **Outline:** We will construct A to be a permutation oracle that can be interpreted as specifying an infinite directed graph of nodes with outdegree at most 1. We will also define a corresponding language-recognition problem, which will be to report the contents of a node that lies T/S nodes down an incompressible linear chain of nodes that have size-S identifiers, starting from a node that is determined by the input length. The oracle will be explicitly constructed via a diagonalization, so that for each possible reversible machine, there will be a particular input for which our oracle makes that particular reversible machine take too much space or else get the wrong answer. In the cases where the reversible machine takes too much space, we will prove this by equating the machine's operation with the "pebble game" for which Li and Vitányi [105] have already proven lower bounds, and by showing that if the machine does not take too much space, then we can build a shorter description of the chain of nodes using the machine's small intermediate configurations, thus contradicting our choice of an incompressible chain.

For the formal proof of part (b), we need some special definitions.

Definition 3.3. A *graph oracle* is a self-reversible permutation oracle with the following property: There exists a partial function $f\colon \{\mathtt{0},\mathtt{1}\}^* \to \{\mathtt{0},\mathtt{1}\}^*$, called a *successor function*, such that for any bit string (node) $b \in \{\mathtt{0},\mathtt{1}\}^*$ for which f is defined, the oracle's permutation function maps the tape contents b to the tape contents $b\mathtt{\#}f(b)$, and also maps $b\mathtt{\#}f(b)$ back to b, where $\mathtt{\#}$ is a special separator character in the oracle tape alphabet. For all tape contents x not of either of these forms, the oracle's permutation function maps them to themselves. See fig. 3.5.

Given that we will be working only with graph oracles, we can now specify an oracle by specifying just the successor function f that it embodies. But before we actually construct the special oracle A that proves our theorem, let us define, relative to A, the language that we claim separates $\mathbf{RST}(\mathsf{S},\mathsf{T})^A$ from $\mathbf{ST}(\mathsf{S},\mathsf{T})^A$.

Definition 3.4. Given two ST-constructible functions $\mathsf{S}(n)$, $\mathsf{T}(n)$, and graph oracle

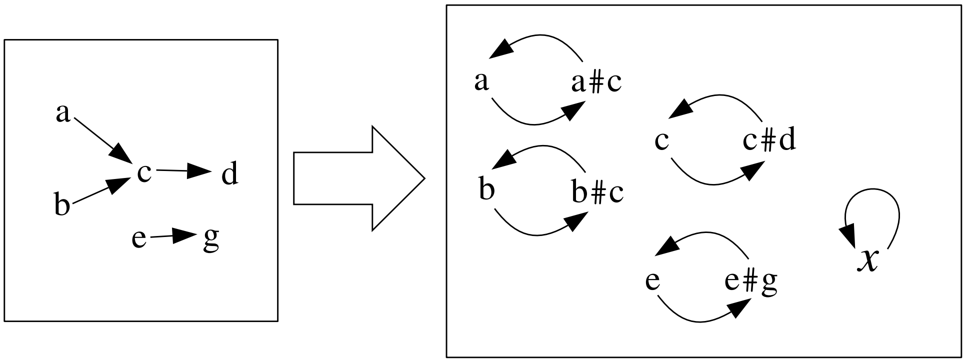


Figure 3.5: Encoding outdegree-1 directed graphs in self-reversible permutation oracles. Letters stand for nodes represented as bit-strings, except for x which represents any other bit-string not explicitly shown. The # is a special separator character.

On the left, we show an example of an outdegree-1 directed graph with bit-string nodes abbreviated a,b,c,d,e,g. The graph function f gives the successor of each node: $f(\mathrm{a}) = \mathrm{c}$, $f(\mathrm{c}) = \mathrm{d}$, etc. This f is a partial function; *e.g.* $f(\mathrm{d})$ is undefined. For each edge in this graph, there is a corresponding pair of strings that are mapped to each other by the self-reversible oracle. To represent the edge a → c, for example, the permutation oracle maps tape contents "a" to "a#c" and maps "a#c" back to "a". Any other string x (including those for terminal nodes of the graph) is simply mapped to itself. In this way the permutation oracle allows easily and reversibly looking up a node's successor, or uncomputing a node's successor given the node and its successor. But finding a node's predecessor(s), given just the node itself, is designed to be hard. Thus the oracle call resembles the reversible computation of a "one-way" invertible function that is easy to compute, but whose inverse is difficult to compute.

A with successor function f, we define the *difficult language* $L(A)$ to be the language decided by the irreversible machine described by the following pseudocode:

```
Given input string $w$,
    Let $n = |w|$, compute $\mathsf{S} = \mathsf{S}(n), \mathsf{T} = \mathsf{T}(n)$.
    Let bit-string $b = \mathtt{0}^{\mathsf{S}}$.
    Repeat the following, $t = \lfloor \mathsf{T}/\mathsf{S} \rfloor$ times:
        Write $b$ on oracle tape, and call oracle.
        If result is of the form $b\#c$, with $c$ a bit-string,
            assign $b \leftarrow c$ (note $c = f(b)$),
        else, quit loop early.
    Accept iff $b[0] = \mathtt{1}$.
```

In other words, given a string of length n, construct a string of zeros of length $\mathsf{S}(n)$. Treat this string as a node identifier, and use oracle queries to proceed down its chain of successors for up to $\lfloor \mathsf{T}/\mathsf{S} \rfloor$ nodes. Finally, return the first bit of the final node's bit-string identifier.

We will be explicitly constructing the successor function f so that it always returns a string of the same length as its input. Given the corresponding oracle, the above algorithm requires only space $\mathcal{O}(\mathsf{S})$ and time $\mathcal{O}(\mathsf{T})$ on on irreversible machine in any standard serial model of computation. (Recall that S, T are ST-constructible.) Therefore the language $L(A)$ will be in the class $\mathbf{ST}(\mathsf{S}, \mathsf{T})^A$.

Now, we will specify how to construct f so that the language $L(A)$ will not be computable by any reversible machine that takes space $\mathcal{O}(\mathsf{S})$ and time $\mathcal{O}(\mathsf{T})$. The way we will do this is to make each of the node identifiers be a different incompressible string. Intuition suggests that the only way to decide $L(A)$ is to actually follow the entire chain of nodes, to see what the final one is. But having obtained a node's successor, the reversible machine cannot easily get rid of its incompressible records of the prior nodes. The graph oracle provides no convenient way to compute f^{-1} and find a node's predecessor, even if the successor function f happens to be invertible. Thus the reversible machine will tend to accumulate records of previous nodes, of size $\mathsf{S}(n_{\text{in}})$ each, and thus, for sufficiently long enough chains, it will take more than a constant factor times $\mathsf{S}(n_{\text{in}})$ space. The reversible machine could conceivably find and uncompute predecessor nodes by searching them all exhaustively, but this would take too much time.

The situation with this oracle language resembles the non-oracle problem of iterating a one-way function, *i.e.* an invertible function whose inverse much is harder to compute than the function itself (*e.g.*, MD5). Public-key cryptography depends on the (unproven, but empirically reasonable) assumption that some functions are one-way. The same assumption might allow us to show that $\mathbf{RST}(\mathsf{S}, \mathsf{T}) \neq \mathbf{ST}(\mathsf{S}, \mathsf{T})$ without an oracle, by using a one-way function instead.

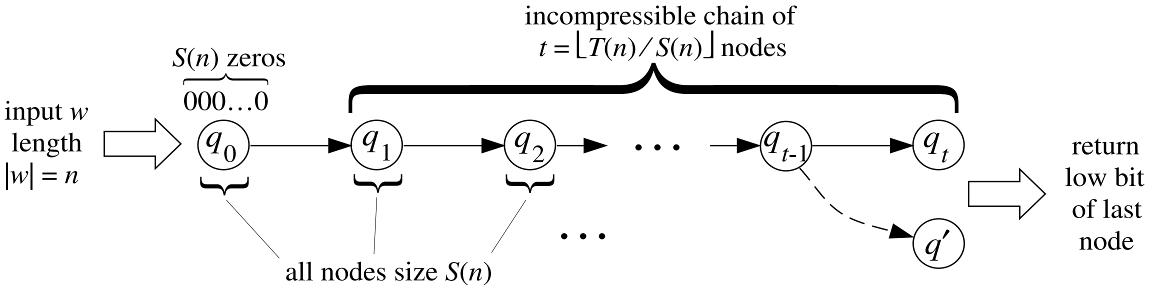


Figure 3.6: The problem graph defined by our oracle for inputs of size n. The "correct answer" is just the first bit of the final node q_t. If the reversible machine M_i that we are trying to foil happens to get the right answer, but never asks for the successor of node q_{t-1}, we redefine q_{t-1}'s successor to be a new node q' having a different initial bit.

Oracle construction. We now construct a particular oracle A and prove that $L(A) \notin \mathbf{RST}(\mathsf{S},\mathsf{T})^A$.

First, fix some standard enumeration of all reversible oracle-querying machines. The enumeration is possible because reversible Turing machines, for example, can be characterized by local syntactic restrictions on their transition function, as in Lange *et al.*, so we can enumerate all machines and pick out the reversible ones. Let $(M_1, c_1), (M_2, c_2), \ldots$ be this enumeration dovetailed together with an enumeration of the positive integers. If a given machine always runs in space $\mathcal{O}(\mathsf{S})$ and time $\mathcal{O}(\mathsf{T})$ then it will eventually appear in the enumeration paired with a large enough c_i so that the machine M_i takes space less than $c_i + c_i S(n_{\text{in}})$ and time less than $c_i + c_i \mathsf{T}(n_{\text{in}})$ for any input length n_{in}.

We will construct the oracle A so that each machine M_i will fail to decide $L(A)$ within these bounds. When considering M_i, $f(q)$ will have already been specified for all oracle queries q asked by machines $M_1, M_2 \ldots, M_{i-1}$ when given certain inputs of lengths $n_1, n_2, \ldots, n_{i-1}$, respectively. Now, choose n_i (henceforth called n), the input length for which our oracle definition will foil M_i, to be such that $\mathsf{S}(n)$ is greater than the maximum length z of any of those earlier machines' oracle queries. Some other lower bounds on the size of n will be mentioned as we go along.

Later we will specify a description system s_i based on M_i, c_i, the value of n, and all the $f(q)$ values defined so far (for bit-strings smaller than $\mathsf{S}(n)$). The description system will be a total computable function, *i.e.*, there is an algorithm that computes $s_i(d)$ for any d and always halts. We will use this description system to define $f(q)$ for bit-strings of length $\mathsf{S}(n)$, as follows:

Let x be a bit-string of length $\mathsf{T}(n)$ that is incompressible in description system s_i (to be defined as we go along). This x will be used as the sequence of size-$\mathsf{S}(n)$

node identifiers that will define our graph for inputs of size n.

Break x up into a sequence of $t(n) \equiv \lfloor \mathsf{T}/\mathsf{S} \rfloor$ bit-strings of length $\mathsf{S}(n)$ each; call these our graph nodes or *query strings* $q_1, \ldots, q_t$. We will design our description system s_i so that all the q_j's must be different. How? By allowing descriptions of the form (j, k, x'), where j and k are the indices of two equal nodes $q_j = q_k$, $j < k$, and x' is x with the q_k substring spliced out. The description system would be defined to generate x from such a description by simply looking up the string q_j in x' and inserting a copy of it in the kth position. The indices j and k would take $\mathcal{O}(\log(\mathsf{T}/\mathsf{S}))$ space, which is $\mathcal{O}(\log \mathsf{T})$ space, which is $o(\mathsf{S})$ space, whereas we are saving $\mathsf{S}(n)$ space by not explicitly including the repetition of q_j. Therefore as long as n is sufficiently large, the total length of this description of x would be less than $\mathsf{T}(n)$. With x being incompressible in a description system that permits such descriptions, we know that $q_1, \ldots, q_t$ includes no repetitions.

Now we can specify exactly how the oracle defines our problem graph for inputs of size n, as follows. Define query string $q_0 = \mathtt{0}^{\mathsf{S}}$ (a string of S $\mathtt{0}$-bits). Provisionally, set $f(q_{j-1}) = q_j$ for all $1 \leq j \leq t$. These assignments are possible since all the q_j's are different, as we just proved. (They also must be different from q_0, but this is easy to ensure as well.) Given these assignments, all strings of length n are in the language $L(A)$ if and only if $q_t[0] = \mathtt{1}$, due to the earlier definition of $L(A)$. (Definition 3.2.)

Suppose temporarily that our oracle definition was completed by letting f remain undefined over all strings w for which we have not yet specified $f(w)$. (*I.e.*, let $A(w) = w$ for these strings.) Under that assumption, simulate M_i's behavior on the input $\mathtt{0}^n$. If M_i runs for more than $c_i + c_i T$ steps, then it takes too much time, and we are through addressing it. Otherwise, M_i either accepts ($\mathtt{1}$) or rejects ($\mathtt{0}$). If this answer is different from $q_t[0]$, then M_i already fails to accept the language $L(A)$, and we are through with it.

Alternatively, suppose M_i's answer is correct with the given q_j's and it halts within $c_i + c_i T$ steps. But now suppose that M_i never asked any query dependent on $f(q_{t-1})$ during its run on input $\mathtt{0}^n$. That is, suppose M_i never asked either query q_{t-1} or query $q_{t-1}\#q_t$. In that case, let us change our definition of $f(q_{t-1})$ as follows, to change the correct answer to be the opposite of what M_i gave. Let q' be a bit-string whose successor was never requested in any query by M_i, and whose first bit is the opposite of M_i's answer. To ensure such strings exist, note there are $\frac{1}{2}2^{\mathsf{S}}$ bit-strings of length S having the desired initial bit, but M_i can make at most $c_i + c_i T$ queries since that is its running time. We know $\mathsf{T} \prec 2^{\mathsf{S}}$, so with sufficiently large n, $\frac{1}{2}2^{\mathsf{S}} > c_i + c_i \mathsf{T}$, and we can find our node q'. Now, given q', we change $f(q_{t-1})$ to be q'. This cannot possibly affect the behavior of M_i since it never asked about $f(q_{t-1})$. But the correct answer is changed to the first bit of q', the new node number t in the chain. Thus with this new partial specification of f, M_i fails to correctly decide $L(A)$, and we can go on to foil other machines.

Finally, suppose M_i does ask query q_{t-1}. We now show how to complete the definition of our description system s_i, source of our incompressible x, so that if M_i does ask query q_{t-1}, then it must at some point take more than $c_i + c_i S$ space.

To do this, we show that M_i can always be interpreted as following the rules of Bennett's reversible "pebble game," introduced in [19] and analyzed by Li and Vitányi in [105].

Pebble game rules. The game is played on a linear list of nodes, which we will identify with query strings $q_1, \ldots, q_t$. At any time during the game some set of nodes is *pebbled.* Initially, no nodes are pebbled. At any time, the *player* (in our case, M_i) may, as a move in the game, change the pebbled vs. unpebbled status of node q_1 or any node q_j for which the previous node q_{j-1} is pebbled. Only one such move may be made at a time.

The idea of the pebbled set is that we will make it correspond to the set of nodes that is currently "stored in memory" by M_i. Pebbling or unpebbling node q_j will require querying the oracle with query string q_{j-1} or $q_{j-1}\#q_j$, respectively. The goal of the pebble game is to eventually place a pebble on the final node q_t. This corresponds to the fact already established that M_i must at some point ask query q_{t-1} or the oracle can be constructed to foil it trivially.

Li and Vitányi's analysis of the pebble game [105] showed that no strategy can win the game for 2^k nodes or more without at some time having more than k nodes pebbled at once. We will show that our machine M_i and its space usage can be modeled using the pebble game, so that for some sufficiently large n, the space required to store the necessary number of pebbled nodes will exceed M_i's allowable storage capacity $c_i + c_i \mathsf{S}$.

For the oracle A as defined so far, consider the complete sequence of configurations of M_i given input 0^n, notated $C_1, C_2, \ldots, C_{\mathsf{T}'}$, where $\mathsf{T}' \leq c_i + c_i \mathsf{T}$ is M_i's total running time, in terms of the number of primitive operations (including oracle calls) performed.

Now, for any time point τ, $1 \leq \tau \leq \mathsf{T}'$, and for any node q_j in the chain of nodes $q_1, \ldots, q_t$, define *the previous query involving* q_j (written $\mathrm{prev}(q_j)$)to mean the most recent oracle query in M_i's history before time τ in which the query string (the one that is present on the oracle tape at the start of the query) is one of $\{q_{j-1}, q_j, q_j\#q_{j+1}, q_{j-1}\#q_j\}$. There may of course be no such query in which case $\mathrm{prev}(q_j)$ does not exist. Similarly define *the next query involving* q_j (written $\mathrm{next}(q_j)$) to mean the most imminent such query in M_i's future after time τ.

Definition 3. Node q_j *is pebbled at time* τ iff at time τ either (a) $\mathrm{prev}(q_j)$ exists and is either (a1) q_{j-1}, (a2) q_j, or (a3) $q_j\#q_{j+1}$, or (b) $\mathrm{next}(q_j)$ exists and is (b1) q_j, (b2) $q_j\#q_{j+1}$, or (b3) $q_{j-1}\#q_j$. (Exception: the final node q_t is only considered pebbled in cases (a1) and (b3).)

Note that this definition implies that q_j is *not* pebbled iff $\mathrm{prev}(q_j) = q_{j-1}\#q_j$ (or

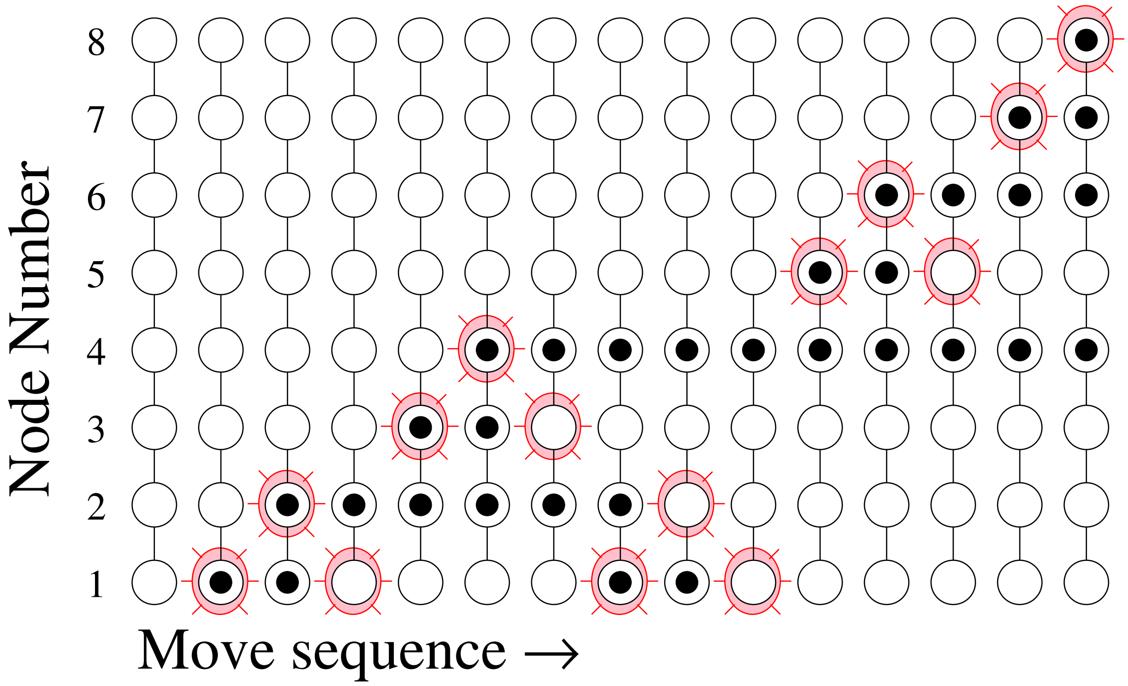


Figure 3.7: Bennett's reversible pebble game strategy. Highlights point out the move made at each step. (Compare with fig. 3.2(a), page 64, rotated 90°.)

A node q_j can be pebbled or unpebbled only if it is node q_1 or if the previous node q_{j-1} is pebbled. The strategy invented by Bennett [19], illustrated here, was shown by Li and Vitányi to be optimal [106] in terms of the number of pebbles required. But even with this optimal strategy, to pebble node 2^k we must at some time have more than k nodes pebbled. In this example, we reach node $2^3 = 8$ but must use 4 pebbles to do so. (After pebbling node 8, we can remove all pebbles by undoing the sequence of moves.) The fact that a constant-size supply of pebbles can only reach outwards along the chain a constant distance is crucial to our proof.

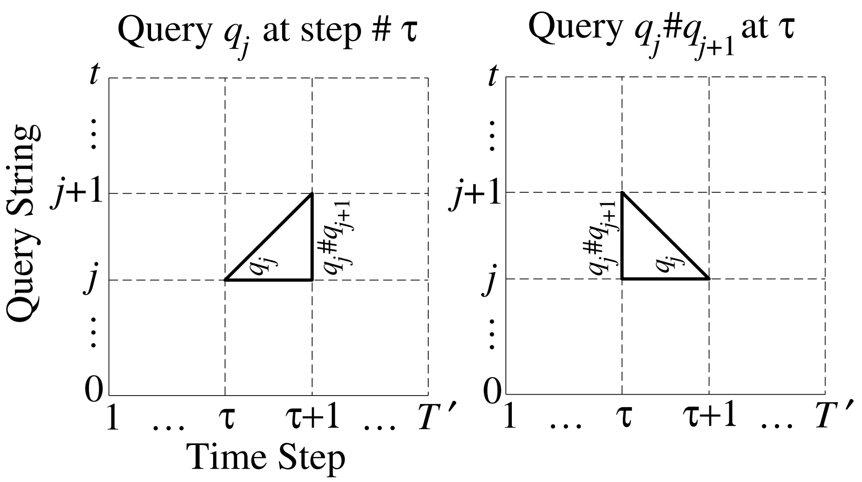


Figure 3.8: Triangle representation of oracle queries.

The shape and direction of the triangle is meant to evoke the fact that at the times just before and after an oracle query, the oracle tape contains the shorter string q_j at one of the times, and the longer string $q_j\#q_{j+1}$ at the other time. The set of triangles defines the set of pebbled nodes at any time, as illustrated in figure 3.9.

---

nonexistent) and $\text{next}(q_j) = q_{j-1}$.

Figure 3.9 illustrates the intuition behind this definition using the graphical notation introduced in fig. 3.8. This graphical notation is especially nice because it evokes the image of playing the pebble game or running Bennett's algorithm (compare fig. 3.9 with figs. 3.7 and 3.2).

The times at which a node is to be considered "pebbled" during a machine's execution are indicated by the solid horizontal lines on 3.9. These times are determined, according to definition 3 above, solely by the arrangement of triangles (representing oracle queries, see fig. 3.8) on the chart. Each vertex of a triangle generates a line of pebbled times for the corresponding node, extending horizontally away from the triangle until it hits another triangle. Query string 0 is never considered pebbled because it is not considered to be a node.

Let p denote the number of distinct nodes out of $q_1, \ldots, q_t$ that are pebbled at time τ. We now lower bound the size of C_τ, *i.e.* M_i's space usage at time τ.

Lemma 2. *Space to pebble p nodes.* $|C_\tau| > \frac{1}{4}pS$.

Proof. Suppose C_τ were no larger than $\frac{1}{4}pS$ bits. Then we can show that x (the sequence of all q_j's) is compressible to a shorter description d which we will now specify. Our description system s_i will be defined to process descriptions of the required form.

First, note that for each node q_j that is pebbled at time τ, that node is pebbled

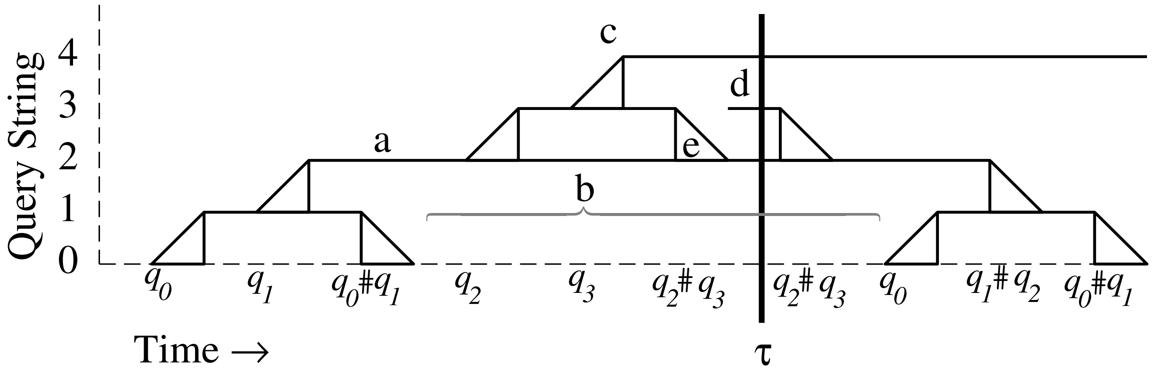


Figure 3.9: Visualizing the definition of the set of pebbled nodes. The times at which a node is pebbled (indicated by solid horizontal lines on the chart) are determined, by definition, solely by the identities and timing of oracle queries and the corresponding arrangement of triangles (see fig. 3.8) on the chart. Each vertex of a triangle generates a line of pebbled times for the corresponding node, extending horizontally away from the triangle until it encounters another triangle. (Except query string 0 is never pebbled, because it is not considered to be a node.)

The above example shows a pattern of queries similar to the one that would occur if one tried to apply Bennett's [19] optimal pebble game strategy. (Compare with figs. 3.7 and 3.2.)

Node 2 is considered pebbled at time (a) both because of the previous and next queries (triangles) involving node 2. Node 1 is not pebbled at times (b) because the previous and next queries are $q_0\#q_1$ and q_0 respectively. Node 4 is pebbled at all times after (c) because even though there is no next query involving node 4, the previous query involving node 4 exists and is of the right form (q_3). Node 3 is pebbled at time (d) because although the previous query (e) is of the wrong form ($q_2\#q_3$), the next query is okay.

Query (e) does not change the set of pebbled nodes and so is not considered to be a move in the pebble game. All the other queries are considered to be pebbling or unpebbling moves in the pebble game, depending on the direction of the corresponding triangle.

In the machine configuration C_τ at time τ, nodes 2, 3, and 4 are pebbled. But note that the query string for node 2 can be found by simulating the machine backwards from time τ until query (e), and reading q_2 off of the oracle tape. And if q_3 is given, we can continue simulating backwards until we get to time (c), and read q_4 off the oracle tape as well. The ability to perform this sort of simulation, for any arrangement of triangles, either forwards or backwards in time as needed to find out more than a constant number of the pebbled nodes is what makes our incompressibility argument work.

either because of the previous query involving q_j, because of the next query involving q_j, or both. Therefore, either at least $\frac{1}{2}p$ nodes are pebbled because of their previous query, or at least $\frac{1}{2}p$ nodes are pebbled because of their next query. Let D be a direction (forwards or backwards) from time τ in which one can find queries causing $h \geq \frac{1}{2}p$ nodes to be pebbled.

We now specify the shorter description d that describes x. It will contain an explicit description of C_τ, which by our assumption is no longer than $\frac{1}{4}p\mathsf{S}$. It will also specify the direction D and contain a concatenation of all the q_j's that are *not* pebbled because of queries in direction D. (Space: $(t-h)\mathsf{S}$.) For each of the h nodes q_j that *are* pebbled because of a query in direction D, the description d will contain the node index j and an integer $\Delta\tau_j$ giving the number of steps from step τ to the time of the query. Also we include a short tag k_j indicating which of the 3 possible cases of queries causes the node to be pebbled. Each of the indices j takes space $\mathcal{O}(\log t) \prec \log \mathsf{T} \prec \mathsf{S}$, and each $\Delta\tau_j$ takes space $\mathcal{O}(\log \mathsf{T}) \prec \mathsf{S}$. The tag is constant size. Thus for sufficiently large n, all h of the $(j, \Delta\tau_j, k_j)$ tuples together take less than $\frac{1}{2}h\mathsf{S}$ space. Total space so far: less than tS. If $t\mathsf{S} < \mathsf{T}$, then x will contain some additional bits beyond the concatenation of $q_1 q_2 \ldots q_t$, in which case d includes those extra bits as well. The total length of d will still be less than $\mathsf{T} = |x|$.

We now demonstrate that the description d is sufficient to reconstruct x, and give an algorithm for doing so. The function computed by this algorithm tells how our description system s will handle descriptions of the form outlined above.

The algorithm will work by simulating M_i's operation in direction D starting from configuration C_τ, and reading the identifiers of pebbled nodes from M_i's simulated oracle tape as it proceeds. We can figure out which oracle queries correspond to which nodes by referring to the stored times $\Delta\tau_j$ and tags k_j. Once we have extracted the identifiers of all nodes pebbled in direction D, we print all the nodes out in the proper order.

As an example, refer again to fig. 3.9. In the machine configuration marked at time τ, nodes 2, 3, and 4 are pebbled. But note that the query string for node 2 can be found by simulating the machine backwards from time τ until query (e), and reading q_2 off of the oracle tape. And if q_3 is known, we can continue simulating backwards until we get to time (c), and read q_4 off the oracle tape as well. The ability to perform this sort of simulation, for any arrangement of triangles, either forwards or backwards in time as needed to find out at least half of the pebbled nodes is what makes our incompressibility argument work. The algorithm is described and verified in more detail in the appendix.

Given d, the algorithm produces x, and with n chosen large enough, the length of the description will be smaller than x itself, contradicting the assumption of x's incompressibility relative to s. Therefore for these sufficiently large n, all configurations in which p nodes are pebbled must actually be larger than $\frac{1}{4}p\mathsf{S}$. This completes the proof of lemma 2. ■

Now, given the definition of the set of pebbled nodes from earlier (defn. 3), it is easy to see how M_i's execution history can be interpreted as the playing of a pebble game. Whenever M_i performs a query q_j and node q_{j+1} was not already pebbled immediately prior to this query, we say that M_i *is pebbling node* q_{j+1} as a move in the pebble game. Similarly, whenever M_i performs a query $q_j\#q_{j+1}$ and node q_{j+1} is not pebbled immediately after this query, we say that M_i *is unpebbling node* q_{j+1}. All other oracle queries and computations by M_i are considered as pauses between pebble game moves of these two forms. For example, in fig. 3.9, query (e) (the first occurrence of $q_2\#q_3$ is not considered a move in the pebble game, since it doesn't change the set of pebbled nodes as defined by definition 3.

It is obvious that under the above interpretation, all moves must obey the main pebble game rule, *i.e.* that the pebbled status of node q_j can only change if $j = 1$ or if node q_{j-1} is pebbled during the change. The move is a query, and the presence of the query means the node q_{j-1} is pebbled both before and after the query, by definition 3, unless $j = 1$; q_0 is not considered to be a node.

To show that no nodes are *initially* pebbled is a only a little bit harder. Suppose that some node q_j was pebbled in M_i's initial configuration. Then a shorter description of x (for sufficiently large n) can be given as $(j, \Delta\tau_j, x')$, where x' is x with q_j spliced out. This description could be processed via simulation of M_i to produce x in the same way as in lemma 2, except that this time, the starting configuration C_1 can be produced directly from the known values of M_i and n, and need not be explicitly included in the description. Of course the description system s needs to be able to process descriptions of this form. Then the incompressibility of x in s shows that the assumption that q_j is initially pebbled is inconsistent.

Thus M_i exactly obeys all the rules of the Bennett pebble game. Now, Li and Vitányi have shown [105] that any strategy for the pebble game that eventually pebbles a node at or beyond node 2^k must at some time have at least $k + 1$ nodes pebbled at once. So let us simply choose n large enough so that $t(n) \geq 2^k$ for some $k \geq 4(c_i + 1)$, and also so that $S \geq c_i$. Then at times τ when p is maximum, M_i's space usage is $|C_\tau| > \frac{1}{4}p\mathsf{S} > \frac{1}{4}kS \geq (c_i + 1)\mathsf{S} \geq c_i + c_iS$.

The above discussion establishes that machine M_i takes more than space $c_i + c_i\mathsf{S}$ if it correctly decides membership in $L(A)$ for inputs of length $n_i = n$ and takes only time $c_i + c_i\mathsf{T}$, so long as the oracle A is consistent with the definition above. Since machine M_i's behavior on the input $\mathsf{0}^n$ only depends on the values of the successor function $f(b)$ for bit-strings b up to a certain size (call it z), we are free to extend the oracle definition to similarly foil machine M_{i+1} by picking n_{i+1} so that $S(n_{i+1}) > z$. If one continues the oracle definition process in this fashion for further M_i's *ad infinitum*, then for the resulting oracle, it will be the case that for any M_i and constant c_i in the entire infinite enumeration, the machine will either get the wrong answer or take more than time $c_i + c_i\mathsf{T}$ or space $c_i + c_i\mathsf{S}$ on input 0^{n_i}. Thus, no reversible machine can actually decide $L(A)$ in time $\mathcal{O}(\mathsf{T})$ and space $\mathcal{O}(\mathsf{S})$, and so $L(A) \notin \mathbf{RST}(\mathsf{S},\mathsf{T})^A$.

Note that this entire oracle construction, as described, is computable. If we are given procedures for computing $\mathsf{S}(n)$ and $\mathsf{T}(n)$, we can write an effective procedure that, given any finite oracle query, returns A's response to the query. The details of the oracle construction algorithm follow directly from the above definition of A, but would be too tedious to present here.

This concludes our proof of theorem 3.1.

□

Note that in the above proof, we used the fact that the number of pebbles required to get to the final node grows larger than any constant as n increases. But the actual rate of growth can be used as well, to give us an interesting lower bound.

Corollary 1. *Lower bound on space for linear-time relativizable reversible simulation of irreversible machines.* For all ST-constructible S,T and computable S' such that $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$ and $\mathsf{S}' \prec \mathsf{S}\log(\mathsf{T}/\mathsf{S})$, there exists a computable, self-reversible oracle A such that $\mathbf{RST}(\mathsf{S}',\mathsf{T})^A \neq \mathbf{ST}(\mathsf{S},\mathsf{T})^A$.

Proof. Essentially the same as for Theorem 1 part (b), but with S' in place of S in appropriate places. In the last part of the proof, M_i is shown to take more than $c_i + c_i\mathsf{S}'$ space by using Lemma 2 together with the fact that $p > \lfloor \lg \lfloor \mathsf{T}/\mathsf{S} \rfloor \rfloor$ pebbles are required to reach the final node. □

This result implies that any general linear-time simulation of irreversible machines by reversible ones that is relativizable with respect to all self-reversible oracles must take space $\mathbf{\Omega}(\mathsf{S}\log(\mathsf{T}/\mathsf{S}))$.

The most space-efficient linear-time reversible simulation technique that is currently known was provided by Bennett ([19], p. 770), and analyzed by Levine and Sherman [103] to take space $\mathcal{O}(\mathsf{S}(\mathsf{T}/\mathsf{S})^{1/(0.58\lg(\mathsf{T}/\mathsf{S}))})$. Bennett's simulation can be easily seen to work with all self-reversible oracles, so it gives a relativizable upper bound on space. There is a gap between it and our lower bound, due to the fact that the space-optimal pebble-game strategy referred to in our proof takes *more* than linear time in the number of nodes. A lower bound on the number of pebbles used by *linear* time pebble game strategies would allow us to expand our lower bound on space, hopefully to converge with the existing upper bound.

3.4.3 Non-relativized separation

We now explain how the same type of proof can be applied to show a non-relativized separation of $\mathbf{RST}(\mathsf{S},\mathsf{T})$ and $\mathbf{ST}(\mathsf{S},\mathsf{T})$ in certain cases, when inputs are accessed in a specialized way that is similar to an oracle query.

Input framework. Machine inputs will be provided in the form of a random-access read-only memory I, which may consist of 2^b b-bit words for any integer $b \geq 0$. The length of this input may be considered to be $n(b) = b2^b$ bits; let $b(n)$ be the inverse of this function. The machine will have a special *input access tape* which is unbounded in one direction, initially empty, and is used for reversibly accessing the input ROM via the following special operations.

Get input size. If the input access tape is empty before this operation, after the operation it will contain b written as a binary string. If the tape contains b before the operation, afterwards it will be empty. In all other circumstances, the query is a no-op.

Access input word. If the input access tape contains a binary string a of length b before the operation, afterwards it will contain the pair $(a, I[a])$ where $I[a]$ is a length-b binary string giving the contents of the input word located at address a. If the tape contains this pair before the operation, afterwards it will contain just a. Otherwise, nothing happens.

Theorem 2. *Non-relativized separation of reversible and irreversible spacetime.* For models using the above input framework, and for $\mathsf{S}(n) = b(n)$ and any ST-constructible $\mathsf{T}(n)$ such that $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$, $\mathbf{RST}(\mathsf{S},\mathsf{T}) \neq \mathbf{ST}(\mathsf{S},\mathsf{T})$.

Proof. (Sketch following proof of theorem 1.) For input I of length $n = b2^b$, define result bit $r(I)$ to be the first bit in the b-bit string given by

$$\underbrace{I[I[\ldots I[}_{\lfloor \mathsf{T}/\mathsf{S} \rfloor}\mathtt{0}^b]\ldots]].$$

Let language $L = \{I : r(I) = \mathtt{1}\}$. $L \in \mathsf{ST}$ because an irreversible machine can simply follow the chain of $\lfloor \mathsf{T}/\mathsf{S} \rfloor$ pointers from address $\mathtt{0}^b$, using space $\mathcal{O}(\mathsf{S})$ (not counting the input) and time $\mathcal{O}(\mathsf{T})$.

Assume there is a reversible machine M that decides L in $c + c\mathsf{S}$ space and $c + c\mathsf{T}$ time for some c. Let b be sufficiently large for the proof below to work. Let s be a certain description system to be defined. Let $t = \lfloor \mathsf{T}/\mathsf{S} \rfloor$. Let x be a length-$t\mathsf{S}$ string incompressible in s. Let $w_1 \ldots w_t = x$ where all w_i are size b. Restrict s so that all the words w_i must be different from each other and from $\mathtt{0}^b$. Let I be an input of length $n = b2^b$ such that $I[\mathtt{0}^b] = w_1$, and $I[w_i] = w_{i+1}$ for $1 \leq i < t$, and $I[a] = \mathtt{0}^b$ for every other address a. M must at some time access $I[w_{t-1}]$ because otherwise we could change the first bit of $I[w_{t-1}]$ to be the opposite of whatever M's answer is, and M would give the wrong answer. Assign a set of pebbled nodes to each configuration of M's execution on input I like in the oracle proof, except that this time, input access operations take the place of oracle calls. Show, as in lemma 2, that the size of a configuration is at least $\frac{1}{4}p\mathsf{S}$ where p is the number of pebbled nodes, by defining s to allow descriptions that are interpreted by simulating M and reading pebbled nodes from the input access tape. As before, the machine must therefore take space $\mathbf{\Omega}(\mathsf{S}\log(\mathsf{T}/\mathsf{S}))$ which for sufficiently large n contradicts our assumption that the space is bounded by $c + c\mathsf{S}$. Thus $L \notin \mathbf{RST}(\mathsf{S}, \mathsf{T})$. ■

Corollary 2. *Non-relativized lower bound on space for linear-time reversible simulations.* For $\mathsf{S} = b(n)$, computable $\mathsf{S}' \prec \mathsf{S}\log(\mathsf{T}/\mathsf{S})$, and ST-constructible $\mathsf{T}(n)$ such that $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$, $\mathbf{RST}(\mathsf{S}', \mathsf{T}) \neq \mathbf{ST}(\mathsf{S}, \mathsf{T})$.

Proof. As in corollary 1 but with theorem 2. □

Such a T exists because b can be found in space and time $\mathcal{O}(\log b)$ using the "get input size" operation, after which $\mathsf{T} = b^2$, for example, can be found in space $\mathcal{O}(\log b)$ and time $\mathcal{O}(\log^2 b)$. Thus, any reversible machine that simulates irreversible ones without slowdown takes $\mathbf{\Omega}(\mathsf{S}\log(\mathsf{T}/\mathsf{S}))$ space in some cases.

3.4.4 Decompression algorithm

It is probably not obvious to the reader that the algorithm that we briefly mentioned in the proof of lemma 2 in §3.4.2 can be made to work properly. In this section we give the complete algorithm and explain why it works.

The algorithm, shown in figure 3.10, essentially just simulates M_i's operation in direction D starting from configuration C_τ, and reads the identifiers of the pebbled nodes off of M_i's simulated oracle tape. The bulk of the algorithm is in the details showing how to simulate all oracle queries correctly.

There is a small subtlety in the fact that this algorithm has, built into it, some of the values of f that are defined by the oracle. Yet the algorithm is part of the definition of our description system s_i, which is used to pick x and define the $f(q_j)$ values. This would be a circularity that might prevent the oracle from being well-defined, if not for the fact that the portion of f that is built in, that is, $f(b)$ for $|b| < \mathsf{S}$, is disjoint from the portion of f that depends on this algorithm, that is, only values of $f(b)$ for $|b| \geq \mathsf{S}(n_i)$. Thus there is no circularity.

The $f()$ values for the entire infinite oracle can be enumerated by enumerating all values of i in sequence, and for each one, computing the appropriate values of M_i and c_i, and choosing an n_i that satisfies all the explicit and implicit lower bounds on n that we mentioned above. Then, n_i is used in the above algorithm to allow us to define s_i and choose the appropriate x, which determines $f(b)$ for all b where $|b| = \mathsf{S}(n_i)$; these values of f can then be added to the table for use in the algorithm later when running on higher values of i.

We now explain why the simulation carried out by the (oracle-less) decompression algorithm imitates the real oracle-calling program exactly. When we come to an oracle query operation where the queried bit-string(s) do not appear in our $q[j]$ array and do not have a matching $\Delta\tau_j$, then we know the bit-string(s) must not correspond to a real node in $q_1, \ldots, q_t$, because if they did, then either they were not pebbled due to queries in direction D, in which case they would have been in the description d and would have been present in the initial q array, or else the first query that involved them must have been before the current one (or else some $\Delta\tau_j$ would match), in which case they would have been added to the q array earlier.

Moreover, when we get to a single query q_j, we know we can look up q_{j+1} to answer the query, because it must already have been stored. Either q_{j+1} was not pebbled in direction D in which case it was stored originally, or it was pebbled in direction D in which case the first query involving it must have been before this one, since this query is not of the type that would have caused the node to be pebbled in direction D. In either case we will already have a value in array entry $q[j+1]$.

Given any description d derived from the execution history of a real M_i, the simulation will eventually find values for all nodes, since either they were given initially or they are found eventually as we simulate. Thus the algorithm prints x, as required for the proof of lemma 2.

3.4.5 Can this proof be carried farther?

Given the work above, an obviously desirable next step would be to show that $\mathbf{RST}(\mathsf{S},\mathsf{T}) \neq \mathbf{ST}(\mathsf{S},\mathsf{T})$ for a larger class of space-time functions S,T in a reasonable serial model of computation *without* an oracle. A similar problem of following a chain of nodes may still be useful for this. But when there is no oracle, and when the required time is larger than the input length $\mathsf{T} \succ n$, there is no opportunity to specify an incompressible chain of nodes to follow. Instead, the function f mapping nodes to their successors must be provided by some actual computation that is specified by the relatively short input. It will be helpful if f is non-invertible or is a one-way invertible function, whose inverse might be hard to compute. But the function will still have some structure, and so it may be very difficult to prove that there are no shortcuts that might allow the result of repeated applications of the function to be computed reversibly using little time or space.

3.5 Summary of reversible complexity results for traditional models

Let us summarize the above results on complexity in reversible machines. In sec. 3.2.2, p. 54, we described a variety of measures of the cost or "complexity" of a computation. Now we will summarize how reversible and irreversible models compare under different measures of computational complexity.

Let $\mathcal{M}$ denote an arbitrary (not necessarily reversible) abstract model of computation such as a Turing machine or RAM machine, in which primitive operations may

```
Given description $d$ as described in the text,
Let $q[1]\ldots q[t]$ be a table of node values,
    initially all NULL.
Initialize all $q[j]$'s not pebbled in direction $D$,
    as specified by description $d$.
Simulate $M_i$ in direction $D$ from configuration $C_\tau$,
as follows:
    To simulate a single operation of $M_i$:
        If it's a non-query operation, simulate it
            straightforwardly, and proceed.
        Otherwise, it's an oracle query.
        Examine oracle tape.
        If it's not of the form $b$ or $b\#c$ for
            bit-strings $b, c$, $|b| = |c|$, do nothing.
        If $|b| < \mathsf{S}$, look up $f(b)$ in a finite table,
            and set the oracle tape appropriately.
        If $|b| > \mathsf{S}$, do nothing for this operation.
        If the query is of the form $b$, then
            If current time matches some $\Delta\tau_j$,
                set $q[j] = b$.
            If $b = q[j]$ for some $j < t$,
                set oracle tape to $b\#q[j+1]$,
            else go ahead to the next operation.
        If query is of the form $b\#c$, then
            For each $\Delta\tau_j$ matching current time,
                set $q[j]$ to $b$ or $c$ depending on tag $k_j$.
            If $b = q[j]$ and $c = q[j+1]$ for some $j$,
                set oracle tape to $b$,
            else do nothing for this operation.
    Increment time counter.
    Repeat until time exceeds largest $\Delta\tau_j$.
Print all $q[j]$'s.
```

Figure 3.10: Algorithm to print the incompressible chain of nodes x via simulation of the reversible machine M_i.

be reversible or irreversible, and unbounded memory is available. Let $\mathrm{R}(\mathcal{M})$ denote the corresponding reversible model of computation, like $\mathcal{M}$ but with all primitive operations constrained to be perfectly logically reversible, and with an extra stack for history information made available to each active processing element from the original machine.

Number of ticks. Under the cost measure T (number of "ticks" of some synchronous, computational "clock"), a reversible model $\mathrm{R}(\mathcal{M})$ is exactly as efficient as an arbitrary model $\mathcal{M}$. This follows from the Landauer-Lecerf-Bennett ideas as we discussed in §3.3.3, p. 61, which apply to parallel models as well as to serial models.

Memory requirement. Under the cost measure S (maximum memory used at any time during the computation), $\mathrm{R}(\mathcal{M})$ is exactly as efficient as $\mathcal{M}$. This follows from the Lange-McKenzie-Tapp technique we discussed in §3.3.5.3, p. 65. However, if the model includes an input stream that can only flow one way and whose length is not included in S, then Pin's proof [136] applies, and the reversible model is structly less space-efficient, because there are regular languages that cannot be recognized by constant-space reversible machines.

Memory and number of ops. Under the cost measure (S, T), without additional assumptions, we only know that $\mathrm{R}(\mathcal{M})$ is no more efficient than $\mathcal{M}$. However, given one-way external inputs, reversible models are less efficient by our argument in the previous paragraph. Moreover, this is also true given a certain oracle, or given random-access external inputs, by our two new theorems from §3.4. We conjecture that reversible models are less (S, T)-efficient even given only internal inputs and simple primitive operations, but this has not yet been proven.

Memory times num. ops. The statements of the previous paragraph also apply to the cost measure ST (the product of the traditional space and time). Therefore, probably reversible models are in general strictly less space-time efficient, in traditional complexity-theory terms.

Note that all the above comparisons deal in measures of complexity that are characterized in abstract, computational terms, such as the number of operations performed, rather than as real physical quantities; and the models of computation that were compared were all idealized abstract models, rather than models of machines as they could be physically implemented.

Normally in computer science it is often assumed that such abstract models are a good enough approximation to reality so that correct conclusions can be inferred from the resulting abstract theory. In our case, the theory would seem to indicate that machines that are constrained to be reversible are strictly inferior to unconstrained machines, including machines that are completely irreversible.

However, in chapter 6 we will show that this conclusion is actually in error, in the sense that if one uses more physically realistic models of machines and of costs, machines that are completely *ir*reversible are instead strictly inferior to machines that are allowed to be reversible to some degree, and are sometimes even inferior to fully reversible machines. We anticipate that this inferiority will make itself felt in a variety of of present-day and projected future computing technologies.

Therefore, for the purpose of deciding between reversible and irreversible modes of computation in the real world, we see that traditional computer science and traditional complexity theory are inadequate; they give the opposite of the correct answer in some cases!

This underscores our overall point, which is that computer scientists must not become mired in the traditional models of computation, but instead should strive to keep their models up-to-date with all the new factors that become important as technology improves.

We believe that computer science, as a field, should look ahead and try to anticipate the ultimate physical limits of computer technology, and begin studying models that are accurate enough to give the right answer even in that limit.

In that spirit, chapter 4 reviews quantum computation, which will stretch the bounds of computational efficiency, if it turns out to be feasible. Chapter 5 further motivates our quest for an ultimate physical model of computation, and outlines some plausible candidates that should remain valid at least through the foreseeable future. Chapter 6 shows why the ultimate model will need to permit an arbitrarily high (if not perfect) level of logical and physical reversibility. Then, Part II of this thesis will present a variety of engineering designs and analyses demonstrating that reversible computation is indeed quite feasible, and that most concepts from ordinary irreversible computation remain applicable to reversible computing as well.

Chapter 4

Quantum computation

In chapter 2 we saw various ways in which fundamental physics limits what we can do with computers. But even within these limits, physics may also afford new, previously unforeseen opportunities.

In this chapter we review the field of *quantum computation*, which studies a particular new kind of computation that takes advantage of known physical principles in a new way which appears to be fundamentally more powerful than the kinds of computation employed in any existing computers. Quantum computing may or may not turn out to be practical in the long run, but its eventual feasibility has not yet been conclusively ruled out. Therefore it is important for us, when considering the ultimate limits of computing, to at least be aware of the ways in which quantum computing may potentially belie the conventional wisdom about what is possible.

Moreover, the basic operations used in quantum computation are inherently reversible, and thus constitute a potential realm of application for some of the reversible algorithm techniques discussed later in this thesis.

Quantitative Church's Thesis. The "Quantitative Church's Thesis" [176, 175] claims that Turing machines are as efficient as any realistic computer, within a polynomial factor. However, Feynman [59] has pointed out that Turing machines seem to be unable to efficiently simulate quantum physics; that is, they seem to require an exponential slowdown to simulate it (although this has not been proven). This leads naturally to the supposition that a computer that was designed to take full advantage of quantum physical principles might be found to be exponentially faster than a Turing machine, at least for some problems, thus disproving the Quantitative Church's Thesis. Such a development could lead to eventual practical applications, if and when such *quantum computers* become buildable.

Shor's Factoring Algorithm. However, this idea remained pure speculation until the last several years, when a series of papers on the power of quantum computers

[47, 23, 22, 21, 149] culminated in Peter Shor's 1994 proof [146, 147] that a (somewhat idealized) quantum computer could factor large integers in polynomial time in the number of bits n in the integer. This was an astounding discovery, since mathematicians throughout history have searched for an efficient way to factor numbers without success, since at least the time of Euclid. The best known classical algorithm [102] takes exponential time.[1]

4.1 Some fundamental quantum concepts

Hilbert spaces. The most important thing to understand about the difference between the quantum and classical paradigms of physics is that in the quantum paradigm, the classical notion of the state of a system is not sufficient to describe the dynamical evolution of the system. In fact, if $\mathcal{S}$ is the space of possible states of a classical system, then a corresponding quantum description of the system involves a consideration of the much larger space of *functions* from $\mathcal{S}$ to the complex numbers $\mathbb{C}$, *i.e.* the space $\mathcal{H} = \mathbb{C}^{\mathcal{S}}$, to use the set theory notation. The number of dimensions of $\mathcal{H}$ is equal to the number of *elements* of $\mathcal{S}$. This space $\mathcal{H}$ is called the *Hilbert space* of the system.

Amplitudes and probabilities. So again, the quantum state of a system consists of a function $\Psi : \mathcal{S} \to \mathbb{C}$ from the classical states of the system to the complex numbers. These complex numbers are called *amplitudes* and their physical significance is that the square of the absolute value of the amplitude of a classical state x is interpreted as the probability that the system would be found to be in state x if the system were observed, *i.e.* $\Pr(x|\Psi) = |\Psi(x)|^2 = \Re(\Psi(x))^2 + \Im(\Psi(x))^2$, where $\Re(z)$ and $\Im(z)$ represent the real and imaginary parts of the complex number z, respectively. A simple probability distribution over classical states would suffice instead, if a static description of the system were all that was required. But if the dynamical evolution of the system is to be modeled in a quantum-theoretic way, the full power of a complex function is necessary; the probabilities alone do not suffice to describe how a quantum system can change over time. The function from states to amplitudes is commonly referred to as a *superposition of states*.

Unitary transformations. The fundamental principle of change in quantum systems is the *unitary transformation*. If we imagine the amplitude function Ψ being identified with a vector of complex numbers indexed by the states in $\mathcal{S}$, then a unitary transformation is simply a multiplication of those vectors by a transformation matrix whose inverse equals its conjugate transpose. (To briefly review some definitions from linear algebra and complex arithmetic, the *transpose* M^{T} of a matrix M is defined

---

[1] More precisely, $O(\exp(n^{1/3}\log(n^{2/3})))$.

by $M^{\mathrm{T}}(x,y) = M(y,x)$ *i.e.* the matrix is flipped around its diagonal axis, and the *conjugate* M^* of M is defined by $M^*(x,y) = \Re(M(x,y)) - \Im(M(x,y))i$, that is, the sign of the imaginary part of each element is negated. The inverse M^{-1} of M is the matrix such that $M^{-1}M = I$, the identity matrix.)

A unitary transformation, intuitively speaking, corresponds to a length-preserving and information-preserving rotation in the vector space. The length-preserving requirement ensures that the total probability of the set of states always remains the same (1 if normalized), and the information-preserving requirement reflects what seems to be a universal conservation principle in fundamental physics (which we discussed in §2.5), namely that all changes are information-preserving on a microscopic level; or, in quantum-mechanical terms, the complete quantum state (amplitude vector) of an isolated system at any time determines the quantum state of the system at all past and future times.

Measurement phenomena. We have already mentioned that a measurement of the state of a quantum system (or of part of the state) yields a result with a probability equal to the square of that state's amplitude. (Or, if only part of the state is measured, the probability is equal to the sum of the squares of the amplitudes of all the global states that are consistent with the state of the part being measured.) However, such a measurement has an apparent side-effect on the system, namely that the probabilities of all states inconsistent with the observed result drop to zero, and the amplitudes of all the states that are consistent with the observed result are scaled up accordingly, so that the total probability over all states remains 1.

In other words, after observing a measurement (that is, allowing the measurement to affect the outside world), one cannot in practice just undo the measurement to restore the amplitudes of all states to their original values, putting the genie back in the bottle, as it were. Instead, the original superposition seems to be effectively destroyed, and all further measurements on the system will be in accordance with the new *collapsed* superposition.

In the traditional *Copenhagen* interpretation of this phenomenon, this event of the collapse of the global wavefunction (which would be a non-unitary and non-local transition) is said to *actually* occur. But there is an alternative explanation for the observed phenomena that is, in our view, philosophically much simpler and thus more plausible. It does not require postulating any new basic principle of "wavefunction collapse," rather, the apparent phenomenon of collapse can be seen to follow automatically from the basic nature of quantum mechanics.

This alternative is Everett's theory of the Universal Wavefunction [57] which says that regardless of whatever interactions occur between a quantum system and the external world, these interactions continue to obey unitarity, and the entire system as a whole (system being studied, plus rest of world) remains in the superposed state predicted by quantum mechanics. If the appropriate inverse transformation could be applied to the system as a whole, the original state could in principle be restored. In a measurement experiment, if information about the measurement does not leak out of the measurement apparatus into the outside world, there is no reason why the measurement, if carefully controlled, cannot be "undone" to restore the original state. (In fact, recent experiments reported in [189] confirm this.)

However, we do not have control over the quantum interactions that take place after a photon (say) that is carrying state information leaves an experiment and strikes, say, an observer's eyeball; after that interaction takes place, the information about the state of the experiment gets all mixed up with the states of billions of particles, and although a superposition is still present, the states corresponding to the different outcomes of the experiment have drifted so far about from each other through random interactions with other particles that there is essentially zero probability that they will ever drift back together to become the same state again, which is necessary in order for their amplitudes to again add up and interfere with each other. It doesn't really matter whether the photon strikes an eyeball or a rock. The different states corresponding to different outcomes of the experiment drift so far apart from each other (in Hilbert space) that it is an excellent approximation to treat them as completely independent and non-interacting. (But if the photon strikes a waveguide that directs it back into the experiment in a controlled way, that's different.)

Therefore, within the context of any state in which information has leaked out and interacted with an uncontrolled, un-modeled external environment, we can appropriately shift to using a model in which the value of this "measured" information is simply chosen according to the $|\Psi|^2$ distribution, and the amplitudes of the other states (those that are inconsistent with the leaked information) are zero. The only alternative would be to include all of the zillions of interacting particles of the external environment in one's model of the state of the system; this alternative is of course impossible for any "environment" that is not itself a known and carefully-controlled quantum system.

In any case, to avoid confusion, a more precise statement of the "measurement phenomenon" described earlier would define a measurement as any event in which state information about a part of a system interacts with an uncontrolled, unmodeled environment.

Unfortunately, distinguishing experimentally between the Copenhagen and Everett interpretations is difficult, if not impossible. (However, for an argument that it is possible, see [46].) Many philosophers have found the Everett interpretation to be highly implausible, and have dismissed it out of hand, because it implies the existence of enormous numbers of alternate versions of our universe that we are unable to interact with. However, based on my own personal view of the ideal rational method for comparing scientific theories, Everett's is clearly the *simpler* and less *ad hoc* theory, in the sense of having a more concise mathematical formulation, and therefore it is more likely; the fact that it predicts this amazing and untestable consequence of a universe that is far more elaborate than we can possibly observe is, in my view, completely irrelevant to assessing the theory's plausibility as a scientific explanation that thoroughly explains the phenomena that we *can* observe.

Interestingly, the future development of quantum computers (if successful) can be seen as stretching the bounds of the plausibility of the Copenhagen interpretation, because the functionality of a quantum computer depends on the assumption that global superposition states of large and complex systems are indeed possible. The larger the quantum computer that we can successfully build, the more implausible seems the Copenhagen viewpoint, which arbitrarily demands that the simple quantum theory (which says that all physics works through unitary transformations), depite applying perfectly well to large, complex quantum computing systems, cannot be applied identically all the way up to cover extremely large systems, *e.g.*, the whole universe.

4.2 Quantum complexity theory

After Deutsch introduced his quantum generalization of the Turing machine [45], researchers wondered whether this computational model has computational capabilities greater than those of classical Turing machines. In his original paper, Deutsch showed that quantum computers could exploit "quantum parallelism" to simultaneously compute function values for N inputs using only one mechanism. This works because the unitary transforms that apply to quantum states operate on the amplitudes of all the possible classical states of the system simultaneously. Thus, a single unitary transformation that implements the transition function f of a computation can simultaneously take the state x_i to $f(x_i)$ for all i in some index of the N initial states.

Unfortunately, this parallelism does not effectively allow one to do N times as much work with it as without it, because the N results of one computation cannot all be measured, since (as described in the previous section) communication of state information to the outside world effectively isolates the possible values of the measured state variable from each other, and so effectively causes states inconsistent with the measured information to have nonzero amplitude. The only way that information about the amplitudes of *different* mutually exclusive states can be combined is by taking unitary, linear transformations of those amplitudes *before* measuring state information.

However, for certain problems, such unitary combinations of the amplitudes of different states may provide information useful for solving the problem. Deutsch hinted at this in his original paper, showing that the XOR of the values of a boolean function on two different inputs could be computed in the time needed to evaluate the function once, if a certain transformation of the result states was performed before measuring them. The catch is that half the time, at random, the measurement yields no information—so the expected *rate* of finding these XORs is the same as with a classical algorithm that first computed the value for one input, and then the other, and XORed them.

In a later paper [47], Deutsch did much better—together with Jozsa he showed that a certain property of functions could be determined with certainty exponentially faster by quantum programs than by classical ones, if the function is given as a black box as input to the program. The property (for a function that returns 0 or 1) is whether the function is *variable* (it has value 0 for some inputs, and 1 for some inputs), or *biased* (it has one value for more inputs than the other value) if we are given that it is not both. If, on the contrary, it is both—say if its value is 1 on two-thirds of the inputs—the quantum algorithm may return either answer.

Unfortunately, it is hard to think of a realistic scenario where such an ability might be useful. For example, if the given function is a simple boolean formula applied to its input bits, we may be interested in knowing whether the function is variable (which corresponds to the famous SAT or "satisfiability" problem), but if it is, who cares whether it is biased! Unfortunately, if the function is highly biased, as is generally the case for hard SAT problems, then the algorithm will almost always answer "biased" instead of "variable," giving us no help with the satisfiability question. A classical algorithm could do just as well on SAT by trying input assignments at random. The Deutsch-Jozsa algorithm is really only helpful if we know that the input function is either constant or unbiased, and we cannot tolerate any non-zero probability of failure in determining which one it is. This seems like an unnatural problem.

But in any case, following the Deutsch-Jozsa paper, analysis of the power of quantum computers developed rapidly with papers [23, 22, 21] that defined various quantum complexity classes and compared them with various classical complexity classes in relativized oracle settings similar to Deutsch and Jozsa's. Quantum operations were also found to have uses in implementing various cryptographic operations; see the end of [23] for a summary. Quantum analogues to the popular classical complexity classes such as **BPP** (bounded-error probabilistic polynomial-time) and **ZPP** (zero-error probabilistic polynomial-time) were defined, and various of the quantum classes were shown to be larger than the various classical classes—but only in relativized oracle settings, such as we used in §3.4.2.

However, none of the oracle problems addressed seemed particularly evocative of real problems until Simon's [149] paper, which showed that the following problem was in **ZQP** (zero-error quantum polynomial-time): We are given a function f, and told that either f is 1-to-1, or else it is 2-to-1 and there is some bit-mask s such that $f(x) = f(s \oplus x)$ (where $\oplus$ is bitwise exclusive-or) for all input bit-patterns x. The problem is to determine whether the former or the latter is true, and if the latter, to find s. This seems a better than the earlier problems because it actually returns a significant amount of information about its input in the form of finding the XOR-mask s (if it exists). Anyway, Simon showed that quantum computers could solve this problem with certainty using a polynomial number of queries of the input function. Classical algorithms require exponentially many tries to achieve a reasonable probability of success.

The extraordinary thing about Simon's construction was its use of a particular unitary transformation equivalent to a special-case of the discrete Fourier transformation that had been introduced earlier by Bernstein and Vazirani [21]. Originally this Fourier transform was used to solve a certain simple oracle problem using $\mathcal{O}(1)$ queries on a quantum computer as opposed to the $\boldsymbol{\Theta}(n)$ queries that were classically required. The Fourier transform is linear and invertible; it turns out that it is unitary as well, and a discrete Fourier transform on functions of n-bit inputs can be performed on a quantum computer in time polynomial in n using a recursive procedure related to the classical fast-Fourier-transform (FFT) algorithm [91].

Simon's ingenious use of the quantum Fourier-transform algorithm to reduce an exponentially-hard problem to polynomial time was the original inspiration for Shor's application of a more general version of the transform to a difficult and plausibly important problem: factoring large integers. (The problem is important, at least to certain government agencies, because efficient factoring is the key to cracking RSA ([38] §33.7, p. 831), the popular public-key cryptography algorithm.) We now summarize Shor's algorithm.

4.3 Outline of Shor's Algorithm

Shor's algorithm depends on an old reduction from number theory, which translates the problem of the factorization of N to the problem of finding the *order* of a number $x \pmod{N}$. To understand the "order" concept, recall that if x is relatively prime to N, then multiplication by $x \pmod{N}$ is one-to-one. Therefore, the series $x^0, x^1, x^2, \ldots \pmod{N}$ eventually gets back to 1 and cycles around again, *i.e.*, $\exists r > 0 : x^0 \equiv x^r \pmod{N}$. The order of $x \pmod{N}$ is defined to be the least such r. The connection with factoring is that if r is even, then either $(x^{r/2} - 1) \bmod N$ or $(x^{r/2} + 1) \bmod N$ has a common factor with N, which can then be easily found using Euclid's algorithm for finding the greatest common divisor (gcd) ([38], §33.2, p. 808.)

Therefore, if a polynomial-time algorithm for finding the order of a number mod N is available, then it can be used to factor N as follows:

1. Pick a random $x < N$.

2. Compute $f = \gcd(x, N)$; if $f \neq 1$, return f. (It's a factor.)
3. Find the least r such that $x^r \equiv 1 \pmod{N}$.
4. If either $\gcd(x^{r/2} - 1, N)$ or $\gcd(x^{r/2} + 1, N)$ is not 1, return it, it's a factor.
5. Otherwise, go to step 1 and repeat.

The number of repitions of the above loop required to find a factor with probability ≥ 0.5 can be shown to be only polynomial in the length of N, therefore if all the steps 1-4 are polynomial, then the algorithm as a whole takes only polynomial time.

The bottleneck in the algorithm is of course the computation of r, which Shor implements via a clever application of the Fourier transform to quantum parallelism. The quantum computer is made to compute x^r for all $r < N^2$ simultaneously. As discussed earlier, the series of x^r ($\pmod{N}$) is cyclically repeating if x is relatively prime to N; the period of repitition is the order of x. This cycle of repeating values is stored simultaneously in N^2 distinct states corresponding to the N^2 different input values of r; each state is present with the same amplitude. Then, the quantum Fourier transform is applied to the superposition of result states. Instead of representing an equal superposition of all the r and $x^r \bmod N$, the function from states to amplitudes now encodes a superposition of frequency spectra for finding the different values of $x^r \bmod N$. These spectra will have amplitude peaks at points correspondsing to multiples of the basic repitition period of x^r. If we then measure the state, it will (with high probability) lie very near one of the peaks, and the value of the state measured will let us guess the repitition period (which is the answer we are looking for) with high probability.

Let us examine how this is done in more detail, with reference to the example illustrated in figs. 4.1 and 4.2. In this example, the number to factor is $N = 3 \times 11 = 33$. Let q be the smallest integral power of 2 greater than N^2, and let $\ell = \lg q$. In the real algorithm q would be 2048, but we will take $q = 256$ instead for ease of visualization. We will have two quantum registers (a, b), where a is an ℓ-bit register ranging from 0 to $q - 1$, and b ranges from 0 to $N - 1$.

In figs. 4.1 and 4.2 we illustrate the superposition state of the joint space of these two registers, following each stage of Shor's algorithm. In these figures, the position along the horizontal axis represents the state of register a, and the position along the vertical axis represents the state of register b. At every point on the resulting two-dimensional surface representing the combined state space, we place a blob of ink whose darkness corresponds to the absolute amplitude of that state. (Additionally, when displayed on a color output device, the hue of the ink in these figures indicates the phase of the amplitude.) White areas correspond to states having zero or nearly zero amplitude.

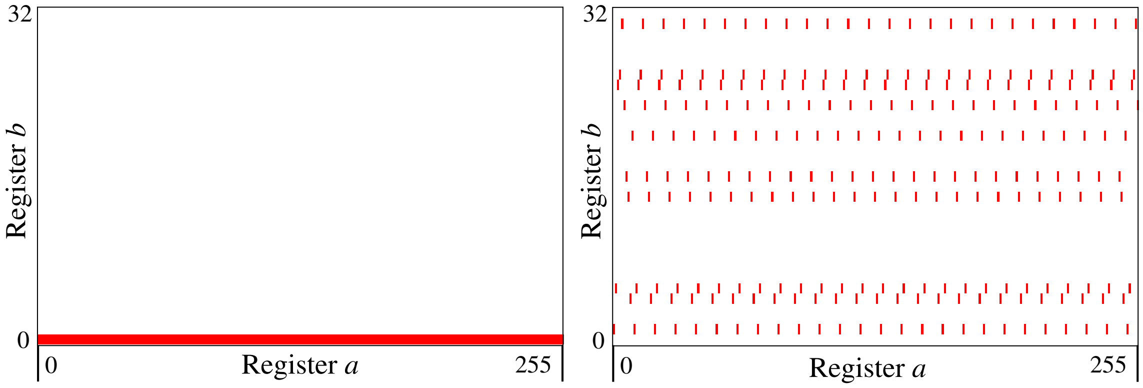


Figure 4.1: Example of Shor's algorithm with $N = 33$, $q = 256$, $x = 5$. Left: superposition over all values of a, with $b = 0$. Right: superposition after computing $b = x^a \mod N$. Across a there is periodic cycling of b through the 10 values 1, 5, 25, 26, 31, 23, 16, 14, 4, 20.

We start with an initial quantum state of $|0, 0\rangle$, that is, $a = 0$, $b = 0$. First we prepare an equal superposition of all values of a while leaving $b = 0$. The left side of fig. 4.1 shows the amplitude spread out over all the values of a, in the row corresponding to $b = 0$. This is the superposition

$$\sum_{a=0}^{q-1} |a, 0\rangle / \sqrt{q}.$$

Next, x is chosen at random (in our case, to be 5), and we perform a classical reversible computation to transform the value of b from 0 to $x^a \mod N$. The right side of fig. 4.1 shows how the blob of amplitude associated with each value of a is moved vertically to the state also having the correct value of b. With increasing a, we see that b cycles periodically among various values, with a period that is (by definition) the order of x mod N, which in this case is 10.

Finally, leaving b, alone we perform a Fourier transform over the value of register a. This is defined by

$$\psi'(a', b) = \sum_{a=0}^{q-1} \exp(2\pi i a a'/q) \psi(a, b) / \sqrt{q}$$

Figure 4.2 illustrates how the amplitude in each row associated with each value of b is moved horizontally to cluster in peaks, whose number corresponds to the period of the original amplitude distribution. Since the period is the same for each value of b, the peaks in each row line up, although they have a different phase in different rows. (The color version of this document shows the amplitudes in color, with the hue denoting the phase.)

After this, register a may be sampled, and several samples will be sufficient to tell us its period, 10 in our case. Now the rest is easy: Half of 10 is 5, and $x^5 \bmod N$ is 23 in our case. Twenty-three minus 1 is 22, which has a common factor (11) with our N, and twenty-three plus 1 is 24, which also has a common factor (3) with our N.

The details of exactly how the quantum Fourier transform works are beyond the scope of this short survey. For more detailed expositions of Shor's algorithm, see Ekert and Jozsa's description in [54, 53], and Shor's original papers [146, 147].

4.4 Important open problems

Here are some of what I believe to be the most important, and also the most difficult open questions in quantum computing:

4.4.1 Can quantum computers solve **NP**-complete problems in polynomial time?

This is perhaps the most interesting question about quantum computing. If the answer to this question were positive, then quantum computing could revolutionize computing, as we know it. There are a wide range of practical problems in constraint-satisfaction, combinatorial search, and other areas that have been shown to be in **NP**, but for which no efficient classical algorithms are known.

The ability to solve **NP**-complete problems efficiently would also revolutionize all of mathematics, because it would enable us to quickly determine, for any given mathematical statement, whether or not there is a fairly simple proof (or disproof) of the statement, and if there is, to find it. Automatically checking a (suitably formal) proof for correctness can be done in polynomial time in the proof length; therefore, finding a proof of a given length can be done in nondeterministic polynomial time in the target length. Therefore a method for solving **NP** problems in polynomial time could find proofs in time polynomial in the proof length.

Currently, it seems unlikely that quantum computers could solve **NP**-complete problems, due to the fact that the only known quantum algorithms that dominate classical algorithms either involve unrealistic oracle-dependent promise problems [21, 149], or introduce only polynomial speedups [77, 78], or only simulate quantum mechanics [25], or depend on the ability to reduce the problem to one involving periodicities for which the quantum Fourier transform is useful [147, 26]; one would not expect such periodicities *a priori* to be characteristic of all problems in **NP**.

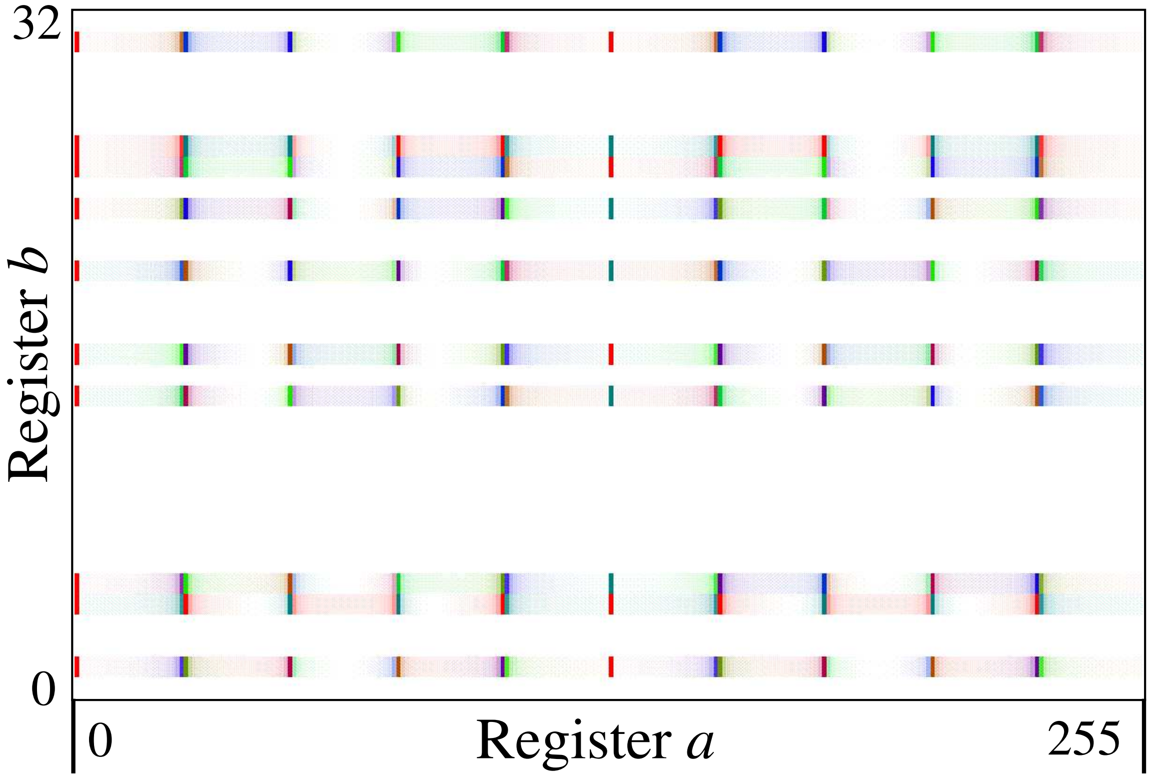


Figure 4.2: Superposition after performing quantum Fourier transform over register a in Shor's algorithm, when factoring $N = 33$ with the choice of $x = 5$. The value of a is now peaked at 10 points spaced 25.6 units apart. The number of peaks is the period (order) of the function 5^a mod N, shown in the right half of fig. 4.1. After the transform, sampling a permits determining the period, and halving it, we find 5^5 mod $33 = 23$, which is 1 away from numbers that have common factors with 33, and the problem is solved.

However, it is conceivable that someone may yet discover a clever quantum algorithm for general simulation of nondeterministic Turing machines. One possible approach might be to reduce some **NP**-hard problem to the factoring or discrete logarithm problem; however, complexity theorists have attempted such reductions for many years without success.

A more sophisticated approach might involve finding some other unitary transformation other than the Fourier transform that could allow states associated with parallel paths in a quantum simulation of a nondeterministic machine to constructively interfere in ways that might yield useful information about the structure of the search space, and help to pin down the solutions. Hogg [85] has investigated quantum algorithms that enhance the probability density found along solution paths in **NP** search problems, but not enough to allow measurements of the machine state for such problems to yield solutions in expected polynomial time.

On the other side of the question, Bennett *et al.* [20] have provided suggestive evidence against the $\mathbf{NP} \subseteq \mathbf{BQP}$ conjecture, by showing that when R is a random oracle, $\mathbf{NP}^R \supset \mathbf{BQP}^R$ with probability 1. However, it is worth noting that since $\mathbf{BQP} \supseteq \mathbf{P}$, an actual proof that $\mathbf{NP} \supset \mathbf{BQP}$ would imply that $\mathbf{NP} \supset \mathbf{P}$, a conjecture whose proof has long eluded complexity theorists. Thus, it seems unlikely that the possibility of quantum computers subsuming nondeterministic computers will be conclusively ruled out anytime soon.

4.4.2 Are quantum computers strictly more powerful than classical computers (with a bounded probability of error)? *I.e.*, $\mathbf{BQP} \supset \mathbf{BPP}$?

This question at first appears the same as the previous one, but there are two important differences. First, it may be the case that $\mathbf{P} = \mathbf{NP}$, in which case both quantum and classical computers could solve all problems in **NP** efficiently, and so the quantum computers might not be any more powerful than the classical ones.

Secondly, if the answer to the question is negative, *i.e.* if $\mathbf{BQP} \subseteq \mathbf{BPP}$, then this has important implications for physics, because it might mean that existing classical computers could therefore simulate arbitrary quantum systems with only a polynomial slowdown, which is not currently known to be possible [59]; current classical simulations of quantum systems all suffer from an exponential slowdown.

A faster method for simulating quantum physics would revolutionize much of theoretical physics, because it would allow many more predictions to be derived from quantum theories, predictions which could then be compared with experiment to refine the theories; and it would also reduce the current dependence on approximation methods in many important areas of applied physics, such as modeling molecular interactions, *e.g.* for drug design.

Unfortunately for these tantazlizing prospects, the existence of Shor's factoring algorithm, together with the long-standing failure of many generations of brilliant mathematicians to find a classical equivalent, seems to leave little hope for a quantum-classical equivalence. Even if quantum computers are not as powerful as nondeterministic Turing machines, they may be strictly more powerful than deterministic ones.

4.4.3 Can errors caused by imprecision and decoherence be controlled sufficiently to allow arbitrarily complex quantum computations to take place with an arbitrarily small probability of failure?

A number of papers have expressed pessimism regarding the question of error accumulation in quantum computers, *e.g.*, [96, 95, 173, 174, 33, 132]. These papers show that in the absence of error correction, the probability of error increases exponentially with both the time and space complexity of the computation, and the expected error-free running time for various experimental setups has been estimated to be roughly on the order of the time to perform a single computational step, seemingly ruling out the possibility of doing interesting quantum computations.

However, Coppersmith [36] has shown that simple imprecision does not cripple the quantum factoring algorithm, and several more recent papers [28, 32, 155] have addressed the more difficult issue of correcting errors due to decoherence of the quantum states. They work by encoding a bit value redundantly in a superposition of many bit values in such a way that up to n independent interactions of bits with the environment can take place without communicating any information about the value of the encoded bit to the environment. If fewer than n bits interact with the environment, than the system can exactly recover the originally-encoded superposition and then regenerate its redundant representation. The only problem is that this reconstruction process will in general be subject to errors as well. More sophisticated techniques might take that into account. In summary, although these papers appear to be on the right track to a solution, a more complete theory of quantum error correction is still needed, and remains to be worked out.

Cesar Miquel's 1996 preprint [127] reports results of some simulation experiments on error-correcting versions of Shor's algorithm in the presence of errors.

4.4.4 How do we build it, physically?

Although the question of how to implement quantum computations physically is of course a question of utmost importance for the future of the field, to a large extent the details of the physical implementation are orthogonal to most of the theoretical issues dealt with in the literature we have reviewed. There is another huge literature, mainly under the rubric of experimental physics (rather than quantum theory or computer science), which deals with constructing physical realizations of systems of controlled interactions between quantum states. For example, researchers in quantum optics study how to manipulate information encoded in the polarization staes of photons; "cavity QED" workers study the interactions between photons and electron spins on individual atoms [56, 168]; and other experimentalists work with vibrational states in assemblages of interacting atoms [35].

An intriguing recent development in implementation techniques has been the investigation of NMR "ensemble quantum computing," in which the nuclear spins of atoms in molecules in solution are made to interact using nuclear magnetic resonance techniques [39, 76, 40]. The NMR experiments have had the most success of any techniques to date; quantum logic operations involving 2 and 3 bits have been successfully demonstrated.

Even more recently, Mooij *et al.* [128] have described how to implement quantum computation on a chip surface in microlithgraphed superconducting Josephson-junction circuits. In this scheme, quantum bits are encoded in the direction of a quantized current flow in a superconducting loop. The system is projected to have very long decoherence times, making it fairly amenable to quantum error correction algorithms, and moreover, quantum circuits of arbitrary size and complexity can be readily patterned. At the moment, this approach seems the most likely candidate for implementing a practical quantum computer in the near future.

Some older proposals for implementation technologies for quantum computing, from various communities, include Teich *et al.* '88 [158], Lloyd '93 & '94 [112, 113], DiVincenzo '95 a [49], Sleator & Weinfurter '95 [150], Barenco *et al.* '95 b [10], and Chuang & Yamamoto '95 [34].

The main lesson to be learned from this long list of proposals is that the details of the physical implementation of quantum computers are "just" an engineering concern, rather than a theoretical issue of fundamental importance. Researchers since Feynman [60] have noted that there seems to be nothing fundamental in quantum physics that precludes using it for computation, and indeed, the multiplicity of ideas listed above seems to bear that out. Although certainly the development path of many particular techniques will be beset with problems, it seems likely that eventually our technologies for manipulating quantum particles will mature to the point where some form of complex controlled assemblages of quantum states will be built fairly readily—that is, if it is useful to do so. The question of whether it would be useful can only be answered by the theoretical studies such as those we surveyed in this chapter; no matter how a quantum computer is finally built, the theorems and algorithms produced by those studies will still apply.

4.5 Summary of quantum computation

Although it appears that the basic computational steps of quantum computers may soon be implementable, and that a large quantum computer may be able to factor numbers faster than a classical computer, many practical problems such as error-correction remain to be solved before we can scale up to large enough computers to be useful. Another possible show-stopper is that factoring and a handful of other problems relating to cryptography [26] may turn out be the only real-world problems amenable to fast quantum solutions, which may not provide enough motivation to support the development of quantum computers. Although cracking RSA is a tantalizing prospect, it would not necessarily change the world radically—RSA might just be replaced by another code that is less amenable to quantum solution, especially perhaps one using quantum crypography. However, there is hope that applications such as the use of quantum computers to efficiently simulate models of real quantum physical systems [25] might revolutionize physics as we know it.

In chapter 2 and in this chapter, we have seen how some basic physical principles affect the limits of what is possible with computers, sometimes in surprising ways. In the next chapter, we will discuss the benefits to be gained from developing "ultimate" models of computation that accurately reflect these limits, and we will propose some candidates for an ultimate model. The ultimate model may or may not turn out be able to use large-scale quantum coherence in the way that we have discussed in this chapter, but in chapter 6 we will see that, at the very least, the ultimate model must be reversible.

Chapter 5

Ultimate physical models of computation

In this chapter we elaborate in some depth our motivating long-term goal (first mentioned in §1.2) of describing "ultimate" physically-based models of computation. We contrast this concept with the models that have traditionally been studied in computer science. We outline several plausible candidates for what the ultimate model ought to look like, based on the fundamental physical observations from chapters 2 and 4.

Traditional complexity theory. As we saw in chapter 3, the subfield of computer science known as "the theory of computational complexity" traditionally deals with how the time and space required for a computer to solve problems in a given class depends on the size of the problem. Unfortunately, different models of computation, such as Turing machines, RAM machines, and various parallel models, are found to differ by polynomial factors in the speeds at which they can solve particular classes of problems. The algorithms that seem fastest in one model may not turn out to be fastest in another. One reason for this is that issues of the physical movement and routing of information are often considered part of the architecture, rather than part of the algorithm. Thus, as computing technology continues to develop, we occasionally find ourselves in a situation where the old models have become inapplicable, and previous work on finding the run-times of problems or on developing efficient algorithms must often be redone.

Ultimate models of computation. In our work, we attempt to leap ahead, and get away from the model-relativism of traditional complexity theory, by proposing a new theory based on "ultimate" models of computation, in which we forecast a "best possible" computing technology that takes full advantage of the computational power of the known laws of physics, while also recognizing the limitations imposed by these laws, including the three-dimensionality of space, the finiteness of the speed of light, and the second law of thermodynamics. In this sort of model, the way information and entropy are routed become an explicit part of the algorithm.

A tight Church's thesis. For our ultimate models, we conjecture what we call a "Tight Church's Thesis," which claims that any model of computation that predicts a smaller order of growth for the time to solve any class of problem than is predicted for our model is not actually physically realistic, in the sense that its growth predictions will break down in any real implementation as the problem size is made larger. It is conjectured that the only fundamental physical effect limiting the accuracy of our model will be gravity; *i.e.*, our model's predictions will break down only when the size of the computer described by our model becomes so large that even if it were built in space, it would collapse under its own self-gravity. Thus, even if not truly "ultimate," our class of model is, at least, expected to remain accurate for all computers that are built for a very long time. As we already stated in §2.3, in this thesis we will not attempt to explore the high-gravity regime.

5.1 What is a model of computation?

A model of computation is simply an abstract formal framework within which we can describe a computation process.

Role of infinities. Many models invoke infinities, such as the infinitely-long tape in a Turing machine, or an infinite cellular automaton array. However, in the more realistic models, any such infinities will be mere mathematical conveniences, and do not cause the model to be infinitely more computationally powerful than anything we might achieve in the actual universe, which may be finite.

Finiteness of the universe. Speaking more precisely, the best guess of modern cosmology is that the total volume and quantity of mass-energy in the part of the universe that is causally connected to us is finite. Therefore, with reference to the fundamental limits discussed in section 2.2, the maximum entropy or informational complexity of the universe (or in computer science terms, its maximum storage capacity) is finite as well. However, the universe is expanding, and according to some of the latest studies may very well continue to expand forever, in which case its entropy may indeed, over time, increase without bound. But at any particular time, everything is finite: propagation speeds, processing rates, information capacities, *etc.*, as we reviewed throughout chapter 2. In realistic models of computation, such quantities should be finite as well.

Turing machines are realistic. The Turing machine is a good example of a realistic model, because the only infinity in the model, namely the infinite tape, is a mathematical convenience that is inessential to the model. It can be replaced by a tape that is finite at any moment, but that can grow unboundedly large, as needed, given unbounded amounts of time. Similarly, the finite controller for the tape-head is allowed to be unboundedly large, but is fixed for any particular machine.

The above property of Turing machines, which we might call *unbounded space*, might even conceivably be realistic, in the very long term, if the universe is ever-growing and its maximum entropy increases without bound. In the near term, there will certainly be many much tighter real-world constraints on storage capacities. But whether unbounded space is realistic or not, it is a useful property because it frees our model from having to worry about any specific space bound, which would be highly situation-dependent anyway, and focus instead on other issues.

Note also that unbounded space does not imply infinite complexity in the initial configuration; indeed such complexity would make the model nonphysical.

Families of fixed-space machines. As we pointed out, Turing machine models usually provide unbounded space through either an infinite blank tape, or a tape that may be indefinitely extended as needed during the course of a computation. Another way to permit unbounded space within finite models is to specify a particular fixed, finite amount of space as part of the initial condition for any instance of the model, but define "the model" to encompass an infinite family of instances having arbitrarily large fixed amounts of space. This conception corresponds nicely to what we do when we build a particular computer; its information capacity is finite, and the machine cannot of its own accord increase its capacity; nevertheless for any given finite computation, a machine can be built that has sufficient capacity to perform it.

In this work we will often describe models of computation in this way, in terms of a family of machines, each instance of which has fixed capacity, but where for any desired capacity, some machine in the family has that capacity. Moreover, the instances must all be Turing-computable; we are not allowed to hide the values of an uncomputable function in the family of instances, or at least, such models would not be considered realistic.

It is important to realize that models of computation that consist of computable families of fixed-capacity machines are equivalent in power to Turing machines with extensible tapes, on any problem class that has computable space bounds, even though all particular machine instances in our model family are finite-state machines. This is because we are allowed to pick larger machine instances as needed, as problem sizes become larger.

If no bound can be placed on the space requirements for a given problem class, we can handle this by extending our model with a protocol whereby a particular fixed-capacity machine is simply rebuilt with a larger capacity if it runs out of space during a particular problem instance.

Of course, if we include the human builders in the concept of the system, or if we create machines that construct more of themselves automatically out of raw materials, expanding along with the universe, then even a particular instance of a computing system could, in reality, have unbounded capacity just like a theoretical Turing machine with extensible tape.

We have broadly outlined what we consider to be a legal candidate for a realistic model of computation, in terms of what quantities may or may not be infinite. Now we survey more closely the range of existing models.

5.2 Existing models of computation

Computer science has historically used many different abstract models of what constitutes a computer. One of the earliest computing models explored by mathematicians was the concept of a *recursive function* (*cf.* the text [141]). Around the same time, Turing proposed his tape machine model. Shortly thereafter, von Neumann explored the power of cellular automata (*cf.* [182]). Over the decades, many other models of computation have popped up, from register machines to pointer machines and RAM machines to parallel PRAMs, hypercubes, butterfly networks, meshes, *etc.* The models mentioned so far seem at first glance to be reasonable models of real computers. Typically the models assume the availability of some unbounded resource (such as the infinite blank tape in the Turing machine, or the infinite grid in a cellular automaton), but as we discussed above, generally the models' complexity predictions do not actually require constructing impossible machines having an infinity of resources, but rather just a reasonable ability to construct more of the resource when needed.

All of the models mentioned above have the property that they have been shown to be equivalent, in the sense of obeying the "Quantitative Church's Thesis" (or Strong Church's Thesis) [176, 175], which we mentioned in chapter 4 (p. 91), which claims that all "reasonable" models of computation are equivalent in power to the Turing machine, within a polynomial factor. That is, the running time to solve a problem class on any of these models is at worst a polynomial function of the running time needed to solve the problem class on any of the others.

Additionally, models have been proposed that seem patently unreasonable, such as nondeterministic Turing machines ([133] §2.7, p. 45), alternating machines (which quickly solve problems in the cumulative polynomial hierarchy, [133] §17.2, p. 425), machines that can query infinitely complex or uncomputable "oracles" ([133] §14.3, p. 339), infinite-precision analog models that can perform infinite amounts of computation in finite time [151], and so forth. Some of these models, such as nondeterministic Turing machines, have not actually been proven to be impossible to realize in polynomial time on a normal Turing machine, but most researchers would still consider them unreasonable models, and would be very surprised if they were to turn out to be equivalent to Turing machines within a polynomial factor. Therefore we will not deal with such models henceforth in this work.

The strong Church's thesis is useful because it permits researchers studying the computational complexity of problems and algorithms to derive their results in whichever model they prefer, with the assurance that any real computer will be able to utilize their favored algorithm with at worst a polynomial factor slowdown.

Such polynomial factor slowdowns are perfectly ignorable when one is just studying the theoretical relationships between broad complexity classes (such as **P** and **EXPTIME**), but in the real world, differences by polynomial factors do matter, and when picking an algorithm to solve a real-world problem, it is important to consider how much time the algorithm will take on a real computer, not just on some abstract model that may be polynomially slower—or faster—than the real computer for the given problem. If we do resort to using an abstract model, then one might argue that it should at least be one that accurately models the true asymptotic difficulty of solving problems in a computer architecture we can actually build.

What's more, as our skill in computer technology increases, architectures change, and so the appropriate abstract models change as well.

For example, for some very early computers having magnetic tape storage and extremely little memory, a simple Turing machine may once have been an appropriate model. But very quickly, the memory of computers increased to the point where the finite state machine in a TM tape head was no longer a reasonable model of how the state of the CPU's memory was updated—a lookup table for all possible state transitions would be far too large. Real computers calculated their next state based only on the contents of a few randomly-accessed memory locations pointed to by a small set of registers. This architecture inspired the register machine, pointer machine, and RAM machine models of computation. The state memory previously modeled as "finite" was now large enough to represent most inputs, and so now memory (or disk) was considered the new kind of practically-unlimited storage, and the sequential-access tape was dropped from the model.

The next major architectural development was the creation of parallel machines. The Turing and RAM machines suffered from the drawback that they could only operate on small amounts of data at a time, one piece after the other, rather than being able to simultaneously perform many identical (or perhaps different) operations on the data stored at many different places in storage. New architectures had many CPUs that operated in parallel. Thus, to analyze appropriate algorithms for these machines, parallel models of computation were born. These appeared in great variety: PRAMs, log-depth boolean circuits, hypercube networks, butterfly networks, meshes; each appropriate for some particular sort of parallel architecture.

Next I will argue that a number of properties of many of the models supposedly representing today's computer architectures are actually physically unrealistic and misleading, in the sense that a computer built according to the given architecture or abstract model would, in actuality, be physically unable to be scaled up arbitrarily while still achieving the same performance that would have been predicted from the abstract model. At various points, the models will break down, because they fail to take into account one or several of the fundamental realities of physics.

As we have said before, the ultimate goal of this work is to develop a model of computation that does not ignore any fundamental physical principles, and yet utilizes the full computational power afforded by physics (within a constant factor). The advantage of having such a model is that when we analyze an algorithm for it, we will be assured that the analysis will still hold true no matter how much we scale up the size of the problem or the computer. Also, since the model is designed to take full advantage of known physics, we will know that whenever we prove that a given problem requires a certain *minimum* order of growth in the time to solve it, no future advances in computer technology—barring some newly discovered fundamental laws of physics—can ever nullify the validity of that result.

There is of course always the possibility that our civilization's technological advancement will plateau for economic or other reasons, before we reach the level where all of the physical effects that we address come into play, in which case the models developed here might never actually be appropriate for application to real computers. However, it is our optimistic assumption that this will not be the case.

5.3 Problems with the existing models

This section lists ways in which existing computational models fail to recognize fundamental scaling limits imposed by physics.

Unlimited amounts of unit-access-time RAM. Models of computation such as the RAM machine typically assume that there is some unlimited amount of storage space, and moreover that any individual piece of it is accessible within constant time. This is physically unrealistic, because presumably, in any particular technology, there will always be a limit on the number of bits that can be encoded per unit of spatial volume, and the speed of light sets a lower bound on the access time to bits located in the more distant volume elements.

Super-cubic connectivity. Some types of graph-based models of computation prevent you from accessing arbitrarily many storage locations in constant time, by only allowing you to step from one storage node to the next in a graph. But if the graph is a binary tree, say, then exponentially many nodes may still be reached in a given time. This is also unrealistic, because only $\mathcal{O}(d^3)$ storage locations may be physically located within a distance d in three-dimensional space, so that even if the information travels at the speed of light, only $\mathcal{O}(t^3)$ different locations can be accessible within a given time t.

Similarly, many parallel computing architectures such as hypercubes have more than cubic connectivity between their processing nodes, so that as one scales up the number of processors, the wires between them take up more and more space and get longer and longer, until most of the computer consists of wires between processors, and delays scale up in ways not accounted for by the model. If the model accounts for the delays it won't be unrealistic, but still, it cannot be presumed to be making best possible use of the space available. Three-dimensional mesh architectures and cellular automata do not suffer from this difficulty.

Free irreversible operations. Next, nearly all of the existing models allow processing elements to perform irreversible operations (*i.e.*, transitions into states that have more than one predecessor) without considering the impact of the necessary resulting energy dissipation that is implied by thermodynamics. As we saw in ch. 2, microscopic dynamics is reversible; therefore the information that is thrown away when performing an irreversible operation must be somehow exported from the system.

As we will demonstrate in more detail in ch. 6, this exporting of information will necessarily constrain the scaling of irreversible computations, since as the radius of the computer increases by a factor of n, the number of processors increases by $\mathcal{O}(n^3)$ whereas the surface area only increases by $\mathcal{O}(n^2)$, so that the rate of production of unwanted information will eventually overwhelm our ability to remove it.

A realistic computing model should instead be reversible, so that the means of disposal of unwanted information must be explicitly accounted for in the algorithm.

We should note that some irreversible models such as Turing machines or 2-D cellular automata have sub-cubic connectivity, and are therefore physically realistic because the unused dimension(s) can be used to deliver free energy and dispose of waste heat. However, models that don't use 3 dimensions can't be assumed to be taking full advantage of physics.

Free reversible operations. Even reversible models of computation sometimes assume that reversible operations require zero energy expenditure. This is true, but only in the limit of taking infinite time to perform the operation. As we will see later, it seems that any particular computing technology will require dissipating energy per operation that scales proportionally to speed, to make up for frictional/resistive losses during the operation. However it is not clear that there is any fundamental limit to how low the friction-related scaling factor may be. (An alternative approach that allows exactly zero energy per operation is to run quadratically more slowly, by depending on a random-walk through the computer's state space to perform the computation. [17])

The fact that even reversible operations require some energy to progress forwards with a fixed lower-bound on the time per step means that even a *reversible* computer cannot be scaled arbitrarily without slowing its clock rate, because eventually there will not be enough surface area to feed in enough energy (and pump out enough heat) to keep the whole volume running at the desired speed despite the frictional or resistive losses. Chapter 6 will address this issue in detail.

Note however, that with reversible operations, the energy per operation even at high frequencies might eventually be made much lower than $k_{\mathrm{B}}T \ln 2$, the minimum energy for irreversible bit-erase operations, so correspondingly, the machine can be made much larger than if it were irreversible before slowing its clock rate, by some constant factor that increases as technological improvements decrease the magnitude of frictional losses compared to the fundamental irreducible energy cost of destroying information.

Error-free operation. Additionally, most traditional models of computation assume that computational operation is error-free, or that the error rate can be made as small as desired with no impact on the speed of a system. A better model would be to have a fixed probability of error per primitive operation, with the magnitude of the probability depending on the particular technology being used. Error-correction techniques in robust architectures can be used to convert a small but fixed probability of error per operation into an arbitrarily small probability of error for the whole computation. However, we will not deal with error models and error correction techniques extensively in our research. We presume that given good engineering, the combination of a low base rate of errors with efficient error-correction techniques would be effective enough that there would not be a large impact on our overall asymptotic scaling results.

Non-quantum operation. Traditional computational models take the classical-physics viewpoint that the computer is in a definite classical state at any given time. However, as we saw in ch. 4, recent research in quantum computing seems to indicate that computers that use quantum superpositions of states and take advantage of interference effects between them may be asymptotically faster than classical computers on some problems. Thus, non-quantum computational models may fail to take advantage of the full computational power offered by physics.

However, most people find quantum computers and algorithms much more difficult to reason about than classical ones, and furthermore, quantum computers have not yet been conclusively proven to be more powerful than classical computers. Therefore in this work we will focus on non-quantum computational models, rather than quantum ones. Either the two models are really equivalent in power, in which case the classical model should be used because of its simplicity, or else the quantum model is more powerful and should instead be considered the "ultimate" model of computation. But, quantum features can be added to our models without changing most of our overall conclusions.

Digitality. Most models of computation also are digital, and do not take advantage of the seemingly continuous range of values that some physical quantities may take. We will not either, other than allowing continuous amplitudes in our quantum computers, because it is not clear either that physics really is continuous (rather than discrete at some very fine scale), or that its continuousness, even if real, confers any additional computational power. Some research has shown that systems of point bodies obeying Newtonian mechanics can perform infinite amounts of computation in finite time if their initial positions and velocities are set with infinite precision [151]. However, our universe is not Newtonian, and anyway infinite precision in initial configurations is not a reasonable assumption. For some discussions of the power of analog computation models, see [177, 151].

Unlimited computer size. Finally, many parallel computing models presume that a parallel machine may have an arbitrarily large number of processing elements. However, this may not be realistic, since the total mass and maximum entropy of the accessible universe may be bounded. Also, given a fixed processor density, gravitational effects will come into play with a large enough number of processors. There are certainly many other technological and economic limits to computer size that apply at even smaller scales. Nevertheless, it is convenient for modeling purposes to ignore all these facts, since the precise maximum number of processors that can be attained does not qualitatively affect the form that the best algorithms will take at scales that are well below the maximum.

5.4 Some candidates for an ultimate model

We now outline what we believe are some plausible candidates for ultimate physically-based models of computation. Our basic model is a reversible three-dimensional mesh of processing elements. We present three variations of the basic model, which vary in their power. The reason we must do this is that the latter two variations, which are more powerful, depend on the future development of hypothetical technologies that, although perhaps plausible in principle, may not be physically achievable in practice at a useful scale. However, if these technologies do turn out to be feasible, then one of them is actually the ultimate model, and our basic model is not. However, all three models at least share the basic property of reversibility which is the focus of this thesis.

Since we are unable to say with certainty which of these models is the right one, we have not yet attained the ultimate model of computation. However, we feel that with these models we have narrowed the scope of possible models, and have taken a significant step in the right direction.

5.4.1 The reversible 3-D mesh (R3M) model

Our basic, most feasible proposed model is something we will call the reversible 3-D mesh, or R3M. As presently conceived, the R3M model describes a family of fixed-capacity machines. Each machine consists of a uniform three-dimensional array of processing elements, each of which has the capability of fully logically reversible and arbitrarily thermodynamically reversible operation. (Irreversible operations can also be permitted, but these should be entirely optional, and under program control.) Each processor is connected locally to (say) its 6 nearest neighbors. The following parameters of the model are considered to be fixed by a given implementation technology, and are not permitted to vary among the instances in a given family of machines:

- Hardware functionality of each processing element (PE).
- Finite storage capacity of each PE.
- Finite information rate (bandwidth) of each connection between neighboring PEs.
- Finite maximum frequency (minimum cycle time) of each PE.
- Non-zero minimum spacing between neighboring PEs.
- Finite speed at which signals travel between neighboring PEs.

The following parameters are permitted to vary among different instances of a given family, but are fixed for any given instance.

- Number of processing elements in each dimension across the PE grid.
- Actual spacing between neighboring PEs.
- Actual clock frequency at which the PEs are operated.

In addition to the information-processing architecture itself, the R3M model also explicitly accounts for the generation and transport of entropy within the computer, using the following parameters which are fixed for all machines using a given technology:

- Non-zero "entropy coefficient" of each PE. (See ch. 6.)
- Finite maximum entropy flux density within PE grid.
- Finite velocity of entropy transport.
- Entropy generated in case of (optional) information erasure.

Entropy is generated by each PE at a rate that is proportional to the square of its operating frequency, with the constant of proportionality given by the entropy coefficient. This entropy travels at a constant velocity from where it is generated, along some preferred dimension through the grid. (This represents a flow of coolant.) The entropy flux is not permitted to exceed the given maximum. At the edge of the grid, the entropy is considered to be emitted into space, and we don't worry about it further.

Processing elements at the edge of the mesh are permitted to irreversibly convert information to entropy, as frequently as desired, and emit it outwards as well. For processors internal to the mesh, any entropy generation due to information erasure must be explicitly accounted for as contributing to the entropy flow through the machine. Internal processors must refrain from performing more information erasure than can be accommodated given the technology's entropy flux capacity.

If an internal processor wants to discard information to the outside world, it can either send the information to the edge in digital form, and have it dissipated there, or dissipate it internally, and send it out using the coolant flow. The two approaches are asymptotically completely equivalent. The determination of which one is better depends entirely on the constant factors involved in the particular technology.

In chapter 6 we will discuss why the R3M is a realistic model, and show that it scales better physically than *any* irreversible model of computation. Thus it is a a good candidate for the ultimate model, unless there is some way to do better. Let us briefly mention two possible ways we might do better.

5.4.2 The ballistic 3-D mesh (B3M) model

This is just like the R3M except that the entropy coefficient is allowed to be exactly zero. In this model, reversible computation is completely dissipationless and the clock frequency of the PEs can be the same for all instances of a given family of machines, independent of the mesh size. As we will see in 6, a B3M is asymptotically strictly more powerful than any R3M model under various cost measures. Unfortunately it is probably not physically possible to achieve exactly zero dissipation at a fixed speed in any real system. However, it might be possible to approach zero dissipation so closely that dissipation doesn't become a concern at any realistic scale.

5.4.3 The quantum 3-D mesh (Q3M) model

This is just like the R3M, except that the machine can be in a global quantum superposition of states, and the operation of each PE consists of a local unitary transformation on this superposition. A Q3M can simulate other quantum computation models in polynomial time, and thus can factor in polynomial time. It thus may be exponentially faster than the R3M or B3M on some problems.

Feasibility. The use of quantum error correction codes [28, 32, 155] may conceivably permit a Q3M to operate dissipatively while still maintaining a coherent superposition, so a Q3M may be possible even if a B3M is not. However, we are still very far from any practical realization, so in any case the Q3M is a very tentative proposal.

5.5 A "tight" Church's thesis

We are now in a position to state an asymptotically *tight* version of Church's thesis, and conjecture that it holds for one of our proposed models.

Conjecture 5.1. (*Tight Church's Thesis.*) It is conjectured that for at least one of the three models of computation described above (R3M, B3M, Q3M), the following two properties hold:

(a) Implementability. For some suitable choice of constant bounds on all the $\mathcal{O}(1)$ parameters in the model, it is possible to actually implement the models in such a way that, given access to sufficient quantities of matter and organized energy (work, not heat), and sufficient construction time, a civilization could build computers that actually realize all the order of growth predictions (for run-time and space) of the model (at least, below the high-gravity regime).

(b) Optimality. For any other model of computation, if the model meets the above implementability criterion, then it implies an order of growth for all problems that is asymptotically at least as large or larger than the order of growth predicted by the model.

The upshot of the above conjecture is to claim that at least one of the types of reversible models that we are considering is physically realizable (at least the R3M is, and maybe the B3M and Q3M are also), and also that at least one of the realizable models is at least as powerful as any other realizable model of computation, on all problems. That is, *any* physically realizable computing device could be simulated in our ultimate computer with *at most* a constant factor slowdown and a constant factor increase in space usage. (Moreover, we conjecture reasonable, non-astronomical constants.)

We cannot rigorously prove our conjecture, in part because the laws of physics are not yet completely understood, but in chapter 6 we at least present a number of rigorous analyses that provide strong support in favor of it. Therefore we go ahead and introduce our conjecture to the research community, partly to see if anyone can find an argument to shoot it down.

5.6 Ultimate computational complexity

In any case, given the apparent correctness of our conjecture, theorists can now proceed to derive classical and quantum bounds on the ultimate computational complexity of various classes of problems.

Some of this work can proceed by merely embedding algorithms for earlier models into our framework. Any multi-tape Turing machine (with a constant number of tapes) and any reversible cellular automaton in up to three dimensions, and any irreversible cellular automaton in up to 2 dimensions, can be simulated with no increase in asymptotic space or run time by our R3M model, so bounds on order growth of problems derived for these machines will immediately apply to our model.

However, translating algorithms for many other models into our framework is more difficult. This is because of the unphysical assumptions made by those other models, often in their interprocessor communication networks: for example, the unit access time to an unbounded shared memory in a PRAM. Simulating these nonphysical models in our models requires an increase in asymptotic run time, but so would *any* physical implementation of those models. This is exactly why we wish to get away from those other models; performance results derived with them are misleading because they are not physically realizable.

Note this is also true of quantum computing models that assume that quantum gate operations can be applied to any selection of qubits out of an arbitrarily large number of qubits in the computer, in constant time. Instead, arbitrarily-chosen qubits will generally be distributed in space, and must be moved to be close together before they can interact. So, simulating those earlier quantum computing models in our Q3M model will introduce some slowdowns.

Note, however, that for all the "reasonable" models of computation that have previously been proposed, the slowdowns incurred by simulating them in R3M or Q3M are at worst polynomial. So although some of the earlier reasonable models may exaggerate the achievable speed of some algorithms, the exaggerations are not too bad. Still, we would like to make sure that the orders of growth of problems are expressed with the *right* polynomial.

To find good algorithms for solving problems in our R3M and Q3M models, it will often be a good strategy to start with an algorithm for some traditional model, simulate it straightforwardly in our model (possibly with increased run time if the original model was unrealistic) and then figure out how to perhaps modify the resulting algorithm to optimize its performance in our model. However, it may turn out that the best algorithm for a given problem in R3M or Q3M may operate in a totally different fashion from the best algorithm for that problem in a traditional model. In some cases, the best algorithm in our model may be slower than the best algorithm in a traditional model. But it should be remembered that the best algorithm in our model *is really the best algorithm* in an absolute sense, because the other models are not physically realistic, and the runtime growth functions derived for them are generally *not physically achievable* beyond a certain scale. Our model, on the other hand, is not only physically realistic, but it takes full advantage of what physics offers. (At least, this is our conjecture.)

5.7 Summary of discussion of ultimate models

Most existing models of computation are quantitatively either too weak, in the sense that asymptotically faster computers are physically possible, or too strong, in the sense that any physical implementation would perform asymptotically more slowly on some problems than the model predicts.

In this chapter, we proposed three models of computation that attempt to be *just right* in the sense that they take full advantage of the computational power that physics offers, but do not exceed it. We propose a "Tight Church's Thesis" that conjectures that at least one of our three models achieves this goal, at least for all sizes of machines that might be achieved in the foreseeable future.

If the Tight Church's Thesis is correct, then bounds on the computational complexity of problem classes that are derived in our models represent bounds on the true difficulty of solving those problems in our universe. Lower bounds we derive will always still apply, no matter how computer technology is improved. And upper bounds we derive will always still apply no matter how large our inputs become.

Our models include both quantum and classical models. The quantum model is thought to be more powerful, in which case it, rather than one of the classical models, is the actual *just right* model of computation that we are looking for. However, the quantum model is difficult to implement and to reason about, and it has not yet been conclusively proven to be more powerful, so one of the classical models may still be preferred. In any case, we conjecture that one of these three models correctly represents the computational capabilities of our universe (within a constant factor), and we suggest that all three models are appropriate targets for further complexity-theoretic study.

In the next chapter, we show why we believe the ultimate model *must at least* be capable of strict *reversible* (if not also ballistic and quantum-coherent) operation, by demonstrating that reversible 3-D meshes are more powerful asymptotically than *any* physical implementation of *any* irreversible model of computation.

Chapter 6

Reversibility and physical scaling laws

In this chapter we analyze how the use of reversibility can improve how well various measures of computational cost-efficiency will scale as we increase the size of our machines, or the size of the problems we are trying to solve. Our analysis establishes that *only* reversible computers are capable of realizing the maximum level of computational scalability that is afforded by the laws of physics.

Moreover, even with today's relatively primitive level of technology, substantially reversible computing can already be the most cost-effective solution in contexts where energy dissipation is a dominant concern, such as in portable devices, or large supercomputing systems. Also, we expect that as device technology improves, reversible operation will become more and more favored.

In chapter 3 we discussed how existing reversible models of computation compare with irreversible models when using a variety of non-physical measures of cost, such as are used in traditional computational complexity theory. Using those measures, we saw that reversibility did not improve efficiency, and in some models could be proven to actually degrade efficiency (§3.4), when carried to the extreme of total reversibility. But the problem with taking those results at face value is that, as we saw in the previous chapter, traditional computation models and cost measures are not realistic; they do not reflect real costs and the physical constraints on computation that we discussed in chapter 2.

In section 3.2.2, p. 54, we introduced some new cost measures which we proposed were more physically appropriate than are the quantities that are traditionally measured in computational complexity theory. In sec. 6.2 (p. 124) we will perform an analysis of physically realistic models of computation using various such physical cost measures, and show that using those models and measures, reversibility can be seen to increase overall efficiency.

Let us now introduce the general classes of models which we will analyze.

6.1 Types of architectures studied

For convenience, in this chapter we will use the term *architecture* to denote the concept of a model of a family of physically-implementable machines, such as those we proposed in ch. 5. Section 6.2 of this chapter analyzes the properties of several very general classes of architectures, which we will define below: fully irreversible architectures (FIA), time-proportionally reversible architectures (TPRA), and ballistically reversible architectures (BRA). All three of these classes will have a number of features in common.

6.1.1 Shared properties

The machine classes we study will all be imagined to be implemented in some fixed underlying technology, which we take to mean that several quantities are fixed across all three classes of machines:

1. There is a fixed minimum physical size (mass and volume) for storing a bit of computational state.
2. There is a fixed maximum physical entropy density allowable within the machines in question, including in their cooling systems.

   The above two items can be justified on the basis of the limits presented in §2.2, along with the argument that mass densities, energies, and temperatures will not be able to be increased indefinitely in any computer technology realizable in the foreseeable future.
3. There is a fixed maximum rate at which bit-operations can be performed per unit of mass and per unit of volume in the machine.

   This limit follows from the fundamental Margolus-Levitin bound we mentioned in §2.4; much tighter bounds than this will certainly hold for all technologies through the foreseeable future.

Now let us distinguish the three classes of architectures that we will study.

6.1.2 Fully irreversible architecture

A *fully irreversible architecture* FIA is one in which there is a fixed constant lower bound, independent of the machine size or of any adjustable parameters of the architecture, on the average number of bits of *computational information* that are lost (converted to entropy) per primitive computational operation that is performed. Note that this does not count the mere conversion of bits that may *already* be entropy from a controlled digital form to an uncontrolled physical form. We are concerned here only with the amount of *new* entropy that is generated per operation due to the architecture.

An architecture is fully irreversible if, for example, it routinely uses ordinary irreversible logic gates, which must produce entropy every time they erase a bit, according to Landauer's principle (§2.5).

6.1.3 Time-proportionately reversible architecture

A *time-proportionately reversible architecture* TPRA is one that provides the option to reduce the average entropy S generated per primitive operation to an arbitrarily small amount that is asymptotically proportional to the inverse of the amount of time t_{op} over which individual operations are performed; that is, $S \sim 1/t_{\mathrm{op}}$. In such architectures, the "degree of irreversibility" (entropy generated per operation) is inverse to this time, so the "degree of reversibility" can be considered *proportionate* to time. Thus we use the adjective "time-proportionately reversible," to describe these machines, the motivation being that this is much more precise than alternative adjectives such as "adiabatic," "asymptotically reversible," and "quasistatic" which have often been used in the past when referring to technologies that have this particular property. (See §7.3 for further discussion of this terminology issue.)

As we will see in chapters 7 and 8, a large number of existing and proposed logic-device technologies are capable of implementing time-proportionate reversibility; so the TPRA model is certainly realistic for purposes of an asymptotic scaling analysis. However, the constant of proportionality (which we call the "entropy coefficient") varies greatly across different technologies, so the range of validity of the asymptotic analysis depends significantly on the technology in question.

6.1.4 Ballistic reversible architecture

This next class of "architectures" may or may not actually be realistic, but it will be a useful point of comparison, which will help us interpret the results of our analysis of the TPRA. The ballistically reversible architecture BRA is a model based on an imagined technology where the entropy generated per constant-time operation can be made exactly zero, or at least so close to zero that the difference does not matter for any achievable scale of machines.

A BRA is the conceptual limit of a TPRA in which the entropy coefficient becomes arbitrarily small. It is appropriate to consider this limit because we do not yet know of

Symbol	Name of architecture class	Entropy generated per operation
FIA	Fully irreversible architecture	$\Theta(1)$
TPRA	Time-proportionally reversible architecture	$\Theta(1/t_{\text{op}})$
BRA	Ballistically reversible architecture	0

Table 6.1: The three classes of physical machine models that are compared in this chapter. The defining difference between them is in how the average entropy generated per computational operation scales in relation to the length of time t_{op} over which the operation is performed.

---

any fundamental physical restrictions on how low the entropy coefficient can actually be made to be.

Note that in both the TPRA and the BRA we specify that *fully* logically reversible operation is permitted, but not required. In these models, we also provide the option to perform logically irreversible operations which generate constant entropy. (Moreover, the type of operation to use should be selectable at run time.) This allows these models to use the external universe as a garbage-information dump, just like the FIA does; this option ensures that our reversible machines will be *at least* as powerful as the FIA, since it will be subsumed as a special case, one in which the time-proportionate reversibility feature is effectively unused.

Table 6.1 summarizes the three classes of architectures we will compare in this chapter.

A general feature of these analyses will be attention to some of the subtle ways in which several kinds of physical constraints, such as limits on entropy density and propagation speed, interact with each other to determine the form of the most cost-efficient possible machines.

The structure of the rest of this chapter will be, roughly, to proceed from the simpler, less compelling physical cost measures and analyses to more sophisticated and realistic ones.

6.2 Analyses under various physical costs

In this section we determine, for various cost measures $\$$, the *reversible advantage* $\mathcal{A}_{\text{r}}$ under the given cost measure. We define $\mathcal{A}_{\text{r}}$ as the asymptotically fastest-growing value of the cost-efficiency ratio $\%_{\$\text{rev}}/\%_{\$\text{irr}}$, as a function of cost, for any class of computational tasks. Equation (3.1), p. 54 defined cost-efficiency as

$$\%_{\$} = \frac{\$_{\min}}{\$}.$$

Therefore, letting $\$_{\mathrm{i}}$ and $\$_{\mathrm{r}}$ be the costs on an irreversible and reversible machine, respectively,

$$\begin{aligned} \mathcal{A}_{\mathrm{r}} &= \%_{\$\mathrm{rev}} / \%_{\$\mathrm{irr}} \\ &= (\$_{\min}/\$_{\mathrm{r}})/(\$_{\min}/\$_{\mathrm{i}}) \\ &= \$_{\mathrm{i}}/\$_{\mathrm{r}}, \end{aligned}$$

that is, the reversible advantage is equal to the ratio of the cost on an irreversible machine to the cost on a reversible machine. (And similarly for the ballistic advantage $\mathcal{A}_{\mathrm{b}}$.)

We will often normalize $\mathcal{A}_{\mathrm{r}}$ by expressing it as a function of $\$_{\mathrm{r}}$, the cost on the reversible machine. So, if we write $\mathcal{A}_{\mathrm{r}} \sim f(\$_{\mathrm{r}})$, this means there are classes of computations such that, for instances that cost $\$_{\mathrm{r}}$ to perform on a TPRA (reversible) machine, the cost to perform them on an FIA (irreversible) machine in general is $\boldsymbol{\Theta}(f(\$_{\mathrm{r}}))$ times larger. If $f \sim 1$ this indicates no reversible advantage; any $f \succ 1$ indicates an asymptotically unbounded reversible advantage, as the cost level increases.

It is important to keep in mind that the true reversible advantage is determined by the *best possible* efficiency of each of the two classes of machines on the problem in question. To show a reversible advantage greater than $\boldsymbol{\Theta}(1)$ (no asymptotic advantage), we have to show that *no* FIA machine can perform a given computation with less than a given asymptotic cost that *is* achievable on a TPRA.

Moreover, throughout this section we will be concerned only with *sustainable* costs; that is, an assessment of a computation's cost will only be considered to be fair if a long series of $N \gg 1$ repetitions of computations like it could be performed on the given machine class with no more than N times the cost. This will allow us to marginalize factors such as the time required for set-up of the initial state and read-out of the result. It is assumed that this is fair to do, because there are many useful computations that *are* of a form that requires numerous sequential iterations of a procedure.

Some of the analyses and results in this section were first reported in our earlier publications [70, 71].

6.2.1 Entropy cost

Perhaps the simplest physical measure of cost, which also gets us away from the bias towards the abstract time and space cost-measures featured in traditional complexity theory, is the idea of the cost of a computation being proportional to just the amount of new entropy that it generates.

This measure makes sense for several reasons:

1. Entropy takes up space, and when too much of it accumulates within a fixed-size system, it causes the system to become disordered in uncontrollable ways. For example, a computer might melt if it produces too much entropy without removing it.

2. As we saw in §2.5.3, energy is required to support the existence of entropy in any system at non-zero temperature. Therefore it costs us free energy whenever entropy is generated. As we mentioned in §2.5.4, the coolest accessible place to dump large amounts of entropy is the cosmic microwave background at ≈ 3 K, so each bit's worth of sustained entropy generation costs us at least $\approx 3{\times}10^{-23}$ J (≈ 0.2 meV) of energy which cannot be recovered. (Except maybe by waiting for the universe to cool further, which will take a while!)

3. Even in the distant-future limit, if there is a finite upper bound to the maximum entropy of the universe, then negentropy ($S_{\mathrm{max}}-S_{\mathrm{current}}$) is a truly non-renewable resource; once we use it up, no further entropy-producing operations will be possible. (There's a cost measure for you!) So the efficiency of our use of entropy is crucial if we wish to maximize the total amount of computational work that we accomplish throughout all time.

Scaling comparison. With entropy alone as the cost measure, $\$ = S$, the comparison is of course straightforward. The irreversible FIA machine by definition produces constant entropy per operation, so the cost of any computation scales as the number of primitive operations, $\$ \sim N_{\mathrm{ops}}$.

The ordinary reversible machine TPRA, given unbounded space, can be run in fully logically reversible fashion using Bennett's 1973 algorithm (with the same order N_{ops} as the FIA), and still produce no computational entropy other than, at most, the size n_{in} of the input problem, and that only if the input is no longer needed after the computation. The entropy generation due to friction can be made arbitrarily smaller than n_{in}, by extending the computation over a sufficiently long period of time. Thus the total entropic cost is at most equal to the input size, $\$ \lesssim n_{\mathrm{in}}$.

Similarly for the ballistic BRA, except that we do not have to run the machine indefinitely slowly to achieve that low of a cost.

Thus, unsurprisingly, when entropy is the cost measure, reversible machines completely dominate irreversible ones in their cost-efficiency. Since for arbitrary problem classes, N_{ops} may scale arbitrarily quickly with n_{in}, the reversible advantage factor may be an arbitrarily fast-growing function of the input size.

Of course, using entropy as the sole cost measure is not particularly compelling, because it ignores the opportunity cost of using up some amount of physical space for the amount of time required by the computation. This is particularly apparent for the case of the TPRA which may consume a large amount of space (for example, proportional to N_{ops}) for a large amount of time ($\mathbf{\Omega}(N_{\text{ops}}^2/n_{\text{in}})$ to get the physical entropy generation below $\mathcal{O}(n_{\text{in}})$). When minimizing entropy only, total spacetime resources for a computation will likely be polynomially larger for the reversible computation. Thus it behooves us to consider those costs as well.

Before we study true spacetime costs, let us first consider another measure of cost that is easier to analyze, but still takes into account measures of both run-time and machine size.

6.2.2 Area-time product

For purposes of this section, we will characterize machine size as the surface area A of the least-area surface that encloses all of the computer's active information-processing components. Note that if the "computer" happens to consist of many independent components that are spread far apart from each other over a large surface, then under our definition, the least-area surface enclosing the system may actually consist of many separate small surfaces, one around each component.

In any case, one reason to think of area as a component of a cost measure is that it measures quantities such as desktop footprint, floor space, and land (planetary surface), which have everyday significance as resources. Moreover, present computer manufacturing technology, based on building up structures on the surfaces of silicon wafers, is geared towards building dense circuitry in only two dimensions, so area is a frequent cost measure in that arena as well.

More fundamentally, due to the limits on entropy density we assumed in §6.1, we will see in a moment that minimum surface area determines the maximum sustained rate at which entropy can be produced within the surface. If a system actually does produce entropy at this rate, it thereby subtracts correspondingly from the maximum rate at which entropy can be produced by the remainder of any larger system within which it is enclosed. So, area makes sense as a component of computational cost.

Multiplying the area by time converts it to a measure of the *rent* that the area would yield over the course of the computation if it were rented out for other purposes (such as for alternative computations). This makes sense in intuitive economic terms, and it also corresponds to a bound on the total amount of entropy that the given system could have produced over the given amount of time.

6.2.2.1 Rate of computation as a function of area

For computing the area-time product, let us first ask, how does the maximum rate of computation scale as a function of area?

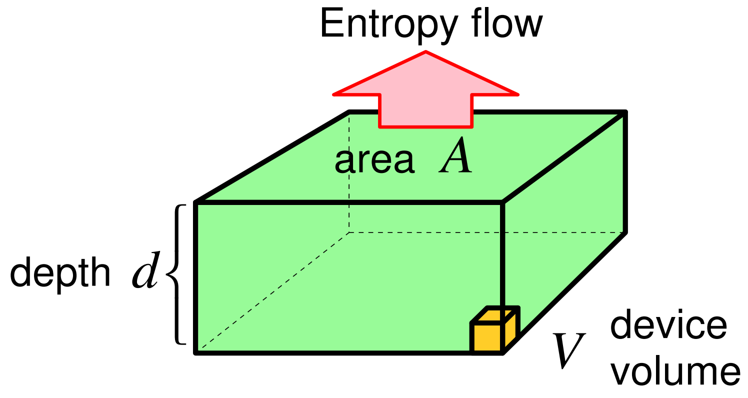
For now, we will characterize the raw processing rate $\mathcal{R}_{\rm op}$ in terms of the number of primitive computational operations (such as logic gate operations) performed per unit of real time. We also assume, for now, that the computation being performed is an inherently logically reversible one that does not require asymptotically more computational steps or memory on a reversible processor; this will allow us to treat time-proportionally reversible operations as equivalent to irreversible operations for our purposes. An example of such a computation would be a simulation of a logically reversible system; we will see other examples in ch. 9.

Irreversible machine. The FIA machine by definition produces $\boldsymbol{\Theta}(1)$ entropy per operation, and we assume as always that entropy densities are limited. As per our arguments in §2.3, the rate of entropy removal per unit area is therefore also limited. Since the total volume within the given area is limited (it's at most $\mathcal{V} \leq \frac{1}{6}\pi^{-1/2}A^{3/2}$), it follows that for a long computation, the highest rate of entropy generation that is sustainable is just equal to the maximum rate at which the entropy may leave through the surface. This rate is bounded by the fixed maximum entropy flux $F_{\rm S}$ times the minimal enclosing area A. Thus $\mathcal{R}_{\rm op} \precsim A$.

Reversible machine. Let the TPRA contain logic devices at constant average density, so that the total number of logic devices $N_{\rm dev}$ is proportional to the TPRA's volume $\mathcal{V}$, and let the TPRA also be roughly spherical (a cube would suffice) so that $\mathcal{V} \sim A^{3/2}$. Then, the number of devices $N_{\rm dev} \sim A^{3/2}$. If each device operation takes time $t_{\rm op}$, and all operations are reversible, the entropy per operation is $\boldsymbol{\Theta}(1/t_{\rm op})$ and so the total rate of entropy generation is $\mathcal{R}_{\rm S} \sim N_{\rm dev}/t_{\rm op}^2 \sim A^{3/2}/t_{\rm op}^2$. Since $\mathcal{R}_{\rm S}$ must be no greater than the rate $\mathcal{O}(A)$ of entropy removal, we have that $t_{\rm op} \succsim A^{1/4}$. Letting $t_{\rm op} \sim A^{1/4}$, we have $\mathcal{R}_{\rm op} = N_{\rm dev}/t_{\rm op} \sim A^{5/4}$.

Thus, within area A, the TPRA can run $\boldsymbol{\Theta}(\sqrt[4]{A})$ times faster than the FIA, on computations that involve only logically reversible operations. For an approximate sphere/cube of diameter $d \sim \sqrt{A}$, the speed advantage of the reversible machines scales as $\boldsymbol{\Theta}(\sqrt{d})$. Many such cubes can be arranged beside each other in a plane without changing the asymptotic area available to each one, forming a flat slab of material of thickness (depth) d, which can perform at a per-area rate of $\boldsymbol{\Theta}(\sqrt{d})$. (See figure 6.1.) An irreversible machine, in contrast, would be capable of only a constant rate per unit area.

Ballistic machine. In this case there is no entropy production, so the maximum rate of operation scales with volume, $\mathcal{R}_{\rm op} \sim A^{3/2}$. This is a factor of $\sqrt{A}$ times faster than the irreversible machine and $\sqrt[4]{A}$ times faster than the time-proportional



$$\left.\begin{array}{lll} \text{Number of devices} & N_{\text{dev}} & = Ad/V \\ \text{Entropy generation rate} & \mathcal{R}_{\text{S}} & = N_{\text{dev}} S/t_{\text{op}} \\ \text{Entropy removal rate} & \mathcal{R}_{\text{S}} & = F_{\text{S}} A \\ \text{Entropy per operation} & S & = k_{\text{S}}/t_{\text{op}} \\ \text{Total rate of operation} & \mathcal{R}_{\text{op}} & = N_{\text{dev}}/t_{\text{op}} \end{array}\right\} \mathcal{R}_{\text{op}} = A\sqrt{\frac{dF_{\text{S}}}{k_{\text{S}}V}}$$

Max rate achieved when $t_{\text{op}} = \sqrt{dk_{\text{S}}/F_{\text{S}}V}$.

Beats irreversible $\mathcal{R}_{\text{op}} = A\frac{F_{\text{S}}}{S}$ whenever $d > \frac{F_{\text{S}}k_{\text{S}}V}{S^2}$.

Figure 6.1: Speed limit for reversible machines of minimum-surface area $\mathbf{\Theta}(A)$ and thickness $d \precsim A^{1/2}$. The maximum rate of computation scales as $\mathbf{\Theta}(A\sqrt{d})$.

reversible machine. For a sphere or slab of thickness d the ballistic machine is $\Theta(d)$ times faster than the irreversible machine, and $\Theta(\sqrt{d})$ times faster than the TPRA.

So the TPRA is, in a sense, "halfway" between irreversible and ballistic machines in terms of rate per unit area; its benefit factor above the irreversible machine is the square root of that for the ballistic machine.

6.2.2.2 Minimum area-time product

Now that we know how speed scales with area, let us see how to choose the shape of a machine so as minimize the area-time product AT for a given computation requiring N_{ops} operations, under our three classes of architectures.

Let us assume we are dealing with a restricted class of computational tasks in which no communication is required between processors during the course of the computation: the task can be expressed as $\Theta(N_{\text{ops}})$ separate computational tasks that can be performed entirely independently of each other. This is approximated by, for example, a brute-force search problem in which a very large number of independent possibilities need to be checked for a solution, and checking each one takes roughly constant time, independent of the number checked. (Remember, we can amortize away the set-up and read-out times, because we are concerned with determining a sustainable rate for many iterated repetitions of the given computation.)

Irreversible. If all the area can be used effectively, $\mathcal{R}_{\text{op}} \sim A$, so the time $\mathcal{T}$ for N_{ops} operations is $\Theta(1/A)$, and so $A\mathcal{T} \sim A(1/A) = 1$. Thus the choice of the area of the machine does not affect the asymptotic area-time product. To see what the area-time product is as a function of N_{ops}, consider spreading the processors arbitrarily far apart over a 2-dimensional plane. The minimum-area surface will then consist of a collection of small surfaces, one enclosing each processor, thus the total surface area will be proportional to the number of processors ($A \sim N_{\text{proc}}$), and if we give each processor a constant-size, constant-time piece of the total problem, the number of processors N_{proc} scales as N_{ops}, and the whole computation takes constant time, and $A\mathcal{T} \sim N_{\text{ops}}$.

Reversible. Let the N_{ops} operations again be performed in parallel on $N_{\text{proc}} \sim N_{\text{ops}}$ processors, but this time in a compact structure with area $A \sim N_{\text{proc}}^{2/3}$. In §6.2.2.1 we already derived that the maximum rate of computation for this TPRA structure is $\Theta(A^{5/4})$, so the minimum time $\mathcal{T}$ for N_{ops} operations is $\Theta(N_{\text{ops}}/A^{5/4})$, or $\Theta(N_{\text{ops}}/N_{\text{ops}}^{5/6}) \sim N_{\text{ops}}^{1/6}$. Thus $A\mathcal{T} \sim N_{\text{ops}}^{5/6}$ in this configuration. Can we do better by spreading the processors out? No, because when we decrease the thickness by a factor of x, the area increases by a factor of $\Theta(x)$, but the time only scales down by $\Theta(\sqrt{x})$, so the area-time product increases by $\Theta(\sqrt{x})$. So the optimal configuration is the one we chose, where the diameter is asymptotically minimal.

Ballistic. In the ballistic machine we perform the N_{ops} operations in parallel on $N_{\mathrm{proc}} \sim N_{\mathrm{ops}}$ processing elements in constant time, and because they produce no entropy we can cram them inside the minimal surface area $A \sim N_{\mathrm{proc}}^{2/3}$ without worrying about entropy removal, and so the area-time product for the whole computation is $\boldsymbol{\Theta}(N_{\mathrm{ops}}^{2/3})$.

Thus for these inherently reversible, completely parallelizable computations, composed of $\boldsymbol{\Theta}(N_{\mathrm{ops}})$ independent constant-time sub-computations, the TPRA reversible model provides an area-time cost-efficiency advantage of $\sqrt[6]{N_{\mathrm{ops}}}$, again the square root of the benefit of $\sqrt[3]{N_{\mathrm{ops}}}$ that would be provided by a perfectly reversible ballistic computer.

In terms of the cost on the reversible machine, the reversible advantage grows as $\$_{\mathrm{r}}^{1/5}$ for this type of problem. This is the highest scaling possible for this cost measure, because for both FIA and TPRA models, the optimal solution for this problem could be achieved using the same structure: a compact, maximally-parallelized structure. This is already the structure that favors reversible operation the most, since structures that are smaller or more spread out will be less limited by entropy removal; and computations that are less parallelizable will require smaller machines for a given N_{ops} to minimize the area-time product.

Thus, we need not consider other types of computational tasks; we have established that the best reversible advantage $\mathcal{A}_{\mathrm{r}}$ for the area-time cost measure is exactly $\boldsymbol{\Theta}(\$_{\mathrm{r}}^{1/5})$. This area-time advantage does not grow as quickly as the reversible entropy advantage of §6.2.1 did, but it still becomes unboundedly large as we compare machines at larger and larger cost levels.

6.2.3 Time cost

We have seen how, given a measure of cost consisting of area times time, reversibility yields a scaling advantage. But what if we don't agree that there should be a surface area factor in the cost? Can reversibility provide any benefits for optimizing run-time, by itself?

For the sort of problem considered in the previous section, in which no communication is required between parts of the computation (during the computation itself), it is clear that reversibility provides no asymptotic speed benefit. To minimize the run-time, the processors performing the independent pieces of the computation can simply be spread far enough apart so that the minimal enclosing area becomes proportional to the number of processors, and then entropy removal no longer constrains the asymptotic minimum time, even in the fully irreversible case. The run-time in all models is then $\boldsymbol{\Theta}(1)$, the time for each individual processor to complete its piece of the computation. (Again, we amortize away set-up time by assuming that many sequential iterations of this computation are required.)

Therefore, in order to show a reversible advantage for time-efficiency, we must consider a different class of sustainable computations, namely one that requires frequent communication between processing elements. This will imply that processing elements cannot be spread arbitrarily far apart without adversely affecting the time for the computation (due to the lightspeed limit). The requirement for a relatively compact structure will then lead to a tradeoff between entropy generation and speed which, as we will show, will favor the reversible machines.

Fortunately, many real computations of interest are indeed of the sort that requires frequent communication. Our canonical example will be the simulation of physical systems; in particular, reversible 3-dimensional lattice simulations (*cf.* [119, 164, 117]; [117] contains many more references). In such computations, each update of a computational cell depends on the results of the updates of its nearest neighbors from the previous time step.

6.2.3.1 Time for 3-D local array simulations

Irreversible time. There is a simple proof of a lower bound on the average time per step for performing 3-D local array computations on an FIA. Consider the problem of simulating a locally-connected $N_{\mathrm{D}} \times N_{\mathrm{D}} \times N_{\mathrm{D}}$ array of cells for a number of steps $N_{\mathrm{st}} \gg N_{\mathrm{D}}$. Consider a segment of this computation consisting of a series of $\mathbf{\Theta}(N_{\mathrm{D}})$ consecutive steps. An element's value at the end of this segment will in general depend on the values (at the start of the segment) of all the elements less than $\mathbf{\Theta}(N_{\mathrm{D}})$ positions away from it, that is, $\mathbf{\Theta}(N_{\mathrm{D}}^3)$ different elements, and on the results of $\mathbf{\Theta}(N_{\mathrm{D}})$ updates of those elements, for a total of $\mathbf{\Theta}(N_{\mathrm{D}}^4)$ operations involved in determining the final value.

If the series of steps is performed within a time $\mathcal{T}$, then all those $\mathbf{\Theta}(N_{\mathrm{D}}^4)$ operations must occur within a sphere of radius $R = c\mathcal{T} \sim \mathcal{T}$ of the final result, in order to possibly affect the final result, given that information propagates no faster than light. This sphere is contained within a surface of area $A \sim \mathcal{T}^2$. By the arguments in §6.2.2.1, the maximum rate $\mathcal{R}_{\mathrm{i}}$ of fully irreversible computation that can be sustained within this region is then bounded by $\mathcal{O}(A) \sim \mathcal{T}^2$. (We care about the sustainable rate because the block of N_{D} steps in question is performed in series with $N_{\mathrm{st}}/N_{\mathrm{D}} \gg 1$ other similar segments operating over the same cells.)

Running at the rate $\mathcal{R}_{\mathrm{i}} \precsim \mathcal{T}^2$ for time $\mathcal{T}$ means that only $N_{\mathrm{ops}} \precsim \mathcal{T}^3$ total operations affecting the result can be performed within that time. For this N_{ops} to be equal to the needed $\mathbf{\Theta}(N_{\mathrm{D}}^4)$, $\mathcal{T}$ must then be $\mathbf{\Omega}(N_{\mathrm{D}}^{4/3})$. If it takes $\mathbf{\Omega}(N_{\mathrm{D}}^{4/3})$ time to perform $\mathbf{\Theta}(N_{\mathrm{D}})$ steps, then the average time per step is $t_{\mathrm{op}} \succsim N_{\mathrm{D}}^{1/3}$.

If we assume that some means is available for ballistic constant-speed communication between neighboring processors over arbitrary distances, then this bound can actually be achieved, using, for example, an array of $N_{\mathrm{D}} \times N_{\mathrm{D}} \times N_{\mathrm{D}}$ processing elements spaced a distance of $\mathbf{\Theta}(N_\mathrm{D}^{1/3})$ apart from their neighbors, each updating its cell once every $t_\mathrm{op} \sim N_\mathrm{D}^{1/3}$ time units, and spending the $\mathbf{\Theta}(N_\mathrm{D}^{1/3})$ time before its next update exchanging results with its neighbors. See figure 6.2.

Each processor produces $S = \mathbf{\Theta}(1)$ entropy per step, so a single column of N_D processors produces entropy at the rate $S/t_\mathrm{op} \sim N_\mathrm{D}^{2/3}$. Fortunately, the cross-sectional area of the column is $\mathbf{\Theta}(N_\mathrm{D}^{1/3}) \times \mathbf{\Theta}(N_\mathrm{D}^{1/3}) \sim N_\mathrm{D}^{2/3}$ and so the flow of entropy can move along the column with no more than constant flux. And if it can be moved ballistically, no additional entropy is generated by this flow.

Ballistic inter-processor communication and ballistic entropy transport seem to be reasonable assumptions because they are very closely approximated by, for example, propagation of photons or information-carrying matter through vacuum, and by propagation of electrons through superconductors.

Ballistic computation, in contrast, may well be more difficult because the need for frequent interactions between information-carrying components may sap energy or introduce exponentially-increasing error; these issues would need to be addressed to build a substantially ballistic computational system. But for purposes of communication only, no interactions need occur during flight, and so those particular problems do not arise.

In any case, it seems a reasonable approximation to conclude that a time per step of $\mathbf{\Theta}(N_\mathrm{D}^{1/3})$ for simulation of diameter-N_D 3-d arrays can actually be achieved on fully irreversible machines. Can we beat this when running in a time-proportionate reversible fashion?

Reversible time. The answer is yes. Consider a TPRA implementation using a similar $N_\mathrm{D} \times N_\mathrm{D} \times N_\mathrm{D}$ array. This time, spread the processors only $\ell = \mathbf{\Theta}(N_\mathrm{D}^{1/4})$ distance apart from their neighbors, and let them take $t_\mathrm{op} \sim \ell$ time for each update computation. (See fig. 6.3.) Then the entropy generated per update is $S \sim 1/t_\mathrm{op} \sim N_\mathrm{D}^{-1/4}$, and the rate of entropy generation per processor is $\mathcal{R}_\mathrm{S} = S/t_\mathrm{op} \sim 1/t_\mathrm{op}^2 \sim (N_\mathrm{D}^{-1/4})^2 = N_\mathrm{D}^{-1/2}$. Thus the rate of entropy generation for a column of N_D processors is $\mathbf{\Theta}(N_\mathrm{D} \cdot N_\mathrm{D}^{-1/2}) \sim N_\mathrm{D}^{1/2}$. The cross-sectional area of the column is $\mathbf{\Theta}(N_\mathrm{D}^{1/4}) \times \mathbf{\Theta}(N_\mathrm{D}^{1/4}) \sim N_\mathrm{D}^{1/2}$, so this rate of entropy generation is sustainable, and the time per step of $\mathbf{\Theta}(N_\mathrm{D}^{1/4})$ is not prevented by entropy removal.

We can show that this asymptotic time of $N_\mathrm{D}^{1/4}$ is minimal for a TPRA, just as $N_\mathrm{D}^{1/3}$ was minimal in the irreversible case. Suppose the average time per step in a sustained TPRA implementation is t_op. The average entropy generated per op is then $S \succsim 1/t_\mathrm{op}$. Performing the $\mathbf{\Theta}(N_\mathrm{D}^4)$ operations that affect a cell during an N_D-step computation then generates $\mathbf{\Omega}(N_\mathrm{D}^4/t_\mathrm{op})$ entropy, and since the N_D steps take exactly time $\mathcal{T} = t_\mathrm{op} N_\mathrm{D}$, the average rate of entropy generation is $\mathbf{\Omega}(N_\mathrm{D}^3/t_\mathrm{op}^2)$. Suppose $t_\mathrm{op} \prec N_\mathrm{D}^{1/4}$: then the rate $\mathcal{R}_\mathrm{S}$ of entropy generation would be $\mathbf{\Omega}(N_\mathrm{D}^3/\mathbf{o}(N_\mathrm{D}^{1/4})^2) \succ$

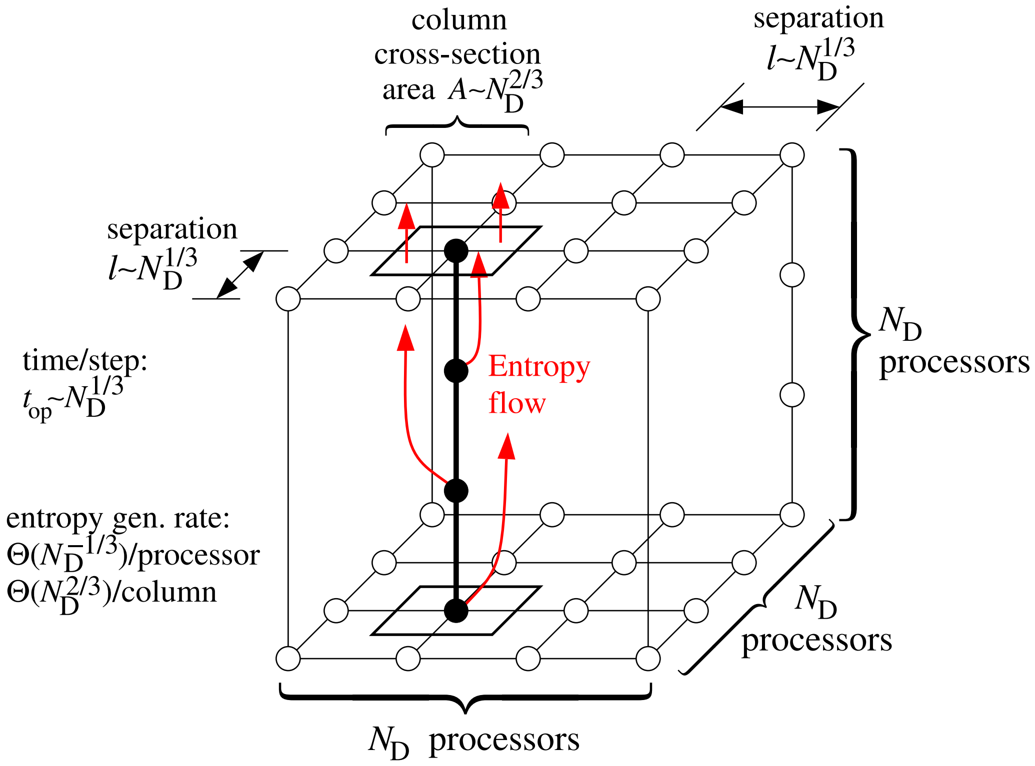


Figure 6.2: A machine configuration that achieves the asymptotically optimal FIA performance of $\mathbf{\Theta}(N_D^{1/3})$ time per step on 3-D local cell-array simulations. The top and bottom layers of a locally-connected $N_D \times N_D \times N_D$ mesh of processors are shown, and a single column of processors through the machine is emphasized in black. Spacing the processors $\mathbf{\Theta}(N_D^{1/3})$ apart gives enough room for the entropy produced by the column to be removed with no more than the maximum achievable flux $F_S = \mathbf{\Theta}(1)$, while still allowing neighbors to communicate with each other within $\mathbf{\Theta}(1)$ steps. Closer spacing would increase the time for entropy removal; sparser spacing would increase the communication time.

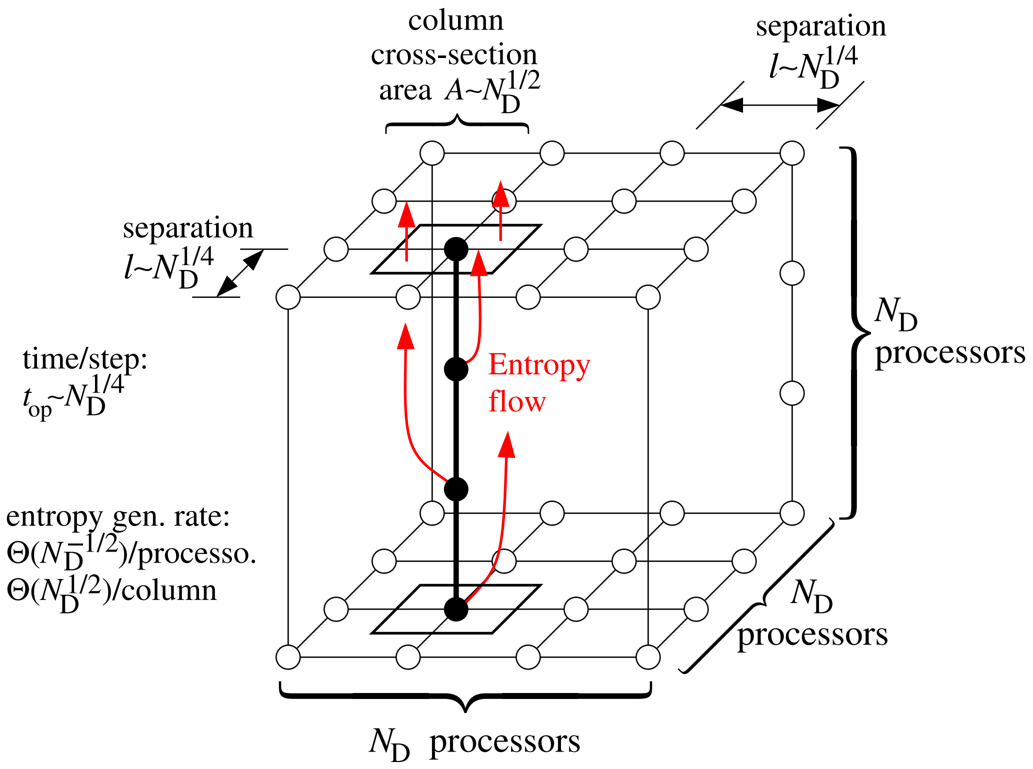


Figure 6.3: A TPRA configuration that is asymptotically strictly faster than the fastest FIA (fig. 6.2) for 3-D simulations of reversible locally-connected cell arrays. The speedup is possible because the lower TPRA entropy per operation, $\Theta(1/t_{\mathrm{op}})$, permits the processors to be packed closer together, and run at a correspondingly faster rate, without the fixed maximum entropy flux F_{S} being exceeded. An inter-neighbor spacing and time per step of $\Theta(N_{\mathrm{D}}^{1/4})$ is optimal among TPRA structures. In contrast, an idealized, perfectly ballistic machine, generating no entropy, could achieve $\Theta(1)$ time per step.

$N_{\mathrm{D}}^3 \cdot N_{\mathrm{D}}^{-1/2} \sim N_{\mathrm{D}}^{5/2}$, that is, $\mathcal{R}_{\mathrm{S}} \succ N_{\mathrm{D}}^{5/2}$. But if $t_{\mathrm{op}} \prec N_{\mathrm{D}}^{1/4}$, then the total time $t_{\mathrm{op}} N_{\mathrm{D}} \prec N_{\mathrm{D}}^{5/4}$, and so the operations must be performed within a sphere of radius $R \prec N_{\mathrm{D}}^{5/4}$, which has area $A \sim R^2 \prec N_{\mathrm{D}}^{5/2}$. Supporting a sustained rate of entropy generation of $\mathcal{R}_{\mathrm{S}} \succ N_{\mathrm{D}}^{5/2}$ within an area $A \prec N_{\mathrm{D}}^{5/2}$ would require an average entropy flux $F_{\mathrm{S}} = \mathcal{R}_{\mathrm{S}}/A \succ 1$, which violates our basic technological assumption of a fixed upper bound on entropy flux. Therefore an average time per step $t_{\mathrm{op}} \prec N_{\mathrm{D}}^{1/4}$ on this problem is actually not possible for a TPRA.

Therefore, for this class of computations, relevant to simulation of physical systems, a time-proportional reversible machine is faster than a fully irreversible machine by a factor of exactly $\Theta(N_{\mathrm{D}}^{1/3})/\Theta(N_{\mathrm{D}}^{1/4}) \sim N_{\mathrm{D}}^{1/12}$. In terms of $\$_{\mathrm{r}} \sim \mathcal{T}$, the reversible advantage is $\mathcal{A}_{\mathrm{r}} \sim \$_{\mathrm{r}}^{1/3}$.

Ballistic time. This situation is trivial. The ballistic machine produces no entropy to remove, so N_{D}^3 processing elements can just be packed together with minimal separation no matter what the value of N_{D}, and so the communication time and the time per step can be made constant, independent of N_{D}.

6.2.3.2 Time cost with non-local communication

In section 6.2.3.1 we saw that on 3-D array simulations with local communication, reversible machines were faster than irreversible machines by a factor of $\Theta(N_{\mathrm{D}}^{1/12})$ where N_{D} was the number of elements across the array in each dimension. Are there other kinds of problems where the reversible advantage is greater as a function of N_{D}? What problems have the highest asymptotic reversible advantage as a function of N_{D}?

One idea to try to improve the reversible advantage is to pose a problem that requires non-local communication between cells, to try to force the machines to be more compact, giving the reversible machine more of an advantage. For an array of cells of diameter $\Theta(N_{\mathrm{D}})$, obviously the farthest we can require a signal to travel before being processed is $\Theta(N_{\mathrm{D}})$ inter-cell distances. However, if we make this logical communication distance be as large as $\Theta(N_{\mathrm{D}})$, then reversibility will confer *no* speed advantage, because the communication time $\Theta(N_{\mathrm{D}})$ will be sufficient for all entropy to be removed even from the irreversible machine in the most compact configuration! So the required communication distance must actually be $\mathbf{o}(N_{\mathrm{D}})$ if we are to achieve any reversible advantage.

An analysis (not detailed here) indicates that the optimal scaling relation d_c between logical communication distance (distance in terms of cells) and array size to achieve maximal reversible advantage is $d_c \sim N_{\mathrm{D}}^{1/2}$. For this problem, the optimal configuration for the irreversible machine turns out to be with distance $\Theta(N_{\mathrm{D}}^{1/6})$ between processors, which gives a minimum time per step of $\Theta(N_{\mathrm{D}}^{2/3})$; the optimal

TPRA and BRA are both packed at fixed density and run with a time per step of $\Theta(N_{\mathrm{D}}^{1/2})$, that is, $\Theta(N_{\mathrm{D}}^{1/6})$ times faster than the irreversible machine. In terms of $\$_{\mathrm{r}} \sim \mathcal{T}$, the reversible advantage is $\mathcal{A}_{\mathrm{r}} \sim \$_{\mathrm{r}}^{1/3}$. This appears to be the maximum speedup possible using time-proportionate reversibility. Still, using the advanced technologies mentioned in chapter 8, we expect that even this rather slow scaling is sufficient to yield significant speedups for reversible machines over irreversible ones at reasonable scales. (However, more detailed analysis is needed.)

Now, as we already discussed in 3.2.2.2, we generally cannot assume that time complexity alone is a good measure of cost. Let us now see what happens to our scaling arguments when we factor in other components of cost as well.

6.2.4 Spacetime cost

The results derived above for the minimum time for array situations might at first appear to be inapplicable to the problem of minimizing the spacetime product, since many of our solutions involved spreading neighboring processors apart with ever-increasing distances between them; the total volume of the computer must thus be enormous!

However, this is actually not the case: all the machines discussed above can be easily converted to configurations in which the total volume of the machine scales no faster than the volume just to store the data being manipulated.

The way this is done is by simply *folding up* each column of processors (normally aligned parallel to the entropy flow) to fill up the entire $\ell \times \ell$ area available between the neighboring columns, thus reducing the thickness of the machine in the direction of entropy flow, to the point where the machine has some fixed density independent of scaling. (See fig. 6.4.)

This transformation changes nothing in our earlier analyses; nothing prevents operation exactly the same as before. We have one additional construction requirement, however; namely that throughout each period that is reserved for signal propagation, each processing element must vacate the paths across the plane (perpendicular to entropy flow) along which interprocessor signals propagate, so as not to the impede the ballistic propagation of signals to and from the processors in neighboring columns.

Therefore, our solutions from the previous section, so reconfigured, optimize volume as well as time, and thus also optimize their product. So for a given problem size, reversibility provides the same asymptotic benefits for spacetime cost (namely, $\mathcal{A}_{\mathrm{r}} \sim N_{\mathrm{D}}^{1/6}$) as it does for time cost alone. Expressed in terms of the number of processors or volume $N_{\mathrm{proc}} \sim \mathcal{V} \sim N_{\mathrm{D}}^3$, we have $\mathcal{A}_{\mathrm{r}} \sim N_{\mathrm{proc}}^{1/18}$. Expressed in terms of the spacetime cost $\$_{\mathrm{r}} = \mathcal{V}\mathcal{T} \sim N_{\mathrm{D}}^3 \cdot N_{\mathrm{D}}^{1/2} \sim N_{\mathrm{D}}^{7/2}$ on the reversible machine, the advantage is $\mathcal{A}_{\mathrm{r}} \sim N_{\mathrm{D}}^{1/6} \sim (\$_{\mathrm{r}}^{2/7})^{1/6} = \$_{\mathrm{r}}^{1/21}$.

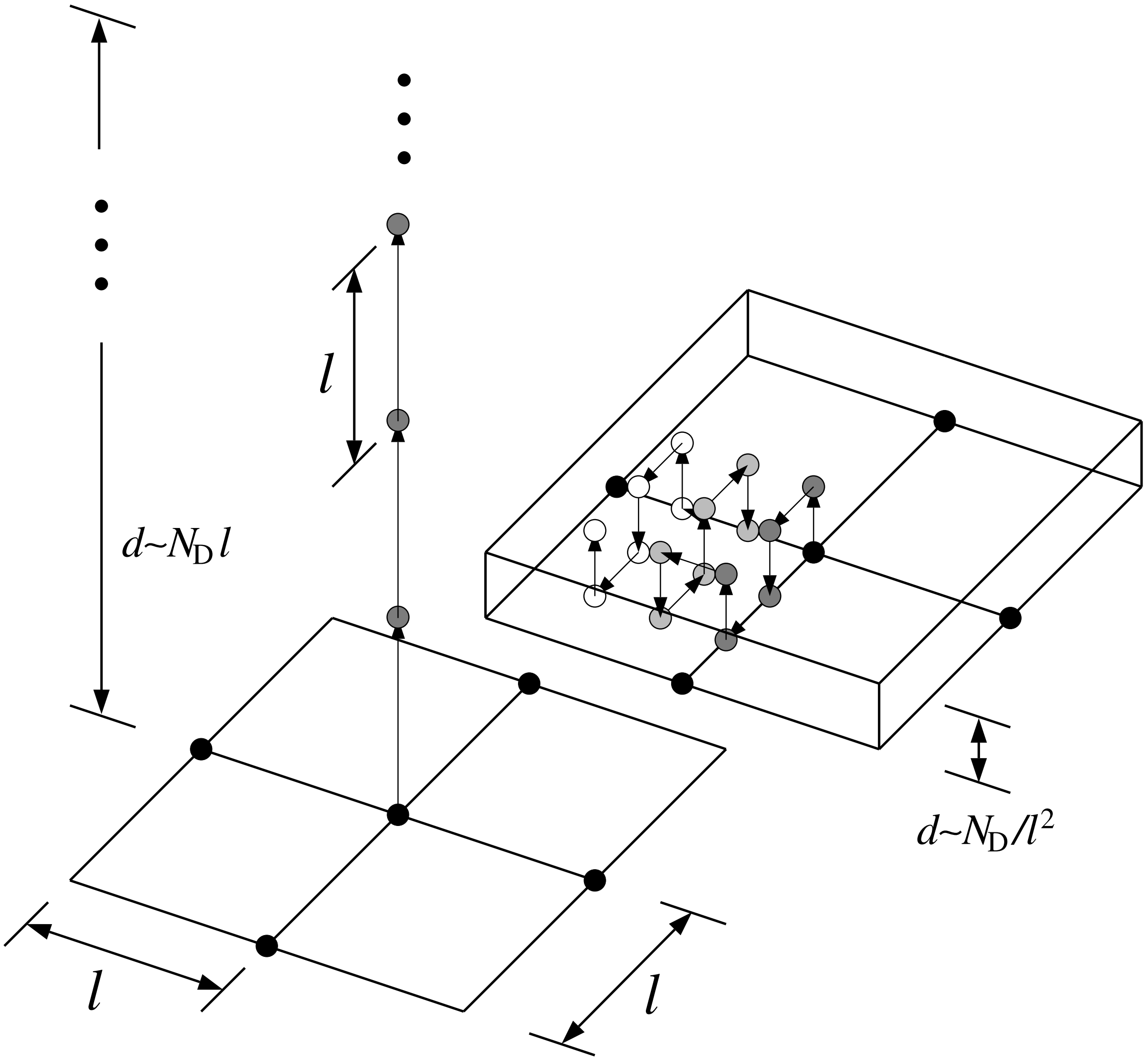


Figure 6.4: How to “fold up” a column of processors to convert a space-inefficient mesh into another structure with the same asymptotic speed but minimum volume. Initially a column of $N_{\rm D}$ (here, 18) processors extends straight up through the machine (full height not shown) from its lowest plane. We take this column, and fold it up at maximum density within the $\ell \times \ell$ area between it and its neighboring columns. The entropy flux through that area does not increase, nor does the distance between any two logically neighboring processors. (Indeed it decreases for neighbors in the same column.) But the thickness d of the machine is decreased by a factor of $\mathbf{\Theta}(\ell^3)$, from $\mathbf{\Theta}(N_{\rm D}\ell)$ to $\mathbf{\Theta}(N_{\rm D}/\ell^2)$; and the volume decreases by the same factor.

6.2.5 Mass-time product

For the array-simulation problem, the mass of our solutions scaled no faster than the mass necessary just to represent the raw information in the array, so our solutions optimized the mass-time product as well. Thus reversibility gives at least the same advantage for mass-time product efficiency as well.

Together with our observations above about spacetime cost, reversibility also minimizes the combined cost measure $(M + \mathcal{V})\mathcal{T}$.

6.2.6 (Area + mass) × time

This case, too, is identical, because for the machines discussed above, $A + M \sim M$ in all machine configurations discussed. The minimum area scaled no faster than the mass, and so for sufficiently large problems made at most a mass-proportionate contribution to the total cost.

6.2.7 Entropy + mass-time

For problems where no communication is required between processors, the mass-time product is proportional to the number of operations, and the entropy production never grows faster than this anyway, so the cost in all models reduces to $\Theta(N_{\mathrm{ops}})$.

For problems such as the array simulations where communication is required, again the total cost is dominated by the mass-time cost in all cases, so reversibility again improves efficiency by the same factor of $\Theta(N_{\mathrm{proc}}^{1/18})$ in the $\sqrt{N_{\mathrm{D}}}$ communication case.

Similarly, we get the same asymptotic results for the comprehensive cost measure $\$_{\mathrm{c}} = S + (A + M + \mathcal{V})T$ from p. 58. (We drop the integral here because we are considering problems where the resource usage does not change significantly over time.)

6.3 Generalizing the results

The scaling results of the previous section were derived under the assumption that the computational task being performed was one that inherently required only reversible operations, so that time-proportionally reversible operations could be considered equivalent in power to logically irreversible operations.

We also depended on the computation being parallelizable, and in the more sophisticated cost measures, we depended on a requirement for frequent communication between relatively nearby processors, and on the absence of a requirement for frequent communication between arbitrarily distant processors. An example of such a computation is the simulation of a spatially and temporally discretized reversible physical system.

Given all these assumptions, just how general are the reversible scaling advantages? Do they cover very many practical applications in large problem classes, other than just physical simulations?

The complete answer to this question is uncertain, but one observation is that Bennett's 1989 algorithm [19, 103] can be utilized to remove the requirement for the *reversibility* of the underlying task, while still permitting almost the same polynomial speedups and cost-efficiency benefits. (However, the assumptions regarding parallelizability and communication requirements remain.)

6.3.1 Speedups for irreversible computations on reversible machines

Bennett's technique [19] allows one to transform a logically irreversible algorithm that requires S memory cells ("space") and T primitive operations ("time") into a reversible algorithm that leaves behind no garbage information (other than input and output) and takes $\mathsf{T}' \sim \mathsf{T}(\mathsf{T}/\mathsf{S})^\varepsilon$ operations, and $\mathsf{S}' \sim \mathsf{S}\log(\mathsf{T}/\mathsf{S})$ memory, for any $\varepsilon > 0$. (See Levine & Sherman 1990 [103] for the derivation.)

A finite irreversible processing element running for N_{st} steps performs $\mathsf{T} \sim N_{\mathrm{st}}$ operations, using $\mathsf{S} \sim 1$ space. Therefore, using Bennett's algorithm, such a run can be simulated reversibly with $\mathsf{T}' \sim N_{\mathrm{st}}^{1+\varepsilon}$, $\mathsf{S}' \sim \log N_{\mathrm{st}}$, accumulating no garbage except for the input, that is, the state of the simulated processor prior to the run.

If we then irreversibly erase this $\Theta(1)$-size input, we generate $\Theta(1)$ entropy, and we can proceed to simulate arbitrarily many consecutive blocks of N_{st} steps in this way, with an average entropy generation per reversible operation of $S_{\mathrm{op}} \sim 1/N_{\mathrm{st}}^{1+\varepsilon}$, and a memory requirement of only $\mathsf{S}' \sim \log N_{\mathrm{st}}$, which is constant in the number of *blocks* of N_{st} steps that are simulated.

If we wish this algorithm to be time-proportionately reversible, the average entropy generated per operation must be $\mathcal{O}(1/t_{\mathrm{op}})$. So we must have $N_{\mathrm{st}}^{1+\varepsilon} \succsim t_{\mathrm{op}}$, or $N_{\mathrm{st}} \succsim t_{\mathrm{op}}^{1/(1+\varepsilon)}$. With the smallest choice, $N_{\mathrm{st}} \sim t_{\mathrm{op}}^{1/(1+\varepsilon)}$, the memory requirement of this algorithm then scales as $\mathsf{S} \sim \log t_{\mathrm{op}}^{1/(1+\varepsilon)} \sim \log t_{\mathrm{op}}$, given constant ε.

By running this algorithm simultaneously on a 3-D array of reversible processors of memory $\Theta(\log t_{\mathrm{op}})$ each, we can sustainably simulate an entire 3-D array of fixed-size irreversible processors in TPRA fashion. Furthermore, we saw in §6.2.3.1 that a 3-D TPRA can run with $t_{\mathrm{op}} \sim N_{\mathrm{D}}^{1/4}$, so a memory per processor of $\mathsf{S} \sim \log N_{\mathrm{D}}^{1/4} \sim \log N_{\mathrm{D}}$ will suffice. Given that the computer must fit within the finite observable universe, $\log N_{\mathrm{D}}$ is bounded by a reasonably small constant, so we may approximate S as

$\mathbf{\Theta}(1)$ for all practical purposes. (Although this is cheating from a pure theoretical perspective.)

With $t_{\mathrm{op}} \sim N_{\mathrm{D}}^{1/4}$, we have that $N_{\mathrm{st}} \sim N_{\mathrm{D}}^{1/[4(1+\varepsilon)]}$, and the Bennett simulation of this many steps takes $N_{\mathrm{ops}} \sim N_{\mathrm{st}}^{1+\varepsilon} \sim [t_{\mathrm{op}}^{1/(1+\varepsilon)}]^{1+\varepsilon} = t_{\mathrm{op}} \sim N_{\mathrm{D}}^{1/4}$ reversible operations, for an average real time, per irreversible step simulated, of

$$\begin{aligned} \mathcal{T} &= t_{\mathrm{op}} N_{\mathrm{ops}} / N_{\mathrm{st}} \\ &\sim N_{\mathrm{D}}^{1/4} N_{\mathrm{D}}^{1/4} / N_{\mathrm{D}}^{1/[4(1+\varepsilon)]} \\ &= N_{\mathrm{D}}^{\frac{1}{2} - \frac{1}{4(1+\varepsilon)}} \\ &= N_{\mathrm{D}}^{\frac{1}{4}\left(\frac{1+2\varepsilon}{1+\varepsilon}\right)} \\ &= N_{\mathrm{D}}^{\frac{1}{4}(1+\varepsilon')} \end{aligned} \tag{6.1}$$

where $\varepsilon' = \varepsilon/(1+\varepsilon)$. The exponent of N_{D} in eq. (6.1) can be made as close to $1/4$ as desired, by taking ε close to 0.

In contrast, as we saw earlier, the 3-D irreversible array being simulated must itself take at least $\mathbf{\Omega}(N_{\mathrm{D}}^{1/3})$ time per step. So a reversible machine can simulate even an *irreversible* 3-D array faster than that array can run by itself! This improved asymptotic speed also leads to improved asymptotic cost-efficiency by the various other measures we have covered. Moreover, the reversible advantages can become arbitrarily asymptotically close to those we calculated in the previous section for the case of simulating 3-D reversible systems.

However, as pointed out by Levine and Sherman [103], one must be careful when using Bennett's algorithm not to take ε *too* close to zero, because the constant factor in the memory requirement increases *exponentially* in $1/\varepsilon$, specifically as $\varepsilon 2^{1/\varepsilon}$. But to beat the irreversible 3-D array's asymptotic performance, we only require $\varepsilon < 1/2$, so the constant factor increase in memory size due to the choice of ε only needs to be more than 2.

The upshot of all this is that, apparently, for *any* class of computations that are sufficiently parallelizable and require the right amount of communication, a TPRA reversible machine family, such as the R3M, can perform that class of computations strictly faster, asymptotically, than any FIA machine family. The class of computations in question does *not* have to be "inherently" reversible in order for this to be true.

6.4 Summary of scaling results

We conclude this chapter with a summary of our discoveries about the asymptotic scaling advantages that can be gained by the use of time-proportionate reversibility.

Cost measure	Task type	Cost in each model			Advantage factors	
		FIA ($\$_\mathrm{i}$)	TPRA ($\$_\mathrm{r}$)	BRA ($\$_\mathrm{b}$)	Reversible $\mathcal{A}_\mathrm{r} = \$_\mathrm{i}/\$_\mathrm{r}$	Ballistic $\mathcal{A}_\mathrm{b} = \$_\mathrm{i}/\$_\mathrm{b}$
$\$ = S$	any	N_ops	n_in	n_in	$N_\mathrm{ops}/n_\mathrm{in}$	$N_\mathrm{ops}/n_\mathrm{in}$
$\$ = \mathcal{T}$	no comm.	1	1	1	1	1
	local comm.	$N_\mathrm{D}^{1/3}$	$N_\mathrm{D}^{1/4}$	1	$N_\mathrm{D}^{1/12}, \$_\mathrm{r}^{1/3}$	$N_\mathrm{D}^{1/3}$
	$\sqrt{N_\mathrm{D}}$ comm.	$N_\mathrm{D}^{2/3}$	$N_\mathrm{D}^{1/2}$	$N_\mathrm{D}^{1/2}$	$N_\mathrm{D}^{1/6}, \$_\mathrm{r}^{1/3}$	$N_\mathrm{D}^{1/6}, \$_\mathrm{b}^{1/3}$
$\$ = A\mathcal{T}$	no comm.	N_ops	$N_\mathrm{ops}^{5/6}$	$N_\mathrm{ops}^{2/3}$	$N_\mathrm{ops}^{1/6}, \$_\mathrm{r}^{1/5}$	$N_\mathrm{ops}^{1/3}, \$_\mathrm{b}^{1/2}$
$\$ = \mathcal{V}\mathcal{T}$,	no comm.	N_D^3	N_D^3	N_D^3	1	1
$M\mathcal{T}$,	local comm.	$N_\mathrm{D}^{3+1/3}$	$N_\mathrm{D}^{3+1/4}$	N_D^3	$N_\mathrm{D}^{1/12}, \$_\mathrm{r}^{1/39}$	$N_\mathrm{D}^{1/3}, \$_\mathrm{b}^{1/9}$
$\ldots, \$_\mathrm{c}$	$\sqrt{N_\mathrm{D}}$ comm.	$N_\mathrm{D}^{3+2/3}$	$N_\mathrm{D}^{3+1/2}$	$N_\mathrm{D}^{3+1/2}$	$N_\mathrm{D}^{1/6}, \$_\mathrm{r}^{1/21}$	$N_\mathrm{D}^{1/6}, \$_\mathrm{b}^{1/21}$

Table 6.2: Summary of asymptotic scaling results for reversible versus irreversible machines. The first column indicates the type of cost measure being used; S being entropy, A surface area, $\mathcal{T}$ real time, $\mathcal{V}$ physical volume, M total gravitating mass, $\$_\mathrm{c}$ the comprehensive cost measure from p. 58. The second column indicates restrictions on the type of computational task for which the results hold, in particular on the communication involved. The quantity N_D refers to the number of elements along each dimension of a 3-D array of finite-state cells.

Under the most comprehensive cost measures, such as $\$_\mathrm{c}$, the reversible advantage $\mathcal{A}_\mathrm{r}$ can scale with factors as high as the 21st root of the reversible cost (18th root of physical machine size), but no more than that. With fully ballistic machines, the comprehensive advantage (in the local communication case) would scale better, $\Theta(\$_\mathrm{b}^{1/9})$.

See table 6.2.

The best asymptotic cost-efficiency advantage for reversible machines is of course gained in the case where total entropy generation is the sole measure of cost. The ratio between irreversible and reversible entropy costs in this case may be an arbitrarily fast-growing function of problem size or reversible entropy cost. But this cost measure is not very satisfying because it ignores the time taken and the opportunity cost due to the temporary use of other resources (A, $\mathcal{V}$, M).

In contrast, considering time costs alone gives a rate of growth for the reversible-to-irreversible speed ratio that, for suitable problem classes, is limited to at most the cube root of the reversible time cost. Thus, a computation on a TPRA machine (such as the R3M of 5.4.1, p. 116) that takes time $\mathcal{T}$ will in some cases require as much as $\Omega(\mathcal{T} \cdot \mathcal{T}^{1/3})$ time in the *fastest possible* fully irreversible (FIA) implementation. In other words, in terms of speed, the class of architectures that permits time-proportional reversible operation strictly dominates the class of architectures that does not.

Of course, in general, time alone is not the only factor in the cost of computation, so we also studied the case where various measures of machine size that influence "rental cost" were included as well. With surface area as the size measure, the best reversible advantage $\mathcal{A}_\mathrm{r}$ grows as the 5th root of cost, or the 4th root of area, or the square root of diameter. When mass and/or volume are included as components of the machine size, the best reversible advantage scales as the 21st root of total cost, or 18th root of the machine's mass or volume. This advantage occurs in computations in which communication distances are proportional to the square root of the logical diameter of the machine.

Such scaling may not appear to be very significant, but we estimate that the constant factors work out so that even given the relatively poor performance of the reversible logic devices available today (which we will discuss in the next chapter), at the extremely high end of machines buildable with current technology, reversible operation is apparently required for optimal efficiency.

Such a machine would be rather large and very expensive (we estimate tens of billions of dollars), but as the underlying device technology improves, machines that gain a cost-efficiency advantage through reversibility will become buildable at lower and lower cost levels. In chapter 8 we will show that if certain proposals for future reversible logic devices work as predicted, any computer larger than about a micron in diameter will require reversibility in order to achieve optimal efficiency.

Part II

Engineering reversible computational systems

Chapter 7

Adiabatic circuits

Part I of this thesis explored the general motivation for and properties of reversible machines, without reference to any particular implementation technology. In Part II, beginning with this chapter, we address a variety of engineering and implementation issues in reversible computing, showing ways to actually design, build, and program reversible computers.

Virtually all computers today are built using semiconductor VLSI (very-large scale integration) technology, in which, typically, metal-oxide-semiconductor field-effect transistors (MOSFETs) are wired together to form CMOS (complementary MOS) logic gates.

Unfortunately, the way these CMOS logic gates are currently designed, they are operationally irreversible, and thus have fixed lower bounds on their energy dissipation and entropy generation, which we will analyze in some detail in §7.1. These bounds have consequences for the maximum cost-efficiency of computation using irreversible CMOS (hereafter abbreviated "iCMOS") under various cost measures.

The irreversibility of traditional logic elements has led researchers to ask whether transistor electronics could instead be configured in such a way as to form reversible logic elements. The answer is yes, there is a class of reversible logic circuit styles called "adiabatic" circuits (a somewhat misleading name, as we will explain in §7.3), whose history we will review. We will then describe in detail *SCRL* (split-level charge recovery logic), a particular variant of adiabatic circuit technology that was developed a few years ago by members of our research group [192, 193, 191, 194]. SCRL has several properties that make it particularly suitable for achieving the asymptotic efficiency benefits that we discussed in ch. 6. It is capable of full reversibility. It can be built cheaply and easily using today's commercially available VLSI fabrication processes. And SCRL's energy dissipation roughly matches the TPRA (time-proportionately reversible) model of chapter 6.

Next, we describe our SCRL-based design of FLATTOP, a simple adiabatic circuit that we recently fabricated [72]. This circuit can be viewed as the first ever universal reversible processor core, capable, in principle, of fulfilling the scaling laws derived in the previous chapter, and thereby computing asymptotically faster than any irreversible technology in machines at a sufficiently large scale. (However, in practice, the FlatTop chip is really just a proof of concept.)

Unfortunately, with some additional analysis (not contained in this thesis) we have found that the constant factors of present-day VLSI (very large-scale integration) semiconductor technology imply that the cost level at which reversible computing will actually dominate in speed is, in the present technology generation, economically infeasible (roughly speaking, in the range of billions of dollars). In the following chapter, we will review a variety of possible future device technologies for which, in contrast, reversibility wins out for all but the tiniest of machines. Then, chapter 9 will review reversible programming issues.

7.1 Maximizing the efficiency of iCMOS

Before we describe SCRL, in this section we will establish a baseline for comparison by roughly estimating the optimum performance of present and future irreversible CMOS, under a variety of cost-efficiency measures.

The central motivation for adiabatic circuits is the avoidance of the $\sim CV^2$ energy dissipation that (as we will see) is necessarily incurred by ordinary irreversible logic circuits whenever they switch a signal from one logic level to another. In this section we briefly review traditional irreversible CMOS logic, and the reasons for its energy dissipation, and analyze in some detail the present and future minimum energy dissipation of such circuits, based on the semiconductor industry's technology projections for the next decade. This analysis can serve as a baseline for comparison when we wish to look at the capabilities of adiabatic circuits.

7.1.1 Basic iCMOS review

A detailed description of irreversible CMOS technology can be found in any standard, reasonably recent VLSI textbook, for example [84, 187, 139]. Here, we only briefly review the basics.

CMOS logic gates may appear in the simple *static* form, or in various *dynamic* variations. In this context, "static" and "dynamic" refer to whether the output of the logic gate is always tied to a fixed voltage source, or is sometimes allowed to float freely. The underlying energy issues in the two circuit styles are basically similar, so to ease our analysis, we will focus on the simplest static logic style. The use of alternative styles of irreversible CMOS logic can improve energy efficiency beyond that of ordinary static CMOS by, at most, only a small constant factor.

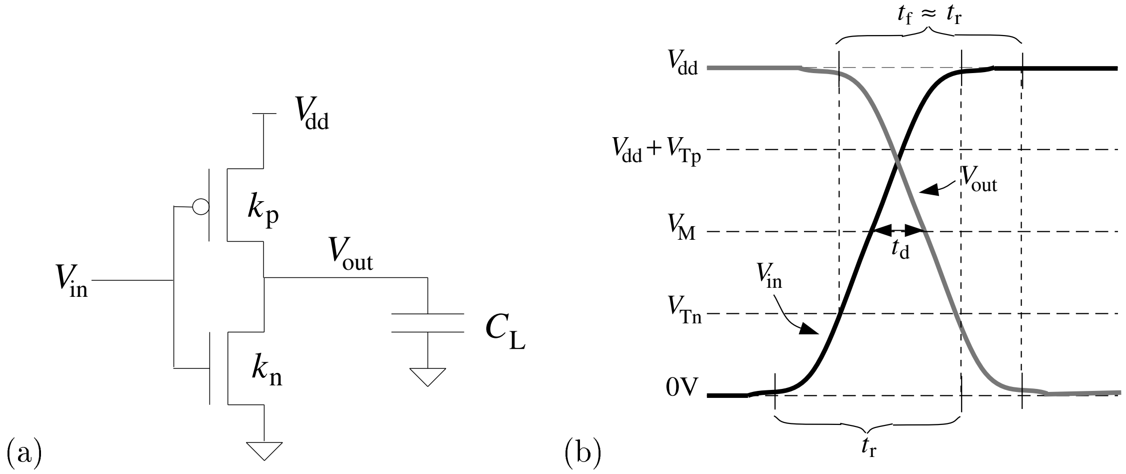


Figure 7.1: (a) An ordinary CMOS inverter, consisting of an n-FET (bottom) and a p-FET (top). Given an input logic value A, the inverter computes its inverse $\bar{A}$. With more complex networks of p-FETs and n-FET in the pull-up and pull-down networks, arbitrary inverting logic functions of multiple inputs can be similarly implemented.

(b) Dynamic behavior. The n-FET conducts significantly when V_{in} is above the threshold voltage V_{Tn}, and the p-FET conducts when V_{in} is below the level $V_{\text{dd}} + V_{\text{Tp}}$ (where V_{Tp} is negative). When the input voltage V_{in} goes to the high level V_{dd}, representing a logic `1`, the output voltage V_{out} goes to the low level of 0 V, representing a `0`, and vice-versa. The delay t_{d} is affected by the *load capacitance* C_{L}, the supply and threshold voltages, and the *gain factors* (also called *device transconductance parameters*) k_{n} and k_{p} of the n-FET and p-FET devices. If the inverter is driven by a similar inverter and if $k_{\text{p}} \approx k_{\text{n}}$, then the rise and fall times t_{r} and t_{f} of input and output will be about equal.

Figure 7.1 shows a static CMOS inverter, consisting of two MOSFETs, one n-type and one p-type. (The n and p refer to whether the primary charge carriers in the device are negatively charged electrons or positively charged "holes.") The p-FET connects the output to a high voltage when the input is low, and the n-FET connects the output to a low voltage when the input is high.

The inverter structure can be generalized to compute any *inverting* boolean function of many inputs (a one-bit function that is monotonically non-increasing in the values of the input bits), by replacing the p-FET and n-FET with appropriate networks of many p-FET and n-FET devices, respectively.

Now, suppose we wish to minimize the total entropy generation of a computation. To see how to do this, let consider how energy dissipation in a CMOS circuit scales with various parameters.

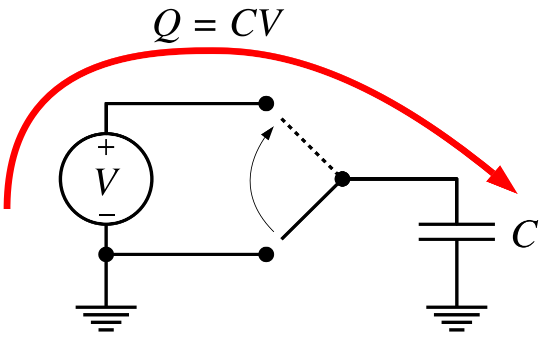


Figure 7.2: Energy dissipation in conventional switching. Whenever a node is switched by connecting it to a constant-voltage V power supply, such as occurs in the ordinary CMOS gate of fig. 7.1, the transfer of the charge $Q = CV$ from the constant-voltage source to the node capacitance C results in an energy dissipation of $\frac{1}{2}CV^2$, where V is the voltage swing and C is the capacitance being switched.

7.1.1.1 iCMOS energy dissipation

We have seen that standard CMOS circuits charge and discharge loads by connecting them to constant-voltage power supplies. Because of this, whenever the voltage of a node is switched from one level to another, through a voltage change V, the energy dissipated is at least $\frac{1}{2}CV^2$, where C is the total capacitance being switched.

To understand this, refer to fig. 7.2. An amount $Q = CV$ of charge is delivered to the node from voltage V, so the energy supplied is at least $E_{\mathrm{supp}} = QV = CV^2$. The energy stored on the node is

$$\begin{aligned} E_{\mathrm{stor}} &= \int dE = \int v(q)\,dq = \int \frac{q}{C}\,dq = \frac{1}{C}\int_0^Q q\,dq \\ &= \frac{1}{C}\,\frac{1}{2}q^2\bigg|_{q=0}^{Q} = \frac{1}{2C}Q^2 = \frac{1}{2C}(CV)^2 \\ &= \frac{1}{2}CV^2. \end{aligned}$$

(For additional textual explanation, see [143], §27-4, p. 521.) The remaining energy $E_{\mathrm{supp}} - E_{\mathrm{stor}} = CV^2 - \frac{1}{2}CV^2 = \frac{1}{2}CV^2$ can not be accommodated in the circuit model; no matter the precise mechanism by which it has left the system, this energy is now considered unmodeled, statistical, thermalized; so as a matter of definition we call it "dissipated." Essentially it becomes heat.

This energy dissipation and the accompanying entropy increase occurs under *any* means of setting a voltage by delivering charge directly from a constant-voltage source.

There is no way it can be avoided by, for example, simply adjusting the resistance or inductance of the switch: note that the above analysis depends in no way on such parameters. We call this $\frac{1}{2}CV^2$ dissipation the *switching energy* E_{sw}.

Of course, if the circuit is a small part of a larger system, then the dissipation to heat of a certain amount of energy by the circuit need not imply that this energy will never again be available for work: for example, given a cool thermal reservoir, the computer can, as a side effect, power a heat engine which can recover a portion of the dissipated energy as work. But in the absence of any specific mechanism tailored to capture the supplied energy in a more direct way, the energy $E = \frac{1}{2}CV^2$ must at still at least be *temporarily* dissipated (thermalized), which results in an immediate, irreversible entropy generation of at least $S = CV^2/2T$, where T is the temperature of the circuit.

Moreover, if the switch that we are using is a static CMOS logic gate, there may be additional dissipation in the gate if the n-FET and p-FET are simultaneously conducting significantly during some portion of the input transition, resulting in a current from power to ground through both transistors. Generally this will occur if $V_{\mathrm{dd}} \gtrsim V_{\mathrm{Tn}} + |V_{\mathrm{Tp}}|$.

This *short-circuit dissipation* is somewhat more complex to model than switching energy, because it depends on the dynamic behavior of the CMOS devices and the input signal. But we can make some useful approximations when the input is driven by a similar gate: if $V_{\mathrm{dd}} \gg V_{\mathrm{Tn}} + |V_{\mathrm{Tp}}|$, then a rough dynamic analysis shows that the short-circuit dissipation E_{ss} will be around CV_{dd}^2, comparable to the switching energy. However, if $V_{\mathrm{dd}} \lesssim V_{\mathrm{Tn}} + |V_{\mathrm{Tp}}|$, then the short-circuit dissipation will be small compared to the switching energy.

Finally, for sufficiently slowly-switching CMOS circuits, another source of dissipation may become important, namely leakage energy due to sub-threshold conduction of MOS devices even when turned off. Due to thermal effects, the transistor off-current cannot be made less than the on-current by more than a factor of roughly $f = e^{V_{\mathrm{dd}}/\phi_T}$, where $\phi_T = k_{\mathrm{B}}T/q_{\mathrm{e}}$ is the thermal voltage of electrons in a device at temperature T. Therefore when device idle times are more than a factor of f times larger than switching times, the *leakage energy* E_{leak} due to the off-current may be the dominant contributor to the total energy dissipation per operation.

7.1.2 iCMOS entropy generation.

All of the total energy $E_{\mathrm{tot}} = E_{\mathrm{sw}} + E_{\mathrm{ss}} + E_{\mathrm{leak}}$ dissipated per operation, due to these various causes, becomes heat, generating an amount of new entropy $S = E_{\mathrm{tot}}/T$, where T is the operating temperature of the devices. This entropy generation can be made smaller by raising the operating temperature, but only up to a point, because higher temperatures eventually lead to large leakage currents, and higher operating voltages (because we must obey $V_{\mathrm{dd}} \gtrsim \phi_T$ or else transistors will not be significantly more conductive when on than when off), and will eventually preclude correct functionality altogether, by melting the device, if not by some other effect that cripples the device's functionality far below its melting temperature.

The relationships between all these factors are complex, but if we just assume that there is some maximum temperature $T_{\max}$ for a working CMOS device, and if as an initial rough estimate we place $T_{\max}$ at around 150 degrees Celsius (423 K) (we do not know of any semiconductor chips that operate hotter than this), then this gives us a rough lower bound on entropy generation of $S \geq \frac{1}{2}CV^2/(423K) = 1.18\times10^{-3}CV^2/K = (86\ \mathrm{nat/aJ})CV^2$, which is an unavoidable lower bound as long as C and V are assumed fixed.

We can still lower bound the entropy generation even if V is allowed to vary. Since we know that $V \gtrsim \phi_T = k_{\mathrm{B}}T/q_{\mathrm{e}}$ for correct functionality, we can define a new lower bound in terms of temperature and load capacitance:

$$\begin{aligned} S &\geq \frac{1}{2}CV^2/T \\ &\gtrsim \frac{C(k_{\mathrm{B}}T/q_{\mathrm{e}})^2}{2T} \\ &= \frac{Ck_{\mathrm{B}}^2T}{2q_{\mathrm{e}}^2}. \end{aligned}$$

Or, if we define a quantity $C_T \equiv q_{\mathrm{e}}/\phi_T$, which we will call the *thermal capacitance*, we have

$$S \gtrsim \frac{1}{2}\cdot\frac{C}{C_T}\ \mathrm{nat}. \tag{7.1}$$

For example, at a temperature of 300 K (room temperature), the thermal capacitance is $C_T \approx 6.2\times10^{-3}$ fF, so our entropy bound becomes

$$S \gtrsim (80.7\ \mathrm{nat/fF})\ C.$$

Since $\phi_T \propto T$, $C_T \propto 1/T$ and so the minimum S in (7.1) scales as T, showing that actually, when voltage is adjustable, low temperatures are best for minimizing entropy generation, at least until the point where the minimum V_{dd} is no longer limited only by the thermal voltage.

The form of eq. (7.1) might seem to suggest that ordinary static CMOS circuits could theoretically generate less than a bit of entropy per switching operation, at any given temperature, if the node capacitance is just made sufficiently small compared to the thermal capacitance. However, we will now see that the capacitance and voltage *cannot* together be made low enough to generate less than 1 bit of entropy per switching event, while still permitting reliable operation of these irreversible circuits.

What is the interpretation of the thermal capacitance C_T? First, from our above definition, it is the node capacitance for which at least 1/2 nat of entropy is generated by dissipative switching through a voltage swing of ϕ_T. But it is also the node capacitance at which the addition of *one electron* raises the node voltage by ϕ_T. Since individual electrons experience an average excitation equal to the thermal energy $E_T = k_\mathrm{B}T$, the voltage on a node with capacitance only C_T will routinely be different than the expected level by amounts comparable to the thermal voltage. Therefore such a node cannot reliably store a logic value encoded by a voltage difference of only ϕ_T. For that, a significantly greater capacitance is required.

Additionally, switching a node by tying it to a constant-voltage source at voltage V is a logically irreversible operation, since after the transition is completed, the information about the previous logical state of the node has been lost. (There is simply no mechanism in these simple circuits for retaining this information in a controlled form, just as there is no mechanism, for the $\frac{1}{2}CV^2$ of supplied energy that doesn't end up on the capacitor, of retaining that energy in a controlled form, *i.e.*, not heat.) So, if the node was equally likely to be in either of two states before the operation, one of them at voltage V and the other at voltage 0, and after the operation it is reliably in a single state (corresponding to voltage V) then the average entropy generation is at least 1 bit $= \ln 2$ nat. Thus the average energy dissipation must be $k_\mathrm{B}T\ln 2$ to provide for this entropy (see §2.5.3, p. 46).

Since no energy is dissipated in the half of the cases when the node is unchanged, the energy dissipation in the other half of the cases must be twice this, $2k_\mathrm{B}T\ln 2$, in order for the average dissipation to be $k_\mathrm{B}T\ln 2$. In order for the switching energy $\frac{1}{2}CV^2$ to be greater than this, we must have

$$\begin{aligned}\frac{1}{2}CV^2 &\geq 2(\ln 2)k_\mathrm{B}T = 2(\ln 2)E_T \\ C &\geq 4(\ln 2)\frac{E_T}{V^2}\end{aligned}$$

so if $V \approx \phi_T$,

$$\begin{aligned}C &\gtrsim 4(\ln 2)\frac{E_T}{\phi_T^2} \\ &= 4(\ln 2)\,C_T.\end{aligned}$$

For example, at $T = 300$ K, and $V \approx \phi_T$, we have $C \gtrsim 0.017$ fF.

In other words, the minimum average entropy generation per irreversible switching event *must* be

$$S \geq 2 \text{ bits}$$

if reliable erasure of a random bit is to be possible.

More generally, consider nodes in any constant-voltage switching technology in which short-circuit energy and leakage energy are negligible, so that the $\frac{1}{2}CV^2$ switching energy is the only dissipation. Suppose a node is to hold a logic `1` in exactly one randomly-selected case out of N cases (instances distributed in space or in time), and is `0` in the rest of the cases, and we wish that, with high probability, the logic value should become `0` after the switching operation in *every case*, given that the node becomes tied to a constant voltage source whose level represents `0`. The entropy generated is

$$S \geq \ln N \text{ nats,} \tag{7.2}$$

but energy is only dissipated in the single case where the bit is `1`. In order for the environment to hold the increased entropy, the energy dissipated during switching in this case must be $E \geq ST \geq k_{\mathrm{B}}T \ln N$. In other words, node capacitance C and swing voltage V must be such that

$$\frac{1}{2}CV^2 \geq E_T \ln N \tag{7.3}$$

in order for a node to switch correctly in every one of N instances (with high probability). Another way to write this formula is

$$\frac{1}{2}\left(\frac{C}{C_T}\right)\left(\frac{V}{\phi_T}\right)^2 \geq \ln N,$$

which shows explicitly the relationship between node capacitance, thermal capacitance, node voltage, and thermal voltage. If we take $V \approx \phi_T$, we get a lower bound on node capacitance of

$$C \gtrsim 2(\ln N)\, C_T.$$

So, for example, to achieve a nice N value of 10^{27}, corresponding to a computer with 10^9 circuit nodes not making a single thermal error in 1 Gs (~32 years) of operation at a clock speed of 1 GHz, we must have a capacitance *per circuit node* of $C \gtrsim 0.77$ fF, if the operating voltage is almost as low as the room-temperature thermal voltage. Present-day gate capacitances of minimum-sized transistors are already near these levels. If some circuit nodes actually have smaller capacitances than this, supply voltages cannot actually approach the thermal voltage without sacrificing reliability to some extent.

It is interesting that we are able to derive the above relationship between reliability and node capacitance solely on the basis of the entropy generation and the switching energy. This can be compared with the traditional approach of developing a sophisticated thermal noise model, which finds that voltage samples follow a normal distribution, with a mean squared error (on a capacitance-C node) of $\Delta V^2 = k_{\mathrm{B}}T/C$ ([129], §1.12, p. 31), and therefore to sample N instances to an accuracy of $k_{\mathrm{B}}T$ with high reliability, C must scale up with $\ln N$ roughly as we have described, or else there will be a significant chance that a thermal fluctuation will, in one of the N instances, place the voltage sample enough standard deviations out on the tail of the distribution to cause a sampling error, and thus qualitatively incorrect functionality of the logic.

7.1.2.1 Voltage bounds.

To express the above voltage bound quantitatively, we can rewrite eq. (7.3) as a lower bound on the swing voltage V in terms of the node capacitance and thermal constants:

$$V \geq \sqrt{\frac{2E_T \ln N}{C}}$$

or equivalently

$$V \geq \phi_T \sqrt{2 \ln N \frac{C_T}{C}}.$$

Thus, as capacitances decrease towards and below the thermal capacitance C_T, minimum node swing voltages must increase to larger and larger multiples of the thermal voltage, proportional to the square root of the capacitance decrease, in order to maintain a given level of reliability. This is an important lower bound on operating voltage which must be taken into account when considering the minimum energy dissipation of irreversible switching circuits.

Of course, continually increasing voltages in order to shrink circuits is unrealistic since high electric fields will cause gate oxide breakdown. So in the long term, as nodes shrink in all dimensions proportionally to some characteristic feature size ℓ, and node capacitances and voltages decrease proportionally, the above analysis really demands that we must eventually start scaling switching energy down as $CV^2 \sim \ell^3$ (which is intuitive for another reason, namely that otherwise, assuming switching frequency does not decrease, the power dissipation density in the channel would increase indefinitely, which would eventually cause damage such as melting and loss of structure) and the only way to reduce the switching energy in an irreversible circuit while still maintaining thermal reliability is, by the above analysis, to scale the temperature down with ℓ^3 as well. So as trends in irreversible circuits continue, active cooling to cryogenic temperatures will eventually become a necessity in order to maintain good reliability.

Another lower bound on operating voltage for CMOS circuits comes from the fact that the device thresholds V_{Tn} and V_{Tp} cannot be set with complete precision, due to the statistical nature of existing dopant implantation processes. If V_{dd} is not significantly larger than the variability σ_{VT} of the V_{T} values, then some devices may fail to switch on and off as desired, and functionality may be compromised.

Our lower bounds on operating voltage may be summed up as follows:

$$\begin{aligned}
&V \gtrsim \phi_T && \text{(to switch FETs strongly on and off)}\\
&V \geq \phi_T \sqrt{2 \ln N \frac{C_T}{C}} && \text{(for reliability despite thermal noise)}\\
&V \gg \sigma_{\mathrm{VT}} && \text{(to avoid defects due to threshold variation).}
\end{aligned}$$

Actually operating at the minimum voltage that is permitted by the above requirements may or may not maximize cost-efficiency, depending on the particular measure of cost that we are trying to minimize. Let us now see how to choose the operating voltage of irreversible static CMOS circuits so as to maximize efficiency under a variety of cost measures.

To sum up our discussion of entropy generation in irreversible CMOS, we saw that one should arrange that $\frac{1}{2}CV^2$ for each switching event is as small as possible, while remaining above $E_T \ln N$ (see eq. 7.3). Lowering the operating temperature helps decrease entropy production if it allows V to decrease. If $\frac{1}{2}CV^2 = 2E_T \ln N$, then entropy generation must be at least $\ln N$ nats per switching event. This is $\alpha N \ln N$ nats for an error-free computation composed of N primitive operations, if a fraction α of the operations cause switching.

Of course, other technological considerations may prevent $\frac{1}{2}CV^2$ from coming anywhere close to room-temperature E_T. The minimum load capacitance is determined by factors such as the minimum transistor gate area, and V is independently bounded below by the variability σ_{VT} of device threshold voltages due to the statistical nature of ion implantation in the channel region. An interesting property of any given CMOS fabrication process is the ration between the minimal $\frac{1}{2}CV^2$ and $E_{T=300\mathrm{K}}$ in properly-functioning logic circuits in that process. In §7.1.3 we estimate this quantity for several present and projected future generations of CMOS technology.

7.1.3 The SIA semiconductor roadmap

Table 7.1 shows some parameters for future generations of CMOS VLSI technology as forecasted by the Semiconductor Industry Association, in [145]. We will be referring to these numbers for our calculations throughout the rest of this section.

Given these numbers, we can take a stab at an actual numeric computation of the minimum entropy generation per switching event. See table 7.2. Let us explain these

	Year of first product shipment						
	1997	1999	2001	2003	2006	2009	2012
Overall characteristics:							
Trans. density,[a] 10^6 trans/cm^2	3.7	6.2	10	18	39	84	180
Chip size,[b] mm^2	300	340	385	430	520	620	750
Clock freq.[c], GHz	0.75	1.25	1.5	2.1	3.5	6	10
Supply voltage[d], V	2.5–1.8	1.8–1.5	1.5–1.2	1.5–1.2	1.2–.9	.9–.6	.6–.5
Max. power[e], W	70	90	110	130	160	170	175
Technology requirements:							
μP drawn channel length[f], nm	200	140	120	100	70	50	35
DRAM $\frac{1}{2}$-pitch[f], nm	250	180	150	130	100	70	50
T_{ox} equivalent[g], nm	4–5	3–4	2–3	2–3	1.5–2	< 1.5	< 1
CV/I delay[g], ps	16–17	12–13	10–12	9–10	7	4–5	3–4
V_{T} 3σ varia.[g], ±mV	60	50	45	40	40	40	40
Src./drn. junction depth,[g] nm	50–100	36–72	30–60	26–52	20–40	15–30	10–20

Table 7.1: Selected numbers from the 1997 edition of the Semiconductor Industry Association's national semiconductor roadmap [145]. These numbers are used for the calculations in tables 7.2 and 7.4.

---

[a] Logic transistor density in a packed, high-volume, cost-performance microprocessor, including on-chip SRAM. From [145], p. 14.

[b] Size for a μprocessor, year 1, before subsequent shrinks; [145] p. 15.

[c] On-chip local clock frequency for high-performance chips, [145], p. 16.

[d] Minimum logic power supply voltage V_{dd}, [145], p. 17.

[e] Maximum power consumption for a high-performance processor with heat sink, [145] p. 17.

[f] Minimum feature sizes, [145], pp. 14, 85.

[g] The last four lines in the table are all from [145], p. 46.

	Year of first product shipment						
	1997	1999	2001	2003	2006	2009	2012
Capacitance calculations:							
Gate oxide thickness $\tilde{T}_{\text{ox}}$, nm	4.5	3.5	2.5	2.5	1.75	1.5	1
Gate areal capac. C_{ox}, fF/μm^2	7.81	10.0	14.1	14.1	20.1	23.4	35.1
Min. gate cap. C_{gmin}, aF	312	197	202	141	98.4	58.6	43.0
Est. min. load cap. $\tilde{C}_{\text{Lmin}}$, fF	5.00	3.15	3.24	2.25	1.57	.937	.689
Voltage calculations:							
Transistors/die, 10^6	11.1	21.1	38.5	77.4	203	521	1350
N trans./defect (90% yield), 10^9	.111	.211	.385	.774	2.03	5.21	13.5
Defect probability p, 10^{-9}	9.01	4.74	2.60	1.29	.493	.192	.0741
Number n of σ_{VT}'s	5.75	5.86	5.96	6.07	6.23	6.37	6.52
Est. min. V_{dd}: $\tilde{V}_{\text{min}}$, mV	230	195	179	162	166	170	174
Energy and entropy:							
Switching en. $E_{\text{sw}} = \frac{1}{2}CV^2$, aJ	132	59.9	51.9	29.5	21.6	13.5	10.4
Est. min. ent. $\tilde{S}_{\text{min}}$, knat	23.9	10.8	9.40	5.35	3.92	2.45	1.89
Inefficiency factor	385	174	151	86.0	63.0	39.4	30.4
Min. perm. energy loss, aJ	.903	.409	.354	.202	.148	.092	.071

Table 7.2: Calculations of minimum capacitance, supply voltage, energy dissipation, and entropy generation for irreversible CMOS, based on the SIA roadmap data from table 7.1. The minimum entropy generation per switching operation that is required given a 90% die yield ranges from 24 kilonats to 1.9 knats, which is greater than that required for a thermal reliability of less than 1 error in 10^{27} switching operations, by factors ranging from 385 in current technology, to 30 in projected technology for the year 2012.

---

calculations.

First, we perform some calculations to work towards finding out the load capacitance of typical small but non-trivial logic nodes (*e.g.*, NAND gates with output fanning out to 4 similar NAND gates) in each technology generation. This is important because we observed earlier that the lower bound on entropy generation is affected by load capacitance.

Estimated gate oxide thickness. We derive an estimate $\tilde{T}_{\text{ox}}$ for the gate oxide thickness in each technology generation by taking the average of the high and low values given by SIA, or the high value if no low value is given.

Gate capacitance per unit area. The dielectric constant of SiO_2 (see table 7.3) is ≈ 351 fF/cm. If we divide this by $\tilde{T}_{\text{ox}}$, we get per-area gate capacitances C_{ox} ranging

Symbol	Approximate value	Meaning
ϵ_0	8.85 aF/μm	Dielectric constant of vacuum
ϵ_{Si}	$11.7\epsilon_0 \approx 105$ aF/μm	Dielectric constant of silicon
ϵ_{ox}	$3.97\epsilon_0 \approx 35.1$ aF/μm	Dielectric constant of SiO_2

Table 7.3: Some important dielectric constants for semiconductor electronics calculations. Taken from the frontispieces of [84, 139].

from 7.81 fF/μm^2 up to 35.1 fF/μm^2 as the technology improves.

Minimum gate capacitance. The gate-to-channel capacitance C_{gmin} of a minimum-sized transistor can be calculated from C_{ox}, SIA's minimum gate length, and a minimum width which is assumed to be equal to the minimum length; we find values ranging from 312 aF in 1997 to only 43 aF in 2012.

Estimated load capacitance. In a minimum-size static CMOS NAND gate adjusted so that the effective gain factor k of pull-up and pull-down networks are equal to the minimal transistor gain factor in the worst case, n-FET and p-FETs will both be sized to about double the minimum width—the n-FETs because two of them are in series, the p-FETs because hole mobility is only about half of electron mobility. An input impinging on the NAND feeds to one transistor of each type, so the load placed on the input by the NAND is about 4 times the minimum gate capacitance, plus fringing capacitances which we will ignore. If the output of each NAND gate fans out to about 4 other NAND gates, then the total load capacitance on the NAND output is only about 16 times the minimum gate capacitance. There is also a contribution from wiring and from the source-drain regions of the gate generating the signal, but if wire lengths are kept short, this can be absorbed into the factor of 4 without decreasing fan-out very much. So based on this, we just multiply C_{gmin} by 16 to find an estimated load capacitance $\tilde{C}_{\mathrm{L}}$ ranging from about 5 femto-Farads in 1997, to 0.69 fF in 2012.

Transistors per die. From SIA's figures for transistor density and chip size, we can calculate the total number of transistors per chip. It ranges from 11 million up to 1.35 billion.

Transistors per defect. Suppose we require that the loss in die yield due to random threshold variations should be less than 10%. That is, less than one die in 10 should contain any transistor having a large enough error in its threshold value to cause the transistor not to be in the correct (on vs. off) state when required. Multiplying by the transistors per die gives us the number N of transistors that need be produced on average before one is produced that has such a defect. That is, in a set of N transistors, the expected number of defects should be 1. We might call this N the *process reliability number*. Values range from $\frac{1}{10}$ billion to 13.5 billion, as the number of transistors per chip increases in the SIA projections.

Defect probability. The sum of expectation values over a set of independent events is additive. So if the expected number of defects in N transistors is 1, and the statistical results of ion implantation in different transistors is independent (a plausible assumption), the expected number of defects in 1 transistor must be $1/N$. Thus the probability p that a given transistor will have a defect must be $1/N$. The required defect probabilities thus range from 9×10^{-9} to 74×10^{-12} as the transistor count increases.

Number of $\boldsymbol{\sigma}_{\mathrm{VT}}$s required. Roughly speaking, an n-FET in static CMOS will cause incorrect functionality either if it does not turn on when V_{GS} is V_{dd}, or if it does not turn off when V_{GS} is zero. Therefore variation in thresholds should leave the threshold within the range 0 V to V_{dd}. To minimize V_{dd} for a given level of threshold variability while remaining within reliability constraints, the nominal V_{Tn} should be exactly halfway between 0 V and V_{dd}, so that the transistor only fails if variation places the actual V_{Tn} far out on one of the tails of the threshold distribution. In this situation, if the total probability of V_{Tn} being out on the tail in both directions is p, then the probability for either side (too high or too low) is $p/2$, since these events are mutually exclusive. (Similarly for p-FETs.)

Given a probability of being out on one tail of at most $p/2$, we can compute a lower bound on how many σ_{VT}s are required before we are far enough out from the mean V_{T} so that the total probability of being that far out is no more than $p/2$. In a normal distribution, the total probability $\mathfrak{R}(n)$ of being at least n σs away from the mean in a particular direction is bounded above as

$$\mathfrak{R}(n) \leq \frac{1}{n} \cdot \frac{1}{\sqrt{2\pi}} e^{-n^2/2}. \tag{7.4}$$

And in fact, in the limit of $n \to \infty$, the probability out on the tail approaches this value exactly (*cf.* Feller 1950 [58], ch. VII, p. 175, eq. (1.8).)

Therefore, to have $\mathfrak{R}(n) \leq p/2$, we only require that n be greater than or equal to the value given by solving

$$2/p = \sqrt{2\pi} \cdot n e^{n^2/2}$$

for n, which for given p we can easily do numerically by computer using Newton's method. For our p's we find values of n ranging from 5.75 to 6.52 σ_{VT}s; this relatively narrow range is afforded by the roughly exponential decay of the tail of the normal distribution.

Minimum V_{dd}. Now we are finally in a position to actually calculate the minimum value of V_{dd} for each technology generation. With $V_{\mathrm{dd}} = 2|V_{\mathrm{T}}|$, we must have $V_{\mathrm{dd}} \geq 2n\sigma_{\mathrm{VT}}$ in order for the total probability of an error-inducing threshold defect (either too high or too low) to be less than p. Given SIA's values for $3\sigma_{\mathrm{VT}}$, this yields minimal V_{dd} voltages ranging between 230 mV and 162 mV.

These values are all several times greater than the thermal voltages of 26–34 mV found at normal operating temperature, so transistors that are turned off will conduct several times less current than ones that are turned on (in the worst case, we estimate, by at least a factor of 3), meaning that correct functionality is maintained, and leakage does not contribute very much to energy dissipation in circuits that switch frequently. However, if much lower levels of threshold variability than those given in the SIA roadmap were to be attained, low-temperature operation would be required in order for transistors to be able to turn off sufficiently at the implied lower voltage levels; this would tend to increase entropy generation, and thus reduce the advantages of the lower energy dissipation.

Minimum energy/switching event. Given the minimum capacitance of a useful logic node and a minimum swing voltage, we are now in a position to calculate the minimum switching energy $E_{\mathrm{sw}} = \frac{1}{2}CV^2$ for such a node.

Minimum entropy/switching event. The minimum attainable entropy is lower-bounded by the minimum energy divided by the maximum temperature. Raising the temperature has a variety of complicated effects on CMOS device behavior, so this bound will not be exact. But if we assume that operating temperatures can't be much higher than, say, 127°C (400 K) while preserving correct functionality, we can get a rough lower bound on entropy generation.

Inefficiency factor. Suppose we wish the probability of a thermally-induced error to be 10^{-27} (this would correspond to, for example, a billion logic nodes switching at 1 gigahertz with only 1 error expected per gigasecond (31 years) of operation). Then the number of nats of entropy generation per switching event only needs to be $\ln 10^{27} \approx 62.2$ in an ideal switching circuit. So our CMOS circuits are generating more entropy than the ideal switching circuits by factors ranging from 385 down to 30. So, thermal reliability does not become the limiting factor on voltages for the foreseeable future of irreversible CMOS, although if the technology could continue to be improved for several more generations beyond the 2012 technology, this might change.

This concludes our discussion of the minimum entropy generation per operation attainable in irreversible CMOS circuits. In summary, entropy generation per switching event is expected to be greater than 1.8 kilonats through at least the year 2012. Insofar as each logic gate operation involves about one switching event, this also corresponds roughly to the entropy generation per primitive operation.

Even if unforeseen technological breakthroughs were to undercut this limit, an entropy generation of at least *tens* of nats per operation is required in order for correct voltage-switching to occur with high reliability in *any* irreversible switching technology, due to the argument that led to eq. (7.2), p. 154.

7.1.4 Minimizing permanent energy dissipation in iCMOS

Note that the energy dissipated by a CMOS circuit internally is *not all lost*. If a circuit is maintained at a temperature T_{H} that is as high as allowed by reliability constraints, in order to minimize entropy production, then most of the energy dissipated internally can be recovered, by using the computer as the heat source for a Carnot-cycle heat engine (*cf.* [143], §19-6, p. 371–376) with a relatively low-temperature reservoir at temperature $T_{\mathrm{L}} \ll T_{\mathrm{H}}$. If the heat is *emitted* from the computer into the heat engine at temperature T_{H} (*i.e.* no temperature degradation during transport), then a fraction $(T_{\mathrm{H}} - T_{\mathrm{L}})/T_{\mathrm{H}}$ of this heat can be converted to work by the heat engine.

In other words, the total energy that is *really* lost when S entropy is generated is only ST_{L}. If the ~2.73 K cosmic microwave background can be used as the low-temperature reservoir, then theoretically only $\sim 4 \times 10^{-23}$ J of energy need be permanently lost for each bit of entropy generated in the computer, even though a hundred times more energy than this is temporarily "dissipated" whenever circuit nodes switch. This shows that energy dissipated in a circuit does *not* correspond to a permanent loss of work. It is only the *entropy* generation of the computer that truly determines the ultimate energy cost.

Thus, true energy loss is really minimized by minimizing entropy generation. If a 2.73 K reservoir is available, the minimum energy loss for CMOS ranges from about 0.9 aJ down to .07 aJ, as shown in the last line of table 7.2.

Actually, in practice there will always be some temperature degradation during the transport of heat from the transistors to the heat engine input, due to the non-infinite "heat capacities" of materials. However, if this temperature degradation is small compared to $T_{\mathrm{H}} - T_{\mathrm{L}}$, then the above results will be approximately correct.

Also, in practice, the 3 K microwave background may not actually be readily available as a thermal reservoir. In this case, the ultimate reservoir would be the atmosphere instead, and T_{L} would be on the order of $300K$, and the minimum energies would be $\sim$ 100 times higher.

7.1.5 Maximizing per-area processing rate for iCMOS

Consider a cost measure like in §6.2.2, consisting of the rental cost of the land area or floor space required for a computation, or in other words the surface area of the computer times the time taken by the computation. Suppose we wish to minimize that cost measure.

To do this, we would like to know how to maximize the rate of computation that can be achieved per unit of surface area. This rate is limited if there is an upper bound on the rate of entropy removal through the machine, and a lower bound on the entropy generated per operation.

To maximize the computation rate per unit of outer surface area in irreversible CMOS, one should just maximize the ratio between the maximum entropy flux F_{S} in the cooling system and the entropy generation per operation, then pack in enough layers of circuits below each unit of surface area so that entropy is being generated at a rate per unit area corresponding to F_{S}.

One key parameter to be chosen is the circuit operating temperature T. A lower operating temperature means more entropy generation for a given energy dissipation, but also more entropy flux for a given heat flux; these factors cancel out, meaning the relevant quantity is the ratio between the maximum heat flux and the minimum energy dissipation.

However, lower temperature probably means a lower heat flux, since at least some components of heat flow will increase with increased temperature differences between inside and outside. So within the bounds set by the reliability requirements at a given operating voltage, the machine should be operated as hot as possible. This is also consistent with minimizing the total entropy production, given the fixed lower bound on energy dissipation in CMOS, and also with minimizing the total permanent energy loss.

From SIA's figures for chip size and power, we can compute what heat flux they are assuming. If we take this as our maximum heat flux, we can combine this with our calculated figures for the minimum energy dissipation from table 7.2 to find the maximum rate of operation per unit area. See table 7.4.

What is the minimum thickness, in layers and in meters, for an irreversible CMOS machine that achieves this maximum speed per unit area? To determine this, we need to know both the minimum area per logic gate, minimum circuit layer thickness, and the maximum speed of operation of individual logic gates when operating at the minimum voltage.

The minimum area per logic gate is easy to calculate from the SIA figures, and an advanced wafer-thinning or SOI (silicon-on-insulator) process might be expected to achieve a thickness per circuit layer as low as $\sim 10\ \mu\mathrm{m}$ or less.

However, accurately determining the maximum speed of operation per gate requires a more detailed analysis. We would like to know an effective resistance R for our logic gates, when driven at our minimized power supply voltage, such that in a characteristic time $t_c \approx RC_{\mathrm{L}}$ the output node is charged most of the way to the desired voltage level.

	Year of first product shipment						
	1997	1999	2001	2003	2006	2009	2012
Rate calculations							
Heat flux F_{E}, W/cm^2	23.3	26.5	28.6	30.2	30.8	27.4	23.3
Max rate $\mathcal{R}_{\mathrm{pA}}$, ops/ns-$\mu$m^2	.882	2.20	2.76	5.13	7.09	10.1	11.2
Min $A\mathcal{T}$/op, ns μm^2	1.13	.454	.362	.195	.141	.0986	.0891

Table 7.4: Calculations of heat flux, rate of operation per outer surface area, and minimum outer-area rental cost per operation for (layered) irreversible CMOS, based on the SIA roadmap data from table 7.1. We find that irreversible CMOS can at best perform only ∼1–10 operations per nanosecond per square micron of outer area, given the heat flux implied by SIA's figures. In other words, the area-time cost per operation in ns μm^2 units ranges from ∼1 to ∼0.1.

Unfortunately, determining such an R is rather complex. First, the instantaneous resistance of each MOSFET is not constant during node charging; it varies as the drain voltage changes. In multi-input logic gates, some transistors will in general have varying source voltages as well. Moreover, the effective resistance of the active logic network (pull-up or pull-down) will generally be data-dependent. For example, the pull-up network of a NAND gate, which consists of two transistors in parallel, conducts best if both inputs are low, rather than just one being low. Finally, the instantaneous resistance depends on the supply voltage in a complex way that depends on threshold voltage and source voltage, and that in small devices is affected by a variety of difficult-to-model short-channel effects, such as channel-length modulation, velocity saturation, mobility degradation, and drain-induced barrier lowering (*cf.* [165] and §2.3 of [139]).

We carried through a rough computation by hand (with help from the `Emacs calc` tool) based on a model described in (Rabaey 1996, [139], §2.3, p. 54, eqs. (2.57)–(2.59)), and originally proposed by Toh *et al.* (1988, [165]), which incorporates threshold voltage, mobility degradation, and velocity saturation effects. At this point we are still not completely confident in the accuracy of that calculation, so we will not detail it here. But one tentative result is that when operating at the minimum supply voltages we derived earlier, we end up with a maximum operation frequency that is only slightly higher than those projected by SIA (it is within a factor of 2), although in our analysis, the maximum operation frequency increases less rapidly than in SIA's, as technology improves. The reason for the remaining discrepancy between our calculations and SIA's is unclear, and will probably remain so until we obtain a fuller description of SIA's assumptions and analysis methods.

Another tentative result of this analysis is that for maximizing total computation rate per unit area, our choice of lower supply voltage results in an overall speedup factor (compared to SIA's projections) that ranges from 127 down to 2.5 as technology improves; part of the reason why the improvement decreases seems to be that SIA's choice of supply voltage converges from 10 times ours down to only 3 times ours, as the generations progress.

Unfortunately, the low-voltage approach is not significantly more cost-effective (in terms of rate per unit of silicon surface) than SIA's. This is not surprising, since in these calculation we were not trying to maximize overall cost-effectiveness, but rather only performance per unit of outer area; we were ignoring the material cost of stacking up more layers of surface over that area. The number of circuit layers we need for optimal per-area performance ranges from 80 (in 1997) down to 4 (in 2012), and the cost of these extra layers roughly negates the benefit of the increased performance.

7.1.6 Maximizing iCMOS cost-efficiency

When maximizing cost-efficiency in terms of circuit mass or wafer area, rather than outermost-surface area, the analysis becomes more complex. For one thing, it depends on the nature of the computation to be performed.

If the computation consists of many small independent computations, requiring only local communication in 2-D, for example, then the circuitry can be spread out in a single layer, and the task is to maximize the rate per unit of silicon area. In this case, we are not entropy-limited, so we cannot assume, as we did in the previous section, that the optimal operating voltage is just the minimum voltage that is consistent with reliability constraints. Lower signal voltages in general increase the effective resistance of transistors, and lead to longer charging times $t \sim RC$. But the voltage cannot be too large either, or it will cause oxide breakdown and other undesirable effects, and it could possibly cause overheating even if the circuit is just a single layer. An accurate analysis would need to take these concerns into account, as well as all of the complex short-channel effects that arise when scaling to smaller device sizes. Optimizing voltage in the face of all these concerns cannot be done via solving a simple analytical equation; instead one must write a program to search for the optimum point numerically.

Alternatively, if the computation requires frequent communication, such as for example between neighboring cells in a 3-D mesh being simulated, then the analysis is made complex for other reasons, namely because we cannot spread everything out in a single layer without incurring communication delays, as we discussed in ch. 6, §6.2.3.1. If a near-ballistic means for communication is available—such as an optical or transmission-line system of interconnects between processors—then when entropy generation becomes the dominant concern, the optimal structure is the one from that earlier section, in which we lower the clock speed and spread the processing elements out in proportion to the cube root of the logical diameter of the communication network.

But in this case as well, carrying out the relevant calculations for future generations of semiconductor technology would be complex and error-prone, due especially to uncertainties in the technical specifications of the communication network.

In both cases (2-D and 3-D computations), we will deem a detailed numeric calculation of cost-efficiency to be beyond the scope of this thesis, and we will relegate it to the domain of future work. Still, such a calculation will be important eventually, if we wish to be able to compute the exact scale above which reversibility becomes advantageous, in each succeeding technology generation. In the current technology generation, a very rough hand-calculation suggests that even in the most optimistic scenario for reversible computing (namely, ballistic communication between nodes at logical distance $\sqrt{N_{\mathrm{D}}}$, see §6.2.3.2, p. 136), reversibility doesn't improve cost-efficiency until we reach a cost level on the order of \$25 billion. Later we will argue that over time, the case for reversibility ought to improve, for as long as the RC of CMOS technology keeps improving; however, we will also argue that at some point, making further RC improvements will require moving to a radically different technology, such as the superconducting Josephson-junction technology of Likharev [108]. In any case, making more accurate projections of the advantages of reversibility in foreseeable generations of CMOS technology would be very desirable in order to gauge the near-term applicability of this research.

This concludes our analysis of the best possible performance of normal *irreversible* CMOS circuits under various efficiency measures for the foreseeable future. This can serve as a baseline for comparison when looking at reversible circuits.

Now, let us see how reversible circuit techniques came about.

7.2 Historical development of adiabatic circuits

We now review the historical development of reversible logic circuit techniques.

Correcting an attribution. Toffoli (1980, [161]) suggested that the idea of dissipationless computing using reversible circuits originated with John von Neumann, but I have been unable to confirm this claim. To quote Toffoli's paper:

> The idea that universal computing capabilities could be obtained from reversible, dissipationless (and, of course, nonlinear) physical circuits apparently first occurred to von Neumann, as reported in a posthumous paper (Wigington 1961 [188]).

However, based on a careful reading of Wigington's paper, I believe that Toffoli's characterization of it is incorrect.

Wigington's paper [188] is an explanation of a patent [181] submitted by von Neumann in 1954 and posthumously granted to him in 1957. The paper describes a computing scheme in which logic values are represented by the relative phase of AC signals, rather than by the DC voltage level used in conventional systems. Wigington also cites similar work (on "Parametron" circuits) that apparently occurred around the same time in Japan.

The logic circuits that Wigington discusses do indeed include some elements (namely, nonlinear reactances) that can be assumed to have arbitrarily low dissipation (and whose operation is therefore physically reversible), but the described circuits also include attenuators that are placed in the signal paths *explicitly* in order to dissipate the energy of signals whose phase information is no longer needed, in exact analogy to the practice in traditional DC logic circuits of dumping a node's static energy through a dissipative switch whenever its logic value is no longer needed. Without these attenuators, von Neumann's circuits would not reliably implement the computing scheme described. In fact, the need for dissipation is guaranteed by the logic system used. The fundamental logical operation in von Neumann's AC circuits is to take three binary input signals, and from them generate a boolean output signal whose logic value is the majority value of the inputs. The input signals are consumed in the process, in the attenuators. Since the resulting operation is a many-to-one transformation of the logical state of the circuit, it destroys logical information, and thus *cannot* be implemented in a physically reversible way, by Landauer's argument (§2.5.1, p. 41). No matter how we try to modify von Neumann's circuit, we will fail to achieve dissipationless, reversible operation, so long as the logical state transformation operation that is performed is a many-to-one operation such as Wigington describes.

Therefore, Toffoli's characterization of the von Neumann/Wigington paper seems mistaken. Von Neumann had certainly had a great many original ideas during his lifetime, and it is conceivable that he thought about the idea of reversible, dissipationless computing, but if so, the Wigington paper certainly provides no evidence to support that attribution.

Earliest adiabatic circuits. To our knowledge, the first description of a logic circuit technique that seriously attempts to avoid the $\sim CV^2$ dissipation associated with conventional logic is that of Watkins 1967 [186]. Watkins describes a technique whereby capacitive loads in a circuit are charged gradually through the control transistors from a power supply whose voltage fluctuates cyclically according to prescribed waveforms. When the transistors in Watkins' circuit are first turned on, there is no voltage across them and thus no dissipation through them. While the load is being charged up through the transistor, there is a small voltage across it, and thus some dissipation, but this dissipation can be made as low as desired by just lengthening the time taken in the charging cycle. Watkins analyzes the energy dissipation in his circuit, and predicts that the energy per cycle asymptotically approaches zero as the cycle time is increased.

Unfortunately, Watkins' energy analysis appears not to have been quite correct, due to his use of semiconductor diodes to discharge the nodes in his circuit. Such diodes have an intrinsic voltage drop ϕ_T across them (*cf.*[94] §2 and [139] §2.2.1, p. 20) which does not decrease as the circuit is run more slowly. Therefore, Watkins' circuits still incur a minimum dissipation of $\sim CV\phi_T$ per operation, no matter how slowly they are run. Therefore, they do not qualify as truly *time-proportionately reversible* (see §6.1.3, p. 123) logic devices that could improve asymptotic cost-efficiency as per the arguments in chapter 6. In any case, for whatever reason, Watkins' proposal appears to have faded into obscurity.[1]

Inductor-based approaches. After Watkins, we next see the idea of dissipationless electronic logic crop up independently in a 1978 proposal by Fredkin and Toffoli [73]. In their idea, energy is shuttled around between inductors and capacitors, but is not dissipated substantially, in a circuit that implements a purely reversible primitive operation, namely a 3-input 3-output Fredkin gate. The assertion is that in such a circuit, energy dissipation can be made arbitrarily small if only the quality factor Q of the LC elements can be made arbitrarily large. Unfortunately, the Fredkin-Toffoli approach was not immediately practical, because it appeared to require large numbers of inductors, roughly one for each logic element, whereas VLSI fabrication technology does not well support high-quality integrated inductances.

However, in 1985 the ball was again picked up by Charles Seitz and colleagues [144] at Caltech, who (apparently independently) describe a logic technique similar to Fredkin and Toffoli's, but in which the inductances are instead shared between many logic circuits, and are brought off-chip, and can therefore be implemented using a technology that is more optimized for providing high-Q resonance than is VLSI. The relatively complex switching circuitry that controls the logical operation of the circuit remains integrated on-chip. This was a key step on the road to making resonant circuits practical. However, Seitz *et al.* only worked out the technique in detail for relatively simple circuits; they leave open the question of whether a general logic family could be worked out to implement *any* combinational or sequential logic with arbitrarily little dissipation. Their proposed solution is, in their own words, not "foolproof" and requires careful tuning of the circuit parameters to ensure correct and dissipationless operation.

---

[1] According to *Science Citation Index*, Watkins' article [186] was only cited a total of four times through 1976, and these citations appear to all be from general review articles, rather than applications of Watkins' research. After 1976, Watkins was not cited at all until Bill Athas and colleagues rediscovered his work in 1997 [4].

Improvement of adiabatic techniques. In 1992 and 1993, a sequence of several important developments proceeded to solve the remaining difficulties with adiabatic switching. First, Koller and Athas [92] devised a simple and general adiabatic logic family, but encountered difficulties because their circuits were not fully logically reversible; Koller and Athas noted that whenever their circuits finally needed to forget some information, they were unable to avoid $\sim CV^2$ dissipation, because ultimately, the only way to clear the logical state of a circuit node whose state is unknown is to tie that node to a reference reservoir at a known voltage level, thereby dissipating the energy of the node (if different from the target level). Being unaware of Fredkin and Toffoli's earlier work showing that sequential circuits need never discard information, Koller and Athas did not know any way around this problem, and went so far as to conjecture (incorrectly) that any sequential logic circuits (*i.e.*, circuits containing feedback loops) would require dissipation.

In the meantime, some fully reversible circuit techniques were discovered independently by nanotechnology enthusiasts Hall [79, 80] and Merkle [122, 123, 125]. However, Merkle did not discuss how to implement sequential circuits, and Hall's technique was essentially non-sequential, and thus algorithmically inefficient. It required saving all intermediate results in hardware, using as many clock rails as there were stages in the computation, then reversing the whole process to recover energy before beginning the next sequential stage. In between stages, an irreversible write of the results of the previous stage was required.

Fully adiabatic CMOS techniques. The first adiabatic circuit technique to put together all the key elements needed for fully universal adiabatic computing was the CRL ("Charge Recovery Logic") technique of Younis and Knight, developed in our research group in 1993 [192]. Like the original Fredkin-Toffoli technique, CRL could implement arbitrary sequential logic. Like Seitz's technique, CRL took advantage of an off-chip resonant element. Like the Koller-Athas technique, it did not require fine tuning of the circuit elements. Putting together these three elements, the Younis-Knight technique provided the first practically implementable circuit style capable of reliable, asymptotically reversible operation.

The initial version of CRL was somewhat baroque, but it was later refined to another version (called SCRL, for "split-level CRL") that was relatively clean and simple [193, 191, 89]. SCRL was used as the basis for all of the reversible circuit design work that we will describe later in this chapter. In section 7.5, we review SCRL in detail.

Recent adiabatic circuits research. Following 1993, there has been a small explosion of literature on and relating to adiabatic circuits of various types. The literature has become too large to review in detail here, but for bibliographical completeness, we include a sampling of some relevant citations: [5, 44, 93, 83, 154, 153, 94, 157, 171, 6, 4].

Patents on adiabatic circuits Additionally, a search for recent patents relating to adiabatic and reversible computing turned up the following patents from 1995 through 1997: [30, 48, 61, 62, 63, 89, 125, 131, 142, 190, 159]. Most of these patents were assigned to large organizations such as IBM, MIT, AT&T, Motorola, and Xerox.

7.3 A comment on terminology

We'd like to pause briefly here for a comment on the term "adiabatic circuits" itself. Originally in thermodynamics, the word *adiabatic* is an adjective literally meaning "without flow of heat" into or out of the system (*cf.* [143], §18-5, p. 352). So, for example, one way to adiabatically compress a gas would be to compress it inside an well-insulated chamber so that the heat produced cannot escape. Such a compression can be thermodynamically reversible: the gas can be allowed to adiabatically re-expand, pushing back against the piston that compressed it while cooling, and the work that was originally applied to compress the gas can be recovered.

As a result of its frequent usage in such contexts, the term "adiabatic" in applied physics has gradually evolved to the point where it is frequently used to refer not to the lack of heat flow precisely, but rather to the overall *thermodynamic reversibility* (or near-reversibility) of a process. Any process that is thermodynamically reversible (at least in the low-speed limit) has come to acquire the moniker "adiabatic."

Note that this new usage is completely orthogonal to the literal meaning of adiabatic, "no heat flow." A process can involve no heat flow into or out of the system, yet be thermodynamically irreversible: for example, when a partition is removed to allow two different gases originally separated in different chambers to mix together. Conversely, a process can involve heat flow, yet be reversible: for example, if the heat is contained within an insulated box which is physically moved via a reversible mechanism out of the region of space identified as "the system."

A more accurate term for thermodynamically reversible processes might be *isentropic* (literally, "with the same entropy"), since thermodynamically reversible processes are, by definition, those processes that generate no new entropy. However, even this term is not precisely applicable to the circuits referred to as "adiabatic circuits," because the circuits are not *perfectly* isentropic except in the limit of zero clock frequency and low temperature (to stem the flow of leakage currents). The phrase "*asymptotically* isentropic" would therefore be a bit more accurate.

However, some of the circuits that have been referred to as "adiabatic" are not even *asymptotically* isentropic, due to the use of diodes with a built-in voltage drop.

Essentially, the term "adiabatic circuit" is so ill-conceived, and so polluted with inaccurate and inconsistent usage that we wish that it could be dropped altogether. To avoid confusion, we would like to advocate the adoption of the following alternative, more accurate lexicon:

- *energy recovery circuits* (ER circuits) — These are circuits that are designed to recover a substantial portion (but not necessarily all) of the energy invested in logic signals (*e.g.* CV^2 static energy). This could include diode approaches.
- *asymptotically isentropic circuits* (AI circuits) — These are ER circuits that, in some appropriate limit (such as low speed and/or low temperature) can generate asymptotically zero entropy per operation. Example: SCRL.
- *time-proportionally reversible circuits* (TPR circuits) — AI circuits in which entropy generation per operation is approximately inversely proportional to the length of time over which operations are performed. Example: SCRL when operated in a regime where leakage currents are small.
- *ballistic circuits* — Hypothetical TPR circuits in which the entropy coefficient is so low that the entropy generation per operation is zero for all practical purposes, even when the circuit is running at its maximum rate. Superconducting technologies would probably be required for this.

However, abolishing an established bit of terminology is, in general, a difficult thing; it confuses people who are accustomed to the old terminology, and complicates keyword searches for material on a given topic. Therefore, despite our academic objections, we bend to popular usage and continue to use the term "adiabatic" when we are referring generally to circuits of any of the above types. When we wish to be more precise than this, we will use one of our more precise terms.

7.4 Basic principles of adiabatic circuits

The core insight behind all adiabatic circuits is that the $\sim CV^2$ minimum dissipation in ordinary switching circuits is due primarily to the fact that such circuits charge a node by connecting it to a *constant voltage* power supply (*cf.* the discussion in §7.1.1.1, p. 150).

Constant current sources. One alternative means that one might think of for charging up a capacitive load is to instead use a *constant current* power supply, operating at some appropriate current over some desired length of time. See figure 7.3. One may assume that the charging pathway has some effective resistance R.

We can analyze the dissipation in this case as follows. Let C be the load capacitance, V the voltage swing, R the resistance in the charging pathway, and t the time

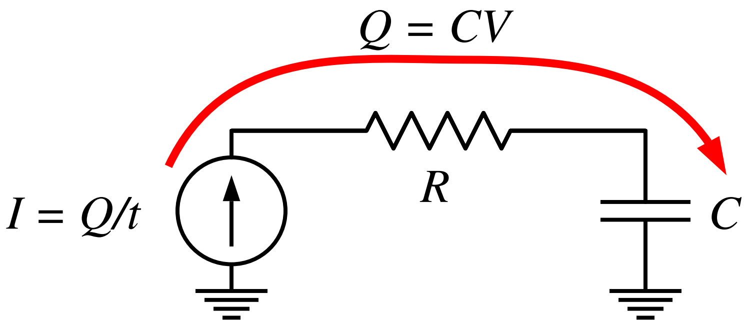


Figure 7.3: Charging a node to voltage V via a constant current over a time t. Compare with fig. 7.2. In this case, the dissipation is not $\frac{1}{2}CV^2$, but rather $CV^2\,\frac{RC}{t}$, which becomes arbitrarily small as the charging time t is increased.

---

we wish to take to charge the node. Then the charge delivered is $Q = CV$, the current should be $I = Q/t$, and the energy dissipated in the circuit is

$$\begin{aligned} E_{\mathrm{diss}} &= \int p\,d\tau = \int i^2 R\,d\tau = I^2 Rt = \left(\frac{Q}{t}\right)^2 Rt \\ &= \frac{Q^2}{t}R = \frac{(CV)^2}{t}R = \boxed{CV^2\,\frac{RC}{t}}. \end{aligned}$$

Note that this dissipation scales down proportionally as the charging time increases. Therefore this particular charging process is an example of what we call a *time-proportionately reversible* process.

Voltage ramps. Can this constant-current procedure be used when charging nodes in a CMOS logic circuit? Well, it can at least be closely approximated, by using a turned-on transistor in place of the resistor, and using a power supply with a linear voltage ramp in place of the constant current source. (See figure 7.4.) An exact analysis of the energy dissipation in this circuit is more complex, but it can be shown to approach that of the constant-current circuit very closely when $t \gg RC$. At the opposite extreme, when $t \ll RC$, the dissipation approaches that of an ordinary constant-supply-voltage switching circuit as in §7.1.1.1. See fig. 7.5, and see Younis [191] for a more detailed discussion.

The same basic technique can also be used to discharge a logic node, with the supply voltage ramping the other way, from the logic level V back down to 0.

Note however that these low-dissipation characteristics are only maintained as long as we charge and discharge all nodes *only* using this technique. If, on the other hand we ever turn on a transistor when there is a voltage difference across it, there

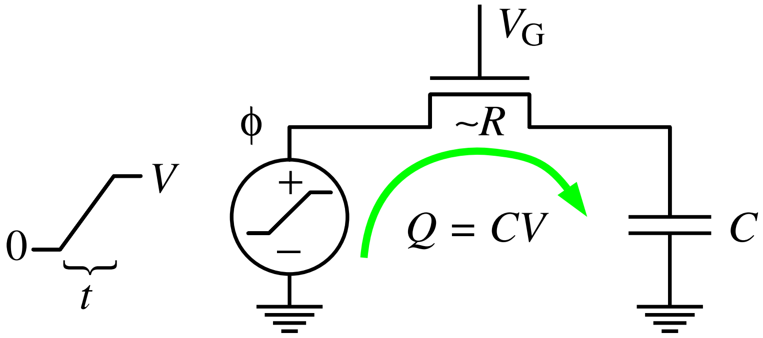


Figure 7.4: Compare with figure 7.3. Instead of an ideal current source, we have power supply that provides a voltage signal ϕ that ramps up from 0 to V over a time t. Instead of an ideal resistor, we have a turned-on CMOS transistor, with gate voltage biased at some value $V_G > V + V_T$ that allows the transistor to conduct well over the entire voltage range from 0 to V, with an approximate effective resistance of R. For $t \gg RC$, $E_{\text{diss}} \approx CV^2 \frac{RC}{t}$; for $t \ll RC$, $E_{\text{diss}} \approx \frac{1}{2}CV^2$.

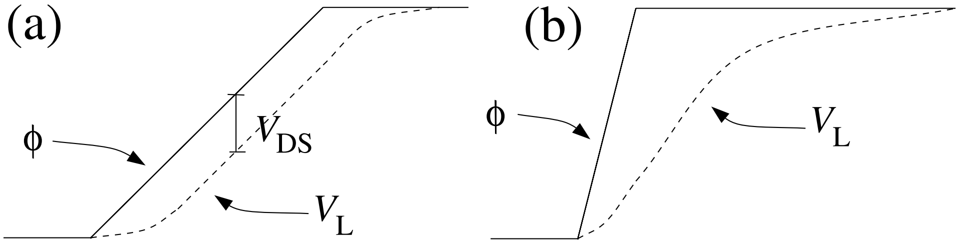


Figure 7.5: Voltage curves for slow and fast adiabatic charging using a voltage ramp. If the supply signal ϕ ramps up much faster than the characteristic RC time of the circuit, then the load voltage V_L will lag behind the ramp and approach the supply level in a characteristic exponential-decay curve. When ϕ reaches its peak, the voltage difference across the transistor is still almost the full swing V, leading to the dissipation being almost $\frac{1}{2}CV^2$.

On the other hand, if ϕ ramps up very slowly, V_L will track it, with only a small lag V_{DS} that is determined by the rise time, the transistor's characteristic transconductance k, and the drive voltage $V_{dr} = V_G - V_T$.

will be a $\frac{1}{2}CV^2$ dissipation, as in any switching circuit. So in a fully adiabatic logic circuit, we need the rule that a transistor can only be turned on if is no voltage difference across it. In truly asymptotically isentropic circuits, this constraint leads to the consequence that logical information cannot be thrown away—since essentially, in these circuits, the only operation that throws away information is to dissipatively connect a node to another one at a different voltage level.

One might object that in this technique we are merely moving the dissipation from inside the circuit to the power supply which must generate this swinging logic signal. But, as we will see later, there are a number of ways to generate the necessary signal using a *resonant* element, in which circuit energy oscillates back and forth between the on-chip capacitance and an off-chip inductance. If the resonant frequency is low and the off-chip elements have a high quality index, the off-chip elements need not dissipate significant energy either.

The above discussion outlines the basic principles of adiabatic circuits, but does not get into the details of how build complex logic circuits using those principles. There are now a number of different adiabatic logic techniques available for doing this. In the next section we review our technique of choice: SCRL.

7.5 The SCRL technique

As we said earlier, SCRL was the first adiabatic circuit technique to simultaneously be capable of (1) asymptotically approaching true zero energy per operation, (2) being integrated on a large scale using standard CMOS process technology, and (3) operating in pipelined, sequential fashion. In this section, we review in detail the operation of SCRL, and show some new graphical depictions that we find helpful for understanding its structure and operation.

7.5.1 Basic SCRL components

We start by reviewing the basic elements of which SCRL circuits are composed.

Note: The following description breaks down SCRL circuits into functional elements in a slightly different way than in the original work of Younis and Knight [193, 191]. We find our alternative decomposition a bit simpler to explain.

In our version, SCRL circuits are separated into two types of components: (1) clock-driven generalized inverters, and (2) bidirectional latches.

7.5.1.1 SCRL clocked generalized inverters

A clocked SCRL inverter is illustrated in figure 7.6. It is very simple, composed merely of two MOSFET transistors, one n-type and one p-type. In fact, it has exactly the same internal structure as an inverter in ordinary static CMOS (see fig. 7.1 in §7.1.1), but it is wired and used slightly differently. Rather than being connected to constant-voltage supply rails, the FETs are connected to a swinging, clocked power supply. The n-FET is connected to a clock-supply signal $\overline{\phi}$ that swings between 0 and $V_{\rm dd}/2$—that is, half of the full signal voltage—with a particular waveform. The p-FET is connected to a clock ϕ that swings between $V_{\rm dd}/2$ and $V_{\rm dd}$. We assume $V_{\rm Tn} \approx V_{\rm Tp}$, and the voltage swing $V_{\rm dd}$ itself is chosen to be more than twice $V_{\rm T}$, so that the transistors will conduct bi-directionally through the entire swing range of the clocks to which they are respectively connected.

The operation of the device is as follows. Initially, all circuit nodes are at the "neutral" level $V_{\rm dd}/2$, representing "no information." Being enhancement-mode devices, both transistors are nonconducting at this time. Then, the input voltage $V_{\rm in}$ swings to a level 0 or $V_{\rm dd}$, representing binary 0 or 1, as in conventional logic. At this point, one of the two transistors becomes conducting (n-FET on input 1, p-FET on input 0), but no current flows because $\phi = V_{\rm out} = \overline{\phi}$.

Next, the rails ϕ and $\overline{\phi}$ swing simultaneously, in a roughly linear ramp taking some non-infinitesimal rise time $t_{\rm r}$, to their respective extremes. (They are said to "split,", thus the S in SCRL.) The output level $V_{\rm out}$ will track the rail to which it is connected. At this point the output is considered valid, and its value can, for example, be sampled by a latch (which we will get to in a moment) for later use. In the meantime, $V_{\rm in}$ must remain (roughly) constant at its 0 or 1 logic level—this is crucial for preventing dissipation.

At some point after the subsequent stages have finished using the $V_{\rm out}$ signal, the supply rails are brought back together to $V_{\rm dd}/2$. Again, $V_{\rm out}$ tracks the rail to which it has remained connected this whole time. Then, the input $V_{\rm in}$ is free to return to the neutral level, turning off both transistors again.

Of course, as in ordinary static CMOS, this SCRL inverter structure can be generalized to compute any n-input inverting logic function, such as NAND or NOR, by simply replacing the p-FET and n-FET with complementary networks of p-FETs and n-FETs, respectively. (See figure 7.7.) The operation of such gates is essentially the same as that of the simple inverter. All internal nodes are initially at $V_{\rm dd}/2$, and when the rails are split, all nodes that are connected to one or the other rail track it, at least up to a threshold drop away from the extreme point. (No node can be connected to both rails if the networks are properly complementary.) In general, all n inputs must be held constant during the entire time that the rails are non-neutral. Then, when the rails re-merge, all nodes that were pulled away from the neutral level

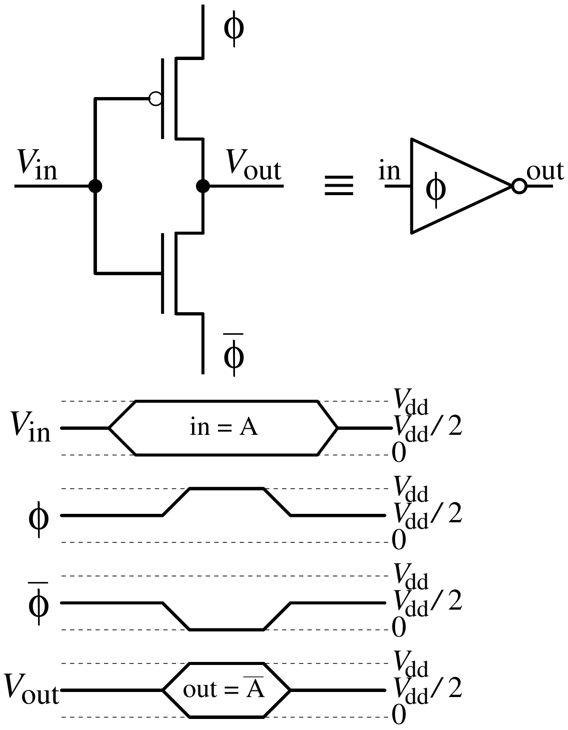


Figure 7.6: An SCRL inverter. It has the same structure as an ordinary static CMOS inverter (fig. 7.1a), except that the supply rails are tied to swinging clock signals rather than to constant-voltage supplies, and there are various additional assumptions and constraints on circuit operation. In between cycles, all nodes are at $V_{\rm dd}/2$. The input $V_{\rm in}$ is assumed to swing to either 0 or $V_{\rm dd}$ (representing a logic 0 or 1 as usual) during some period, and for a shorter period enclosed within this, the clock rails ϕ and its logical inverse $\overline{\phi}$ swing at a constant rate from $V_{\rm dd}/2$ to $V_{\rm dd}$ and 0, respectively, remain in this state for a time, and then return smoothly to the neutral level $V_{\rm dd}/2$. (See the timing diagram in the bottom half of the figure.) During this time, the output is valid and can be sampled by a latch (fig. 7.8) to be used in further stages of processing. Note that the input must not change while there is a voltage across either transistor: this is the key property that avoids CV^2 dissipation. On the right is an icon convenient for representing this element.

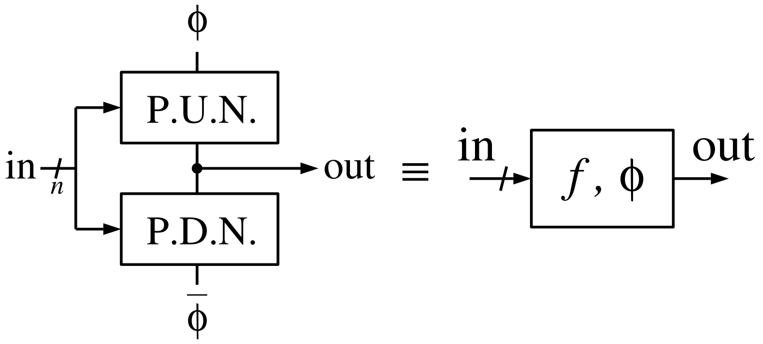


Figure 7.7: Generalized SCRL inverter. As in ordinary static CMOS, the SCRL inverter can be generalized to compute any inverting logic function f (e.g., NAND, NOR) of n inputs by simply replacing the single p-FET with an arbitrary pull-up network of p-FETs, and the n-FET with the complementary network of n-FETs. As in ordinary CMOS, it is best not to make the logic gates have *too* many inputs, due to the impact on conductance if there are many transistors in series in the pullup/pulldown network. Lower conductance decreases speed in static CMOS, and in SCRL, it also increases the entropy coefficient and the minimum energy dissipation. To imitate the conductance of an inverter, the transistors in a multi-input gate can be made wider, but this of course consumes more wafer surface area.

are gradually pulled back.

A warning: Associated with internal nodes in the pullup/pulldown networks of a generalized SCRL inverter, there may be a component of dissipation that does not scale down with frequency. Fortunately, as we will see in §7.6.4, this problem is easy to fix.

7.5.1.2 SCRL bidirectional latches

Given only generalized SCRL inverters, and no other components, one could potentially proceed to create combinational logic of any desired depth, by using a different pair of clock signals for each level of logic, and having the clocks for earlier stages split before the clocks for the later stages do, and re-merge after the clocks for the later stages do. This would be similar to the "retractile cascade" approach of Hall [79, 80]. But the problems with this simple approach are that (1) it would need as many pairs of clock rails as there are stages in the logic, and (2) the earlier stages must remain idle while the later stages are computing, and therefore there can be no pipelining, and no sequential circuits with feedback.

To solve these problems, SCRL introduces an additional component in between logic stages, something we call a "bidirectional latch" (fig. 7.8). Through a "forward"

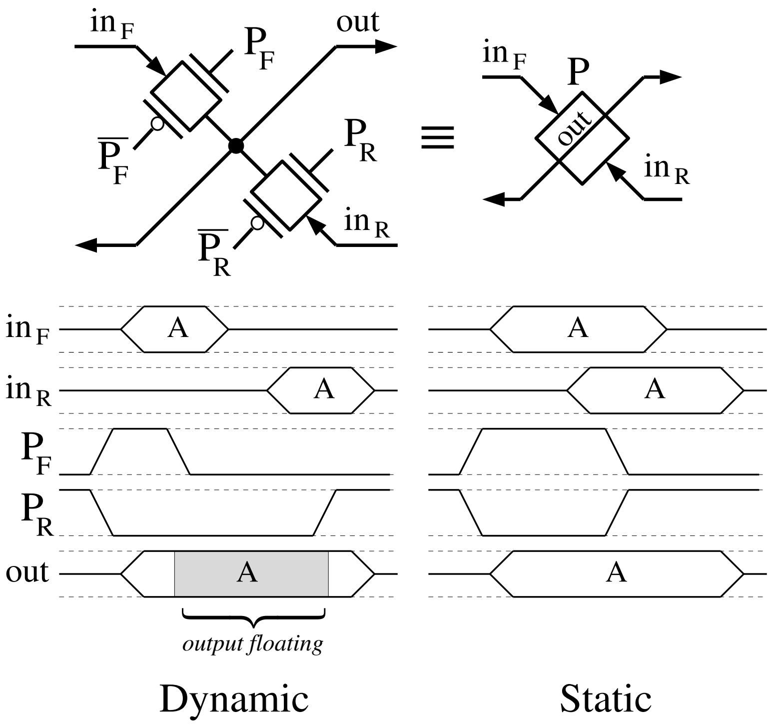


Figure 7.8: SCRL bidirectional latch. This special circuit element, composed of 2 CMOS pass gates with appropriately clocked controls, is essential for being able to pipeline SCRL logic stages and to make arbitrary sequential (as opposed to combinational) circuits of any depth using only a constant number of clock phases. An icon for the element is shown on the upper right.

Initially, nodes in_{F}, in_{R}, and *out* are all neutral. The "forward" F pass gate turns on, and the "reverse" R pass gate turns off. Input signal in_{F} goes valid with a logic value A, driving the output line through gate F. Before the input signal goes neutral, gate F closes, so that the output signal will continue to remain valid after the input goes neutral. Meanwhile, a later stage of the computation is reconstructing the value A, and presents it again on input in_{R}. After this happens, gate R opens, tying the output node to in_{R} which is at the same level, so there is no dissipation. Then in_{R} goes neutral gradually, drawing *out* back to the neutral level. The latch is now ready to process another input on in_{F}.

Depending on the relative timing and the presence/absence of overlap of the in_{F} and in_{R} signals, the latch may operate in either a dynamic mode (bottom left), or in a static mode (bottom right).

pass gate “F,” the latch is adiabatically drawn from its initial neutral level to the logic level in_{F} produced by the preceding logic stage. The pass gate shuts off, and then the latch holds its value, allowing the preceding stage to reset and prepare to accept a new input, while in the meantime the succeeding logic stage uses the value held on the latch as its own input for further computations. But after the succeeding logic stage finishes, there is a small problem: How do we clear the latch to accept a new input from the preceding stage? We cannot just dump the latch to a constant voltage because that would be irreversible and dissipative. Instead, the latch must be discharged adiabatically by a controlling component that knows what level to discharge it from. The *preceding* logic stage no longer knows what value the latch is holding, because it has already gone on to reset itself and process new data (allowing this was the whole point of the latch). However, the key insight is that the *succeeding* logic stage now contains information that depends on the latched value. If that succeeding logic stage has computed some *invertible* function of its inputs, then the value in the latch can be reconstructed based on the information that the succeeding stage has calculated, and using this knowledge, the latch can be reversibly cleared.

To provide this adiabatic “unwriting” functionality, the latch provides a second write port, in the form of a second “reverse” pass gate “R”. Logic in the succeeding stage presents a reconstructed copy of the latched value on input in_{R}, then the reverse pass gate opens, and is drawn back to the neutral level through pass gate R. Also, the latch provides a second “read port” in the form of a wire of the output node leading back to the preceding stage, which uses this input to reconstruct and clear the values stored in the preceding level of latches.

Two different alternative timing disciplines for these latches are shown in the bottom half of fig. 7.8. Note that the pass gates are turned off and on adiabatically by gradually-swinging ramps, and that they are never turned on when there is a voltage across them.

7.5.2 SCRL pipelines

Putting it all together, figure 7.9 shows the structure of a complete SCRL pipeline. The arrows in this figure represent many parallel wires, each function block represents a parallel set of logic gates all using the same clock, and each bidirectional-latch icon represents a parallel set of bidirectional latches, all on the same clock. The direction of the arrows shows the direction of information flow. As you can see, each set of “forward” logic gates that computes a logic function is paired with a corresponding set of gates, pointing the other way, which is used to uncompute the latched values from the previous logic stage. Each forward or reverse stage, and each latch, operates on a different set of clock signals. However, after some small number of stages, the earlier clock signals may be used again. This allows arbitrary sequential circuits with feedback loops (such as CPUs) to be constructed.

Normally, each logic stage can only compute an inverting function, and so there is a potential difficulty that if one initially has a value but not its complement, one cannot, in a single later stage, have access both to the value and its complement. This difficulty can be fixed by having a 2-level retractile cascade within each stage of logic, as illustrated in the bottom part of fig. 7.9. An alternative way to fix the problem would be to maintain a dual-rail signaling discipline, with complements of every logic value always available, but this would in general require more area.

7.5.3 Timing disciplines

There are a variety of alternative timing disciplines in SCRL. These vary in terms of the number of clock phases, and whether they are dynamic or fully static. The simplest fully-static discipline is 3-phase; the timing diagram for this is illustrated in fig. 7.10. If one wishes to permit dynamic operation (floating nodes), 2 phases will suffice.

We used 3-phase SCRL in our designs, because when running at very slow clock speeds, dynamic circuits would have been vulnerable to incorrect functionality, because (at normal temperatures) the charge stored capacitively on a dynamic node may leak out over long periods. With fully static circuits, we could be more confident that functionality would remain correct even when running at the very slow speeds that minimize energy dissipation, speeds at which the switching currents become nearly as small as the leakage currents.

For more detailed descriptions of the various timing disciplines see Younis 1994 [191].

7.6 SCRL circuit analyses

In this section, we carry through a variety of CMOS circuit analyses in order to better understand interesting aspects of SCRL's scaling behavior. First, §7.6.1 presents a simplified model of SCRL that we will use in our analyses. Then in §7.6.2, we derive an expression for the switching energy dissipation in SCRL in any given technology, in terms of raw characteristics of the technology, such as the threshold voltages and transconductance parameters of its transistors. In §7.6.3, we extend this by taking leakage currents into account, and derive analytical expressions showing how to adjust speed and threshold voltage to minimize total dissipation in SCRL at a given temperature. Those analyses are based on a fairly simple model of CMOS transistors, which becomes somewhat inaccurate in very small devices.

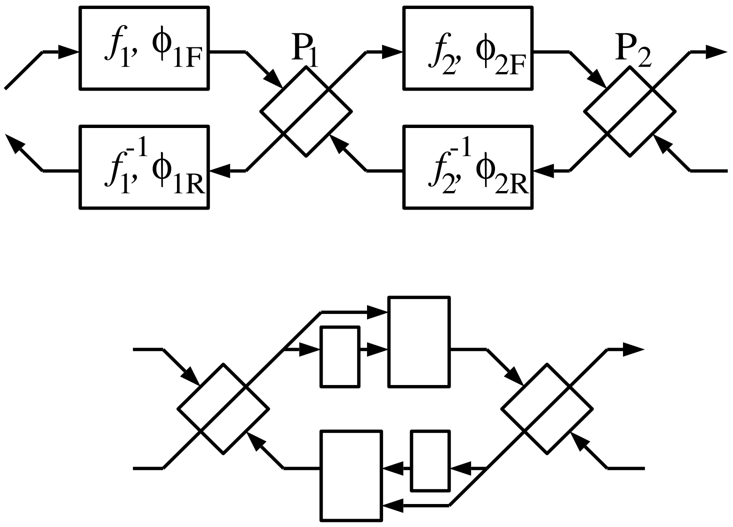
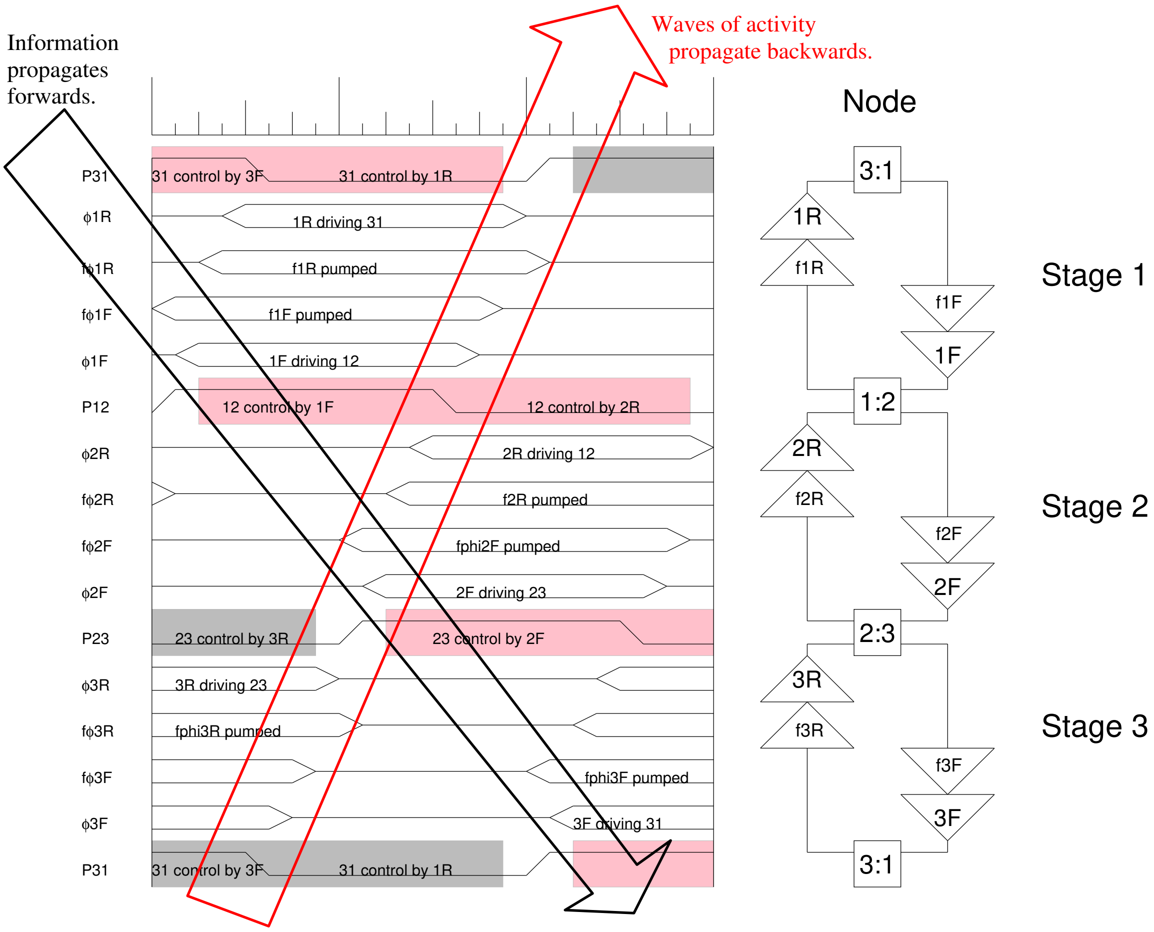


Figure 7.9: SCRL pipeline. A pipeline of arbitrary sequential logic in SCRL can be constructed by chaining together generalized inverters and bidirectional latching in the following way. A parallel set of generalized inverters is grouped together into a multi-input, multi-output function f_1 which must be *invertible*, and this block is paired with a corresponding block that computes the inverse function f^{-1}. The two blocks are powered by two clock phases ϕ_{1F} and ϕ_{1R} that are offset relative to each other. Then comes a bidirectional latch P_1 and another pair of functional blocks.

The basic operational cycle is that f_1 computes and its output X_1 is latched onto P_1. Then f_2 computes and its output X_2 is latched onto P_2, then f_2^{-1} computes X_1 from X_2, and "unlatches" P_1 back to the neutral level. Now f_1 can process a new input and store the result Y_1 on P_1, at the same time that a further stage f_3 is using the value computed from the earlier value X_1. In this way, waves of information propagate down the pipeline as they are being processed. The pipeline can even loop back on itself, as long as phases are matched properly. If the clock timing is inverted, information flows in the opposite direction.

The bottom part of the figure illustrates how non-inverting logic functions can be computed in a single SCRL stage by the use of a second intermediate level of logic. The second level uses a clock whose active period is enclosed within that of the first level's clock, like a 2-level version of one of Hall's retractile cascades. With 2 levels per stage in an SCRL pipeline, one can do universal reversible sequential logic.



Information propagates forwards.

Waves of activity propagate backwards.

Figure 7.10: Full timing diagram for 3-phase, non-inverting, static SCRL. This was the timing discipline used in our designs. On the right is a vertical representation of a sequence of 3 pipeline stages, using a slightly different notation from that presented in figs. 7.6–7.8: the squares are bidirectional latches, and the triangles are generalized SCRL inverters. Each element's clock is shown exactly to its left on the timing diagram. Time goes from left to right on the timing diagram, and information flows from top to bottom in the pipeline. This scheme requires 16 clock signals and their inverses, and 24 distinct non-overlapping transition steps per complete clock period. The shaded regions indicate times when valid logic values are present on the various latches.

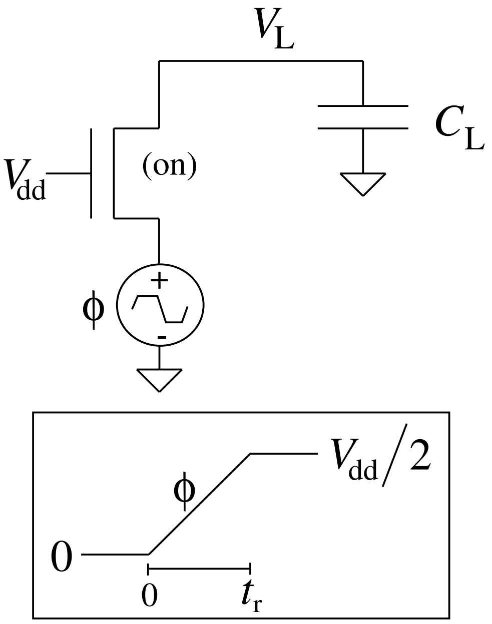


Figure 7.11: Simplified circuit model for SCRL analysis. Compare with figs. 7.6 (p. 176) through 7.8 (p. 178). In the simple model, we only look at a single transition, and we use a single transistor to represent the path through an arbitrary pulldown network and a transmission gate in a latch.

Following this, section 7.6.4 reveals a case in ordinary SCRL where the dissipation seems to be larger than in the ideal model, and shows a way to fix it. Then later, in §7.9, we will talk about some of the long-term limits involved in scaling CMOS and SCRL technology to smaller length scales.

7.6.1 A simple SCRL model for analysis

Switching energy is dissipated in an SCRL circuit whenever the voltages on some logic gate's power supply rails $\phi, \overline{\phi}$ change. Energy is dissipated within the transistors of the gate's pullup/pulldown networks, and also in the transistors of the transmission gate in the bidirectional latch attached to the gate's output. However, to simplify the analysis, we will lump together all the turned-on transistors within which dissipation occurs during a transition, and treat them as if they were a single transistor, as in figure 7.11.

We can consider a number of different cases for switching. A gate's output node voltage may be switched either through the gate's pulldown network of n-FETs or through its pullup network of p-FETs. And the switching activity may either be to clear the output or to set the output. When an output node is cleared, its voltage goes from a valid level (0 or $V_{\rm dd}$) to the neutral value $V_{\rm dd}/2$; when it is set, its value goes from $V_{\rm dd}/2$ to 0 or $V_{\rm dd}$.

However, all these cases are symmetrically similar to each other with regards to how their energy dissipation scales with speed, threshold voltage, and temperature. Therefore, rather than analyzing them all separately, we will just consider one case: where the voltage $V_{\rm L}$ on the load capacitance $C_{\rm L}$ on the output node is charged up from 0V to $V_{\rm dd}/2$, through a turned-on n-FET which represents the gate's pull-down network and N pass transistor.

In our analysis, we will ignore any dissipation that occurs during switching in transistors along paths that do not actually connect all the way through to the gate's output. For example, we ignore energy dissipation that occurs when switching with the transmission gate turned off. As another example, referring forward to the NAND gate in figure 7.15 (p. 200), we can see that if input A is high and input B is low, then transistor T3 will be turned on, and so there will be some dissipation through it, even though it does not connect through to the output. Ignoring such dissipations is a simplification that is fairly well justified, because these dissipations involve driving relatively small capacitances compared to the external load. In adiabatic charging, there is a quadratic dependence of dissipation on the capacitance being driven. So the total dissipation we are ignoring should not be large, compared to the dissipation that we are including.

We assume that the p-FETs and n-FETs in the SCRL circuit have been sized so that their gain factors are equal, $k_{\rm n} = k_{\rm p} = k$ (matching the rise/fall delay times), and we assume that the p-FET and n-FET threshold voltages are also equal, $V_{\rm t0n} = V_{\rm t0p} = V_{\rm t0}$, so that the analysis of the dissipation through the pulldown network comes out the same for the pullup network.

7.6.2 Switching losses as a function of technology parameters

To determine the energy dissipation of our model circuit (fig. 7.11), we would like to know the voltage on the load at each moment during the transition, $V_{\rm L}(t)$, because this would tell us the instantaneous drain-to-source voltage $V_{\rm DS}(t)$ across the transistor, which we could plug into the device's current-voltage relation to give us the instantaneous current $I(t)$, and thence the instantaneous power, which we could integrate over time to find the total energy dissipation of the transition $E_{\rm tr}$:

$$\begin{aligned} E_{\rm tr} &= \int_{t=0}^{\infty} P(t)dt && (7.5) \\ &= \int_{t=0}^{\infty} I(t)V_{\rm DS}(t)dt && (7.6) \end{aligned}$$

Unfortunately, $V_{\mathrm{L}}(t)$ itself is determined by integrating the current $I(t)$ flowing into the load capacitance C_{L}, so that determining closed-form formulas for $I(t)$ and $V_{\mathrm{DS}}(t)$ requires solving a tricky differential equation, which we will not attempt here. Instead, we will approximate the energy dissipation by treating the limiting case where the supply rise time t_{r} is very large compared to the characteristic RC time constant of the circuit, where R is the effective resistance of the turned-on transistor. Cases where the rise time is comparable to RC will not be adequately addressed by the below analysis.

To understand this limiting case, refer back to the diagrams in figure 7.5 (p. 173). Diagram (a) shows qualitatively what would happen if the supply rail were to rise very quickly compared to RC. Essentially the output voltage would rise at an exponentially-decaying rate and asymptotically approach the supply voltage, just as happens in a regular CMOS inverter whose input switches very quickly. The energy dissipation E_{fast} for this fast-switching case is well known to be, as in §7.1.1.1, p. 150,

$$E_{\mathrm{fast}} = \frac{1}{2} C_{\mathrm{L}} (\Delta V)^2, \tag{7.7}$$

which in our case is (with $\Delta V = V_{\mathrm{dd}}/2$)

$$E_{\mathrm{fast}} = \frac{1}{8} C_{\mathrm{L}} V_{\mathrm{dd}}^2. \tag{7.8}$$

On the other hand, figure 7.5(b) shows what happens in the case which we will now analyze, where the supply rail rises very slowly. The output voltage V_{L} will initially rise slowly, but as the voltage drop V_{DS} across the transistor increases, the current $I(t)$ through the transistor will also rise, until an equilibrium is reached at which point V_{L} is rising at the same rate as the input voltage, but lagging behind it by a small amount $V_{\mathrm{DS}} = IR$. Then, when the input voltage stops rising, the output voltage will finish the approach to $V_{\mathrm{dd}}/2$ in asymptotic fashion, with an RC time constant.

We note that if the input rises slowly, V_{DS} is always small compared to $V_{\mathrm{dd}}/2$, and so $V_{\mathrm{L}}(t) \approx \phi(t)$. During the transition, $d\phi/dt$ is constant, and so the current $I = C_{\mathrm{L}} \frac{dV_{\mathrm{L}}}{dt}$ through the transistor will be approximately constant as well. I will be the quotient of the total charge $Q = C_{\mathrm{L}} V_{\mathrm{dd}}/2$ that is transfered to the load capacitance, divided by the supply rail rise time t_{r}, since that is the time during which almost all of this charge is transfered.

$$I(t) \approx I = Q/t_{\mathrm{r}} = \frac{C_{\mathrm{L}} V_{\mathrm{dd}}/2}{t_{\mathrm{r}}} \tag{7.9}$$

Now, armed with this constant current I, we can use the standard MOSFET triode-regime current-voltage formula (*cf.* [139], §2.3.2, p. 44, eq. 2.47) to derive a closed form expression for V_{DS}. The reason we use the triode-regime rather than the saturation-regime formula is that turned-on transistors in SCRL are never in saturation.[2]

In the following, V_{GS} is the gate-to-source voltage, and V_{T} the threshold voltage. Everything except k (the transistor's gain factor) is here implicitly a function of t.

$$I = k\left((V_{\mathrm{GS}} - V_{\mathrm{T}})V_{\mathrm{DS}} - \frac{V_{\mathrm{DS}}^2}{2}\right) \tag{7.10}$$

Let's write $V_{\mathrm{GS}} - V_{\mathrm{T}}$ as V_{dr} (drive voltage) for conciseness.

$$I = k\left(V_{\mathrm{dr}}V_{\mathrm{DS}} - \frac{V_{\mathrm{DS}}^2}{2}\right) \tag{7.11}$$

We can easily solve this equation for V_{DS}, using the quadratic formula.

$$\frac{I}{k} = V_{\mathrm{dr}}V_{\mathrm{DS}} - \frac{V_{\mathrm{DS}}^2}{2} \tag{7.12}$$

$$\frac{1}{2}V_{\mathrm{DS}}^2 - V_{\mathrm{dr}}V_{\mathrm{DS}} + \frac{I}{k} = 0 \tag{7.13}$$

$$V_{\mathrm{DS}} = \frac{V_{\mathrm{dr}} \pm \sqrt{(-V_{\mathrm{dr}})^2 - 4\left(\frac{1}{2}\right)\left(\frac{I}{k}\right)}}{2\left(\frac{1}{2}\right)} \tag{7.14}$$

$$= V_{\mathrm{dr}} - \sqrt{V_{\mathrm{dr}}^2 - 2\frac{I}{k}} \tag{7.15}$$

Now, let us make a further simplification of eq. 7.15. We observe that our earlier approximation, that $I(t)$ was constant, assumed that t_{r} is large, and therefore that I is small (from eq. 7.9). With $I \ll kV_{\mathrm{dr}}^2$, this will allow us to approximate eq. 7.15 as follows. We observe that V_{DS} will be approximately linear in I for these small Is.

---

[2] This formula may not be appropriate for turned-on transistors if V_{dd} is about as small as the thermal voltage $\phi_T = k_{\mathrm{B}}T/q_{\mathrm{e}}$, since then even turned-on transistors may only be in moderate or weak inversion, and the current may scale exponentially with V_{GS} rather than according to the triode formula. This is one area where the present analysis needs refinement.

V_{DS} will pass through 0 at $I = 0$, and the slope is given by dV_{DS}/dI:

$$\begin{aligned}
\frac{dV_{\mathrm{DS}}}{dI} &= \frac{d}{dI}\left(V_{\mathrm{dr}} - \sqrt{V_{\mathrm{dr}}^2 - 2\frac{I}{k}}\right) && (7.16)\\
&= -\frac{1}{2}\left(V_{\mathrm{dr}}^2 - 2\frac{I}{k}\right)^{-\frac{1}{2}}\left(\frac{-2}{k}\right) && (7.17)\\
&= \frac{1}{k\sqrt{V_{\mathrm{dr}}^2 - 2\frac{I}{k}}} && (7.18)\\
&\approx \frac{1}{k\sqrt{V_{\mathrm{dr}}^2}} \quad \text{(for small } I\text{)} && (7.19)\\
&= \frac{1}{kV_{\mathrm{dr}}}. && (7.20)
\end{aligned}$$

Given this slope, and the fact that $V_{\mathrm{DS}} = 0$ when $I = 0$, we can therefore simplify eq. 7.15 to the very concise form

$$V_{\mathrm{DS}} \approx I/kV_{\mathrm{dr}}. \qquad (7.21)$$

Now, the drive voltage V_{dr} is itself actually time-dependent, because it is defined in terms of the gate-to-source voltage V_{GS}, and although the gate voltage is constant, the transistor source voltage changes linearly over time t_{r}, from 0 to $V_{\mathrm{dd}}/2$, following $\phi(t)$.

$$\begin{aligned}
V_{\mathrm{dr}}(t) &\equiv V_{\mathrm{GS}}(t) - V_{\mathrm{T}}(t) && (7.22)\\
&= [V_{\mathrm{G}} - V_{\mathrm{S}}(t)] - V_{\mathrm{T}}(t) && (7.23)\\
&= \left(V_{\mathrm{dd}} - \frac{V_{\mathrm{dd}}}{2}\frac{t}{t_{\mathrm{r}}}\right) - V_{\mathrm{T}}(t) && (7.24)\\
&= (V_{\mathrm{dd}} - V_{\mathrm{T}}(t)) - \frac{V_{\mathrm{dd}}}{2}\frac{t}{t_{\mathrm{r}}} && (7.25)
\end{aligned}$$

Moreover, $V_{\mathrm{T}}(t)$ as well will vary along with the supply voltage, due to the changing body effect as the source voltage changes. For example, when the supply voltage is at $V_{\mathrm{dd}}/2$, V_{T} might be perhaps (as a roughly estimated typical value) 50% above the minimum value V_{T0} that it has when $\phi = 0V$. Using the correct formulas for V_{GS} and V_{T}, the energy integral in equation 7.6 would still a bit too complicated to conveniently evaluate, although if we really cared to do it, we could.

But instead, let's just make the rough approximation that $V_{\mathrm{dr}}(t)$ is constant, and is equal to

$$V_{\mathrm{dr}} = \frac{3}{4}V_{\mathrm{dd}} - b_{\mathrm{avg}}V_{\mathrm{T0}}, \tag{7.26}$$

taking the average of the initial (V_{dd}) and final ($V_{\mathrm{dd}}/2$) values of $V_{\mathrm{GS}}(t)$, with an average body-effect factor $b_{\mathrm{avg}} = V_{\mathrm{T}}/V_{\mathrm{T0}}$ for a typical body-effected V_{T}. The reason for expressing the body-effected threshold V_{T} as a multiple of V_{T0} is that it will later allow us to derive a very simple expression for the switching energy.

Now, with our approximate constant expressions for I (eq. 7.9) and V_{dr} (eq. 7.26), we can consider V_{DS} as given by eq. 7.21 to be roughly constant, which allows us finally to approximate the transition energy integral (eq. 7.6) and derive a fairly simple expression for E_{tr} in the slow-transition limiting case. We set the upper bound on the integral to be time t_{r} rather than ∞, in observance of the fact that in the slow-transition limit, most of the energy dissipation occurs by time t_{r}.

$$\begin{aligned}
E_{\mathrm{tr}} &\approx \int_{t=0}^{t_{\mathrm{r}}} I(t)V_{\mathrm{DS}}(t)dt && (7.27)\\
&\approx IV_{\mathrm{DS}}t_{\mathrm{r}} && (7.28)\\
&= I\left(\frac{I}{kV_{\mathrm{dr}}}\right)t_{\mathrm{r}} && (7.29)\\
&= \frac{I^2t_{\mathrm{r}}}{kV_{\mathrm{dr}}} && (7.30)\\
&= \frac{\left(\frac{C_{\mathrm{L}}V_{\mathrm{dd}}/2}{t_{\mathrm{r}}}\right)^2 t_{\mathrm{r}}}{kV_{\mathrm{dr}}} && (7.31)\\
&= \frac{C_{\mathrm{L}}^2V_{\mathrm{dd}}^2}{4t_{\mathrm{r}}kV_{\mathrm{dr}}} && (7.32)
\end{aligned}$$

Now, we would like to take another simplifying step, by assuming that our maximum power supply voltage V_{dd} is being scaled proportionately to V_{T0}, and is equal to

$$V_{\mathrm{dd}} = n_{\mathrm{dd}}V_{\mathrm{T0}} \tag{7.33}$$

where n_{dd} indicates the scaling factor used for determining $V_{\mathrm{dd}}/V_{\mathrm{T0}}$. SCRL will not work properly if V_{dd} is too close to the threshold voltage V_{T0}. A reasonable value for n_{dd} for SCRL might be 4. Anyway, given eqs. 7.33 and 7.26, we can substitute V_{dd} and V_{dr} in eq. 7.32 to re-express it in terms of a single voltage parameter V_{T0}, the zero-bias threshold voltage:

$$
\begin{aligned}
E_{\mathrm{tr}} &= \frac{C_{\mathrm{L}}^2 (n_{\mathrm{dd}} V_{\mathrm{T0}})^2}{4 t_{\mathrm{r}} k \left(\frac{3}{4} n_{\mathrm{dd}} V_{\mathrm{T0}} - b_{\mathrm{avg}} V_{\mathrm{T0}}\right)} & (7.34) \\
&= \frac{C_{\mathrm{L}}^2 n_{\mathrm{dd}}^2 V_{\mathrm{T0}}^2}{4 t_{\mathrm{r}} k \left(\frac{3}{4} n_{\mathrm{dd}} - b_{\mathrm{avg}}\right) V_{\mathrm{T0}}} & (7.35) \\
&= \left(\frac{n_{\mathrm{dd}}^2}{3 n_{\mathrm{dd}} - 4 b_{\mathrm{avg}}}\right) \frac{C_{\mathrm{L}}^2 V_{\mathrm{T0}}}{t_{\mathrm{r}} k}, & (7.36)
\end{aligned}
$$

and let us finally just make this a bit more concise by renaming the factor containing n_{dd} as just

$$c_{\mathrm{dd}} \equiv n_{\mathrm{dd}}^2 / (3 n_{\mathrm{dd}} - 4 b_{\mathrm{avg}}). \tag{7.37}$$

To illustrate what a typical value of c_{dd} might be, if $n_{\mathrm{dd}} = 4$ and $b_{\mathrm{avg}} = 1.25$ (*i.e.*, average body-effected threshold 25% above V_{T0}), then $c_{\mathrm{dd}} \approx 1.45$.

Anyway, we can now write the transition energy formula (7.36) as just

$$\boxed{E_{\mathrm{tr}} = c_{\mathrm{dd}} \frac{C_{\mathrm{L}}^2 V_{\mathrm{T0}}}{t_{\mathrm{r}} k}.} \tag{7.38}$$

There are a couple of very interesting things to note about equation 7.38, when compared to equations like eq. 7.7 that govern the dissipation in fast SCRL transitions or ordinary CMOS transitions.

The first thing is that the transition energy in eq. 7.38 scales in proportion to the square of the load capacitance, in contrast to traditional CMOS where the CV^2 dissipation scales only linearly with capacitance. The reason is that higher capacitance means higher currents through our transistors, and thus a larger voltage drop across them, in addition to greater charge to move across that drop. So in designing SCRL circuits we must be even more careful to get load capacitances small than we are in regular CMOS. Unless most of the capacitance is in the interconnects, minimum-sized transistors are favored. If most capacitance is in transistor gates and PN junctions, then increasing transistor widths increases energy dissipation roughly linearly (not quadratically, because k is scaled too). The flip side of this coin is that SCRL benefits greatly from improved process technologies that allow smaller, less capacitive transistors.

The other very interesting point is that given a constant n_{dd} ratio between supply and threshold voltages, and everything else but V_{T0} also constant, the switching energy of SCRL circuits *decreases only linearly with decreasing threshold voltage*, in contrast to the quadratic drop of traditional CMOS due to its CV^2 switching energy.

Intuitively, the reason is because as voltages go down in SCRL, the effective on-resistance of our transistors increases, so the voltage drop across the transistors during transitions is increased, causing higher dissipation. In standard CMOS, the voltage drop across the transistors during switching is already as high as possible, and so making them more resistive doesn't affect the dissipation at all.

Equation 7.38 is interesting and useful on its own, because it allows us to predict the switching energy of SCRL circuits constructed in particular process technologies, and helps guide us in designing these circuits. But now, let's go a little further, and use eq. 7.38 as part of a more sophisticated analysis of SCRL energy dissipation that includes the effects of leakage.

7.6.3 Minimizing the sum of switching and leakage energy

In this section we explore how to minimize the energy dissipation of SCRL when taking leakage into account. First, in §7.6.3.1 we see how to minimize energy dissipation when the speed of operation is adjustable but all other technology parameters are held fixed. Then, in §7.6.3.2 we will see how to minimize dissipation when the choice of device threshold voltage (and supply voltage) is also adjustable, but other parameters such as device geometry and operating temperature are fixed.

7.6.3.1 Adjusting speed to minimize dissipation

One often-cited characteristic of the switching energy of adiabatic circuits, based on equations like eq. 7.38, is that it decreases linearly with increasing transition time t_{r}, leading to the conclusion that the energy per operation of SCRL circuits can be made arbitrarily small by just making the transition time larger. However, given current device technologies, this statement is somewhat misleading, because MOS transistors also have a leakage power dissipation that is always present, and thus contributes a term to total energy per operation that increases linearly with increasing time per operation. This means that there is some speed at which the energy per operation of an SCRL circuit is minimized; at faster speeds, the switching energy dominates, and at lower speeds, the leakage energy dominates. In this section we derive a formula for the optimal rise time for minimizing total energy per operation.

Let us consider what happens to a signal wire in an SCRL circuit during a complete cycle, from the time it first holds one valid value to the time it first holds the next. During this time there will be two complete transitions on the wire: one from the old value to $V_{\mathrm{dd}}/2$, the other from $V_{\mathrm{dd}}/2$ to the new value. The total time for the complete cycle depends on the number of phases in the particular SCRL clocking discipline in question. A complete cycle of the 2-phase SCRL described by Younis [191] is the length of 18 transitions; 3-phase and 4-phase SCRL take 24 transitions (*cf.* fig. 7.10, p. 182), etc. These numbers are probably not minimal. Anyway, let n_{t} be the number of transitions per cycle; the total cycle time is then $T = n_{\mathrm{t}} t_{\mathrm{r}}$.

Now we can write down an expression for the total energy dissipation associated with this signal wire per complete cycle, including terms for both the transition energy and the leakage energy, where the leakage energy is expressed in terms of P_{leak}, the average leakage power associated with the signal wire:

$$\begin{aligned} E_{\mathrm{tot}} &= 2E_{\mathrm{tr}} + P_{\mathrm{leak}} T && (7.39) \\ &= 2c_{\mathrm{dd}} \frac{C_{\mathrm{L}}^2 V_{\mathrm{T0}}}{t_{\mathrm{r}} k} + P_{\mathrm{leak}} n_{\mathrm{t}} t_{\mathrm{r}}. && (7.40) \end{aligned}$$

where the multiplication by 2 comes from the above-mentioned fact that an SCRL wire undergoes two transitions per cycle.

We want to find the t_{r} that minimizes E_{tot}. First, let us collapse everything except t_{r} into coefficients a and b:

$$\begin{aligned} a &\equiv 2c_{\mathrm{dd}} C_{\mathrm{L}}^2 V_{\mathrm{T0}}/k && (7.41) \\ b &\equiv P_{\mathrm{leak}} n_{\mathrm{t}} && (7.42) \\ E_{\mathrm{tot}} &= \frac{a}{t_{\mathrm{r}}} + b t_{\mathrm{r}}. && (7.43) \end{aligned}$$

Figure 7.12 shows how the total energy in eq. 7.43 scales with t_{r}, regardless of the values of a and b. We can see that at very high values of t_{r}, E_{tot} is high because of the high leakage energy, and at very low values of t_{r}, E_{tot} is high because of the high switching energy. In between, there is a point where the total energy is minimized.

We can find a formula for the t_{r} at this point; it's just where the derivative of eq. 7.43 equals zero, which turns out to be where the switching energy equals the leakage energy:

$$\frac{d}{dt_{\mathrm{r}}} \left(\frac{a}{t_{\mathrm{r}}} + b t_{\mathrm{r}} \right) = 0 \qquad (7.44)$$

$$-\frac{a}{t_{\mathrm{r}}^2} + b = 0 \qquad (7.45)$$

$$t_{\mathrm{r}} = \sqrt{\frac{a}{b}} = \sqrt{\frac{2c_{\mathrm{dd}} C_{\mathrm{L}}^2 V_{\mathrm{T0}}}{k P_{\mathrm{leak}} n_{\mathrm{t}}}} \qquad (7.46)$$

$$\boxed{t_{\mathrm{r}} = \sqrt{\frac{2c_{\mathrm{dd}}}{n_{\mathrm{t}}}} \cdot C_{\mathrm{L}} \sqrt{\frac{V_{\mathrm{T0}}}{k P_{\mathrm{leak}}}}} \qquad (7.47)$$

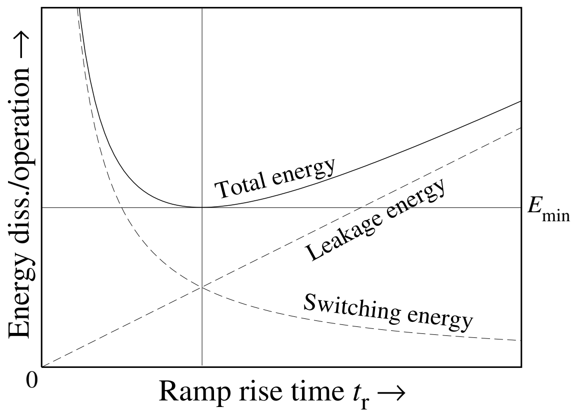


Figure 7.12: How total energy dissipation per operation scales with ramp rise time t_r in SCRL, when leakage is significant. The increasing line is leakage energy, the inversely declining curve is switching energy. Their sum is analytically proven to be minimized when the two components are equal.

At this minimum-energy setting for t_r, the total energy dissipation is:

$$E_{\text{tot}} = \frac{a}{t_r} + bt_r \tag{7.48}$$

$$E_{\min} = \frac{a}{\sqrt{a/b}} + b\sqrt{a/b} \tag{7.49}$$

$$= \sqrt{\frac{a^2}{a/b}} + \sqrt{b^2 a/b} \tag{7.50}$$

$$= \sqrt{ab} + \sqrt{ab} \quad \text{(note identical terms)} \tag{7.51}$$

$$= 2\sqrt{ab} \tag{7.52}$$

$$= 2\sqrt{\frac{2c_{\text{dd}} C_{\text{L}}^2 V_{\text{T0}}}{k} P_{\text{leak}} n_{\text{t}}} \tag{7.53}$$

$$\boxed{E_{\min} = \left(2\sqrt{2c_{\text{dd}} n_{\text{t}}}\right) C_{\text{L}} \sqrt{\frac{V_{\text{T0}} P_{\text{leak}}}{k}}} \tag{7.54}$$

Looking at eq. 7.54, if we want the energy per operation of an SCRL circuit to be as low as possible, we will want to first minimize the wiring capacitance and other parasitic capacitances we need to drive. Then we'd want to maximize the gain factor k of our transistors. However, if we try to increase k by making the transistors wider, this also increases the capacitance, and the leakage power. So narrower transistors are favored.

Ideally we'd like to get a handle on minimum energy by adjusting the threshold voltage, so as to minimize the quantity $V_{\mathrm{T0}}P_{\mathrm{leak}}$ in eq. 7.54. But choosing the optimal V_{T0} is actually a bit tricky, since P_{leak} itself depends on V_{T0}, in a way which we will now analyze.

7.6.3.2 Adjusting voltages to minimize dissipation

In a single transistor across which there is a voltage drop of $V_{\mathrm{DS}} = V_{\mathrm{dd}}$, which we will later see suffices to model the leakage through all the transistors attached to a given SCRL signal wire, the leakage power P_{leak} is given by

$$\begin{aligned} P_{\mathrm{leak}} &= I_{\mathrm{leak}}V_{\mathrm{dd}} && (7.55)\\ &= I_{\mathrm{leak}}n_{\mathrm{dd}}V_{\mathrm{T0}} && (7.56)\end{aligned}$$

and I_{leak} for transistors that are supposed to be "off" ($V_{\mathrm{GS}} \leq V_{\mathrm{T}}$) is given by a standard formula

$$I_{\mathrm{leak}} = I_0 e^{(V_{\mathrm{GS}}-V_{\mathrm{T}})/((1+\alpha)k_{\mathrm{B}}T/q)} \tag{7.57}$$

where I_0 denotes the leakage current when the transistor is just barely on the edge of being off (*i.e.*, when $V_{\mathrm{GS}} = V_{\mathrm{T}}$). k_{B} is Boltzmann's constant, T is the absolute temperature, q is the magnitude of the electron charge, and α is a technology-dependent constant fudge factor, which is ideally 0 but in practice is perhaps closer to 1. This factor is needed because real devices are found empirically to have a greater dependence of leakage on temperature than is predicted by the theoretical ideal.

Now, the leakage in SCRL circuits is not really continuous, but fluctuates during the SCRL cycle as different rails split and merge. In static versions of SCRL such as Younis's 3-phase clocking scheme, we can identify two types of leakage: (1) leakage through the middle of a logic gate across a voltage drop of V_{dd} when the gate's supply rails are split, and (2) leakage through a turned-off pass transistor across a voltage drop of $V_{\mathrm{dd}}/2$. All these leakages occur through off devices that have a V_{GS} of zero; other off devices with $V_{\mathrm{GS}} < 0$ have exponentially less leakage, and so we ignore them. During some transitions, there are also leakages across voltage drops smaller than $V_{\mathrm{dd}}/2$. Some of these happen when $V_{\mathrm{GS}} < 0$, and the others contribute small amounts to the total leakage power.

One may carry out a careful analysis of leakage based on the timing diagram of Younis's 3-phase clocking cycle. We will not relate the analysis in detail here. However, one finds that for each signal wire, there is leakage inside one of the logic gates that drive that wire during $\frac{22}{24}$ of each cycle, and leakage through a pass transistor for about $\frac{19}{24}$ of the cycle (this latter figure is adjusted to take into account the smaller voltage drops that occur during transitions).

Further, the I_0 for the leakage inside logic gates may be different than the I_0 for the leakage through the pass transistors, depending on how the devices are sized relative to each other, and also remembering that if a logic gate is not a simple inverter but rather contains several parallel paths, there may be leakage through all of the paths.

However, all of these factors can incorporated into our definition of the effective I_0 for the SCRL signal wire, as follows. Let $I_{0\mathrm{G}}$ be the effective I_0 in the pullup/pulldown networks of our logic gates (taking into account the widths of devices and number of parallel paths). Let $I_{0\mathrm{P}}$ be the I_0 through our pass transistors (taking into account their widths). Then we just define the effective I_0 for the single-transistor equivalent model of the SCRL signal wire's average leakage as

$$I_0 = I_{0\mathrm{G}}\frac{22}{24} + I_{0\mathrm{P}}\frac{1}{2}\cdot\frac{19}{24} \tag{7.58}$$

where the $\frac{1}{2}$ compensates for the fact that the leakage through the pass transistors involves a voltage drop of $V_{\mathrm{dd}}/2$ rather than V_{dd}. This substitution is valid because the other factor in eq. 7.57 (the exponential) doesn't depend on the magnitude of the V_{DS} voltage drop or on which kind of leakage we are looking at, since $V_{\mathrm{GS}} = 0$ for all the significant leakage.

We further note that almost all of the leakage takes place when $V_{\mathrm{GS}} = 0$ and $V_{\mathrm{SB}} = 0$, so that at these times $V_{\mathrm{T}} = V_{\mathrm{T0}}$, and we can substitute V_{T0} for V_{T} in eq. 7.57. Further, for conciseness let's define convenient notations for the thermal voltage $k_{\mathrm{B}}T/q$ with and without the $(1+\alpha)$ fudge factor.

$$\phi_T \equiv k_{\mathrm{B}}T/q \tag{7.59}$$

$$\phi'_T \equiv (1+\alpha)\phi_T \tag{7.60}$$

Now we can re-express the leakage current as just

$$I_{\mathrm{leak}} \approx I_0 e^{-V_{\mathrm{T0}}/\phi'_T}. \tag{7.61}$$

Although the above method for estimating I_{leak} was developed for the particular case of static 3-phase SCRL, it is fairly clear that the same approach could be carried out similarly for other SCRL clocking schemes as well, with appropriate modifications to eq. 7.58. Remember, however, that in 2-phase SCRL, nodes are not always being actively driven, and so high leakages can harm functionality as well as dissipating power; therefore the analysis later in this section will probably not be appropriate for dynamic 2-phase clocking.

Now that we've gotten I_{leak} expressed in terms of V_{T0}, let's merge eqs. 7.56 & 7.61

back into our expression for E_{min} (eq. 7.54):

$$
\begin{aligned}
E_{\mathrm{min}} &= \left(2\sqrt{2c_{\mathrm{dd}}n_{\mathrm{t}}}\right) C_{\mathrm{L}}\sqrt{\frac{V_{\mathrm{T0}}P_{\mathrm{leak}}}{k}} && (7.62)\\
&= \left(2\sqrt{2c_{\mathrm{dd}}n_{\mathrm{t}}}\right) C_{\mathrm{L}}\cdot \\
&\quad \sqrt{\frac{V_{\mathrm{T0}}(n_{\mathrm{dd}}V_{\mathrm{T0}})I_0e^{-V_{\mathrm{T0}}/\phi'_T}}{k}} && (7.63)\\
&= \left(2\sqrt{2c_{\mathrm{dd}}n_{\mathrm{t}}n_{\mathrm{dd}}}\right)\cdot \\
&\quad C_{\mathrm{L}}V_{\mathrm{T0}}\left(\sqrt{\frac{I_0}{k}}\right)e^{-\frac{1}{2}V_{\mathrm{T0}}/\phi'_T} && (7.64)
\end{aligned}
$$

To make this formula easier to work with, we'll express the factor involving the SCRL power and timing parameters n_{dd} and n_{t} as just s. Also, we note that since I_0 and k both scale roughly proportionally to transistor width, the voltage factor $\sqrt{I_0/k}$ is basically independent of transistor width. It scales up with increasing length however (because k scales down proportionally, but I_0 does not scale down as much), indicating that SCRL favors designing with minimum-length devices and small gate fan-ins. (Larger fan-ins yield a larger effective length.) In such designs, $\sqrt{I_0/k}$ can be thought of as a width-independent voltage v_{c} that is characteristic of the particular device technology being used. It can be interpreted as the drive voltage required to turn on a standard-length transistor strongly enough to conduct current at some fixed multiple of the transistor's zero-drive leakage current I_0.[3]

Given the above definitions, we can re-express the minimum energy as

$$
\begin{aligned}
s &\equiv 2\sqrt{2c_{\mathrm{dd}}n_{\mathrm{t}}n_{\mathrm{dd}}} && (7.65)\\
&= 2\sqrt{\frac{2n_{\mathrm{t}}n_{\mathrm{dd}}^3}{3n_{\mathrm{dd}}-4b_{\mathrm{avg}}}} && (7.66)\\
v_{\mathrm{c}} &\equiv \sqrt{I_0/k} && (7.67)\\
E_{\mathrm{min}} &= sC_{\mathrm{L}}v_{\mathrm{c}}V_{\mathrm{T0}}e^{-\frac{1}{2}V_{\mathrm{T0}}/\phi'_T}. && (7.68)
\end{aligned}
$$

Figure 7.13 shows qualitatively how E_{min} scales as V_{T0} is changed. Perhaps surprisingly, above a certain point, the minimum energy/op of SCRL actually decreases exponentially as the threshold voltage is increased! This contrasts with the situation in standard CMOS, where higher thresholds mean quadratically larger switching energy, determined by equations like eq. 7.7. The difference in SCRL is that higher

---

[3] Perhaps v_{c} is related to the drive voltage needed for strong inversion. This needs further investigation.

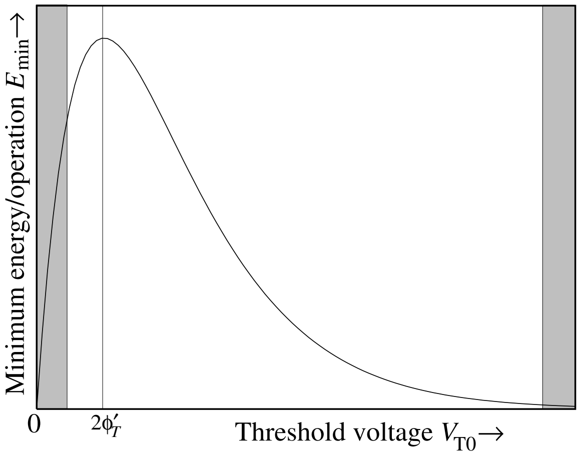


Figure 7.13: How minimum energy/operation scales with V_{T0} in SCRL, as per (7.68). The curve is only meaningful for points to the right of the maximum, but not too far to the right. At the very low voltages at the far left, circuits will not function properly because they will be overwhelmed by leakage currents. At very high voltages far to the right, gate oxides may break down, depending on their thickness. But up to this breakdown point, SCRL minimum energy scales down roughly exponentially, as the ratio V_{T0}/ϕ_T is increased.

---

thresholds mean exponentially smaller leakage power, which allows us to run at exponentially slower speeds and still not have leakage dominate the total energy, which thus allows exponentially less energy to be dissipated during our quasistatic charging at high thresholds.

The curve in fig. 7.13 also suggests that at very low thresholds, the energy/op can be made arbitrarily small as well. However, this part of the curve is probably not accurate. Further analysis ([68]) shows that the maximum point on the curve occurs when $V_{\mathrm{T0}} = 2\phi'_T$, twice the adjusted thermal voltage. At thresholds near or below the thermal voltage, a V_{dd} that is only a small fixed multiple of the threshold voltage will probably not be high enough to produce strong inversion, and the square-law equation (7.10) will probably not accurately represent the source-drain current of our transistors, upon which the above analysis was based. Moreover, at low thresholds, the high leakage power will call for a very short rise time from eq. 7.47; if the rise time is too short, it will not be large compared to the effective RC of our transistors, which will invalidate the assumptions upon which the analysis of section 7.6.2 was based.

Also, all of the analysis above is only reasonably accurate for relatively large devices. As we mentioned 7.1.5, as transistors shrink below present-day sizes, a variety of short-channel effects become increasingly significant in their influence on the I-V characteristics of the device. As we move deeper into this short-channel regime, the analytical expressions on which the above sections were based will become increasingly inaccurate. It was deemed beyond the scope of the present work to correct all of the above analysis to take short-channel effects into account, although such corrections will be important if adiabatic techniques are to be applied to low-energy computing applications in the near future.

7.6.4 Fixing a problematic case for plain SCRL

We believe there may actually be a small problem in ordinary SCRL, as originally described by Younis and Knight, a problem that seems to lead to non-time-proportional dissipation, and perhaps even to a dissipation per operation that is bounded below by $k_{\mathrm{B}}T$ (although this is not yet certain). This particular problem occurs even at low temperatures, at which ordinary leakage currents become exponentially more insignificant.

The problem occurs in multi-input SCRL gates such as NAND and NOR gates, as well as in more complex gates, but not in single inverters. The problem is due to the fact that under some inputs, part of a pull-up or pull-down network may be conducting even if the whole network is not. So for example, in a NAND gate, when the output is high, part of the pull-down network may actually be pulled up as well (see fig. 7.14). So an n-FET, which is designed for passing low voltage, is being asked to pass a high voltage. n-FETs can only conduct well over part of the high range, up to a threshold drop V_{T} away from the high voltage V_{dd}. So, an internal node in the pulldown network will follow the output ramp closely up to this voltage, but after that its rate of increase will slow down, because the effective resistance of the n-FET increases exponentially as its source voltage rises several thermal voltages ϕ_T above the threshold point. Therefore, the voltage drop over this n-FET will become significantly larger than voltage drops in the rest of the circuit, and some non-zero amount of charge will flow over this voltage drop, as the n-FET source voltage gradually edges up to a few thermal voltages above $V_{\mathrm{dd}} - V_{\mathrm{T}}$.

Clearly, this situation will have some impact on the analysis of the energy dissipation of SCRL, but it is not yet clear exactly what the impact will be. It is difficult to derive an analytical expression for the dissipation due to this effect. However, it seems likely that the major result will be that the overall dissipation of SCRL circuits does not decrease anywhere near as quickly as linearly with frequency, once the overall dissipation is low enough that the dissipation due to the above effect becomes significant.

To test this intuition, I performed a simple numerical simulation of this situation for an example circuit, and found that when the ramp time was long enough so that

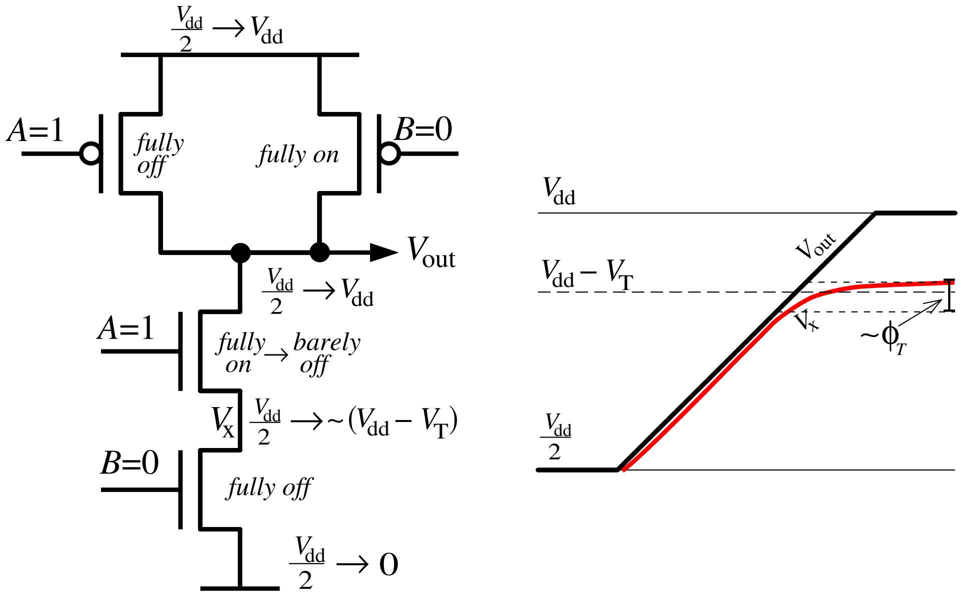


Figure 7.14: A case in the simplest version of SCRL where there may be energy dissipation that does not scale down in proportion to operating frequency. Consider a NAND gate when the inputs are different, and the higher input goes to the innermost n-FET. When the rails split, that n-FET will conduct, and the internal node voltage V_x will track V_{out} arbitrarily closely (depending on the ramp time), until $V_x \approx V_{dd} - V_T$. Then, the FET will begin to cut off, and V_x will lag farther behind V_x, while it continues to increase over a range of several additional ϕ_T's. In the depletion regime, it takes several ϕ_T's worth of V_{GS} voltage change in order for a MOSFET's channel charge (and thus its conductance) to fall off by several factors of e. During this time, drain voltage continues to increase at a relatively faster rate and so charge will be falling over a relatively large voltage drop. NOR gates and more complex gates will suffer from this problem as well.

the total dissipation in the earlier part of the ramp was less than $k_{\mathrm{B}}T$, the dissipation in the subthreshold regime was still $\sim 3000\,k_{\mathrm{B}}T$—but it was still decreasing slowly as the ramp time was lengthened. If the ramp time was lengthened far enough, perhaps the source voltage would continue to track the ramp closely all the way up to V_{dd}, and the dissipation in this transistor would be less than $k_{\mathrm{B}}T$—but then we are talking about ramp times so long that energy losses due to leakage would be significant, and greater than $k_{\mathrm{B}}T$, in other parts of the circuit. The logic would no longer function reliably because leakage currents would be comparable to charging currents. Overall, it is not yet clear whether or not this effect leads to a true $\sim k_{\mathrm{B}}T$ lower bound on dissipation in these simple circuits.

Fortunately, regardless of the precise effect, there is a simple way to fix SCRL to prevent this problem from occurring, and restore guaranteed time-proportionate reversibility to SCRL. That fix is to transform the problematic transistors into CMOS pass gates, with appropriate inputs, which conduct well over the entire voltage range that they might encounter. Thus, at low speeds the voltage drop across the transistors will always remain insignificant. The NAND gate problem in figure 7.14 can be repaired via the addition of just a single p-FET, as shown in fig. 7.15.

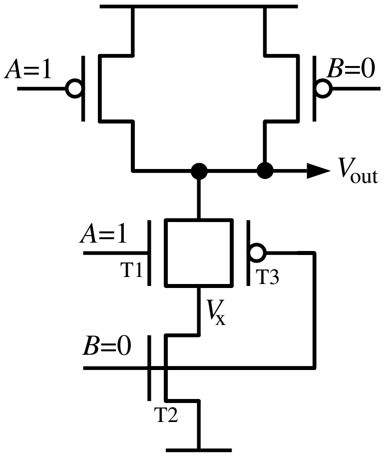
A more general way to fix all SCRL logic gates would be to just use dual-rail (complementary) logic everywhere, to ensure that the appropriate complementary signal for use in pass gates is available. This would have the advantage that it would also eliminate the need for 2-level retractile cascades in non-inverting logic stages, and would lead to greater data-independence of the overall capacitance of the chip, allowing the resonant power supply signal to be tuned more cleanly. The primary disadvantage would be the need for a roughly factor of 2 increase in the number of gates, wafer surface area, and entropy coefficient. But in the spirit of the asymptotic emphasis of this thesis, we remind ourselves that this is only a small constant factor.

That concludes our discussion of general properties of SCRL. In the next section we discuss the particular SCRL-based circuits that we designed and fabricated.

7.7 Experimental SCRL Circuits

The first SCRL circuits to be fabricated were those of Younis [191] in his original experimental tests. Younis fabricated a demonstration chip that included reversible adders and multipliers. Younis tested these chips and measured their power dissipation, and found that it scaled roughly as predicted within the range of sensitivity of the measurements. More accurate measurements of dissipation in other adiabatic circuits have been performed by Solomon and Frank (1995, [153]).

In parallel with my own work, Carlin Vieri has been designing Pendulum, a fully-adiabatic RISC-style processor based on SCRL. I assisted Vieri with various Pendu-



(a)

(b)

A	B	T1	T2	T3	PDN Conducts?
0	0	off	off	on	no
0	1	off	on	off	no
1	0	on → off	off	on	no
1	1	on	on	off	yes

Figure 7.15: A simple way to eliminate the problem discussed in figure 7.14. A p-FET "T3" tied to input B is placed in parallel with the inner n-FET "T1". T3 will be fully turned on in the problem case where A = `1` and B = `0`, and thus node V_x will follow V_{out} all the way up to V_{dd}, and so there will be no problematic voltage drop over T1. The truth table on the right verifies that the logic of the pulldown network remains correct in all cases with the addition of T3.

lum instruction set issues, about which I will have more to say in chapter 9. Scott Rixner and I designed and tested Tick, a non-adiabatic 8-bit version of Pendulum that was intended to gauge the complexity of the reversible instruction set design. (Unfortunately, the chip failed to operate fully, due to inaccuracies in the layout design-rule checking software that we used; design modifications would be necessary to get Tick working.) As of this writing, the fabrication of the fully adiabatic Pendulum prototype has recently been completed, and it is now in the testing phase.

Vieri and colleagues have also designed and fabricated XRAM [179], a fully-adiabatic static memory component, whose design I discussed with Vieri, but was not intimately involved with.

The primary SCRL design effort that I have been centrally responsible for (with assistance from other group members) has been the design of FLATTOP, a chip comprised of an adiabatic mesh-style array of processing elements that are very simple, yet capable of fully universal reversible computation. The FLATTOP chips can be tiled in 2-D or 3-D arrays, and together with the appropriate external resonant rail generators, reversible communications links, and power delivery/entropy removal systems, would constitute a concrete example of a time-proportionately reversible 3-D mesh processor, such as we conjectured was asymptotically optimal in ch. 6. As such, sufficiently large arrays of FLATTOP chips would be, in principle, faster for their size than any possible irreversible machine. At least, this would be the case if the FLATTOP design was repaired to circumvent the dissipative flaw in SCRL that we discussed in §7.6.4, which had not yet been discovered at the time FLATTOP was designed. Also, in reality the FLATTOP machine sizes necessary to outperform the fastest irreversible architectures would be astronomically large. But FLATTOP is still an important proof-of-concept, demonstrating that it is not only possible but fairly straightforward to design a universal, sequential reversible mesh processor using fully adiabatic circuits.

We now discuss some of the background for the FLATTOP design.

7.7.1 The Billiard Ball Model

The basic operation of FLATTOP is to simulate the "Billiard Ball Model" (BBM) of computation, an idealized physical model of reversible computation that was introduced by Fredkin [74] in the course of some of his early work on reversible logic circuits. The model involves computation using idealized perfect spheres, that move ballistically through 2-D space along precise trajectories, and bounce off fixed walls and each other in perfectly elastic collisions. Fredkin showed that using only this behavior, one can construct elementary reversible boolean logic gates (fig. 7.16) and put them together to compose arbitrary reversible logic circuits.

The BBM is of course an idealization. In reality, to avoid the accumulation of

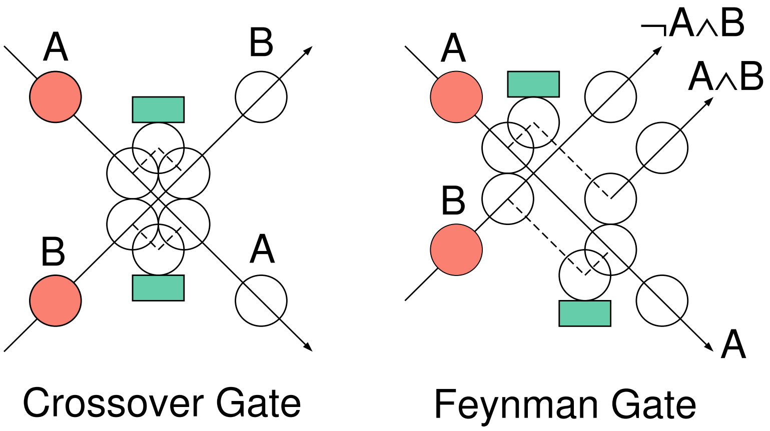


Figure 7.16: Examples of two logic gates in the physical billiard ball model of computation. The "Crossover Gate" on the left permits two ball-signals to effectively pass through each other without delay. The "Feynman Gate" on the right computes a reversible AND/NAND function.

errors in ball trajectories, the balls would have to be made to travel along troughs in a potential energy surface, pushed along by waves of potential to keep their global timing consistent, and to make up for frictional losses. In reality, the collisions would not be perfectly elastic—but this is just an example of time-proportionate reversibility, since real collisions between hard objects become more nearly elastic as the speeds involved are decreased. Energy can be injected into the system, at a rate per interaction that declines with the balls' speed, to keep the system progressing forwards at a constant rate. In talks given at MIT, Fredkin has discussed physically plausible mechanisms for performing the above-mentioned types of corrections.

But the main conceptual importance of the model is that it provides a way to see how extremely simple interactions—ball collisions—can be used to build up arbitrarily complex logic circuits. Further, in essence it is a purely digital model. It only cares about discrete positions and times. In contrast, other researchers have investigated analog models of computation in which an unlimited number of decimal places of precision in physical parameters are used to carry information important to computation [177, 151, 148], but such models are not physically realistic, because in quantum mechanics, bounded systems only have a finite number of distinguishable states, as we noted in §2.2; infinite precision is not a physically realistic assumption.

Furthermore, the BBM is ultimately a parallel model of computation—interactions may be occurring in many parts of the space simultaneously—and a 3-D version of it could asymptotically efficiently simulate any (non-quantum) physical algorithm.

Due to its ultimate digital nature, its simplicity, its reversibility, its asymptotic

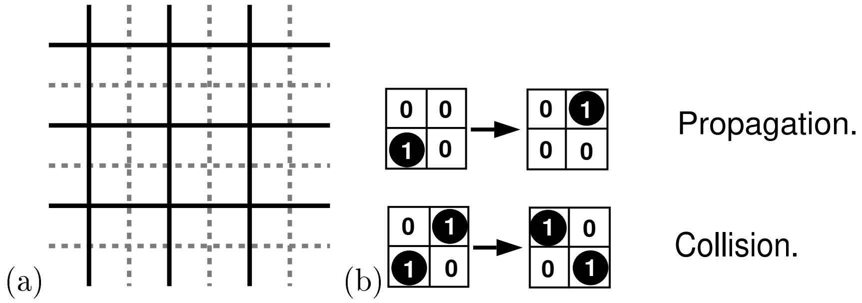


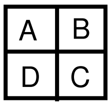
Figure 7.17: The Billiard-Ball Model Cellular Automaton. (a) Updates are performed alternately in two overlapping meshes of 2×2 blocks of cells. This partitioning scheme, also called the "Margolus neighborhood," is an easy way to produce a global reversible dynamics from a local reversible update rule. (b) BBMCA block update rule. In a 2×2 block of cells, a single 1-bit moves to the opposite corner of a block (propagation), whereas 2 bits in opposite corners move to the other 2 corners (collision). All rotations of the illustrated cases also apply. All other configurations remain unchanged. Note that this rule is reversible.

efficiency, and its universality, the billiard ball model is a useful starting point for investigations of reversible computation.

7.7.2 The Billiard Ball Model Cellular Automaton

Making the ultimately digital nature of the billiard ball model even more apparent, in 1983 Norman Margolus invented a digital cellular automaton, with only 1 bit per cell, that precisely and efficiently simulates the billiard ball model on a 2-D grid of cells [116, 115]. In this "Billiard Ball Model Cellular Automaton" (BBMCA), balls are represented by pairs of 1-bits that move along diagonal paths through the grid of cells. The grid is updated by breaking it into two overlapping meshes of 2×2 blocks of cells (fig. 7.17a), which apply on alternate update steps, and within each block transforming its state according to a simple reversible update rule (fig. 7.17b). The reader should be warned that it is somewhat non-obvious how this update rule leads to behavior that imitates the billiard ball model; see [116] or §2.4 of [115] for a detailed description.

Because of the extreme simplicity of the BBMCA update rule, we chose it as our target for our proof-of-concept mesh architecture, which was therefore named FlatTop, after the local billiards pub, "Flat-Top Johnny's." This architecture is not intended to be particularly efficient in terms of its constant factors, or to be particular easy to program directly. But ultimately, since it can asymptotically efficiently simulate any (non-quantum) parallel architecture, it demonstrates the points



$$S = (A + C)\,(B + D)$$
$$A' = S\,A + \overline{S}\,\overline{A}\,(C + B\,D)$$
... and similarly for B, C, D

Figure 7.18: Boolean logic form of BBMCA update rule. The S bit is true when there are bits in both diagonals, in which case the block should remain *static*, unchanged. If there aren't bits in both diagonals, the block may change, and a given bit (e.g., A) should turn on if it was off and the opposite bit was on (propagation rule) or the two adjoining bits were on (collision rule).

---

we are trying to make about asymptotic scaling. Given a large enough array of FlatTop chips, one could efficiently simulate any alternative architecture or programming model on top of it.

A more practical reversible mesh processor would probably have a design and a programming model closer to that of Vieri's RISC-style Pendulum chip. In fact, given a suitable communication network between neighboring processors, Pendulum itself would probably work fine as a processing element. But FlatTop was relatively easy to design, and it gives us the proof-of-concept we wanted. It also seems entirely possible to design a reversible FPGA element that would be intermediate in complexity and programmability between FlatTop and Pendulum, and might permit greater logical density on a wider variety of problems than either.

7.7.3 Logic minimization

After choosing the BBMCA as the target functionality, the next step was to translate the BBMCA update rule into a boolean logic expression that could be easily implemented in a real SCRL logic circuit. After trying several ways of translating the update rule into a boolean formula, we settled on the solution shown in fig. 7.18, which is the simplest such representation of the logic that we have found so far.

The idea behind these formulas is that under the BBMCA update rule, one of two things happen: either the block might change, or it must stay the same. It must stay the same if there are 1 bits in both of the two diagonals across the block. Letting "S" represent this case, we have (with reference to fig. 7.18) that S is (A or C) and (B or D).

Then, the new value of A itself, A', is fairly easy to compute. If S, then A' is just A, unchanged. Otherwise, A' turns on if and only if A was originally off, and either the opposite bit was on (propagation rule), or both of the adjacent bits were on (collision). If all bits are off, they all remain off. One can see by inspection of the possible cases that this rule yields exactly the BBMCA update rules.

To implement this logic in SCRL, we would need at least two logic stages in each processing element: The first stage would take the initial state of the 4 cells in a block as input, and would produce the S signal using a complex gate implementing the formula for S, while passing the input bits through unchanged to the second stage. The second stage would use a complex gate for each cell to compute its new state given S and the original cell states. This was the basic function of each processing element. Since we were using 3-phase SCRL, a third stage was needed to put the data in the correct phase for passing to an adjacent processing element. We also used the third stage to implement a special shift register functionality for initializing the whole array.

7.7.4 FlatTop array design

Figure 7.19 outlines how the array of processing elements was connected. The horizontal-vertical grid shows the cell space partitioned according to the Margolus neighborhood into 2 overlapping grids of 2×2 blocks, as in fig. 7.17a. The updating of each block is handled by a corresponding processing element, which can be visualized as resting at the center of that block. The state of each cell is passed back and forth along wires between the PE's at the centers of the two diagonally-overlapping blocks that contain the given cell. The array of PE's thus naturally forms a mesh that is oriented at a 45° angle relative to the original CA mesh. We orient the chip edges along this diagonal mesh, so that the wires between processing elements can be parallel to the chip edges, which is a constraint required by some fabrication processes.

One artifact of this design is that, if all the processing elements operate simultaneously, they will actually be simulating two parallel non-interacting BBMCA systems. We can separate the PEs into "dark" and "light" processors in a diagonal checkerboard pattern as pictured. One CA system is the one that is processed by light processors on even-numbered time-steps and by dark processors on odd-numbered steps. The other CA system is the one that is processed by dark processors on even-numbered steps and by light processors on odd-numbered steps. The two systems do not interact at all, and they can be used to represent two completely different configurations of balls and mirrors in the BBM. Using both systems can be seen as a way of making more efficient use of the hardware, which would otherwise be only half utilized.

The two systems can be connected together at one or more chip edges to form one larger system with an alternative topology. In fact, we did this in our prototype FlatTop chip. Also, at various points at the chip edges we passed signals to bidirectional I/O pads to connect to neighboring processors in a larger array. Normally in the BBMCA the grid is 2-D, but there is nothing to prevent a topology that is locally 2-D on-chip, but globally 3-D, with the chips connected in a 3-D mesh. As

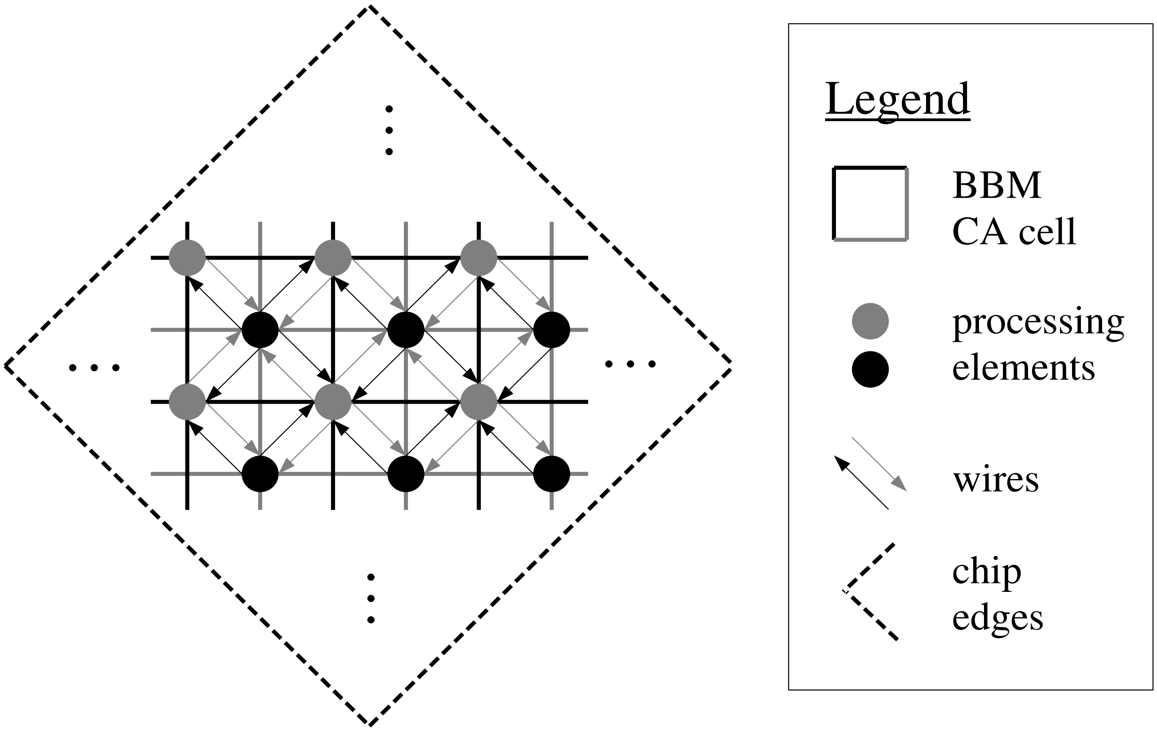


Figure 7.19: Schematic illustration of a grid of FlatTop processing elements. Each 2×2 block of cells in the Margolus partitioning (see fig. 7.17a) is updated by a different PE. The state of a given cell is stored, on alternate time steps, on one or the other of the two wires running between diagonally adjacent PEs. The PEs thus naturally form a square mesh oriented 45° from the BBMCA mesh. To make layout more convenient, the chip edges were oriented parallel to the PE mesh.

NMOS		PMOS	
Var	Value	Var	Value
ϕ_0	1.1V	ϕ_0	0.8993
m_{j}	0.726	m_{j}	0.4905
m_{jsw}	0.2451	m_{jsw}	0.2451
C_{j}	4.67x10^{-4} F/m^2	C_{j}	8.76x10^{-4} F/m^2
C_{jsw}	3.20x10^{-10} F/m	C_{jsw}	2.13x10^{-10} F/m
t_{ox}	9nm	t_{ox}	9nm
μ_{n}	978.1 cm^2/V^2-s	μ_{p}	228.5 cm^2/V^2-s
C_{ox}	3.89x10^{-15} F/μm^2	C_{ox}	3.89x10^{-15} F/μm^2
k'_{n}	3.80x10^{-4} A/V^2	k'_{p}	8.889x10^{-5} A/V^2

Table 7.5: Device parameters for the HP14 process, from Cadence models. These figures were used in our hand-calculations of the minimum energy dissipation per operation in FlatTop.

long as each chip has at least 3 external connections, a globally 3-D network can be made [185].

Appendix A shows most of the Cadence schematics and layout for the FlatTop unit cell and processor array. The design was simulated using Verilog and functioned as expected in simulation. Individual PEs were simulated in HSPICE to verify that there were no errors causing CV^2 dissipation. The chips have been fabricated but have not yet been tested. It is expected that their basic functionality will work, and that fairly low-energy operation is possible, but that the dissipation will not be quite as low as was originally projected due to the SCRL flaw we discussed in §7.6.4.

7.7.5 Minimum energy estimation

After designing FlatTop, we carried out an approximate hand-analysis of the circuit, using the device parameters of the fabrication process we used (table 7.5), and the formulas we derived in §7.6.3.1 (p. 190), to estimate the circuit's minimum energy dissipation per operation when operated at 3.3 V, the standard supply voltage for the process (HP14) that was used. As the circuit is clocked more slowly, switching energies decrease proportionally, but leakage energies increase. As we pointed out in §7.6.3.1, the minimum total energy per operation turns out to be achieved at the speed at which switching energy equals the leakage energy (refer back to fig. 7.12, p. 192).

The energy estimation procedures we used are also described in more detail in our conference paper [72] on FlatTop. The upshot was that the optimal cycle time turned out to be around $12\mu s$, at which point FlatTop would dissipate around 10 fJ per cycle per cell, whereas in our estimation the equivalent iCMOS circuit dissipates around 20 pJ, a reduction of energy dissipation by a factor of 2000! When cooled below room temperature, the chip would have less leakage, and even greater energy efficiency could be obtained at lower speeds.

Unfortunately, there has not yet been an opportunity to actually test the energy dissipation in FlatTop. Indeed, such low levels of dissipation would be hard to measure accurately. Also, the actual dissipation is probably higher than we first estimated, since at design time we did not know about the SCRL bug mentioned in §7.6.4. However, if this bug were fixed, we believe that dissipation in the chip would be roughly as predicted.

7.8 Resonant power supply techniques

In most of the above, we have glossed over the issue of how to build a resonant power supply that can provide the power/clock waveforms that SCRL requires, in such a way that the energy loss per cycle scales down arbitrarily in proportion to frequency. This particular issue has not been the primary focus of my own research, but it is important for the overall scaling results.

Various techniques for powering adiabatic circuits have been described in the literature: [156, 157, 194, 63, 3]. However, many techniques do not have asymptotically zero dissipation. One interesting recent technique is the one developed in our group by Becker and Knight [12, 13]. This technique uses a transmission line with tuned nonuniformities that allow it to end up resonating with any desired waveform; in particular, it has been used to produce the trapezoidal ramp-shaped waveforms used by SCRL[4]. However, due to nonlinear effects of the signal on resistivity in Becker's transmission lines, the dissipation per cycle does not scale down quite in proportion to the frequency f, but apparently rather as $\sqrt{f}$. Thus the scaling results for a reversible system powered by these circuits is not quite as good as the ideal.

An open problem for adiabatic computing is the design of an external resonant element that can provide waveforms usable for adiabatic computing while at the same time retaining the property of having a dissipation per cycle that scales down to arbitrarily small levels, in direct proportion to frequency. It is also important for the cost-efficiency arguments of §6.2 that the cost of the resonant element does not increase substantially as its frequency is decreased. One problem with using Becker's transmission line approach in reversible computing is that the length of the transmission line scales up in proportion to its frequency, therefore increasing the cost

---

[4] As reported by Becker in personal discussions.

of the system when run at low speeds.

However, we (optimistically) suspect that if enough attention is paid to the issue of designing resonant power supplies, a power supply technique having the desired properties can be found. Certainly it has not been proven, to our knowledge, that no such technique can exist. It is an important area for future work to determine with certainty whether an ideal supply technique can exist, and to design a technique having the most favorable scaling properties that are physically possible.

7.9 Scaling SCRL to future technology generations

Although it is difficult to precisely forecast the future performance of CMOS-based technology in the near term (next 10 years), over the long term one can point out some qualitative aspects of how performance scales with technology shrinkage in CMOS-like circuits, based on some fundamental scaling laws, and see how these factors affect SCRL compared with standard irreversible CMOS.

Consider the effect of shrinking all circuit dimensions by a factor of f_ℓ (that is, any length ℓ is shrunk to ℓ/f_ℓ). Intuitively speaking, one must consider the effect of shrinking all dimensions, rather than just one or two, because any dimension that does not shrink will eventually dominate in terms of its parasitic effects, and one could improve performance by shrinking that dimension as well.

Under f_ℓ scaling in all dimensions, the width and length of capacitive elements will decrease, but so will their thickness, so capacitances will decrease by $\sim f_\ell$. Resistive elements will get shorter, but also narrower along the other two dimensions, so their resistance will increase by $\sim f_\ell$. Characteristic RC delays through resistive elements thus do not scale down by much, since these factors cancel out. (In the near term, resistance is dominated by the effective resistance through transistors. These are harder to model accurately. But if their resistance decreases, RC's might decrease for a while, but then eventually the resistance in the wires will come to dominate, so at some point RC cannot decrease further.)

Earlier, we saw that energy dissipation per operation in adiabatic technologies such as SCRL scales as $CV^2\frac{RC}{t}$. If the RC part doesn't improve much beyond a certain point, as we just saw, then what about the CV^2 part? This part corresponds to node energy. For a while, C will scale down as $1/f_\ell$, and V at some point will be forced to scale down at least as fast as $1/f_\ell$ because otherwise, as gate oxides get thinner, the electric field through the oxide would increase and at some point would break down. So CV^2 eventually must scale down at least as fast as f_ℓ^{-3}. This makes sense intuitively, because it corresponds to the energy density in the circuit not increasing beyond a fixed maximum level that the circuit's materials can withstand.

But, can CV^2 continue to decrease indefinitely? In §7.1.2, p. 151, we already demonstrated that in irreversible CMOS, we can show that CV^2 cannot decrease below a reliability-dependent factor times $k_{\mathrm{B}}T$ from a pure thermodynamics argument. But this reliability-based limit affects SCRL as well. The electrons in a circuit at normal temperatures behave like a thermal gas, and this leads to a well-known noise component called "kT/C noise," because anything that samples a signal will find that the σ^2 variance in the sampled voltage is $k_{\mathrm{B}}T/C$, where C is the capacitance of the sampling node (*cf.* p. 155). Such sampling occurs in SCRL all the time; each time a pass gate in an bidirectional latch cuts off, it can can be viewed as sampling the previous logic gate's output voltage on a sampling node whose capacitance is just the load capacitance of the latch output. If thermal noise causes a sampling error comparable to $V_{\mathrm{dd}}/2$, correct logical functionality can be impaired. Therefore, just like in irreversible CMOS, node energies in SCRL cannot be made smaller than a reliability-dependent factor times $k_{\mathrm{B}}T$.

Therefore, as typical node energies decrease with f_ℓ^{-3} at a given temperature, eventually they will reach this point where further decreases cannot be made without sacrificing reliability, so to continue shrinkage, the temperature will have to start scaling down as f_ℓ^{-3} as well. But note that at this point, the entropy generation per operation in SCRL is no longer decreasing along with the shrinkage. We know $S = E/T$, and the dissipation E is decreasing as f^3, but now so is T. So ultimately, SCRL's entropy generation per operation becomes exactly RC/t_{r} times a reliability-dependent constant ($\ln N$), regardless of further circuit shrinks.

Further decreases in entropy coefficients beyond this point would require non-scaling-related decreases in R, such as less resistive wiring materials, a switching device with better I-V characteristics than MOSFETs (perhaps micro-electro-mechanical switches, if they could be made small enough), or even superconducting circuit elements. Irreversible CMOS, on the other hand, could not take full advantage of such improvements, because its dissipation per operation does not fundamentally improve with R, and its entropy generation per operation is bounded below by $\sim \ln N$ nat.

This leads to a long-term advantage of SCRL as the technology scales. As lengths are shrunk by f_ℓ, one can create f_ℓ^3 times as many processing elements out of a given mass of materials. Beyond a point, entropy generation for a fixed mass of reliable irreversible CMOS scales as $\sim$ nat, whereas entropy generation in SCRL is $\sim \frac{RC}{t_{\mathrm{r}}}$ nat, where we assume we have reached a limit where RC cannot be decreased further. But as the number of processors increases with f_ℓ^3, the clock frequency of the irreversible machine must be slowed down for some tasks by a factor $f_\ell^{1/3}$, due to the increasing number of processing elements and the heat-removal arguments of §6.2.3.1, whereas the SCRL machine only needs to be slowed by $f_\ell^{1/4}$. Therefore, as f_ℓ increases, the overall processing rate $\mathcal{R}_{\mathrm{op}}$ of the irreversible machine goes as $f_\ell^{2\,2/3}$ whereas the reversible machine goes as $f_\ell^{2\,3/4}$, for a reversible advantage increasing as $f_\ell^{1/12}$ as devices shrink by a factor of f_ℓ.

The above analysis imagines that it is possible to continue scaling MOSFETs without fundamental differences up to and beyond the point where reliability constraints become dominant. In reality, MOSFETs may hit a wall for other reasons before this (*cf.* [121]), at which point, we might have to switch to a radically different technology for further improvements. Different technologies might have different scaling considerations, but in any irreversible technology, the lower bound of $S = 1$ nat still holds, and in any reversible technology, we expect there will be a characteristic transition time parameter t_c, playing a role similar to the RC parameter in CMOS, that approximately gives the entropy coefficient for the technology (the entropy goes below 1 to the extent that t goes above the characteristic transition time). To the extent that this expectation is true, the reversible advantage with shrinking components will be at least the $f_\ell^{1/12}$ factor mentioned above, and if the characteristic time were to scale down with the length scale as well, so much the better for reversible technology. In that case, processing rate would scale as f_ℓ^3, an advantage of $f_\ell^{1/3}$ over irreversible technology.

7.10 Mostly reversible computation

This thesis primarily focuses on the concept of *arbitrarily* reversible computation. But there are of course benefits to be gained from a more limited use of reversibility in digital circuits. For example, one could construct a processor's functional units using fully adiabatic circuits, and use irreversible switching only to update the high-level processor state between instructions. Or, one could identify the highest-power components of a chip (often, the long buses) and apply energy-recovery techniques only to those sections (this is the direction being explored by the ISI ACMOS group [4, 2, 170]).

Such limited uses of reversibility are potentially quite beneficial in highly energy-limited environments such as portable or embedded systems [66, 1]. And since reversibility need not be complete in order to gain substantial energy savings in these applications, the algorithmic overheads for full reversibility that we explored in §3.4 need not apply.

This line of work can lead to immediate practical, commercially viable products, thereby introducing adiabatic circuit concepts to industry. After this introduction, we expect that gradually over time, users will come to demand more and more computational power using less and less energy, and so the degree to which systems will need to rely on reversible techniques will increase. Eventually, we expect designs for energy-limited systems will converge to encompass the arbitrarily-reversible sort of architectures that we discuss in this thesis.

7.11 Adiabatic Circuits—Conclusion

In this chapter we reviewed the most immediately feasible technology for reversible computing, namely the technology of adiabatic circuits. We discussed the prototype mesh-style processor that we constructed using this technology, which is a proof of concept that illustrates that adiabatic circuit techniques such as SCRL are powerful enough to implement fully reversible parallel machines such as we proposed in ch. 6.

The most important areas for future work on adiabatic circuits, in the context of exploring the limits of computing, include:

- More thorough analysis of the precise performance characteristics of SCRL (or similar techniques) compared with irreversible CMOS in near-future technology generations, to obtain a more precise estimation of the cost level above which reversibility confers a real advantage.
- Research on directions in VLSI technology (such as MEMS switches or superconducting materials) that might lead to much lower-resistance switches, which would benefit adiabatic techniques much more than irreversible techniques.
- More research on resonant power supplies for adiabatic circuits, in search of a technique whose dissipation scales down in proportion to frequency, asymptotically all the way to zero, while still providing waveforms that are suitable for use in SCRL or comparable adiabatic circuit techniques.
- Similarly, design of reversible or even ballistic interconnection technologies for communication between adiabatic circuit chips.
- Design of good parallel reversible processor architectures, building on FLATTOP and Vieri's Pendulum work.

In summary, the future for reversible computing with adiabatic circuits looks interesting, but several new developments are still needed before adiabatic techniques could become competitive with traditional techniques at affording cost-efficient supercomputing. It may well be that in the short run, more effective cooling systems (which benefit irreversible techniques more than reversible techniques) will be easier to develop. However, in the long run there are limits to how much cooling systems can be improved—and cooling a system does not ultimately reduce total energy. Meanwhile, various factors on the horizon threaten the ability of CMOS circuits to keep shrinking indefinitely.

Eventually, to gain further improvements in machine speed, it seems we may well be forced to jump to an alternative, non-CMOS-like, computing technology. Interestingly, many of the alternative technologies that have been proposed by various researchers are capable of efficient reversible computation. In the next chapter, we briefly survey some of these, and then in chapter 9 we go on to discuss issues in the architecture and programming of reversible machines, regardless of the details of the underlying reversible logic technology.

Chapter 8

Future reversible device technologies

In the previous chapter, we reviewed adiabatic circuits, the reversible computing technology that is most similar to the irreversible semiconductor technology that is the basis of essentially all present-day computing.

In this chapter, we look a bit farther afield, and review a number of proposals that have been made for computing technologies that might supersede traditional CMOS, once the limits of MOSFET technology are reached, and manufacturing processes develop to the point where constructing machines based on these alternative device technologies is feasible and economical. A number of the technologies we describe are (time-proportionately) reversible, or at least have reversible variants. We will describe the important parameters of these technologies that impact the scaling issues we discussed in chapter 6. We also will review several proposals that have been made for advanced cooling technologies.

We then calculate, using the formulas developed in §6.2.2.1, p. 128, how large a reversible machine would have to be, in the various proposed reversible technologies and under the various proposed cooling systems, in order for it to be faster per unit area than a machine built using various irreversible technologies. Most of these calculations were previously reported in §7 of our journal article [70].

8.1 Cooling technologies

Ordinary CPU chips in most present-day computers rarely dissipate more than 100 W of heat from a square centimeter of chip surface using normal passive cooling mechanisms, such as conduction through a ceramic package, and natural or forced convection through air. The chip surface is normally at least at room temperature

Cooling technology	Max entropy flux F_S in bits/s cm^2
Digital optic fiber	10^{13}
Typical passive emission	3.5×10^{22}
Drexler's fractal plumbing	3.8×10^{24}
Slow atomic ballistic	10^{26}
Fast atomic ballistic	3×10^{33}
Quantum maximum	5×10^{40}

Table 8.1: Estimates of the maximum entropy flux per unit area achievable with various existing and hypthetical cooling technologies. These are all rather rough estimates, and the last limit is especially arbitrary, since it is technically only valid for black holes having an arbitrarily-chosen 1 Å radius.

(300 K), so the entropy flux attained by these mechanisms is no larger than $F = 100\ \text{W}/(k_B(300\ \text{K})\ln(2)/\text{bit}) = 3.5 \times 10^{22}$ b/s-cm^2.

David Tuckerman [166, 167] has created and tested advanced semiconductor cooling systems which use forced convection of liquid coolant through micron-scale channels etched into the back of a silicon wafer. He has experimentally verified cooling rates on the order of $\sim$ 1000 W from a square centimeter-size chip, and has projected that higher rates are possible.

Drexler (1992, [51]), §11.5.3, p. 332, has designed a nanotechnological cooling system using a fractal plumbing network that ought to be able to remove at least 10kW/cm^2 of heat at 273 K from a flat slab of material up to 1 cm thick. This corresponds to an entropy removal rate of 3.8×10^{24} bit/s cm^2.

This figure corresponds roughly to the heat flux in the cooling systems of current-day nuclear reactors, which transport megawatts of heat through massive pipes on the order of a square meter in cross-section. (According to an acquaintance in the nuclear engineering department.)

If entropy were to be encoded in some material at the atomic scale at a density ρ_S of no more than 1 bit per cubic Ångstrom (roughly the volume of a small atom), and the material moves nearly ballistically through the computer at a speed of $v = 1$ m/s, the maximum entropy flux $F_S = \rho_S v$ is 10^{26} bit/s cm^2. (Allowing most of the machine's volume to be occupied by the cooling material.)

If the material instead moves at a tenth of the speed of light (a very fast speed that is still easy to analyze since relativistic effects are small), then the maximum flux is 3×10^{33} bit/s cm^2.

For materials around the density of water, 1 g/cm^3, Bekenstein's fundamental quantum-mechanical/general-relativistic bound on entropy (see §2.2.1, p. 33 and [15]) implies that even if all the material's mass-energy could be used for storing information, no more than about 1.7×10^6 bits can exist in an region 1 Å across. At a tenth the speed of light this gives an entropy flux of 5×10^{40} bit/s cm^2.

If entropy is removed digitally through 1 mm wide 100 GHz optical fiber available today, the maximum flux is only about 10^{13} bits/s cm^2. The maximum entropy flux that can be achieved using electromagnetic radiation is the blackbody flux, as we described with eq. 2.16 (§2.3, p. 39). We should note that the entropy density $S/\mathcal{V}$ in a thermal photon gas scales in proportion to T^3 ([88], p. 571), so achieving unboundedly high entropy densities using a stream of photons would require unboundedly high temperatures, which we may reasonably disallow.

Further, we should remember that the limit on entropy density given by Bekenstein's bound actually increases as information is encoded across regions of smaller and smaller diameters. If some technology can achieve Bekenstein's limit, then it may change the entire form of the appropriate scaling analysis. However, Bekenstein's bound may not actually be achievable, and in any case it seemingly only applies in the high-gravity realm that we have decided to avoid. So for now, we will stick with our general assumption that for any particular technology, entropy density is finite.

Table 8.1 summarizes the above figures.

Now, let us examine how these different flux limits affect the maximum possible rate of computing per unit area, under various computing technologies.

8.2 Irreversible device technologies

Based on the switching energy issues we discussed in §7.1.1 (p. 148), and typical parameters of modern VLSI fabrication processes, we estimate that the best present-day CMOS irreversible device technologies still generate at least $\sim 10^6$ bits of entropy per device-switching operation. This number will decrease somewhat over successive VLSI technology generations, as power supply voltages and circuit node capacitances decrease. However, as we saw in §7.1.2.1 (p. 155), supply voltages cannot decrease too much because of difficulties in setting device thresholds accurately. Moreover, in order to cope with thermal noise, total entropy generation per operation in irreversible CMOS circuits cannot decrease below a reliability-dependent number of nats per operation.

One very interesting alternative semiconductor logic technology is the "rapid single flux quantum" (RSFQ) superconducting logic family being developed by K. Likharev's research group at SUNY, and colleagues [109, 110, 195, 50]. This technology may be able to dissipate as little as 1 aJ (10^{-18} J) of energy per irreversible bit-operation at a temperature of 5 K, which corresponds to an entropy generation of only 21 kilobits.

		Operations per second per cm^2 surface in each cooling technology		
Irreversible device technology	Entropy generated per bit erased	Typical passive	Fractal Plumbing	Slow atomic ballistic
Modern CMOS	10^6	3.5×10^{16}	3.8×10^{18}	10^{20}
Likharev RSFQ	2.1×10^4	1.7×10^{18}	1.8×10^{20}	4.8×10^{21}
Best possible	1	3.5×10^{22}	3.8×10^{24}	10^{26}

Table 8.2: Maximum rate $\mathcal{R}_{\mathrm{A}}$ of irreversible operations per unit area achievable with various irreversible device technologies and cooling technologies from table 8.1.

Finally, we would like to consider a "best possible" irreversible technology that produces only 1 bit of physical entropy for each bit of information that is discarded. Merkle and Drexler (1996, [126]) argue that their proposed "helical logic" electronic logic technology could perform irreversible bit erasure with an energy dissipation approaching $k_{\mathrm{B}}T \ln 2$, which would create just 1 bit of entropy. Drexler's nanomechanical "rod logic" is also estimated to be capable of performing close to this limit as well (Drexler 1992 [51], §12.4.3d, p. 359). We expect that in general, as computational devices approach the nanoscale, a wide variety of different device designs will be found that are capable of asymptotically approaching the minimum entropy generation of 1 bit of entropy per bit of logical information that is irreversibly erased.

In table 8.2 we combine these entropy generation figures with the entropy flux rates from the previous section to calculate a maximum rate of irreversible bit operations per second, per unit of enclosing surface area, for various combinations of irreversible device technologies and cooling technologies. Note that these limits apply no matter how much extra hardware one packs in along the third dimension! As we saw in §6.2.2.1, ultimately, all irreversible technologies are limited to a fixed processing rate per unit of outer surface area, such as the limits given here.

Now, let us examine some reversible technologies and estimate the scales above which they exceed these irreversible rates of performance per unit area.

8.3 Reversible technologies

We now examine the entropy coefficients of a variety of reversible device technologies. Recall that the entropy coefficient of a technology expresses the amount of entropy generated per device operation, per unit of frequency at which the device is operated.

Based on the SCRL adiabatic circuit technology described in the previous chapter, we calculated the entropy coefficient for typical reversible logic gates fabricated in the fairly recent 0.5 μm VLSI process (HP14) that we used for FlatTop, when operating at room temerature. We obtained a value of about 43 bits/kHz. In an estimated "best available" process with around 10 kΩ transistor on-resistance, 1 V power supply, and 60 fF node capacitance, we estimate a somewhat lower value of $\sim$ 6 bits/kHz.

SCRL's entropy coefficient might be even better in an implementation based on low-resistance micro-electro-mechanical switches, as was suggested by Younis ([191], §2.7.3, p. 34). However, based on calculations I did using figures obtained from the MEMS (micro-electro-mechanical systems) community, although some of the best available MEMS switches apparently might offer an entropy coefficient as low as $\sim$ 0.003 bits/kHz, the size of these switches (on the order of 100 microns) is large enough that they do not end up outperforming MOSFETs in terms of reversible cost-efficiency. In other words, although the individual switches can run faster for a given dissipation per operation, a machine of a given speed per unit area must be larger.

Merkle [123] analyzed the energy dissipation of the reversible transfer of a packet of 100 electrons through a minimal quantum FET, and found it to be around $3{\times}10^{-21} J$ at a rate of 1 GHz. The corresponding entropy coefficient at room temerature is about 1.2 bits/GHz.

Drexler's rod logic, operated reversibly, would dissipate about $2{\times}10^{-21}$ J per operation at a speed of 10 GHz ([51], p. 354). Its entropy coefficient at room temperature thus comes out to 0.070 bits/GHz.

The "parametric quantron" superconducting reversible device of Likharev [108] dissipates about 10^{-24} J during a 1 ns operation at around 4 K ([108] p. 322); its entropy coefficient thus comes out at about 0.026 bits/GHz.

Finally, Merkle and Drexler's proposed helical logic [126] was analyzed by them to dissipate around 10^{-27} J at 10 GHz and 1 K when operated reversibly; its entropy coefficient thus comes out to be 10^{-5} bits/GHz. This is the lowest entropy coefficient that we have encountered so far.

Table 8.3 summarizes the above figures. Armed with them, we are now in a position to calculate the scale at which the various reversible technologies will beat the various irreversible technologies that we mentioned in §8.2. We will measure this scale first in terms of the number of devices required per unit area, then, in technologies for which we know the device volume, this can be used to find the necessary diameter or thickness of the machine.

Based on the analysis of section 6.2.2.1 (p. 128), we can express the number of reversible devices N_A per unit area required to achieve a given rate $\mathcal{R}_\mathrm{A}$ of operations per unit area as

$$N_\mathrm{A} = \mathcal{R}_\mathrm{A}^2 k_\mathrm{S}/F_\mathrm{S},$$

where as usual k_S is the entropy coefficient and F_S is the entropy flux per unit area.

Reversible device technology	Entropy coefficient k_S in bits/GHz
SCRL in HP14	4.3×10^{7}
SCRL in best available CMOS	6×10^{6}
Merkle quantum FET	1.2×10^{0}
Drexler rod logic	7.0×10^{-2}
Likharev parametric quantron	2.5×10^{-2}
Helical logic	1.0×10^{-5}

Table 8.3: Entropy coefficients k_S for some existing and proposed asymptotically reversible logic device technologies

To achieve the same rate of operation achievable by an irreversible machine that produces S bits of entropy per operation and uses the same cooling system, the number becomes

$$N_\mathrm{A} = F_\mathrm{S} k_\mathrm{S} / S^2.$$

Table 8.4 shows the number of reversible devices in various technologies needed to beat the maximum per-area processing rate for the 3 combinations of irreversible technologies and cooling technologies that fall along the diagonal of table 8.2. The parenthesized numbers indicate cases in which the number of devices required may be determined by the maximum rate of operation of the devices, rather than by the entropy limits. The number given is the number of devices that would be required if the individual devices could run with as high a frequency as needed. The actual number required will most likely be higher.

To make sense of the non-parenthesized numbers in table 8.4, we estimate the volumes of the logic devices in various technologies. SCRL logic gates we will take to be about 10 μm × 10 μm × 1 μm = 100 $\mu\mathrm{m}^3$. Merkle's quantum FET we estimate at about $(10\ \mathrm{nm})^3$, a rod logic interlock as 40 nm^3 ([51], §12.4.2, p. 357), and a helical logic switch as 10^7 nm^3 ([126], §5.2, p. 330). Given these values we produce the results in table 8.5.

The parenthesized numbers in table 8.5 need explanation. The entries that say "any" indicate that even if the given reversible devices are arranged over a surface in only a single layer, they will still be faster than any machine built with the given irreversible technology within that surface. As for the 0.1 mm figure we calculated for 10^{12} helical logic devices per square centimeter beating the best possible irreversible technology given a 10^{26} bit/cm^2 entropy flux, it is probably inaccurate because the individual helical logic devices probably couldn't be made to run at the implied rate of 100 THz.

	Irreversible device and cooling technology combination		
	best CMOS/passive	RSFQ/convective	best/atomic
Entropy S, bits/op	10^6	2.1×10^4	1
Flux F, bits/s cm^2	3.5×10^{22}	3.8×10^{24}	10^{26}
Rate R, ops/s cm^2	3.5×10^{16}	1.8×10^{20}	10^{26}
Reversible Technology	Devices required per square cm to beat the above rate		
SCRL/best CMOS	2.1×10^8	5.2×10^{13}	6×10^{23}
Quantum FET	(42)	(107)	1.2×10^{17}
Rod logic	(2.4)	6×10^5	7×10^{15}
Helical logic	(3.5×10^{-4})	(86)	(10^{12})

Table 8.4: Numbers N_{A} of reversible machines per unit area required to beat different irreversible device technologies with different cooling strategies. Parenthesized numbers indicate lower bounds, where the real bounds depend on the maximum rate of operation of the devices.

	Irreversible device and cooling technology combination		
Reversible Technology	best CMOS/passive	RSFQ/convective	best/atomic
SCRL/best CMOS	0.21 mm	52 m	(4 au)
Quantum FET	(any)	(any)	1.2 mm
Rod logic	(any)	(any)	2.8 μm
Helical logic	(any)	(any)	(0.1 mm)

Table 8.5: Thicknesses d of reversible machines that can beat different irreversible technologies in terms of operations per unit area. “Any” indicates that even a single layer of the given reversible devices will suffice to beat the given irreversible technology.

The entry in the upper right corner of the table indicates that a machine built with current CMOS reversible technology, such as SCRL, would have to be the size of the inner solar system (!!) before it would be faster per unit area than the most efficient possible irreversible technology. Needless to say, a machine this large, composed mostly of solid silicon, would collapse under its own gravity.

In any case, the table indicates overall that most of the listed reversible technologies outperform most irreversible technologies, in terms of raw numbers of operations per second per unit area, for a wide range of cooling capabilities and for machines at a reasonable scale. Current CMOS reversible technology does not perform so well against the most efficient conceivable irreversible technologies, but it can still beat machines based on contemporary irreversible CMOS technology at reasonable scales.

One caveat to the above results is that in general a reversible device operation is not quite as computationally useful as an irreversible operation, due to the algorithmic issues we discussed in §3.3. However, for problems that have efficient reversible algorithms, like physical simulations (see §9.5.6), a small constant number of reversible device operations should suffice to do as much useful computational work as a single irreversible operation. The diameters in table 8.5 should therefore be increased by a factor of the same small constant.

8.4 Future device technologies—Conclusion

In this chapter, we listed a number of existing and proposed device technologies for both irreversible and reversible logic, and a variety of existing and hypothetical cooling technologies. Many of the technologies described cannot currently be built, but it is plausible that someday they might, and in any case all the technologies described serve as interesting points for comparison.

For each device technology, we gave the explicit numerical parameters determining its entropy generation, and from this, we determined limiting rates of operation per unit area for the irreversible technologies. Then, based on the superior scaling laws we have derived for time-proportionately reversible machines, we estimated the thickness of the reversible machines that would beat the irreversible machines' performance per unit area.

The upshot is that although present reversible technology is not so great, many of the proposed future reversible technologies would outperform *any* irreversible technology in terms of rate per unit area, even when considering only very thin layers—on the order of microns to millimeters thick—of packed logic devices in the given technology. This result holds firm unless a way is found to remove entropy from a system at a flux much higher than our rather arbitrarily-chosen maximum rate of 10^{26} bits per square centimeter per second. (A rate corresponding to 1 bit per cubic Ångstrom, moving at an arbitrary 1 m/s.)

These figures argue that in the long term, as computing technology moves down into the nanometer realm, and (eventually) away from conventional bulk-semiconductor techniques, reversibility will become a clear win in any macroscopic-scale computers built from such nano-scale devices.

This long-term trend makes it interesting and important to study reversible computer architectures and algorithms even today, because no matter the precise details of the future nano-scale device technologies that might become dominant, we can expect that using them in an asymptotically reversible way will confer substantially more computational power, in many applications for all but the smallest-scale machines. We will need reversible architectures and algorithms eventually; we can get a head start by designing them today. In the next chapter, we describe what we have learned along that direction so far.

Chapter 9

Design and programming of reversible processors

In the previous two chapters we reviewed a variety of computing technologies, ranging from the standard VLSI technology of today to visionary nanotechnologies that we cannot yet manufacture, that are all fundamentally capable of operating reversibly in a time-proportionate way that facilitates the central scaling advantages that we discussed in chapter 6. As technology improves, the potential advantage from reversibility becomes increasingly great.

However, independently of the precise hardware, we can ask: How should a reversible machine be programmed, so as to realize these potential benefits? In this chapter, we begin to answer this question, by describing our experience illustrating that it is actually quite straightforward to design reversible microprocessor instruction sets and reversible programming languages that allow the inherent asymptotic cost-efficiency of the underlying logic hardware to be preserved at higher levels. We describe the assembly language, high-level language, and compiler for the reversible processor designed in our group. We also give examples of reversible programs and algorithms, including a constant-space, linear-time simulation of a reversible physical system.

The overall message of this chapter is that once some basic concepts of reversible computing are understood, programming reversible hardware need not be significantly more difficult than programming normal machines. In this chapter we primarily focus on serial programming, but we close with some thoughts on the need for fundamentally new kinds of abstraction and programming techniques for expressing parallel "physical" algorithms.

9.1 Context of this work

First we briefly review the historical context of the present reversible systems design effort.

9.1.1 Previous reversible architectures

The first reversible computer architectures that we know of were designed by Barton (1978, [11]) and Ressler (1981, [140]) as thesis projects at MIT. These designs were based on the conservative (reversible and 1-bit-conserving) logic model developed by Fredkin and Toffoli (*cf.* [74]). The "conservative" aspect of the model actually seems rather irrelevant to efficiency issues, since both conservative and nonconservative logic systems can simulate each other with only small constant-factor overheads.

Several years later (1994), Hall [81] described a reversible instruction set architecture based on his "retractile cascade" reversible circuit style and the PDP-10 instruction set.

9.1.2 Pendulum architecture

In 1995, Vieri [179] developed the first version of the Pendulum architecture. Pendulum was unique in that it was the first reversible computing architecture designed for implementation in a real reversible silicon technology (SCRL, see §7.5). For conceptual simplicity and ease of implementation, it was a RISC (Reduced Instruction Set Computing) style architecture, in contrast to the CISC basis of Hall's architecture.

The original version of the Pendulum instruction set archietcture (PISA) had a few drawbacks. There was a lack of software control over garbage data, which meant that the architecture was not capable of realizing the full potential asymptotic cost-efficiency afforded by SCRL. (For example, one could not efficiently run Bennett's 1989 algorithm [19] or reversible physical simulations on it.) Also, the instruction set did not guarantee full reversibility independently of program correctness, which precluded some of the possible alternative applications for reversibility, such as bi-directional debugging (see ch. 10).

In our work we therefore studied several improved variations of PISA (*cf.* [64, 67]), which were used in our prototype reversible processor [65] and in the compiler design effort. Yet another improved and simplified version of PISA is being implemented now by Vieri for his Ph.D. dissertation research [178]. Since reversible architectures still only exist for purposes of isolated academic research, there has not yet been much need to standardize on a particular version of the instruction set. This would be easy enough to do at a later time, if and when more widespread interest develops.

Appendix B lists the version of PISA that we used for developing our reversible high-level language and compiler, which we will discuss later, in §9.4.

9.2 Reversible instruction set architectures

We now delve into some of the important issues in the design of reversible instruction set architectures in a bit more detail.

9.2.1 Asymptotic efficiency

One important desideratum for a reversible instruction set architecture is that it should be possible to write programs for it that perform tasks with the same *asymptotic* efficiency that could be achieved by a custom reversible circuit for that task. That is, the architecture should not hide the underlying efficiency of the circuit model. A sufficient condition for this is if a program for the processor can efficiently simulate a model of the hardware.

Of course, a single serial von Neumann style processor cannot be expected to be asymptotically as efficient as an arbitrarily-large parallel circuit. So in order to judge such an processor fairly, we imagine that it just represents a single node in an arbitrarily-large mesh of such processors, and ask whether the resulting mesh can simulate arbitrary circuits efficiently.

One consequence of this criterion is that the architecture should permit running arbitrarily-long reversible physical simulations with only *constant* space usage. This immediately rules out instruction set architectures that provide reversibility by constantly pushing garbage data onto an ever-growing stack. In such architectures, space usage increases asymptotically with time, and so cannot be bounded by a constant in an arbitrarily long simulation.

For instance, the first version of the Pendulum architecture [179] had this problem, since all branches and many data operations pushed information onto a “garbage stack” which could not be uncomputed except by reversing the entire processor. So one could not write a constant-space loop, for example.

Later versions of Pendulum avoided this problem by allowing garbage data to be uncomputed in software, and by using paired branches to avoid the generation of garbage during control flow operations. This type of branching is an important concept in reversible instruction sets and deserves further discussion.

9.2.2 Use of paired branches

The control flow instructions such as branches and jumps in normal architecures are generally not reversible, because after branching to a location, there is in general no way of telling which of many possible locations one might have branched from.

Branch stacks. One way to make branches reversible would be to treat every branch like a subroutine call, in that the address branched from (the old value of the program counter, or PC) becomes pushed onto a special stack, along with the value of a branch counter that has been keeping time since the previous branch. In reverse, when the branch counter reaches zero, one pops the previous PC and branch counter values from the stack, which undoes the branch.

Vieri's original Pendulum architecture took this approach, but avoided the need for the branch counter by placing special "come-from" instructions at branch destinations. These would trigger the popping of the old PC value when encountered when running in reverse.

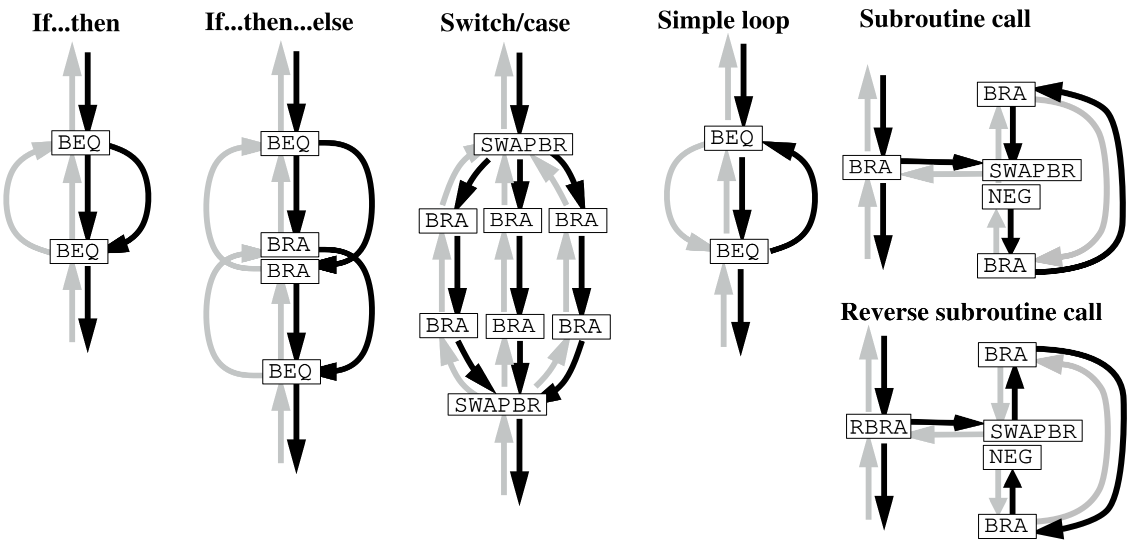
Unfortunately, though it is simple, the branch stack approach is not asymptotically efficient because of the potentially large size of the stack of branch information that must be maintained. A loop that executes N times would require $\mathbf{\Theta}(N)$ space with this approach, even if it only explicitly manipulates a constant number $\mathbf{\Theta}(1)$ of variables.

A better approach. To solve this problem, and avoid generating extra garbage data on every branch instruction, one can take the approach of using *paired branches*. That is, the destination of each branch instruction should be another branch instruction, which refers back to the original instruction and takes care of absorbing the old PC value, or performing the backwards branch when running in reverse. The resulting control flow constructs are completely time-symmetric.

Control flow structures. Any of the usual high-level patterns of structured control flow can be implemented using paired branches. Figure 9.1 schematically illustrates some examples. Detailed examples of some of these can be seen in the example program in §9.3, p. 232.

If/then statement. A reversible implementation of a plain conditional statement (if/then) requires two branch instructions, one at the beginning and one at the end of the body of the conditional code. Before the first branch, the condition being tested is computed, and the desination address is loaded. Then if the condition fails, the branch instruction changes the machine state so that before the next instruction, the PC will be updated to point to the branch at the end of the IF body. For example, the PC could be swapped with a register holding the destination address. Then when the second branch executes (testing the same condition), it toggles the mode back to normal sequential operation, and the code after the second branch uncomputes the condition and unloads the address of the first branch.

There are a variety of straightforward mechanisms which will work for low-level implementation of paired branch functionality like this. We describe one in appendix B.



If...then

If...then...else

Switch/case

Simple loop

Subroutine call

Reverse subroutine call

Figure 9.1: Abstract schematics of some reversible control-flow structures, using the PISA instruction set of Appendix B. Vertical arrows represent sequential flow of control through the program, other arrows represent non-local flow of control. Gray arrows represent the flow of control when the code is run in reverse.

If/then/else. Like if/then, but each of the two conditional branches is paired with an unconditional branch which serves to separate the two alternative paths through the construct. See the second diagram in figure 9.1.

Switch/case statement. The destination address is computed based on data, and we branch to it using a branch instruction that gets its destination from a register. The entry point is a literal branch that refer to the dispatching instruction to receive control from it. At the end of the case body we branch to a literal point in the outer context at which the address of the end of the case body is uncomputed based on the same data that was used to compute the start address.

Simple loop. A simple loop (such as a FOR loop) has the same structure as an IF except that the branches at the start and end are activated at different times. The first branch tests a loop-entry condition, and the second a loop-exit condition—these may or may not be identical. When we first hit the initial branch, the loop-entry condition succeeds and allows us into the body of the loop (the branch is not activated). When we hit the trailing branch, it should activate if we are to perform another loop iteration. We go back up to the first branch, which should now also be activated, to receive us. The process repeats until the final branch condition fails (loop exit condition succeeds) and we fall out of the loop.

A WHILE or UNTIL loop can also be implemented this way, so long as there is some piece of information maintained within the loop that is sufficient to tell when the loop was first entered; this information controls entry to the loop. For example, it suffices to keep a count of the number of times through the loop; the loop is only entered (initial branch deactivated) when this is zero. One must remember though that in that case, the count remains around after the loop is completed.

More complex loops with alternative entry/exit points can be constructed, but they require matching code on the outside of the loop that knows how to dispatch or absorb control to/from the proper points inside the loop.

Subroutine call. There are several ways to implement this; one of the simplest is the structure shown in the figure. The entry/exit points of the subroutine are at the *same* address; the body of the subroutine can loop around so as to exit from the same point where it was entered. The call instruction refers to the entry/exit point. We branch to it, for example by loading up a special branch register with the offset. The body of the subroutine saves away the branch register, using a free register and/or a stack. The offset is negated, and then used again for the branch at the subroutine's return. We branch back to the call instruction, which re-absorbs the offset. This is the approach that was used in most of the recent variants of the Pendulum design.

An alternative mechanism is to have separate entry and exit points at the beginning and end of the subroutine. The code before the call loads up the address of the start of the subroutine into a register. We branch to it (swapping PC and register, for example). The body of the subroutine saves away the address of the branch instruction that we came from. This same address is then used for the branch at the subroutine's return. We branch back to the entry point, which activates and receives the address of the subroutine's end point back into a register. Then the code after the call uncomputes the address of the end-point.

One very nice feature to have in the subroutine call instruction is a way to reverse the processor direction when entering and leaving the subroutine. This way the same subroutine can be used to either compute or uncompute some result, depending on which direction it is called in. This may reduce program size by up to a factor of two, compared to the alternative approach of maintaining two separate versions (one forward and one backward) of every subroutine whose results may need uncomputing.

Summary of paired branches: All the standard structured control-flow constructs are straightforward to implement using paired branches. This approach helps to minimize the generation of garbage data as code is executed.

9.2.3 Reversible logic/arithmetic operations

Many standard machine data instructions are already reversible. Fixed-length integer addition and subtraction, logical and two's complement negation, bit rotation, and exclusive-OR'ing one register into another are all examples. However, other operations, such as ANDing one register into another, normally lose information. There are several means for allowing such operations to be performed reversibly. One is to save on an internal machine stack the word that would otherwise be demolished. But this would not give the programmer the opportunity to uncompute that word later when it is no longer needed. A better approach is to require that operations like AND should write their result into a third register, distinct from the other two inputs, which is either required to be previously clear, or else the result is XOR'ed into it, or added into it, for example. Then a matching instruction should be provided that can uncompute the result. (In the case of XOR'ing the result into the destination register, the uncomputing instruction is just the same instruction over again.)

9.2.4 Data transfer operations

Similarly, operations that move values around, such as between registers and other registers or memory, must either (1) require that the destination be initially clear, (2) XOR/ADD the source into the destination, or (3) swap the source and destination. A swap can be implemented with 3 XORs, or 2 XORs if one location is initially clear.

9.2.5 Hardware-guaranteed reversibility

An important issue to decide about a reversible architecture is: Does it guarantee full reversibility of operation at the hardware level, independently of program correctness? If the hardware does not guarantee reversibility, then there is the risk that an incorrect program could inadvertently cause dissipation, and burn up the hardware.

One could imagine instead guaranteeing reversibility at the software level, in a compiler, but there are some problems with that approach. First, if the compiler allows asymptotically efficient reversible programs, then it can only guarantee reversibility by compiling programs to a set of guaranteed-reversible pseudocode primitives, which then might as well have been implemented directly by the hardware.

To understand why it is hard to introduce a guarantee of reversibility above the instruction set level, consider for example an instruction set whose expanding logic instructions (*e.g.*, NAND) do not guarantee reversibility unless the destination register is initially empty. So, in order to guarantee reversibility, the compiler has to guarantee that the program never accidentally does a NAND into a destination register that contains a value that is possibly non-zero.

Now, suppose one were to write a section of code that starts with some initially empty memory, and then uses that memory as scratch storage for performing some complex computation. One may know how to prove mathematically that after the code segment is finished, the memory it worked with will all have been restored to zero.

For example, suppose I first move to the scratch storage two primes a, b ($a < b$) that I wish to multiply. I generate their product ab and put it elsewhere, then I run a factoring algorithm to compute a, b given ab, and thereby subtract or XOR out a, b from the scratch storage, leaving me with just ab. Since I have proven that my factoring algorithm is correct, I know that the locations that held a, b are now empty, and I can go on to use them for some other computation. The emptiness of the location is an invariant that I can prove is maintained by my code section.

On the other hand, unless the compiler is required to be able to find proofs of such invariants, or the user is required to supply them, the compiler cannot assume that after my user-defined manipulation is completed, the locations are really empty. Therefore, the compiler will have to consider those locations to contain indestructible garbage, and it will have to allocate new memory for use in future operations. If the program involves a long sequence of manipulations like this, it will not be as space-efficient, asymptotically, as it could have been if the compiler was not responsible for guaranteeing reversibility.

9.3 Simple example PISA program: Multipication algorithm

As a simple example of reversible programming techniques, figure 9.2 shows a simple hand-coded multiplication subroutine for one 32-bit version of PISA. See appendix B for detailed specifications of individual instructions. The registers used in the routine are documented in table 9.1. Let us go through this routine line-by-line, to explain its operation. It is based on the simple grade-school multiplication algorithm, in which we just march through the digits of one multiplicand, multiplying them individually by the other multiplicand, and adding up the partial products, shifted appropriately, to form the complete product.

Line 1: `subtop:  BRA subbot` This is the first of a pair of labeled unconditional branches, pointing to each other, that frame the entire subroutine. These permit the subroutine to exit from the same point as where it is entered. They also have the side effect that if the flow of control encounters the subroutine sequentially, it will just skip over it.

Line 2: `mult:  SWAPBR R2` This is a conventional subroutine entry/exit point. On entry, the branch register is saved away into register R2, which is reserved for this purpose. On exit, R2 is swapped back into the branch register, causing control to be transfered back to outside the subroutine.

```
;; Label     Instr  Args            ; Pseudocode description
;; --------  ------ ----            ; ----------------------
   subtop:   BRA    subbot          ; MULT top.
   mult:     SWAPBR R2              ; Subroutine entry/exit point.
             NEG    R2              ; Negate offset to return to caller.
             EXCH   R2 R1           ; Push return offset to stack.
             BRA    alloc4          ; Allocate 4 empty registers (R28-R31).
             ADDI   R31 32          ; limit <- 32
             ADDI   R2 1            ; mask <- 1
   looptop:  BNE    R30 R0 loopbot  ; unless (position != 0) do
             ANDX   R28 R3 R2       ;     bit <- m1&mask
   iftop:    BEQ    R28 R0 ifbot    ;     if (bit != 0) then
             SLLVX  R29 R4 R30      ;        shifted <- m2<<position
             ADD    R5  R29         ;        product += shifted
             SLLVX  R29 R4 R30      ;        shifted -> m2<<position
   ifbot:    BEQ    R28 R0 iftop    ;     end if
             ANDX   R28 R3 R2       ;     bit -> m1&mask
             RL     R2 1            ;     mask <=< 1 (rotate left by 1)
             ADDI   R30 1           ;     position++
   loopbot:  BNE    R30 R31 looptop ; and repeat while (position != limit).
             SUB    R30 R31         ; position -> limit
             ADDI   R2 -1           ; mask -> 1
             ADDI   R31 -32         ; limit -> 32
             RBRA   alloc4          ; Deallocate 4 registers (R28-R31).
             EXCH   R2 R1           ; Pop return address.
   subbot:   BRA    subtop          ; MULT bottom.

   alloctop: BRA    allocbot
   alloc4:   SWAPBR R2          ; This sub-subroutine frees
             NEG    R2          ; 4 registers for use in the
             ADDI   R1  1       ; MULT subroutine.  It leaves
             EXCH   R31 R1      ; the stack pointer pushed
             ADDI   R1  1       ; above, but we don’t mind.
             EXCH   R30 R1
             ADDI   R1  1
             EXCH   R29 R1
             ADDI   R1  1
             EXCH   R28 R1
   allocbot: BRA    alloctop
```

Figure 9.2: Hand-coded reversible assembly-language multiplication routine. The registers used are documented in table 9.1.

Register	Variable name	Purpose
`R0`	`ZERO`	Constant zero.
`R1`	`SP`	Stack pointer.
`R2`	`SRO`	Subroutine return offset.
”	`mask`	Bit in some position 0–31.
`R3`	`m1`	Arg 1: First multiplicand.
`R4`	`m2`	Arg 2: Second multiplicand.
`R5`	`product`	Arg 3: Product accumulator.
`R28`	`bit`	A single bit of `m1`, in place.
`R29`	`shifted`	`bit`, shifted to proper position.
`R30`	`position`	Index of current bit position.
`R31`	`limit`	Bit position limit (32).

Table 9.1: Registers used in the MULT routine shown in figure 9.2.

Line 3: `NEG R2` This negates the subroutine return offset so that when we exit the subroutine we will take exactly the opposite offset of the one that got us into the subroutine, so that we will return to exactly the point where we were called from.

Line 4: `EXCH R2 R1` `R1` is by convention the stack pointer. This instruction pushes `R2` onto the (presumed empty) top-of-stack location, so that `R2` will be available for use in calling further subroutines. (And also to be a temporary variable.)

Line 5: `BRA alloc4` This is a call to the subroutine `alloc4` (see lines 25–36) which simply pushes the upper four registers onto the stack, so we may safely use them for holding temporary values. (This is a “callee saves” register saving convention.) Routines such as `alloc4` may be shared by many subroutines.

Line 6–7: Here we just initialize a couple of registers. `limit` is just a constant 32 for use in the loop termination condition. `mask` is a bit, initially at position 0.

Line 8: `fortop:  BNE R30 R0 forbot` This is the loop entry condition. The loop is entered if the `position` variable is zero, which initially it is. Otherwise the loop would be skipped over.

Line 9: `ANDX R28 R3 R2` Simply extracts the desired bit from the first multiplicand.

Line 10: `BEQ R28 R0 ifbot` Skips the IF body if the extracted bit was zero.

Line 11: `SLLVX R29 R4 R30` Shifts the second multiplicand by an amount corresponding to which bit of the first multiplicand we are currently multiplying by.

Line 12: `ADD R5 R29` Add the appropriately-shifted second multiplicand into the accumulating product.

Line 13: Undo line 11 to clear register `R29`.

Line 14: Absorbs the transfer of control if the IF body was skipped.

Line 15: Undo line 9 to clear register `R28`.

Line 16: `RL R2 1` Shift the bit-mask left to the next position.

Line 17: `ADDI R30 1` Increment the position index.

Line 18: `loopbot:  BNE R30 R31 looptop` Loop exit condition. If we're not yet at the position limit, then branch back to the loop top.

Line 19: `SUB R30 R31` `position` is now equal to `limit`, so subtract `limit` out of it to restore it to zero.

Lines 20–21: Uncompute the constants that we set up in lines 6–7. `mask` is 1 because it has rotated from position 0 by 32 positions, back to position 0.

Line 22: `RBRA alloc4` Reverse-call the register-allocation subroutine, to restore the caller's registers back off the stack.

Line 23: `EXCH R2 R1` Pop return address back off the stack.

Line 24: `subbot:  BRA subtop` Subroutine bottom: branch back up to `subtop`, to get back to the entry/exit point.

Line 25–36: Register allocation subroutine. Alternates between incrementing the stack pointer, and exchanging a register we want to use with the current stack location. Effectively, pushes those registers onto the stack. They can be restored from the stack later by calling the subroutine in reverse.

9.3.1 Discussion

The routine illustrates several of the general reversible programming techniques we discussed in section 9.2. In particular, note the following points:

- Subroutine calls are implemented using save/restore of the branch-register offset, and a single subroutine entry/exit point.
- Any registers we use to hold temporary values are always restored to zero when we are finished with them.
- The subroutine works by accumulating the desired result in one of its arguments. This product can later be uncomputed by simply calling the subroutine again in reverse (using RBRA).
- Similarly, the auxilliary routine `alloc4` is called in reverse at the end of the MULT routine, in order to undo its earlier effects.
- An IF functionality is implemented via a matching pair of branches.
- A looping functionality is implemented using a pair of branches that determine the entry and exit conditions for the loop.
- Note that although the routine uses order n repetitions of the inner loop (where $n = 32$ is the word length), it only uses a constant amount of temporary storage, just as an irreversible version would. This is a good example of an algorithm that requires asymptotically no more space or time to do reversibly than irreversibly.

This concludes our discussion of reversible instruction sets. Many variations on the above theme are of course possible, but the above discussion should address many of the common underlying issues. More details would of course be necessary to support features such as floating-point arithmetic, arithmetic overflows, and asynchronous interrupts. But we believe that most of these features will be similarly straightforward to implement reversibly.

9.4 Reversible programming languages

9.4.1 General issues

If we are seriously considering the implications of building a reversible computer, then naturally we will want to investigate the possibility of programming that computer in a high-level language, rather than directly in machine code.

One approach to high-level programmability would allow programs to be written in a standard, irreversible programming language (for example, C) and then provide an interpreter or translator that allows them to be run on a reversible architecture. This would be straightforward, given the known general algorithms for simulation of irreversible machines on reversible ones (see §3.3).

However, this approach incurs a cost in terms of asymptotic inefficiency. As we saw in chapter 3, general-purpose reversible simulations are expected to require increased asymptotic time or space. However, for a particular problem, there may be an alternative reversible algorithm that is either just as asymptotically efficient as the original irreversible algorithm, or is only insignificantly less efficient. But an asymptotically good reversible algorithm for a problem cannot in general be expected to be a straightforward translation of the best irreversible algorithm. It may require a completely different structure. (For an example, see §9.5.5.) The programmer's ability to write asymptotically well-performing programs for the machine will in general be crippled if its underlying reversibility is hidden from him/her. Since all the known general-purpose simulation techniques incur at least polynomial asymptotic overheads, writing efficient reversible programs requires exposing a universal set of reversible constructs that incur *no* hidden asymptotic overheads, so that the programmer can explicitly manipulate information in a way that constitutes an asymptotically good reversible algorithm for the problem at hand, an algorithm that no automatic "reversibilizing" system could be expected to have discovered.

This leads to an approach wherein the input is allowed to be in a general irreversible language, but if a given program only uses a certain reversible subset of that language's constructs, then that program will be compiled in such a way that it incurs no asymptotic overheads, compared to what the programmer could have written if he were hand-coding in assembly language.

For example, if one is coding in C, but if assignment statements are eschewed in favor of reversible mutation statements such as `+=`, and if all local variables are asserted (see the Unix `assert(3)` manual page) to be restored to zero before function return, and if other `assert()`s are used to inform the compiler of loop entry conditions, and if one avoids frequent use of dynamic memory allocation (because garbage collection is irreversible; see [8, 7]), and overall if one's programs are written essentially in a style that looks basically like assembly language augmented with named variables, nested expressions, and structured control-flow, then it should be possible to compile the resulting programs to efficient reversible machine code without a need for an ever-growing garbage stack, or other asymptotically inefficient run-time support mechanisms.

Coding up a complete PISA-targeted compiler system for even a fairly simple conventional programming language was deemed to be too time-consuming a goal to fit within the scope of the present research project, especially given that there is no commercial interest yet in reversible machines, and also that no lessons of even much academic interest would be expected to be learned from such an exercise.

However, it was deemed feasible and useful to write a simple compiler for an extremely simple toy programming language similar to a reversible subset of C. In reference to the 1-letter programming language naming convention, we called our

```
(defsub mult (m1 m2 prod)
   ;; Use grade-school algorithm:
   (for pos = 0 to 31               ; For each of the 32 bit-positions,
      (if (m1 & (1 << pos)) then    ;   if that bit of m1 is 1, then
         (prod += (m2 << pos)))))   ;     add m2, shifted over to that
                                    ;       position, into prod.
```

Figure 9.3: A simple, efficient multiplication routine in the R language. This is essentially the same algorithm as that used in the hand-coded assembly-language routine in figure 9.2 (p. 233). Note that the high-level code is much more concise. The compiler (see §9.4.3) converts this routine into a sequence of 66 assembly code instructions: not quite as concise as our 36-instruction hand-coded routine, but reasonable.

language "R." [1]

9.4.2 "R," a reversible language

The essence of R is a very simple procedural C-like language based on machine-supported fixed-precision integer arithmetic, with nested expressions, arrays, efficient control flow statements, and with a Lisp-like (parenthesis-based) syntax for ease of parsing. The user-level constructs in the current version of R are documented in appendix C. To quickly illustrate what R code looks like, figure 9.3 shows a simple multiplication subroutine.

R is not actually the first reversible high-level language. Recently, we learned that around 1982, Chris Lutz and Howard Derby created a reversible programming language called "Janus" for a class at CalTech. (Our source is a letter [114] from Lutz to Rolf Landauer, describing the language.) It turns out that Janus's feature set is very similar to R's, which is interesting given that the two languages were developed entirely independently. However, Janus ran only under SIMULA on a DECSYSTEM-20, and it may no longer exist anywhere in usable form.

Also, Henry Baker described a reversible Lisp-like language called "Ψ-lisp" in a 1992 paper [8]. Ψ-lisp was based on so-called "linear" functional languages, *cf.* [7], which have the additional restriction that references must be conserved; there is always exactly one pointer to any given storage cell. Ψ-lisp is theoretically interesting, but it is not clear to us whether it constitutes an asymptotically efficient programming language. Also, the reasons for and implications of linearity seem to us to be mostly

---

[1] After naming our R language, we learned there is a statistics package called "R": see `http://www.ci.tuwien.ac.at/R/contents.html`. So if our R ever hits the big-time, we may want to rename it "RL", or something, to avoid confusion.

orthogonal to the reasons for and implications of reversibility.

9.4.3 The R compiler

In addition to specifying the R language, I also wrote a simple compiler for translating R programs into assembly code executable on a certain version of the Pendulum architecture. The main points of this exercise were (1) to demonstrate that the R language, as envisioned, is easy to implement, and (2) to provide a convenient way to create substantially-sized test programs for the Pendulum architecture.

Since both the Pendulum architecture and the R language design are in flux, the compiler was written so as to make modifications very easy. The compiler is written in Common Lisp, and works through a process similar to macro-expansion, where high-level language constructs are broken down into sequences of lower-level constructs, and the process finally bottoms out when the lowest-level constructs are translated directly into assembly language instructions. This design makes it very easy to add new high-level constructs, or change the compiler to support a different low-level architecture.

The R compiler did indeed turn out to be completely straightforward to implement (no surprises), and it was used to successfully compile a number of test programs, which were run under a simulator for the Pendulum architecture. It would be easy to extend the language and write more programs, if that were a priority.

The R compiler source code, internal constucts, and a brief usage summary are given in appendix D. The source files can also be downloaded from `http://www.ai.mit.edu/~mpf/rc/memos/M08/*.lisp`.

9.5 Reversible algorithms

In this section we summarize what we have learned about efficient reversible serial algorithms for a variety of problems in computer science and physics. These include sorting, arithmetic, matrix operations, graph problems, and physical simulations. Due mainly to a lack of time, we have not written down sample code for any of these except our physical simulation (appendix E). An important area for future work is to specify a broad range of reversible algorithms in complete detail, with sample code. However in most cases the details are fairly obvious and straightforward.

We focus on serial algorithms in this section primarily for simplicity. To support the long-term applications of reversibility in massively parallel computing, the development of good reversible parallel algorithms (for a mesh architecture) for a variety of problems would also be desirable.

9.5.1 Sorting

Sorting a list in place is an inherently non-reversible operation, because information is lost: namely, the original order of the elements. So in general, a reversible sort must produce some extra garbage data. For arbitrary lists, the minimal worst-case garbage is $\mathbf{\Theta}(\log n!)$ since that is the number of bits required to specify the original permutation of the elements.

Simple insertion sort performs $\mathbf{\Theta}(n^2)$ comparisons in the worst case. Hall [81] observed however that only $\mathcal{O}(n \log n)$ garbage bits need be generated to run this algorithm reversibly: n integers each $\mathcal{O}(\log n)$ bits long telling how far each element was moved down the list before being inserted in the proper place. If these numbers are stored as variable-length bit-strings instead of fixed-length words, the garbage space usage becomes $\mathbf{\Theta}(\log n!)$, exactly the minimum.

Similarly, we realized that any of the standard efficient $\mathbf{\Theta}(n \log n)$-time comparison-based sorting algorithms, such as quicksort, are easy to turn into good reversible algorithms, by simply saving away bits giving the result of each comparison, telling whether two elements were exchanged or not on any given step of the algorithm.

Even radix sort, which takes $\mathbf{\Theta}(n)$ time for n $\mathbf{\Theta}(1)$-size elements, can be easily turned into an efficient reversible sort which takes $\mathbf{\Theta}(n)$ time and produces $\mathbf{\Theta}(n)$ garbage.

9.5.2 Arithmetic

Addition or subtraction of one n-bit number into another can be performed reversibly in $\mathbf{\Theta}(n)$ time with no garbage data.

The simple $\mathbf{\Theta}(n^2)$-time grade-school algorithm for multiplication (*e.g.*, see fig. 9.3) can also be performed reversibly with no loss in asymptotic efficiency, and no garbage other than the operands (if they are no longer needed). If the multiplicands are specified to be nonzero, then one of them can be uncomputed based on the product and the other multiplicand, thus reducing the garbage further. (After computing the product, the multiplicand can be uncomputed using $\mathbf{\Theta}(n^2)$-time division.)

If the multiplicands are prime and sorted, then in principle *both* of them could be uncomputed from the product reversibly, and the operation would create *no* garbage (since the number of bits in the product will be roughly the sum of the number of bits in the two multiplicands). However, since there is no known efficient classical algorithm for factoring, doing this is in general very slow.

9.5.3 Matrices

Similarly, the simple $\mathbf{\Theta}(n^3)$ algorithm for multiplying $n \times n$ matrices can also be efficiently reversible. If the left multiplicand is nonsingular, then the right multiplicand can be uncomputed given the product, reversibly and efficiently ($\mathbf{\Theta}(n^3)$).

9.5.4 Searches

A systematic depth-first or breadth-first search of a tree can be carried out reversibly; the nodes are visited in a particular sequence, and the prior node in the sequence can be determined from the current node. So, for example, the naive SAT algorithm, of generating and testing all possible assignments to the boolean variables, incurs no additional asymptotic overheads when performed reversibly.

9.5.5 Graph problems

For the all-pairs shortest-path problem, one of the best algorithms is the Floyd-Warshall algorithm, which takes time $\mathbf{\Theta}(n^3)$, space $\mathbf{\Theta}(n^2)$. This is an example of an algorithm which appears to require asymptotically either more time or space when performed reversibly. The reason is that the Floyd-Warshall algorithm performs $\mathbf{\Theta}(n^3)$ irreversible updates of array elements in working storage, in the worst case. Performing all those updates reversibly would thus require $\mathbf{\Theta}(n^3)$ temporary storage, significantly worse than the original irreversible algorithm.

However, if one instead uses an alternative algorithm, of repeatedly "squaring" a connectivity matrix between graph edges, then all-pairs shortest path can be performed reversibly in time $\mathbf{\Theta}(n^3 \log n)$ and space $\mathbf{\Theta}(n^2 \log n)$—only slightly worse than the irreversible Floyd-Warshall algorithm on both time and space. Thanks are due to F. Thomson Leighton and his collaborators for pointing out the high reversible efficiency of this alternative approach, in personal discussions.

This example of the all-pairs shortest path problem illustrates how the best reversible algorithm for a problem might not necessarily correspond to a simple modification of the best irreversible algorithm for that problem, which is why hiding the fact of a machine's underlying reversibility from the programmer (or algorithm designer) is not a good idea.

9.5.6 Physical simulations

Direct, dynamic spatial simulation of reversible physical systems is often straightforward to perform reversibly with no asymptotic overheads compared to an irreversible version of the simulation. This is true at least when the system model being simulated is reversible. Since physics really is reversible at a low level, such models are often appropriate, and can exhibit much more stable and realistic behavior than competing irreversible dynamic models [75].

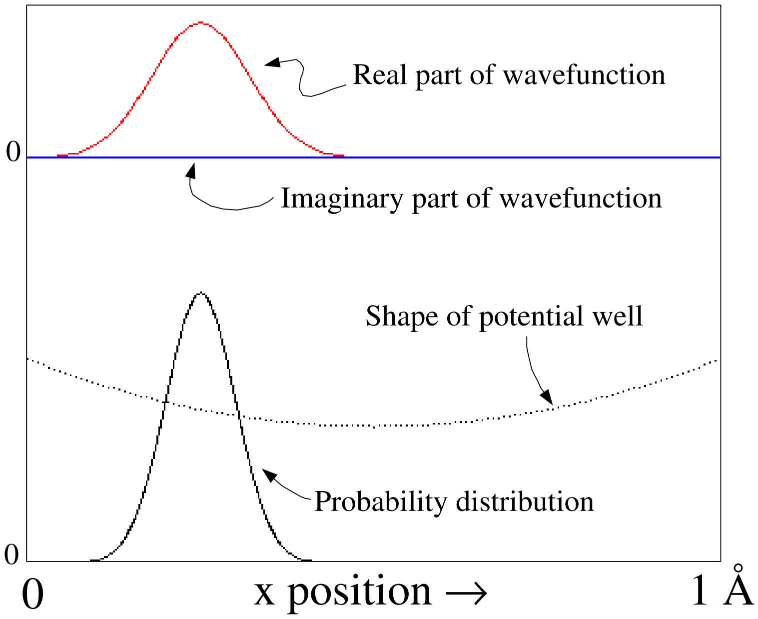


Figure 9.4: Example of an initial state of the Schrödinger wavefunction simulation. A wave for an initially stationary electron is placed up along the side of a 1 Å wide parabolic potential well. Its initial potential is approximately 4000 eV, relative to the bottom of the well. (A very strong potential given the small size of this space!)

As an example, we wrote a reversible simulation of the evolution of a simple quantum wavefunction according to Schrödinger's wave equation (see figures 9.4 and 9.5 for example output). The original irreversible version of the algorithm behaved well for a few hundred steps, but was eventually swamped by ever-growing artifacts of the simulation technique. We then reimplemented our update rule so that it would perform a perfectly reversible transformation of the wavefunction state on each step. The artifacts disappeared; the system was never observed to blow up again even after runs lasting millions of steps.

Moreover, the new version of the algorithm could be implemented under our reversible programming language in such a way that *no* garbage data would be accumulated; the algorithm required only *constant* space on our reversible architecture, independently of the number of simulation steps that were performed.

The mathematical derivation and code for the algorithm (in C, R and PISA versions) are given in appendix E.

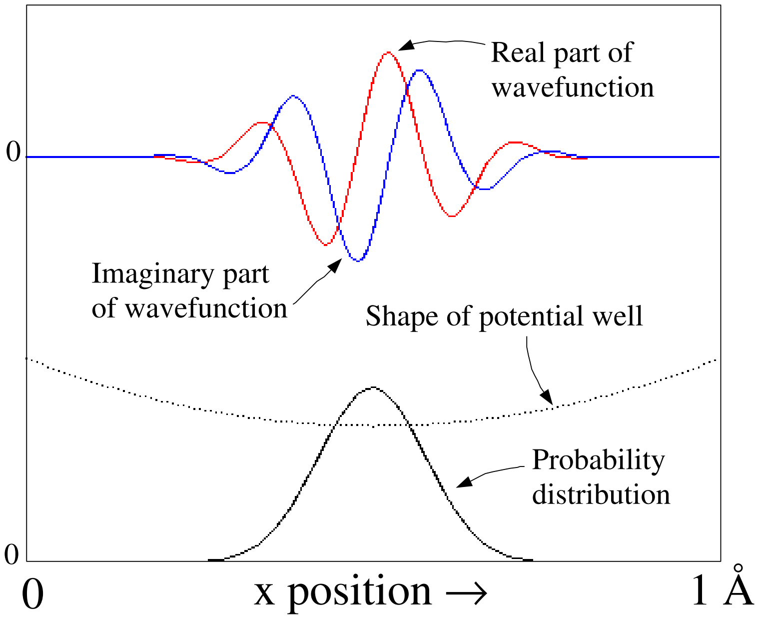


Figure 9.5: The state of the simulated wavefunction of fig. 9.4 after ∼ 1000 simulation steps, representing about $\frac{1}{2}$ atto-second (5×10^{-19} sec) of physical time. The electron wave packet has fallen to the bottom of the potential well, and is now moving to the right at about $\frac{1}{10}$ of the speed of light.

9.6 Operating system issues

So far I have discussed reversible computer programming from the point of view of running a single reversible program at a time. However, if we want to allow for the possibility that eventually reversibility will, for one reason or another, be in widespread use for general-purpose computing, then we should address the issue of how users might be enabled to run multiple programs concurrently on a single reversible computer, since this sort of multitasking has proven to be extremely useful in current computer systems.

Running multiple programs concurrently is traditionally part of the job of an operating system. In this thesis, we have not attempted to study how the wide range of services provided by modern operating systems might be implemented on a reversible computer, since we believe that most of these services would straightforward to implement reversibly (given our general purpose reversible-programming framework) and thus would not be very interesting.

However, we now briefly examine how a multitasking facility might be implemented on a reversible computer, since we know of some interesting problems associated with that goal.

If the multiple reversible programs are not interacting at all with each other, then it appears straightforward to run them concurrently on a reversible processor. There simply needs to be a mechanism for CPU control to be periodically transfered from within the individual programs out to an external scheduler, which decides which other program to run next, and makes the appropriate changes to machine state to transfer control to within the body of that other program, in order to continue from the point where that program left off.

One way to achieve this "escaping to the scheduler" would be to simply have the compiler periodically insert in program code instructions that transfer control to the scheduler. Timed hardware interrupts are also quite possible and straightforward.

In fact, even if the computer's instruction set includes software control over the direction of execution, and even if the running of individual programs involves those programs switching the CPU direction, and running pieces of their code backwards and forwards, it is still straightforward to integrate this activity with process switching, by having the scheduler keep track of which way the CPU was going when a process was last being run, as part of the saved state of that process.

However, a problem arises if we wish to allow multiple concurrently executing reversible programs to communicate with each other. Namely, what happens when a program reverses over an I/O instruction that communicates with another reversible process?

For example, suppose process A is running forwards and outputs a piece of data X through a stream-like facility, and later the datum X is received by another process B, which is also running forwards. Then process A reverses CPU direction, and attempts to undo the operating system call that sent X. How does the OS handle this situation?

Similarly, looking at the situation from B's point of view, suppose B reverses and undoes the call that received X, and then proceeds forwards again. Will the next piece of data received be X again, or will it be the next datum produced by A after it produced X? (We note that this question is related to the question of whether a theoretical reversible finite automaton (RFA) should be defined as having a one-way read-only input, as Pin defined it (see §3.3.2.2, p. 60), or as having a two-way input with operations to both read and unread data.)

There are several possible answers to this question.

- Disallow reversing over I/O calls. Declare such an action to be illegal, an error. However, this approach seems overly restrictive.
- Allow reversing over I/O instructions, but treat this as a no-op; don't actually undo the I/O action. This is like treating each process as if it were running on a separate reversible processor, but with an irreversible communications channel. This seems OK, but still limited.
- Finally, allow reversing over I/O instructions, but when this happens, actually reverse the flow of data in the data stream. Reversing over output is just like pulling back out the last thing that you output when running forwards; reversing over input is like stuffing the last thing you input back into the pipe.

In future work we may experiment with all three options, and will perhaps think of more. The third option seems the most interesting to explore, however. It raises the further question of what happens when you attempt to reverse over an output instruction, but the program at the other end of the pipe has already pulled out and used the last datum X that you sent.

One possibility is that the program B is then forcibly reversed until it gets back to the point where it received X, so that then X gets stuffed back into the pipe, and A retrieves X (running backwards). However, this seems rude; it prevents B from doing whatever it might have been planning to do with X before reversing over the input instruction that got it.

Another possibility treats the situation more symmetrically. Suppose that reversing over an instruction to output data to a stream really is exactly like getting input from the stream—just like the stream were a totally symmetric, big, stretchy hose where you can stuff things in either end, and get things from either end, but where the things inside always remain in the same order relative to each other.

Well, then if you try to get a datum out, but there is no datum inside, then the normal thing is to *block* and wait until the data is available. *I.e.,* when A tries to reverse over its “output X” instruction, it blocks until B decides to reverse over its “input X” instruction. This is just like what happens in a normal operating system when you try to input from a stream when running forwards, and there is nothing in the stream to be read.

This of course leads to still more problems. What if B pushes back into the pipe a different value than the one that A originally sent? (This can happen whenever programs have control over their own execution direction in a non-trivial way.) Program A might not be expecting to get a different value back when he reverses over his “output X” instruction; and so program A might not function as expected. (If reversibility isn’t guaranteed at the hardware level, and we’re not careful, program A might even fail to reverse properly, and may irreversibly dissipate information to heat.)

All we can say about this problem at this point is that a communications mechanism is not a communication protocol, and if we want multiple concurrent reversible programs to communicate meaningfully with each other along reversible channels such as this, we will have to describe more precisely what the intent of those communications would be. It would help if we had in mind some particular candidate application for which multitasking would be a useful abstraction.

9.7 Parallelism

Finally, we note that we have not said much so far about the design and programming of parallel reversible computers. Of course, one can treat a parallel computer as just a set of interacting serial computers, and program it that way. As long as reversible I/O operations are provided, nothing need be broken. And, as we discussed in earlier chapters, the processors could be organized into a mesh structure that would be very regular, relatively easy to program, and asymptotically optimal.

However, even with the right physical architecture, programming a parallel system as a set of independent interacting serial processes is very hard. It would of course be nice to have a means to express parallel reversible algorithms at a high level, and have a compiler do to the work of translating that high-level parallel algorithm into sequential algorithms to run on the individual interconnected processors.

Such a language would allow us to easily express “physical” algorithms, specifying the movements and interactions of whole “fields” of data extending through the machine. The ideal language would somehow implicitly incorporate the realities of physical constraints, such as that large amounts of data can not all be in the same place at the same time. It might describe data operations in terms that traditionally we associate more with physical processes: moving, sifting, mixing, separating, recombining, chemically reacting, etc. Such analogies become increasingly appropriate when the computing system is designed to admit the fact that information is conserved (in the sense that its transformation are reversible), takes up space, and must physically move from one place to another.

Programmers of such a "physical" programming system would use the skills of a computer systems architect, or even those of a mechanical or industrial engineer, such as an assembly-line designer, as well as the skills of a sequential program coder. The programmer's job would be to design a dynamic assembly line for information within the real physical 3-D space of the machine. He would have an advantage over the factory designer in that all parts of his "factory" (namely, the individual processors) can be reprogrammed and reconfigured dynamically at will to serve his needs. Also since the material being manupulated is just information, there are no concerns with weight, structural support, etc. But there are still concerns about flows! The information takes up space. Where does the unwanted information go when it is no longer needed? The programmer would design pathways for the flow of both useful information and garbage information through the machine.

Good programming tools might perform the detailed routing of these pathways automatically, even dynamically as the system runs. But a good programmer should still occasionally find that concerns with the physical nature of information come into his thinking during the algorithm design process. The expert programmer should not mind expanding his expertise to designing algorithms for the manipulation of information considered as a *conserved* material-like thing, embedded in 3-D space. After all, *that* is what information really is like. The ultimate, best-performing algorithms can never be discovered by those who are afraid to step out of the imaginary serial, random-access, bit-destroying world of programming that we have constructed for ourseves up to now.

Chapter 10

Alternative applications for reversibility

Most of this thesis has focused on the benefits of reversibility that are gained through its reduction of the energy dissipation of computation. However, full reversible operation may be useful for other reasons as well. In this short chapter we briefly survey some of the possibilities.

10.1 Auditable/verifiable/trustable computation

One interesting application for full reversibility in a computer system is to assist in meeting requirements for auditable or verifiable computation. This requirement exists in at least two forms for which reversibility might be useful:

- It should be possible to determine, with high confidence, whether a transient hardware error of some sort (e.g., a random bit-flip in memory) might have occurred during a computation, to help us determine whether the result of the computation can be considered valid.
- The system should ensure that any malicious intruder not having physical access to the hardware is unable to destroy any information stored in the system, and that the intruder's presence and complete actions can always be determined after the fact.

Let us examine how reversibility might be useful for helping to satisfy these requirements.

10.1.1 Detecting transient errors

Full reversibility could be used to detect transient hardware errors as follows. The initial state for the computation is set up, and we record the entire initial state, or if that is too large, a cryptographically secure hash of the entire initial state. Then a computation is run, and produces some result. Then we reverse the processor direction, and run in reverse, back to the initial state. Then we compare the state with the stored state or the checksum. If there are any differences, then some error must have occurred during the computation, and the result should be considered untrustworthy.

If the processor is designed to guarantee reversibility at the hardware level, then any differences in initial state will indicate that a hardware error occurred. However, if the processor only guarantees reversibility under an assumption of the correctness of some piece of software, then differences in initial state could indicate the presence of either a hardware error or an error in that software.

Hardware errors. If the error is a transient hardware error due to some random influence from the environment (such as a cosmic ray shower, perhaps), then it is very unlikely that an error would be missed by this detection scheme, as this would require two independent error events, one during the forward computation and one during the reverse computation, that happened to cancel each other out so that the identical initial state was reached even though the final result might have been corrupted.

If such a transient hardware error is detected, then the user can simply try running the computation again, and repeat until the computation proceeds forwards and backwards with no error being detected. In this way, a very high-confidence result can always be obtained eventually, even if there is a significant probability of transient hardware errors occurring during an individual run of the computation.

Of course, an alternative technique for detecting and correcting transient errors, without resorting to reversibility, is to run multiple copies of a system side-by-side, or run a single machine multiple times, and compare the results, and perhaps compare the entire state at checkpoints along the way.

Reversibility may also be useful for detecting the presence of some kinds of permanent hardware faults, since if the fault is permanent, then presumably the difference between initial states would be the same each time. The time at which the fault first makes itself felt could be rapidly determined by running forwards and backwards for different lengths of time in a binary search pattern. Note, however, that some permanent faults, such as a bitwise logical-complement instruction that consistently fails to flip a certain bit (in both the forwards and reverse directions), may maintain reversibility, and so will not be detected by the process of comparing initial states.

Note also that we will not necessarily catch transient or permanent hardware errors if they affect the state-comparison process as well as (or instead of) the main computation.

Software errors. If the system does not guarantee reversibility at the hardware level, the above technique will also catch those software errors that happen to lead to forward computations that fail to match the corresponding reverse computations. However, other kinds of software errors will not be caught. For example, if the system compiler correctly guarantees that all programs will be correctly reversible, then any errors in user programs compiled by that compiler will remain undetected by this technique.

Note also that if we detect an error in reversibility that is a software error, then an easy way to pinpoint its precise location is to run partial computations forwards and backwards, and find exactly how far forwards you must go before the reversibility of the system is corrupted. A binary search can be used to pinpoint the precise time of the error after O(log(n)) partial computations, where n is the number of steps in the entire computation. The machine state at this time can be examined to help determine the cause of the error. This technique will be a useful tool in debugging software that is intended to ensure reversibility on hardware that does not by itself guarantee reversibility. However, this will of course incur the usual reversibility cost of slower execution time, on traditional serial processors.

10.1.2 Logging or limiting effects of unwelcome intrusions

Suppose we have a requirement that any computer cracker that manages to get past system security measures should (1) be unable to actually permanently destroy any data, and (2) have the complete history of his actions on the system be determinable once his interference is discovered.

A traditional approach to item 1 is to make backups of data and log user activities, but this does not help if the cracker destroys data before it manages to get backed up, or corrupts the backup software itself so that new data does not get backed up properly.

However, if the system guarantees full reversibility at some level that is impossible for the cracker to interfere with (e.g., at the hardware level), then by the definition of reversibility, whatever actions he takes cannot permanently destroy any user data, or any information that the cracker had input to the system in order to do his dirty work.

Additionally, once the presence of the cracker is detected, the machine can be disconnected from the network and reversed to recover any desired earlier state of execution, and all of the cracker's actions can be observed, and the clean state prior to his break-in can be recovered.

However, note that reversibility does not protect us from the cracker's corrupting the system's outputs after the time he breaks in. He could in general still alter the running state of the system so that it produced invalid or misleading data until his interference is discovered, and its effects on the system are undone. Any inputs to the system while it was in a corrupted state would have to be re-input once a clean state is restored. All those inputs could be recovered by backing up over the time after the cracker's interference.

But this suggests an alternative technique for achieving the same protection without requiring reversibility. Namely, one could simply have an incorruptible mechanism for recording the initial state of the system (when it is first turned on in a clean-slate state) and for recording every bit of information that flows into the system, including any timing information, if that is important. If the system is deterministic, then that stored information is sufficient for reconstructing the complete machine state at any later time—the system is "reversible" in the sense that we could always back up to the state at any earlier time by simply going back to the initial state and proceeding forwards from there.

The obvious drawback to this technique is that if the system has been running continuously for a long time, say a year, and we only want to back up a small amount, we will have to spend another year to get up to the desired point. But then, the obvious solution is to also have an incorruptible mechanism for checkpointing the system state periodically, so that we can just go forwards from the last checkpoint.

In summary, although reversibility may be useful for tracing the activity of malicious crackers, and preventing them from damaging any data, it is not theoretically any better than just reliably recording the system's initial state and all inputs. It may or may not turn out to be easier to implement. Thus, this application is not, by itself, a convincing justification for reversibility.

10.2 Program debugging

One interesting application of a reversible instruction set is that it makes it very simple to write a bi-directional debugger, which allows stepping backwards as well as forwards through a program. This feature eases the software debugging process, since, when observing that the program is behaving incorrectly, one can simply run in reverse from the point where the problem was first observed, to quickly trace back through the preceding events that led to the errant behavior.

Our simulator for the Pendulum instruction set (written by Matt DeBergalis) has the feature that one can step backwards as well as forwards through the program code, while observing registers. The simulator is thus a simple example of a bi-directional debugger, at the level of assembly instructions.

During the development of the compiler discussed in §9.4.3, these bi-directional debugging capabilities proved very useful several times, for tracking down the causes of incorrect program behavior caused by bugs in the compiler. During the compiler development process, we were using a version of the Pendulum instruction set that guaranteed reversibility independently of program well-formedness. This allowed the bidirectional capabilities of the simulation/debugging envrionment to function even when the compiler still had bugs. Incorrect program behavior was tracked backwards in time until the instructions that had caused the inappropriate behavior were found, at which point the compiler could easily be fixed.

So we have seen that a reversible computing capability can ease debugging. However, reversible computing is not strictly necessary for implementing a bi-directional debugger. For example, Boothe (1998, [27]) describes algorithms that can be used to implement bi-directional debugging environments for normal (irreversible) programming languages. There are many other bi-directional debuggers as well; see for example [172] and the references in [27]. One simple technique that is sometimes used is to save periodic checkpoints of program state, and when stepping backwards, just re-compute forwards from the previous saved checkpoint to reach the state of the program at a desired time-point.

So, alhough pure reversible computing makes bi-directional debugging trivial, it is not strictly necessary to compute reversibly in order to achieve this debugging capability. If one's only requirement is bi-directional debugging, it might be easier to just reinstrument an existing programming environment to achieve this directly, rather than coming up with a full computing reversible computing system from scratch.

10.3 Transaction processing and database rollback

It seems that the operation of "rolling back" the effects of an aborted transaction, which is common in some types of database systems, could possibly be implemented on top of a more general framework for undoing the actions of inter-communicating, reversible processes in a multitasking operating system for a reversible computer.

However, many details of this connection remain to be worked out; I can not yet say with confidence that this sort of application for reversible processing makes sense. Database rollback can already be performed quite well without requiring that the computer system be reversible at all levels. It is not yet clear whether the requirements of this application justify the sort of total, low-level reversibility that we have been discussing.

10.4 Speculative execution in multiprocessors

Similarly, it appears that reversibility might be useful to coordinate the activities of multiple CPUs which are running an underlying sequential algorithm in a parallel multiprocessing system. The individual CPUs might optimistically perform computations on data under the assumption that the data is valid (as in Knight's paper [90]), but when an inconsistency is detected, rather than restarting the processor's computation entirely, the processor might be reversibly rolled back to the point at which it read the bad data, and then proceed from there using the new, correct data.

10.5 Numerical stability in physics simulations

Apart from the performance benefits discussed in previous chapters, there are some advantages to using reversible algorithms when simulating physical systems. Reversibility is a sort of conservation law that is maintained in the real world, and so should also be maintained in the simulation. The flow of information in the physical system can and should be mirrored by the flow of information in the simulation. Failing to do this can lead to the simulated state of the system drifting farther and farther from the set of states that are possible in the real system being simulated.

We saw this behavior in our simulation of the Schrödinger wave equation (§9.5.6, p. 241). In the original irreversible version of the program, errors that crept into the wavefunction would grow in amplitude without bound. The simulation could only run for a certain amount of time before being swamped by ever-increasing artifacts in the wave function and going completely haywire. The reversible version, in contrast, although it was certainly not completely precise, always maintained a reasonably-shaped wave function, and was never observed to become swamped out by artifacts.

Perhaps this makes sense because if the artifacts steadily grow in one time-direction, then that would mean they would have to steadily decrease in the other time-direction. But such asymmetry was unlikely since the reversible algorithm was completely time-symmetric. So any artifacts that appeared could not grow unboundedly; they remained small relative to the desired wave data.

The advantages of reversibility in physical simulations are discussed further by Margolus [115]. Note however that these advantages can be gained as long as the simulation is simply reversible at the relatively high level of its state-update rule. The low-level instructions and circuits in the computer need not be individually reversible to obtain this improved simulation stability, although we saw in chapter 6 that doing so confers an efficiency advantage in large parallel systems.

10.6 Alternative applications: Conclusion

Pure reversible computing has possible applications in areas such as verifiable computation, intrusion detection and data protection, program debugging, transaction processing, and physical simulation. However, in most of the cases we have considered so far, it seems that the same benefits that could be achieved using total reversibility could be achieved using other, perhaps simpler, means as well; thus most of these alternative applications are not, in and of themselves, convincing justifications for the use of reversible computing technology.

However, if one has constructed a reversible system for other (*e.g.*, thermodynamic) reasons, then it is interesting to note that the various above capabilities fall out as a side effect. But we must remember that these alternative applications apply only if the system maintains *full* logical reversibility, but as we have seen in previous chapters, depending on the computations being performed, full reversibility may not be desirable from an asymptotic cost-efficiency standpoint. Even in our own reversible 3-D mesh model, the machine is allowed to be irreversible on its outer surface at least. Unless free energy is very expensive in a given application, it will probably be cheaper to generate some amount of permanent entropy and store it in the external universe, than it is to provide enough reversible digital storage so that a very long computation that is not inherently reversible can still be run perfectly reversibly.

Other applications for pure logical reversibility may yet be discovered, but at this time it appears that the most promising application of reversible computing technology will remain its selective use in making computation more cost-efficient by various measures; thus, that application remains the focus of our research.

Chapter 11

Conclusion and Future Work

In this chapter we summarize our conclusions and point to the important areas for future research along the lines of this thesis.

11.1 Summary of Contributions

In this thesis, we have attempted to comprehensively survey the entire range of current knowledge about reversible computing techniques, with an emphasis on the use of these techniques to make computers more efficient in a variety of ways. In the course of my own research in this area, I have discovered a substantial number of original and very interesting results, the most important of which we summarize here:

- In the context of *traditional* models of computation, purely reversible models appear to be asymptotically sub-optimal when both time and space costs are considered. (§3.4)
- However, given our understanding of the fundamental constraints and opportunities for information processing implied by well-established laws of physics (chs. 2,4), it becomes clear that those traditional models of computation lead to scaling predictions that are either suboptimal or physically unrealistic. Based on the constraints of physics, one can conceive of an *ultimate* physical model of computation that gives exactly the asymptotically best scaling for all problems that is permitted by physics. We conjecture that the ultimate model must take the form of some class of reversible 3-D mesh (either time-proportionately reversible, ballistic, or quantum coherent). (Chapter 5)
- We show that any of the proposed varieties of reversible 3-D mesh are asymptotically *strictly* faster per unit area or per unit mass than *any* irreversible physical computing architecture. The advantage per unit area grows with the

square root of the reversible mesh thickness. However, the advantage per unit mass of the non-quantum approaches is only a very small polynomial, growing at best only with the 18th root of the number of processors. (Or 9th root, if ballistic computation is possible.) (Chapter 6)

- We demonstrate that an asymptotically optimal reversible 3-D mesh could be built today using existing commercial VLSI processes, by giving a complete circuit design for a proof-of-concept universal reversible mesh processing element (FLATTOP). Unfortunately, we also find that despite the superior asymptotic scaling, reversible processing in VLSI is not competitive for supercomputing applications at feasible cost levels. Tipping the scale would require the development of much lower-resistance transistors. (Chapter 7)
- Fortunately, in the long term, a wide variety of advanced superconducting and nano-scale logic device technologies have been proposed that would not only be much faster than traditional VLSI, but would also entail much larger advantages for reversible processing, since they are much more nearly ballistic than are systems based on highly-resistive MOSFETs. Only a very thin layer (microns to millimeters thick) of these future reversible devices would be needed in order for the resulting machine to be faster per unit area than *any possible* irreversible technology of *any thickness*. The advantages would continue to increase with the square root of further increases in machine thickness. (Chapter 8)
- In order to take full advantage of the long-term efficiency benefits offered by reversible computing, it is required to use new microprocessor instruction sets and new high-level programming languages, in order to permit the optimal reversible algorithms to be expressed and executed with their intended efficiency. However, the necessary changes are fairly straightforward. We presented some important principles for reversible instruction sets, and a simple proof-of-concept C-like programming language. Algorithms must in general also be redesigned to take best advantage of the reversible paradigm. We give examples, including an efficient reversible algorithm for simulating quantum mechanics. (Chapter 9)
- In addition to its thermodynamic advantages, pure reversible computing may conceivably have applications in other areas such as audit trails, transaction rollback, and backwards debugging. However, after considering a number of such possibilities, we have not found any very convincing benefits in areas other than in increasing computational efficiency. For other applications, there seem to be equally good solutions that do not require complete reversibility. (Chapter 10)

The upshot of all this is that substantially reversible computing techniques do not seem to be immediately practical through the next 10 years or so of semiconductor technology, but in the much longer term, as we move to a nano-scale computing technology, the issues and techniques described in this thesis will be not only practical, but essential in order to make good use of the physical resources available for computing. We can confidently predict that meter-scale and larger machines composed of good nano-scale reversible components will be much faster, at many types of problems, than *any physically possible* mostly-irreversible computer of *any size*. Designing and programming these superior machines will require processor architectures and algorithms along the lines we have discussed.

Although the manufacturing technology does not yet exist to produce many of the proposed future logic devices, with a bit of optimism, one can look at the present rate of progress of technology, and see that it is at least a fairly good bet that at some point, probably a few decades hence, that technology will exist, and we will need to build computers with it. Based on that projection, it is worthwhile for computer scientists to start thinking now about how to architect and program those machines, and for device physicists to start thinking about how to improve their components with the future reversible revolution in mind.

It is hoped that the research reported in this thesis will serve as an impetus to that work, and a guide for future researchers starting out in this important area.

11.2 Major areas for future research

The future research that will be needed in reversible computing spans a wide variety of disciplines.

Fundamental theoretical physics. In chapter 2 we saw that among the ultimate physical limits on computation, the fundamental limits on entropy density (and entropy flux) are currently not very well understood. (At least, I have not yet encountered a definitive statement of them.) For example, it is not clear to me what is the maximum density with which entropy can be stored within normal atomic matter at manageable temperatures, or whether there are other types of matter that might achieve higher densities and still be useful. Entropy density limits determine entropy flux limits, and thus are important for understanding the limitations of cooling systems.

Another important area for theoretical physics is in further elucidating the physics relating to quantum computing, devising better ways of avoiding decoherence, and so forth.

Of course, it would also be nice to have a simple, complete, unified theory of physics, rather than the somewhat incomplete picture we still have. It is possible that with a complete theory, we might see some surprising new implications for the fundamental limits of computation. Of course, pinning down a complete theory is the holy grail of theoretical physics, and physicists are already hard at work on this problem.

Nano-scale device physics. There is of course also much need for work on designing new physical devices that can be used as computational elements at the nanoscale, devices that have the appropriate physical properties so that they can be operated quickly in time-proportionately reversible fashion, with an entropy coefficient that is low enough so that the devices are nearly ballistic. (One would like to not have to start clocking the devices more slowly than their maximum speed until a very large scale of machines.)

More difficult than just designing new devices is designing *buildable* devices, or more broadly, designing economically feasible pathways for the development of future manufacturing technology that will eventually lead to the ability to build the desired devices. Fortunately, the entire field of nanotechnology (*cf.* [51, 41], and the journal *Nanotechnology*) is already working hard in this direction, since it can be seen that a flexible nano-scale manufacturing infrastructure would benefit society enormously in ways that go far beyond just making faster computers.

Semiconductor device physics. Meanwhile, one direction to try to make effective reversible supercomputers would be to try to find ways to make lower-resistance switches using more conventional fabrication technology. We saw that low-resistance switches benefit adiabatic circuit techniques because they would decrease entropy coefficients. However, they do not benefit traditional irreversible circuits as much because the CV^2 switching energy is independent of switch resistance, and so a dissipation-limited system will not go any faster beyond a certain point no matter how fast its switches are. One promising approach to making lower-resistance switches might be to make smaller, faster micro-electromechanical relays. Other approaches might be possible.

Resonant power supplies. Another element that would be needed for good, scalable adiabatic circuits is a good resonant power supply. Currently, we do not know of a resonant supply technique that has a high Q, the desired scaling properties, and can provide the desired waveforms. Becker and Knight's technique [12, 13] comes close, but the scaling seems not quite right. It might be that the desired scaling isn't possible, in which case adiabatic circuits overall might not scale as well asymptotically as reversible circuits based on other types of devices.

Adiabatic circuit designs. If lower-resistance switches and good resonant power supplies can be found, the adiabatic circuit approach to reversible computing might become attractive enough so that it is worth some effort to try to find an adiabatic logic circuit family that is simpler than SCRL, while still providing all of SCRL's desirable features. SCRL already seems pretty good, but a further improvement by a small constant factor might still be possible.

Reversible architectures. Independent of the precise logic technology used, work can already be done on designing better reversible processor architectures. Our proof-of-concept FlatTop chip was not designed to be easy to program. Vieri's Pendulum architecture [179, 178] is much better, but further improvements in the efficiency of the design are likely possible. For use in parallel mesh architectures, a processor including hardware support for routing might be desirable.

Reversible programming languages. Our proof-of-concept R programming language was intentionally extremely simple. It would be interesting to have reversible versions of more sophisticated programming languages with more advanced features. However, one should be careful when choosing a programming language that it does not necessarily impose any asymptotic inefficiencies compared to programming in raw machine language. Constant-factor overheads are acceptable, but if the language is asymptotically inefficient, then to some degree it ruins the point of developing a reversible architecture to begin with.

Mesh programming languages. Traditional programming languages are tied to the idea of a uniprocessor architecture, and do not help much in expressing effective parallel algorithms. In order to concisely express reversible 3-D mesh algorithms, it would be nice to have a language in which one could express algorithms at the level of specifying where information is located in 3-D space, how it should flow around the system as it is computed and uncomputed, and so forth.

Given the ultimate physical nature of information, we suspect that a good parallel programming language should have a bit of the flavor of a language for specifying the detailed design and layout of a complex assembly line in a factory, in which physical objects (data structures) are assembled and disassembled (computed and uncomputed), and must physically move from place to place, at bounded speeds (no more than c), without ever occupying the same space as each other (consuming more memory than is available at a given node). Moreover, like in a factory, one has to route power to the subsystems, and provide ventilation and cooling systems to remove unwanted heat (entropy)—except that in a reversible mesh architecture, the entropy could be moved out of the system while it is still in digital form, if this is beneficial. Moreover, the production of unwanted entropy could be kept to a minimum by uncomputing objects that are no longer needed, rather than, say, "melting them down" to the "raw material" of free memory—which is an appropriate physical analogy for what we do when we overwrite memory that contains a data object.

Unlike in a factory, however, the structure of the manufacturing machines (programs) in the mesh computer would be relatively easy to reconfigure dynamically, under program control. They could replicate themselves at will, and move around freely from one part of the factory to another. At the programming level in the computer, one need not worry about factors such as gravity, strength of materials, vibrations, and wearing out of parts. (Hopefully, the bottom-level logic devices are sufficiently reliable enough and/or the architecture fault-tolerant enough that the programmer does not have to worry about failure of the underlying physical components.)

Reversible manufacturing. The above analogy between computing and manufacturing also suggests the intriguing idea that in future nanomechanical factories, one might also apply reversible computing principles to manufacturing. In a real nano-scale factory that is producing nano-scale components, from time to time, temporary structures may need to built. The system will be able to operate with less heat dissipation, and thus faster overall, if it uses a knowledge of how a temporary structure was built when disassembling it. If the structure of an atomically-precise object is known precisely, it can theoretically be disassembled to raw materials with no production of entropy. In contrast, disassembling the temporary object using a general approach with no knowledge of the object's structure is bound to be wasteful.

An analogy from everyday experience: when the face of a tall building is repaired, often the contractor will construct a temporary scaffolding, in a regular structure, to aid the work. When the work is finished, they carefully unscrew the pipes making up the scaffolding, in an appropriate order, dissassembling the scaffolding piece by piece. An alternative approach would be to just knock down the whole scaffolding, break it into pieces, melt it down, mold new pipes, and rebuild it next time from scratch. But of course this would be much less energy-efficient.

The same basic principle could be applied in manufacturing at all scales. For very fast nano-scale systems where heat removal is a major concern, this principle would be essential for allowing as many nano-manufacturing operations as possible per unit time, per unit of outer surface area.

Reversible mesh algorithms. Finally, even given a good programming language, one is still left with the task of designing good algorithms for all the computationally expensive problems that one encounters. As we saw in ch. 9, in general the best reversible algorithm for a problem might not be derivable from the best irreversible algorithm in a straightforward way. The same applies to mesh algorithms. If we wish to find the truly best algorithms for solving very hard problems, we should be working on algorithms for reversible 3-D mesh architectures. (And as we saw in ch. 4, perhaps the algorithms should be quantum coherent as well.) It would be very interesting and useful to compile a catalog of the best classical (and quantum) reversible 3-D mesh algorithms that can be devised for a variety of the hard, data-intensive problems that have historically motivated the development of massively parallel supercomputers. That way, once we have developed the technology to start building computers based on high-quality reversible nano-scale components, we will already have a good idea of how to use them effectively.

11.3 Final words

In closing, we hope that this thesis has shed a revealing new light on an fascinating area of computer science, reversible computing, an area that has previously been somewhat obscure, and not very well-understood. We hope that this work will contribute significantly to an eventual consensus in the computing community that it is worth the effort to design and build new types of nano-scale physical logic devices with reversible usage in mind, and to develop and study reversible algorithms to run on the massively parallel reversible meshes that we will someday be able to construct.

It is our fervent hope that such a consensus will arrive sooner rather than later, in order to facilitate a more rapid evolution towards the reversible computing revolution that must eventually occur. By enabling vastly more efficient computing, reversible computing techniques should greatly facilitate many amazing feats of technological and social progress that we expect and hope our civilization will accomplish, over the course of the new millenium that is about to dawn.

Part III

Appendices

Appendix A

FlatTop processor schematics and layouts

This appendix gives the detailed Cadence schematics and layout for the FLATTOP universal parallel reversible processor which we discussed in §7.7. These diagrams were also included in our conference paper [72].

A.1 High-level blocks

Figure A.1 shows a schematic block diagram of a single processing element cell. Note the three blocks, one for each of the three stages in the 3-phase SCRL pipeline which makes up the logic of the cell. The lollipop-shaped icon above each stage represents the set of swinging supply rails, in a particular phase, which drive the stage adiabatically. The cell has four inputs A, B, C, D which come from the four neighboring cells, and four corresponding outputs which go back out to those cells.

The SHIFT in and out signals are global signals shared by all cells; they tell the cells whether to operate in initialization mode or normal mode. In initialization mode the array of cells behaves as a shift register, and the array contents may be shifted in and out; in normal mode, the array just obeys the BBMCA update rules.

The boxes attached to the input are for setting initial conditions during HSPICE simulation of the circuit.

Figure A.2 shows our schematic icon representing an entire processing element. The icon portrays the 2×2 block of BBMCA cells which the PE is updating, with the PE inputs and outputs placed in the appropriate cells. The cell grid is rotated 45 degrees from the representation in fig. 7.19 to show how the CA array is orientated with respect to the edges of the chip. With this orientation, the array of processing elements can communicate along pathways that run parallel to the chip edges, making layout easier.

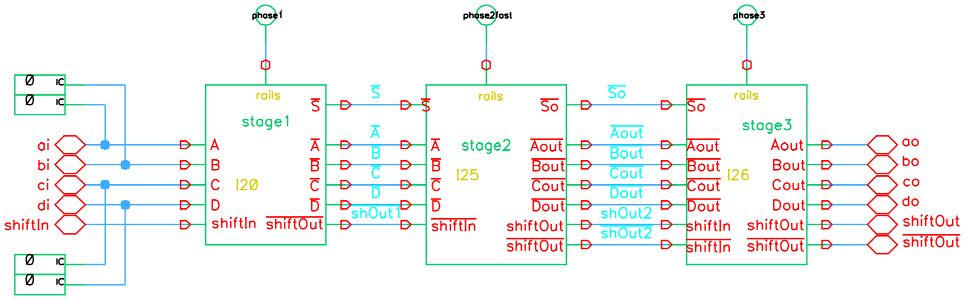


Figure A.1: Cadence schematic block diagram of the FLATTOP PE cell. The three blocks are the 3 SCRL logic stages, each clocked by a separate set of clock rails, indicated by the “lollipop” icons above the blocks, whose phases are offset by 1/3 cycle from each other.

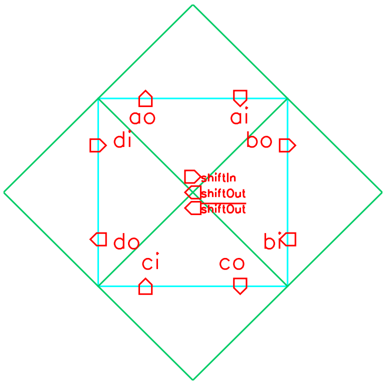


Figure A.2: Icon for a single FLATTOP cell. The depiction graphically illustrates the block of 4 CA cells, oriented by 45° to the inter-PE wiring, that is updated by the PE. The two terminals in each cell connect to and from the neighboring PE in the given direction, whose icon is overlapped with this one, as a reminder that the two PEs take turns updating the same CA cell.

Figure A.3 shows one corner of an array of these processing elements. In the upper left corners are pathways used for initialization and reading out the whole array when used as a shift-register. Along the edges of the array are edge cells which provide connections to pins, allowing the chip to communicate with other chips during normal operation. There are not enough pins to allow communication everywhere along the edge, so in other places the wires at the edge just loop around to feed back into the array. Every PE receives the global shift signals, which run horizontally.

Figure A.4 shows the entire 20×20 array of processing elements which we fabricated for testing purposes.

A.2 Detailed gate schematics

Below are the schematics for the logic in the three stages of each cell. We have not yet had time to size the transistors in our design so as to minimize power, but within each gate, we have, for uniformity, sized its transistors so that the worst-case conductances are the same as that of an inverter with a minimum-width n-FET and a twice-minimum-width p-FET.

Figure A.5 shows the logic in stage 1 which computes the inverses of the inputs, together with the inverse of the S (static) signal used in stage 2. Note that S is turned on when in *sh* (shift) mode. This special case was added to support array initialization.

Figure A.6(a) shows the gate used for computing each second-stage output. The forward part of stage 2 includes four repititions of this gate, differing as to which inputs are fed to the A, B, C, D pins, plus 6 "fast" inverters for generating the $A, B, C, D, S, shift$ signals from their inverses using the $f\phi_2$ and $\overline{f\phi_2}$ rails, plus one other inverter for generating $\overline{S}$ on the stage 2 output.

Finally, figure A.6(b) shows the gate that is repeated four times (with different pin assignments) in the forwards part of stage 3. This gate selects either A or the bit in the opposite corner of the block for passing through to the output, depending on *sh*. Thus if *sh* is on, input bits go to the opposite output bits, enabling the array as a whole to act as one large shift register, which can be used to initialize the array contents.

A.3 Cell layout

Figure A.7 shows the complete layout of a single PE cell. For clarity, the metal3 rails that run vertically across the entire cell are not shown. The cell measures 167.6 μm × 91.3 μm.

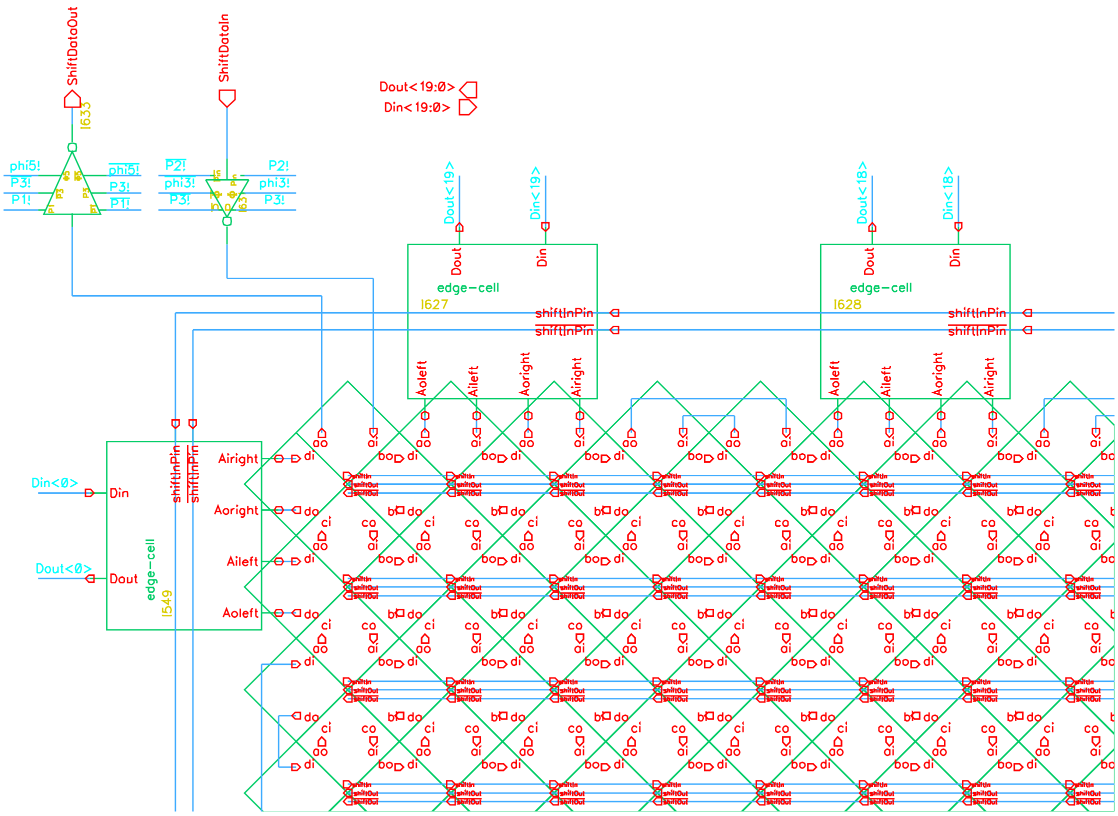


Figure A.3: Schematic diagram of the upper-left corner of a large array of FLATTOP processing elements. The rows and columns of cells of CA array being simulating can be seen running diagonally across the array. You can see how the PE icons (fig. A.2) are overlapped to represent the shared management of each CA cell. Meanwhile, clock-power signals are strapped horizontally across the array. Periodically along the edge of the array are units for inter-chip communication. Other signals along the edges are simply wrapped around to a neighboring edge cell. The buffers in the far upper left corner provide initialization and readout capabilities.

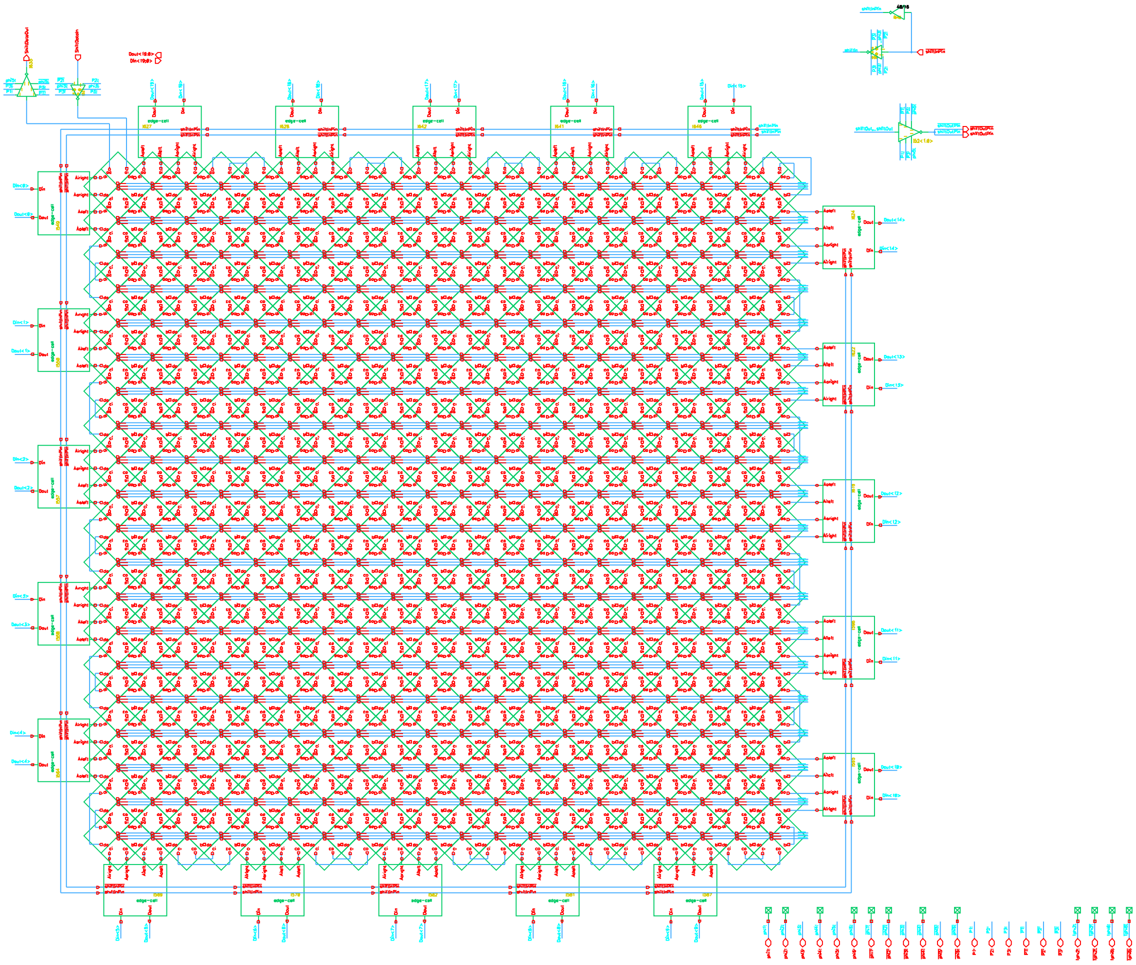


Figure A.4: Schematic block diagram of the full 20×20 array of FLATTOP processing elements which we fabricated on each die. This simply extends the structure shown in fig. A.3.

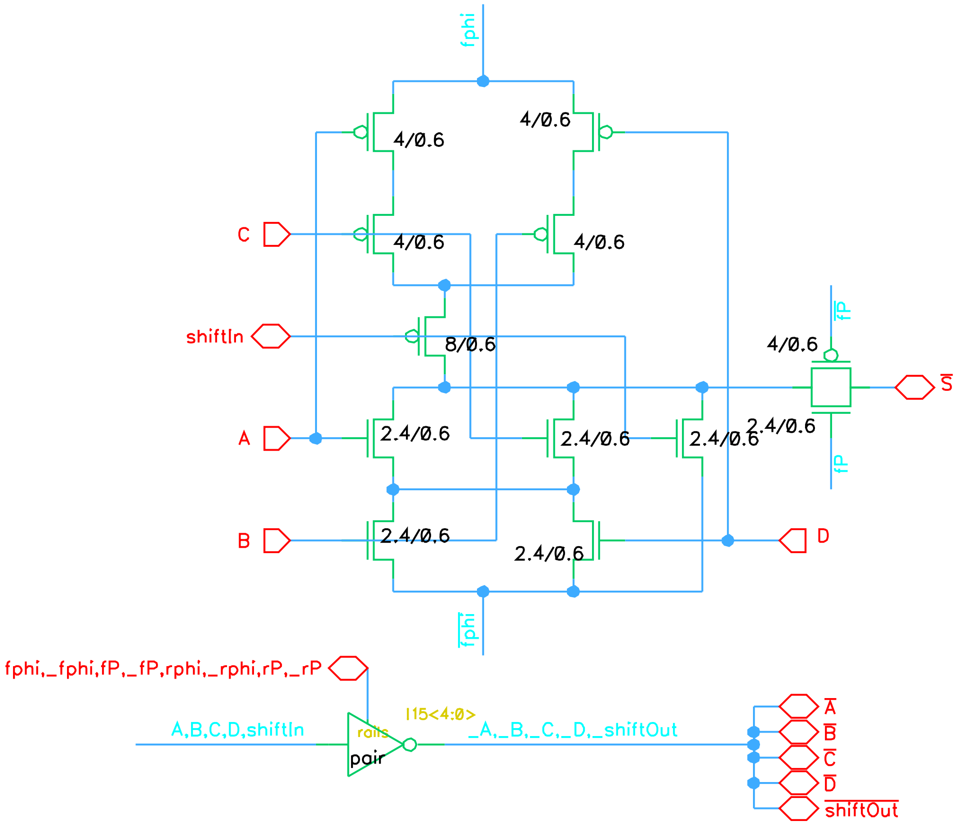
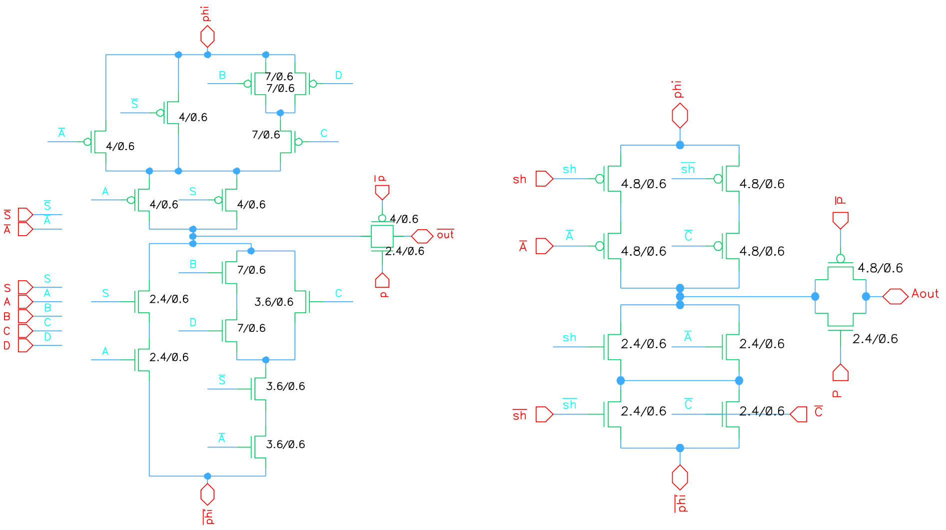


Figure A.5: Stage 1 logic, $\overline{S} = \overline{sh + (A + C)(B + D)}$.



(a) (b)

Figure A.6: (a) Gate for computing $\overline{A_{out}}$ in stage 2. (b) Gate for computing $A' = \overline{\overline{sh}\ \overline{A} + sh\ \overline{C}}$ in stage 3.

Appendix B

The Pendulum instruction set architecture (PISA)

This appendix gives a detailed description of the assembly language instruction set for the Pendulum reversible microprocessor currently being fabricated by Carlin Vieri [178]. Some reversible instruction set issues encountered during the development of this instruction set (which I assisted Vieri with) were discussed in §9. The version of the instruction set described here was the target for the compiler described in §9.4.3 and appendix D.

Vieri's thesis describes these instructions at a more detailed level, that gives the precise instruction word layout for purposes of machine code assembly and instruction decoding in his real hardware implementation. For our purposes of testing compiler techniques, such details were unimportant. Thus, in this reference we only describe the instructions from an assembly language programmer's point of view.

B.1 Overall organization

The PISA instruction set can be divided into three categories: reversible artihmetic/logical operations, ordinary branch instructions, and special instructions.

The set of arithmetic/logical operations is designed to be logically complete, yet purely reversible. To achieve this, the results of operations are generally XOR'ed into separate destination registers, an operation which can be inverted by simply repeating it. Certain "non-expanding" operations can be performed reversibly without a separate destination register.

As for the branch instructions, these are designed to be used in pairs, where each branch instruction points to a corresponding branch that points back to the original instruction, as per the discussion in §9.2.2. Given this, one way to implement branches reversibly is to have the branch instruction add its offset into a special

```
Key:
rsd,rt = 5-bit register identifier.
  No-Op if rsd is same reg as rt.
imm,amt = 16 bit signed immediate
  [imm] = imm sign-extended to 32 bits

"Non-expanding" arith./logical operations:

Mnem.   Args.      Forwards behavior
-----   ---------  ---------------------
NEG     rsd        rsd = -rsd
ADD     rsd,rt     rsd += rt (mod 2^32)
ADDI    rsd,imm    rsd += [imm] (mod 2^32)
SUB     rsd,rt     rsd -= rt (mod 2^32)
XOR     rsd,rt     rsd ^= rt
XORI    rsd,imm    rsd ^= [imm]
RL      rsd,amt    rsd = rsd rol amt
RLV     rsd,rt     rsd = rsd rol rt
RR      rsd,amt    rsd = rsd ror amt
RRV     rsd,rt     rsd = rsd ror rt
```

Figure B.1: “Non-expanding” arithmetic/logical operations in the 32-bit simulator/compiler version of the PISA instruction set.

```
Key:
rd,rs,rt = 5-bit register identifier.
  No-Op if rd is same reg as rs or rt.
imm,amt = 16 bit signed immediate
  [imm] = imm sign-extended to 32 bits

"Expanding" arith./logical operations:

Mnem.   Args.      Forwards behavior
-----   ---------  ---------------------
ANDX    rd,rs,rt   rd ^= rs&rt
ANDIX   rd,rs,imm  rd ^= rs&[imm]
NORX    rd,rs,rt   rd ^= ~(rs|rt)
ORX     rd,rs,rt   rd ^= rs|rt
ORIX    rd,rs,imm  rd ^= rs|[imm]
SLLX    rd,rs,amt  rd ^= rs<<amt
SLLVX   rd,rs,rt   rd ^= rs<<rt
SLTX    rd,rs,rt   rd ^= (rs<rt)?1:0
SLTIX   rd,rs,imm  rd ^= (rs<imm)?1:0
SRAX    rd,rs,amt  rd ^= rs>>amt
SRAVX   rd,rs,rt   rd ^= rs>>rt
SRLX    rd,rs,amt  rd ^= (unsigned)rs>>amt
SRLVX   rd,rs,rt   rd ^= (unsigned)rs>>rt
```

Figure B.2: “Expanding” arithmetic/logical operations in the PISA instruction set.

```
Key:
rd,ra,rb = 5-bit register identifier.
off = 16 bit signed offset
loff = 26 bit signed offset
dir = +1/-1 bit where +1=forward, -1=reverse
BR = internal "branch register"

Branch instructions:

BEQ     ra,rb,off  if ra=rb, BR+=off*dir
BGEZ    rb,off     if rb>=0, BR+=off*dir
BGTZ    rb,off     if rb>0, BR+=off*dir
BLEZ    rb,off     if rb<=0, BR+=off*dir
BLTZ    rb,off     if rb<0, BR+=off*dir
BNE     ra,rb,off  if ra!=rb, BR+=off*dir
BRA     loff       BR+=loff*dir
RBRA    loff       dir=-dir, BR+=loff*dir
SWAPBR  r          r<->BR

PC update between instructions:
  if (BR=0) pc+=dir else pc+=BR*dir

Memory & I/O instructions:

EXCH    rd,ra      rd <-> mem[ra]
READ    ra         ra ^= next word from input str.
SHOW    ra         Copies ra to output stream.
EMIT    ra         Emit ra to garbage stream.
```

Figure B.3: Branching, memory access, and input/output operations in the PISA instruction set.

"branch register" which is normally zero. Between instructions, if the branch register is non-zero, the program counter increments by the branch register value, instead of by the normal 1 instruction. Then the branch at the destination executes, canceling out the value stored in the branch register and resuming normal execution.

In this scheme, even if the programmer forgets to put in the branch at the destination, the resulting behavior will still be reversible. But it will not be useful behavior: the program counter will jump forward through the program in repeated leaps, of size equal to the original offset.

To implement subroutine calls, the branch destination can be the special SWAPBR instruction, which exchanges the branch register with an empty register. The body of the subroutine negates the register, so when the subroutine hits the next SWAPBR it branches back to the location it came from; the branch at that location cancels out the branch register and the processor continues sequentially. SWAPBR can also be used in a complementary way to perform switch statements.

All memory access happens through the EXCH instruction which exchanges a register with a variable memory location. There is an interesting case here, in which an EXCH instruction tries to exchange *itself* with a memory location. The machine can be designed to do nothing in such a case. Or, if the instruction fetch/unfetch mechanism works via an exchange, the register will actually be exchanged with the single constantly-moving value that sits in the instruction register between instructions, and in the current PC location in memory during instructions.

The processor direction can be reversed in software using the special RBRA (reversing branch) instruction, which toggles the processor direction bit while it is performing BRA functionality. This allows subroutines to be called either forwards or in reverse, thus reducing the need for repeated code.

Special instructions to perform reversible output are also available—the SHOW instruction which just exports a copy of a register, and the EMIT instruction which actually reversibly sends the information in the register out of the processor to whatever system it is embedded in. Input instructions could also be defined, and there could be two types: a SEE instruction which just XOR's an input word into a register, but does not consume it (the external system would be responsible for disposing of the original) and a TAKE instruction which would reversibly consume the outside information and bring it into a register, would have to be initially clear. Another option would just be a single IOEX instruction which simply exchanges a register with the value currently present in the external I/O system, which could then move the old value to its output, and move a new value into place from its input.

Finally, there are START/FINISH instructions for marking the start/endpoints of programs when running in the simulation environment. Presumably, a real processor would always be running its operating system, and would never need to halt.

Let us now give all the instructions, in a reference format.

B.2 List of Instructions

Figures B.1, B.2, and B.3 list the name, arguments, and forwards behavior of all the instructions in the 32-bit version of the PISA instruction set that was used in the Pendulum simulator and the R language compiler.

B.3 Arithmetic/logical ops

ADD

Add one register into another.

Usage: `ADD` reg_d reg_s

Arguments:

reg_d — The destination register.

reg_s — The source register.

Description:

Adds the value of register reg_s into register reg_d, that is, modifies reg_d be equal to the previous value of $(reg_d + reg_s) \bmod 2^{32}$. Note that this operation is inherently reversible, no matter the previous values of reg_d and reg_s. It is inverted by SUB with the same arguments.

ADDI

Add an immediate value into a register.

Usage: `ADDI` reg_d imm

Arguments:

reg_d — The destination register.

imm — The immediate value to be added into reg_d.

Description:

Sign-extends the immediate 16-bit value imm to 32 bits and adds it into reg_d. That is, $reg_d \leftarrow (reg_d + imm) \bmod 2^{32}$. Note that this operation is inherently reversible, no matter the previous value of reg_d. It is inverted by another ADDI with the immediate value negated, or by doing NEG of reg_d followed by the identical ADDI, followed by another NEG of reg_d.

ANDX

Exclusive-OR the result of an AND into a register.

Usage: `ANDX` reg_d reg_{s1} reg_{s2}

Arguments:

reg_d — The destination register.

reg_{s1} — The first source operand.

reg_{s2} — The second source operand.

Description:

Computes the bitwise logical AND of the contents of reg_{s1} and reg_{s2}, and exclusive-OR's the result into register reg_d. That is, $reg_d \leftarrow reg_d \oplus (reg_{s1} \wedge reg_{s2})$. Note that due to the XOR, this operation is inherently reversible, no matter the previous values of the registers. It is inverted by repeating the exact same instruction again. Note also that plain AND may be emulated by letting reg_d be a register that was previously 0. ANDX corresponds to 32 Toffoli gates operating in parallel on the corresponding bits of the 3 operands.

`ANDIX` XOR an AND with an immediate value into a register.

Usage: `ANDIX` reg_d reg_s *imm*

Arguments:

reg_d — The destination register.

reg_s — The source register.

imm — The immediate value to AND with.

Description:

Like ANDX, except the source register is AND'ed with the immediate value *imm* instead of with a second source register.

`NORX` Exclusive-OR the result of a NOR into a register.

Usage: `NORX` reg_d reg_{s1} reg_{s2}

Arguments:

reg_d — The destination register.

reg_{s1} — The first source operand.

reg_{s2} — The second source operand.

Description:

Computes the bitwise logical NOR of the contents of reg_{s1} and reg_{s2}, and exclusive-OR's the result into register reg_d. That is, $reg_d \leftarrow reg_d \oplus \overline{(reg_{s1} \vee reg_{s2})}$. Note that due to the XOR, this operation is inherently reversible, no matter the previous values of the registers. It is inverted by repeating the exact same instruction again.

For no particular reason, there are no corresponding NORIX, NANDX, or NANDIX instructions. I believe this was just to keep the instruction set smaller. But NORX itself is not strictly necessary either, since one can emulate it by using ORX and then XORI'ing -1 into the result.

NEG

Two's-complement negate the given register.

Usage: `NEG` reg_{sd}

Arguments:

reg_{sd} — The source/destination register.

Description:

Replace the contents of reg_{sd} with its (two's complement) negative. That is, $reg_{sd} \leftarrow (2^{32} - reg_{sd}) \bmod 2^{32}$. Note that this operation is inherently reversible. It is its own inverse.

ORX

Exclusive-OR the result of an OR into a register.

Usage: `ORX` reg_d reg_{s1} reg_{s2}

Arguments:

reg_d — The destination register.

reg_{s1} — The first source operand.

reg_{s2} — The second source operand.

Description:

Computes the bitwise logical OR of the contents of reg_{s1} and reg_{s2}, and exclusive-OR's the result into register reg_d. That is, $reg_d \leftarrow reg_d \oplus (reg_{s1} \vee reg_{s2})$. Note that due to the XOR, this operation is inherently reversible, no matter the previous values of the registers. It is inverted by repeating the exact same instruction again. Note also that plain AND may be emulated by letting reg_d be a register that was previously 0.

ORIX

XOR an OR with an immediate value into a register.

Usage: `ORIX` *reg_d reg_s imm*

Arguments:

reg_d — The destination register.

reg_s — The source register.

imm — The immediate value to AND with.

Description:

Like ORX, except the source register is AND'ed with the immediate value *imm* instead of with a second source register.

RL

Rotate a register left by a fixed number of bits.

Usage: `RL` reg_{sd} *amt*

Arguments:

reg_{sd} — The source/destination register.

amt — The immediate number of bits to rotate by.

Description:

Rotate the bits in register reg_{sd} left (that is, in the direction from least-significant positions to most-significant positions) by the given number of places (0-31). Bits rotated off the left end of the word rotate back onto the right end. RL is inherently reversible; it is inverted by RR *amt*, or by RL'ing by the amount $(32 - amt)$ mod 32.

RLV

Rotate a register left by a variable number of bits.

Usage: `RLV` reg_{sd} reg_t

Arguments:

reg_{sd} — The source/destination register.

reg_t — The register giving the number of bits to rotate by.

Description:

Rotate the bits in register reg_{sd} left (that is, in the direction from least-significant positions to most-significant positions) by the number of places given by reg_t mod 32. Bits rotated off the left end of the word rotate back onto the right end. RLV is inherently reversible; it is inverted by RRV.

`RR`

Rotate a register right by a fixed number of bits.

Usage: `RR` reg_{sd} amt

Arguments:

reg_{sd} — The source/destination register.

amt — The immediate number of bits to rotate by.

Description:

Rotate the bits in register reg_{sd} right (that is, in the direction from most-significant positions to least-significant positions) by the given number of places (0-31). Bits rotated off the right end of the word rotate back onto the left end. RR is inherently reversible; it is inverted by RL amt, or by RR'ing by the amount $(32 - amt)$ mod 32.

`RRV`

Rotate a register right by a variable number of bits.

Usage: `RRV` reg_{sd} reg_t

Arguments:

reg_{sd} — The source/destination register.

reg_t — The register giving the number of bits to rotate by.

Description:

Rotate the bits in register reg_{sd} right (that is, in the direction from most-significant positions to least-significant positions) by the number of places given by reg_t mod 32. Bits rotated off the right end of the word rotate back onto the left end. RRV is inherently reversible; it is inverted by RLV.

`SLLX`

XOR with result of shifting a register left logically by a fixed number of bits.

Usage: `SLLX` reg_d reg_s amt

Arguments:

reg_d — The destination register.

reg_s — The source register.

amt — The immediate number of bits to shift by.

Description:

Logically shifts the given register left by *amt*, filling in 0's on the right, and XOR's the result into the destination register reg_d. Note that since the shift operation is information-losing, we cannot put the result back into the same register. Instead, we XOR the result into a register as with NANDX and other operations.

SLLVX

XOR with result of shifting a register left logically by a variable number of bits.

Usage: `SLLVX` reg_d reg_s reg_t

Arguments:

reg_d — The destination register.

reg_s — The source register.

reg_t — Register specifying amount to shift by.

Description:

Like SLLX but with a variable number of bits. See RLV.

SRAX

XOR with result of shifting a register right arithmetically by a fixed number of bits.

Usage: `SRAX` reg_d reg_s *amt*

Arguments:

reg_d — The destination register.

reg_s — The source register.

amt — The immediate number of bits to shift by.

Description:

Arithmetically shifts the given register right by *amt*, filling in with copies of the leftmost bit, and XOR's the result into the destination register reg_d. Note that since the shift operation is information-losing, we cannot put the result back into the same register. Instead, we XOR the result into a register as with NANDX and other operations.

SRAVX

XOR with result of shifting a register right arithmetically by a variable number of bits.

Usage: `SLLVX` reg_d reg_s reg_t

Arguments:

reg_d — The destination register.

reg_s — The source register.

reg_t — Register specifying amount to shift by.

Description:

Like SRAX but with a variable number of bits. See RLV.

SRLX

XOR with result of shifting a register right logically by a fixed number of bits.

Usage: `SRLX` reg_d reg_s *amt*

Arguments:

reg_d — The destination register.

reg_s — The source register.

amt — The immediate number of bits to shift by.

Description:

Logically shifts the given register right by *amt*, filling in 0's on the left, and XOR's the result into the destination register reg_d. Note that since the shift operation is information-losing, we cannot put the result back into the same register. Instead, we XOR the result into a register as with NANDX and other operations.

SRLVX

XOR with result of shifting a register right logically by a variable number of bits.

Usage: `SRLVX` reg_d reg_s reg_t

Arguments:

reg_d — The destination register.

reg_s — The source register.

reg_t — Register specifying amount to shift by.

Description:

Like SRLX but with a variable number of bits. See RLV.

SUB

Subtract one register from another.

Usage: `SUB` reg_d reg_s

Arguments:

reg_d — The destination register.

reg_s — The source register.

Description:

Subtracts the value of register reg_s from register reg_d, that is, modifies reg_d be equal to the previous value of $(reg_d - reg_s + 2^{32}) \bmod 2^{32}$. Note that this operation is inherently reversible, no matter the previous values of reg_d and reg_s. It is inverted by ADD with the same arguments.

XOR

Exclusive-OR one register into another.

Usage: `XOR` reg_d reg_s

Arguments:

reg_d — The destination register.

reg_s — The source register.

Description:

Exclusive-OR reg_s into reg_d, that is, sets reg_d equal to $reg_d \oplus reg_s$. This is a self-reversible operation.

XORI

Exclusive-OR an immediate value into a register.

Usage: `XORI` reg_d imm

Arguments:

reg_d — The destination register.

imm — The immediate value to XOR with.

Description:

Exclusive-OR the 16-bit immediate value imm into reg_d, that is, sets reg_d equal to $reg_d \oplus imm$. Self-reversible.

B.4 Ordinary branches

BEQ

Branch if equal.

Usage: `BEQ` reg_a reg_b off

Arguments:

reg_a, reg_b — Registers to compare.

off — Immediate offset.

Description:

If the contents of registers reg_a and reg_b are equal, arrange to branch to an instruction off steps ahead of the current instruction. That is, add the signed 16-bit value off into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

BGEZ

Branch if greater than or equal to zero.

Usage: `BGEZ` reg_a off

Arguments:

reg_a — Register to compare.

off — Immediate offset.

Description:

If the value in register reg_a, interpreted in two's complement form, is greater than or equal to zero (that is, if its high bit is 0), then branch to an instruction off steps ahead of the current instruction. That is, add the signed 16-bit value off into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

BGTZ

Branch if greater than zero.

Usage: `BGTZ` reg_a off

Arguments:

reg_a — Register to compare.

off — Immediate offset.

Description:

If the value in register reg_a, interpreted in two's complement form, is greater than zero (that is, if it is not zero but its high bit is 0), then branch to an instruction off steps ahead of the current instruction. That is, add the signed 16-bit value off into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

BLEZ

Branch if less than or equal to zero.

Usage: `BLEZ` reg_a off

Arguments:

reg_a — Register to compare.

off — Immediate offset.

Description:

If the value in register reg_a, interpreted in two's complement form, is less than or equal to zero (that is, if it is zero or its high bit is 1), then branch to an instruction off steps ahead of the current instruction. That is, add the signed 16-bit value off into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

BLTZ

Branch if less than zero.

Usage: `BLTZ` reg_a off

Arguments:

reg_a — Register to compare.

off — Immediate offset.

Description:

If the value in register reg_a, interpreted in two's complement form, is less than zero (that is, if its high bit is 1), then branch to an instruction off steps ahead of the current instruction. That is, add the signed 16-bit value off into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

BNE

Branch if not equal to zero.

Usage: `BNE` reg_a reg_b off

Arguments:

reg_a, reg_b — Registers to compare.

off — Immediate offset.

Description:

If the contents of registers reg_a and reg_b are equal, then branch to an instruction off steps ahead of the current instruction. That is, add the signed 16-bit value off into the branch register. The destination must be a branch pointing back to the current location, or a SWAPBR.

BRA

Unconditional branch.

Usage: `BRA` $loff$

Arguments:

$loff$ — Long immediate offset.

Description:

Unconditionally branch to a location $loff$ steps ahead of the current instruction. That is, add the signed value $loff$ into the branch register. The value $loff$ is a signed long immediate value, with more bits than the short offsets in the various conditional branches above. The exact number of bits depends on the instruction encoding and the number of bits reserved for opcodes. The destination must be a branch pointing back to the current location, or a SWAPBR.

B.5 Special instructions

`EXCH`

Exchange a register with a memory cell.

Usage: `EXCH` reg_d reg_a

Arguments:

reg_d — The data register.

reg_a — The address register.

Description:

Exchanges the contents of the given data register reg_d with the contents of the RAM memory cell at the 32-bit address given in register reg_a. If the address given happens to be the address of the EXCH instruction being executed, the hardware may treat this as a special case, and for example, ignore the instruction. Whatever it does, it must be reversible, however.

`SWAPBR`

Exchange register with branch register.

Usage: `SWAPBR` reg

Arguments:

reg — Register to swap with the branch register.

Description:

Swap the contents of register reg with the contents of the branch register. This instruction is useful at the entry/exit points of subroutines and switch statements.

`RBRA`

Direction-reversing unconditional branch.

Usage: `RBRA` *loff*

Arguments:

loff — Long immediate offset.

Description:

Like BRA, but also toggles the processor direction bit. After the branch is taken, the processor will proceed in the opposite direction from the one it was traversing originally. Useful for making reverse subroutine calls.

READ

Copy information from input device.

Usage: `READ` *reg*

Arguments:

reg — Register to read data into.

Description:

This instruction XORs the next word of data from the processor's canonical input device into register *reg*. In a multiprocessing architecture, this might be a means to receive information from the interprocessor communication network.

SHOW

Copy information to output device.

Usage: `SHOW` *reg*

Arguments:

reg — Register whose contents to show.

Description:

This instruction copies the information in register *reg* and sends the copy to the processor's canonical output device. In the Pendulum simulator, this is used to echo data to the standard output stream, for viewing program output. In a multiprocessing architecture, this might be a means to send information into the interprocessor communication network.

EMIT

Emit information from the processor.

Usage: `EMIT` *reg*

Arguments:

reg — Register whose contents to emit.

Description:

Like SHOW, but moves the data out of *reg*, instead of making a copy. This is presumed to represent the explicit, reversible removal of unwanted data from the processor to an entropy-removal mechanism. This mechanism might be a sub-processor that first compresses the garbage stream, then erases it using as little energy as possible. Or, it might be a mechanism for reversible, digitally transmitting the information to the edge of a multiprocessor mesh, and then dissipating it there. Or, it might be another mechanism for moving information out to an interprocessor communication network.

START/FINISH

End-points for computation.

Usage: `START`
`FINISH`

Arguments:

Description:

These instructions merely mark the start and end of a program, for ease of simulation. They could be used in a real processor: FINISH could halt the processor, and START could halt if running in reverse. Note, however, that actually halting the processor is an irreversible event, since one has lost the information about how long ago the processor halted. One should be careful not to build a very large, dense reversible mesh processor that will explode when all the processors simultaneously reach the FINISH instruction, as a result of each processing node dissipating the $k_{\mathrm{B}}T\ln 2$ energy to clear the bit of information that tells it that it is still running.

One could fix this problem by having FINISH merely switch to a mode where the processor starts counting the number of time-steps since it halted. When the counter runs out of space, however, the processor must either erase information, or start running again.

Appendix C

The R reversible programming language

This appendix gives a detailed description of the "R" reversible programming language we developed, which we mentioned in §9.4.2.

C.1 Introduction

R is a programming language for reversible machines. The language is currently very incomplete and not particularly stable. This appendix documents the current state of the language, to convey a feel for the language as it stands, and solicit feedback regarding how the language should develop.

The R language compiler translates R source into Pendulum assembly code. In this document we will also describe the workings of the R compiler. Currently the compiler works by applying code transformations similar to macro expansions, to reduce high-level constructs into successively lower level constructs until the expansion bottoms out with Pendulum assembly language instructions.

Because of the many levels of constructs involved in this gradual transformation process, the distinction between the constructs intended for end-use in R source programs and the intermediate constructs used internally by the compiler is currently rather fuzzy. This document will attempt to separate user-level from compiler-level constructs, but the status of constructs may change as the language evolves, and currently there is nothing to prevent an R source program from using constructs at all the different levels. However, the lower-level constructs are perhaps somewhat more likely to change as the language and compiler evolve, so their use in application programs is discouraged.

C.2 What type of language is R?

R is like C in that it is (currently) a procedural language, not a strict functional language, with data types and primitive operations centered around the two's complement fixed-precision integers and the corresponding arithmetic/logical operations that are supported directly by the machine hardware. The language supports simple C-like arrays, `for` loops, `if` statements, and recursive subroutines with arguments.

Reversibility of execution of R programs is guaranteed by the reversibility of the assumed version of the Pendulum instruction set, so long the program does not use the `EMIT` assembly-language instruction (which explicitly permits information to be removed irretrievably from the processor). However, if the user wishes his programs to run not only reversibly but correctly, he is responsible for ensuring that certain conditions are met by his code. Currently, these conditions are not checked automatically. If the conditions are not met, then the program will silently proceed anyway, with nonsensical (but still reversible) behavior. However, this is not as fatal as it sounds, because the reversibility of execution allows the errant program to be debugged, after the misbehavior is discovered, by running it in reverse from the error to see what caused it.

C.3 Overview of R Syntax

R programs are currently represented using nested, parenthesized lists of symbols and numbers, as in Lisp. Similarly to Lisp, the first element of a list may be a symbol that identifes the kind of construct that the list is representing, for example, a function definition, an `if` statement, a `let` construct for variable binding. However, in R, currently some constructs may also be denoted using *infix* notation, in which the identifying symbol is the *second* element of the list instead of the first. Many of these infix lists have a C-like syntax and behavior, for example, the `(a += b)` statement which adds b into a. Infix notation is also often used in subexpressions of a statement which are intended to evaluate to a value, for example, `(a + b)` in the statement `(print (a + b))`.

C.4 User-level Constructs

This section describes constructs that are intended for use in end-user R applications.

C.4.1 Program Structure

The executable portion of a program normally consists of a single `defmain` statement, and any number of `defsub` statements. These statements may appear in any order.

`defmain`

Define program's main routine.

Syntax: `(defmain` *`progname`*
`statement`$_1$
`statement`$_2$ `... )`

Elements:

progname — A symbol naming the program. The name should be a sequence of letters and digits starting with a letter. It should be distinct from the names of all subroutines and static data items in the program.

statement$_1$***, statement***$_2$***,*** ... — Statements to be executed in sequence as the main routine of the program.

Description:

The `defmain` statement is used to define the main routine of a program. It is intended to appear as a top-level form, but may actually appear anywhere a statement may appear. (If executed as a statement, it does nothing.) Currently there is no "command line" or other argument list available to the program; it must either be self-contained along with its data or explicitly read data from an input stream. `Defmain` generates information in the output file that tells the run-time environment where to begin executing. If there are zero or more than one `defmain` statements in a given program, then the result of attempting to run the program is undefined.

`Defmain` currently also has the side effect of causing the entire standard library to be included in the output program. Right now there is only one subroutine in the standard library (named `smf`), so this is not too burdensome.

`defsub`

Define subroutine.

Syntax: `(defsub` *`subname`* `(`*`arg`*$_1$ *`arg`*$_2$ `... )`
`statement`$_1$
`statement`$_2$ `... )`

Elements:

subname — A symbol naming the subroutine. The name should be a sequence of letters and digits starting with a letter. It should be distinct from the names of the main routine, all other subroutines, and all static data items in the program.

***arg*$_1$*, arg*$_2$*,* …** — Formal argument names, with the same alphanumeric format. These names are not required to be distinct from any other names in the program. However a single subroutine cannot have two arguments with the same name. Currently, the compiler does not support subroutines taking more than 29 arguments.

***statement*$_1$*, statement*$_2$*,* …** — Statements to be executed in sequence as the body of the subroutine.

Description:

`Defsub` statements are used to define subroutines within a program. They are intended to appear only as top-level forms, but may actually appear anywhere that a statement may appear. (If executed as a statement, a `defsub` construct does nothing.) If there are two `defsub` statements with the same *subname* in a given program, then the result of attempting to execute that program is undefined.

The formal arguments may be accessed as read-write variables within the body of the subroutine. On entry to the subroutine, the values of these variables are bound to the values of the actual arguments that were passed in via the `call` statement in the caller. The `call` statement must pass exactly the number of arguments required by the subroutine or else the behavior of the subroutine is undefined. On exit, the values of the argument variables become the new values of the actual arguments (see the description of `call`).

Any subroutine may also be called in reverse; see `rcall`.

C.4.2 Control Structure

Within the program's main routine and subroutines, the flow of execution is controlled using `call`, `rcall`, `if`, and `for` statements.

`call`, `rcall`

Call or reverse-call subroutine.

Syntax: (call *subname* *arg*$_1$ *arg*$_2$ …) *or*
(rcall *subname* *arg*$_1$ *arg*$_2$ …)

Elements:

subname — The name of the subroutine to call. If zero or more than one subroutines with that name exist in the program, the result of the call is undefined.

arg*$_1$*, arg*$_2$*, … — Actual arguments to the subroutine. These may be variables, constants, or expressions, with restrictions described below. The number of arguments must match the number of formal arguments listed in the subroutine's `defsub` statement.

Description:

A `call` or `rcall` statement is used to call a subroutine either forwards or in reverse, with arguments. If a particular actual argument is a variable or a memory reference, then the subroutine may actually change the value of its corresponding formal argument, and the caller will see the new value after the call is completed. If the argument is a constant or an expression, then it is an error for the subroutine to return with the corresponding formal argument having a value that is different from the value that the constant or expression evaluates to after the return. (Nonsensical behavior will result.)

`Rcall` differs from `call` only in that with `rcall`, the subroutine body is executed in the reverse direction from the direction in which the `rcall` is executed.

`if`

Conditional execution.

Syntax:

```
(if condition then
    statement1
    statement2 ... )
```

Elements:

condition — An expression representing a condition; considered "true" if its value is non-zero.

statement*$_1$*, statement*$_2$*, … — Statements to execute if the condition is true.

Description:

An `if` statement conditionally executes the body *statement*s if the *condition* expression evaluates to a non-zero value. If the value of the *condition* expression ever has a different value at the end of the body from the value it had at the beginning, then program behavior after that point will in general be nonsensical.

The top-level operation in the *condition* expression may be a normal expression operation, or one of the relational operators `=`, `<`, `>`, `<=`, `>=`, `!=` which have the expected C-like meanings of signed integer comparison. These relational operators are not currently supported for use in expressions in contexts other than the top-level expressions in `if` *conditions*.

Actually the compiler does not yet support all the different relations with all of the possible types of arguments even in `if` *conditions*. The `if` implementation in the compiler needs some major rewriting.

In the future, `if` statements will also be allowed to appear in forms containing `else` clauses, using the syntax

```
(if condition
    if-statement1 if-statement2 ...
 else
    else-statement1 else-statement2 ... ),
```

but this form of `if` is not yet implemented by the compiler.

`for` For loop; definite iteration.

Syntax:

```
(for var = start to end
     statement1
     statement2 ... )
```

Elements:

var — A variable name.

start — Start value expression.

end — End value expression.

***statement*$_1$*, statement*$_2$*,* ...** — Statements to execute on each iteration.

Description:

A `for` statement performs definite iteration. *Var* must not exist as a variable at the point where the `for` construct appears, but it may exist as a name of a static data element, in which case this meaning will be shadowed during the `for`.

Before the loop, the *start* and *end* expressions are evaluated, and *var* is bound to the value of *start*. The scope of *var* is the body of the `for`. On each iteration, the body *statement*s are executed. After each iteration, if *var* is equal to the value of *end* which was computed earlier, the loop terminates; otherwise, *var* is incremented as a mod-2^{32} integer and the loop continues. After the loop, the *start* and *end* expressions are evaluated again in reverse to uncompute their stored values.

It is an error for either the *start* or *end* expressions to evaluate to different values after the loop than they do before the loop. If they do, program behavior afterwards will be nonsensical. The same goes for any of their subexpressions.

Note that although `for` is intended for definite iteration, in which the number of iterations is always exactly the difference between the initially-computed *start* and *end* values, actually there is nothing to prevent the value of *var* from being modified within the body, so that the number of iterations can actually be determined dynamically as the iteration proceeds. One can thus construct "while"-like indefinite iteration functionality using `for` as a primitive. However, this is inconvenient, so the language will eventually explicitly include a `while`-like construct, though it does not do so currently.

C.4.3 Variables

New local variables may be created and bound to values anywhere a statement may appear, using the `let` statement.

`let`

New variable binding.

Syntax:

```
(let (var <- val)
        statement1
        statement2 ... )
```

Elements:

var — A new variable name. This name must not exist as a local variable name at the point where the `let` statement occurs. However, it may exist as the name of a static data item, in which case that meaning will be shadowed within the body of the `let`.

val — An expression to whose value *var* will be bound.

***statement$_1$*, *statement$_2$*, ...** — Statements to execute in the scope where *var* is available as a variable.

Description:

`Let` creates a new local variable *var* and binds it to a value. The body of the `let` may change the value of *var*, but the value of *var* at the end of the body *must* match the value that the *val* expression has at the time the body ends. Otherwise program behavior will be unpredictable thereafter. The *val* expression is actually evaluated twice, once forwards before the body, to generate the value to bind to *var*, and once backwards after the body, to uncompute this value.

Actually the current implementation of `let` does require the value of *val* and all its subexpressions to remain the same at both the start and end of the body. Future implementations may relax this restriction.

Other forms of the `let` construct currently exist, but are not currently documented as user-level constructs.

C.4.4 Data Modification

Currently, R programs modify variables and memory locations using a variety of vaguely C-like data modification constructs: `++`, `-`, `<=<`, `>=>`, `+=`, `-=`, `^=`, `<->`, and others not currently documented as user-level operations.

In general, it is an error for a data modification statement to modify a variable or memory location whose value is used in any subexpressions of the statement. If this happens, program behavior thereafter will be nonsensical.

`++` Integer increment statement.

Syntax: `(`*`place`*` ++)`

Elements:

place — A variable or an expression denoting a memory reference.

Description:

The mod-2^{32} integer word stored in *place*, which may be a variable or a memory reference, is incremented by 1.

`-` (minus sign) Unary negate statement.

Syntax: `(- `*`place`*`)`

Elements:

place — A variable or an expression denoting a memory reference.

Description:

The integer word stored in *place* is negated in two's complement fashion.

`<=<`, `>=>` Rotate left/right.

Syntax: `(place <=< amount)`
`(place >=> amount)`

Elements:

place — A variable or an expression denoting a memory reference.

amount — An expression for the amount to rotate by.

Description:

`<=<` rotates the bits stored in the given *place* to the left by the given *amount*. Rotating left by 1 means the bit stored in most significant bit-location moves to the least significant bit-location, and all the other bits shift over to the next, more significant position. Rotating by some other amount produces the same result as rotating by 1 *amount* times. `>=>` is the same but rotates to the right (exactly undoing `<=<`).

+=, -= Add/subtract statement.

Syntax: `(place += value)`
`(place -= value)`

Elements:

place — A variable, or an expression denoting a memory reference.

value — An expression for the value to add/subtract.

Description:

`+=` adds *value* into *place*, as an integer. `-=` subtracts *value* from *place*.

---

^= Exclusive OR.

Syntax: `(place ^= value)`

Elements:

place — A variable, or an expression denoting a memory reference.

value — An expression for the value to XOR.

Description:

`^=` bitwise exclusive-OR's *value* into *place*.

`<->` Swap.

Syntax: `(`*`place`*$_1$ `<->` *`place`*$_2$`)`

Elements:

place$_1$**,** ***place***$_2$ — Each is a variable or an expression denoting a memory reference.

Description:

`<->` swaps the contents of the two *place*s.

C.4.5 Expressions

Variables and constants count as expression, as do the more complex parenthesized expressions described here. Expressions may be nested arbitrarily deeply. Parentheses for all the subexpressions must all be explicitly present. Currently, all expression operations are of the infix style, where the symbol for the operator appears as the second member of the list; however new kinds of expressions may exist later.

Currently available expression operations include `+`, `-`, `&`, `<<`, `>>`, `*`, `*/`, `_`, and others for internal use by the compiler.

There are also relational operators `=`, `<`, `>`, `<=`, `>=`, `!=` which may currently only be used at top-level expressions in `if` *condition*s. They are not yet documented individually yet, but they have the expected behavior of signed integer comparison. Conceptually they return 1 if the relation holds, and 0 otherwise.

Inside expressions, only expression constructs may be used. Expression constructs may never be used in place of statements.

Expressions are generally evaluated twice each time they are used, once in the forward direction to generate the result, and once in the reverse direction to uncompute it.

There is currently no way within the language to define a new type of expression operator, but this may change later.

`+,-` Sum/difference expression.

Syntax: `(`*`val`*$_1$ `+` *`val`*$_2$`)`
`(`*`val`*$_1$ `-` *`val`*$_2$`)`

Elements:

val_1, val_2 — Expressions for values to add.

Description:

Evaluates to the sum or difference of the values of the two sub-expressions taken as mod-2^{32} integers.

& Bitwise logical AND expression.

Syntax: `(`val_1 `&` val_2`)`

Elements:

val_1, val_2 — Expressions for values to AND.

Description:

Evaluates to the bitwise logical AND of the word values of the two sub-expressions.

<<, >> Logical left/right shift expression.

Syntax:
```
(val << amt)
(val >> amt)
```

Elements:

val — Expression for the value to be shifted.

amt — Expression for the amount to shift by.

Description:

This evaluates to the value of *val* logically shifted left or right as a 32-bit word, by *amt* bit-positions.

***** Pointer dereference expression.

Syntax: `(* address)`

Elements:

address — Expression that evaluates to a memory address.

Description:

This evaluates to a copy of the contents of the memory location at the given *address*. However, this expression may also be used as a *place* which may be modified by any of the data-modification statements above, in which case the actual contents of the location, not a copy, is modified.

It is an error for the contents of an address to be referred to by a subexpression of a statement that modifies that same address; if this is done, behavior thenceforth will be unpredictable.

`*/` Fractional product expression.

Syntax: `(`*`integer */ fraction`*`)`

Elements:

integer — An expression whose value is taken as a signed integer.

fraction — An expression whose value is taken as fraction between -1 and 1.

Description:

This rather odd operator returns the signed 32-bit integer product of the two values, taking one as a signed 32-bit integer and the other as a signed 32-bit fixed-precision fraction between 0 and 1. Another way of saying this is that it is the product of two integers, divided by 2^{32}. Or, it is the upper word of the 64-bit product of the two integers, rather than the lower word.

This operation is useful for doing fixed-precision fractional arithmetic. It is used by the single existing significant test program `sch.r`.

Since the Pendulum architecture naturally does not support this rather unusual operation directly, the compiler transforms it into a call to the standard library routine `SMF` (Signed Multiplication by Fraction). `SMF` is itself written in R, but for efficiency it uses some optimized internal compiler constructs that are not yet intended for general use.

`_` (underscore) Array dereference expression.

Syntax: `(`*`array _ index`*`)`

Elements:

array — Expression for the address of element 0 of an array in memory.

index — Expression for the index of the array element to access.

Description:

This type of expression evaluates to a copy of the contents of the element numbered *index* in the sequential array of memory locations whose element number 0 is pointed to by *array*. However, this expression may also be used as a *place* in any of the data-modification statements, in which case it is the real array element that will be modified, not just a copy of it.

It is an error for an array element or other memory location to be examined by a subexpression of a statement that ends up modifying that location.

C.4.6 Static Data

Two constructs, `defword` and `defarray`, allow single words and regions of memory to be named and initialized to definite values when the program is loaded.

`defword`

Define a global variable.

Syntax: `(defword` ***`name value`***`)`

Elements:

name — An alphanumeric symbol naming this variable. Must be distinct from the names of routines and other static data items.

value — A 32-bit constant integer giving the initial value of the variable.

Description:

`Defword` is intended for use as a top-level form but actually it may appear anywhere a statement may appear. When executed as a statement it does nothing.

`Defword` defines the name *name* to globally refer to a particular unique memory location, whose initial value when the program is loaded is *value*. This meaning of *name* can be shadowed within subroutines that have *name* as a formal argument, or within the body of a `let` statement that binds that name. The *name* can be used as a *place* in data-modification statements.

Actually the *name* will only be recognized to refer to the memory location at statements in the program that occur textually *after* the `defword` declaration.

`defarray`

Define a global array.

Syntax: (defarray *name*
value$_0$ *value*$_1$ ...)

Elements:

name — Unique alphanumeric name for the array.

value$_0$**,** ***value***$_1$**,** ... — Integer constants giving the initial values of all the array elements.

Description:

`Defarray` is intended for use as a top-level form, but actually it may appear anywhere a statement may appear. When executed as a statement it does nothing.

`Defarray` sets aside a contiguous region of memory, containing a number of words equal to the number of *value* arguments, and defines the name *name* to globally (that is, after the `defarray`) refer to the address of the first word in the region. The words are initialized to the given *value*s when the program is first loaded. *Name* can be used as an *array* in array-dereference operations. It is a compile-time error to attempt to change the value of *name*. However, *name* can be shadowed by subroutine arguments and local variable declarations.

C.4.7 Input/Output

Currently there are no input constructs in R. However, there are two user-level output constructs, `printword` and `println`. These are rather ad-hoc. The set of I/O constructs is a part of R that is particularly likely to change in later versions of the language.

`printword` Output a representation of a word of data.

Syntax: `(printword` *`val`* `)`

Elements:

val — An expression for the word to print.

Description:

`Printword` sends to the output stream a representation of the value of the *val* expression, as a 32-bit integer. Currently the representation consists of outputing the value `0` followed by the given value, to distinguish the output from that produced by `println`.

The *val* expression is evaluated twice, once to compute the value and again to uncompute it.

`println` Output a representation of a line-break delimiter.

Syntax: `(println)`

Elements:

None.

Description:

`Println` sends to the output stream a representation of a line-break delimiter. Currently this consists of outputing the single word `1`.

C.5 Example Programs

Figure 9.3, p. 238 showed a simple example of a multiplication subroutine written in R.

As an additional example of R programming style and of many of the user-level constructs described above, appendix E, §E.3 (p. 396) shows the first significant R test program, `sch.r`, in its entirety. The character “`;`” indicates a comment that runs to the end of the line.

This program simulates the quantum-mechanical behavior of an electron oscillating at near the speed of light in a 1-dimensional parabolic potential well about 1 Ångstrom (10^{-10} m) wide. It takes about 1 minute to complete each 5×10^{-22} second long simulation step under the PendVM Pendulum virtual machine emulation program, running on a Sun SPARCstation 20.

An interesting feature of this program is that although it is perfectly reversible, its outer loop can run for indefinitely long periods, without either slowing down or filling up the memory with garbage data.

The current version of the compiler successfully compiles this program to correct (though not optimally efficient) Pendulum code, which is shown in §E.4 (p. 397). When run, the compiled program produces exactly the correct output.

C.6 Compiler Internals

Appendix D describes the R compilation infrastructure and documents the low-level R constructs that are not recommended for prime time.

C.7 Conclusions

R is a pretty cool little language, but it has a long way to go. It would be nice to have support for floating-point arithmetic, strings, structures with named fields, dynamic memory allocation, various built-in abstract data types, type checking and other error checking, exception handling, *etc., etc.* Not to mention object-oriented programming. It would be nice to have the option to use high-level irreversible operations, and have the compiler deal intelligently with the garbage data.

But anyway, the above describes revision 0.0 of the language, as a proof of concept and a starting point for further development.

Appendix D

The R language compiler

This appendix gives the documentation and complete code for RCOMP, the compiler for the R programming language.

D.1 R Compiler User's Guide

Currently the R compiler does not have a very convenient user interface. But this section describes the interface, such as it is.

The R compiler resides in the files `~mpf/rcomp/*.lisp` at the MIT AI Lab. The files are also listed in §D.4 below, and can be downloaded from the web via the URL `http://www.ai.mit.edu/~mpf/rc/rcomp`. To use RCOMP, start a Common Lisp interpreter, change to a directory containing the source files, and type `(load "loader")`. This will load up the system.

Write your R program in a file such as `program.r`. Then type at Lisp, `(rcompile-file "program.r")`. First, the source will be printed with comments stripped out, and then, after a delay, the entire Pendulum assembly code for the compiled program will be printed. The example program `sch.r` takes about 15 seconds to compile on a Sparc 20.

Alternatively, one can type `(rcompile-file "program.r" :debug t)`, and the full program will be printed out after every tiny little step in the compilation process. The output from this is voluminous, but by doing an incremental search (`^S`/`^R`) in Emacs for the characters "`==>`" in this output, one can easily scroll through it to see what is going on at each compilation step. However, it is not recommended to try this option with very large programs.

When there are compilation problems, it is helpful to try compiling individual subroutines and statements in isolation, using the calls `(rc` *source*`)` and `(rcd` *source*`)`, where *source* is a Lisp expression that evaluates to the list representing the fragment of source code to be compiled. `Rc` prints out the compiled assembly code, and `rcd` prints out the complete appearance of the program after each compilation step. For example, `(rc '(a += b))` outputs the single Pendulum assembly language instruction "`ADD $2 $3`."

D.2 Compilation technique

The RCOMP compiler was written in Common Lisp, which provides a portable and well-defined language base that runs on many hardware platforms, and a comprehensive standard set of built-in list-manipulation functions, that allow very easy manipulation of program code fragments represented in a Lisp-like (s-expression) format.

The basic principle of RCOMP operation is essentially macro expansion. A high-level language construct is interpreted like a macro; it is expanded in-place to a sequence of compiler-internal constructs, which are in turn expanded to even lower-level constructs until the whole process bottoms out with expressions representing assembly language instructions in the target architecture.

In fact, the earlier versions of RCOMP, which were very limited in their capabilities, actually used Common Lisp's built-in macro expansion facility to do all the work. The entire compiler was implemented as just a set of macro definitions.

However, this raw approach was not capable of performing important compiler functions such as register assignment, since each statement was compiled separately without knowledge of its context. So we replaced the reliance on built-in macro expansion with our own custom macro-like facility which carried an *environment* structure along through the code expansion process. The environment keeps track of the current variable assignments, mapping the lexical variables to registers and stack locations.

Unfortunately, the macro-expansion type of approach is also not capable of performing certain types of code optimizations, in particular optimizations that might operate across statement boundaries, such as peephole optimizations. So the code produced by RCOMP is not particularly well-minimized. The situation could be improved somewhat by adding a post-processing phase to the compilation in which the sequence of assembly instructions is scanned for known patterns that can be simplified. But a more aggressive and thorough approach to optimization would probably benefit from some higher-level knowledge, and this would probably require a different overall compiler architecture than our macro-expansion-based approach.

However, RCOMP was not intended to be a study in advanced compiler techniques. Rather, for us the imperative was to make the compiler particularly simple to modify and extend, in order to easily handle new constructs, implement existing ones differently, or port the entire compiler to a different target reversible instruction set. All these things are easy in our current design.

The reason for having such flexibility is simply that reversible programming languages are still very much experimental and in flux. It is desirable at this point to be able to very rapidly experiment with different high-level constructs and different machine instructions.

Another advantage of the macro-based design is that it was very simple to program, and the compiler code is fairly easy to read. The definition of most language constructs is just a literal expression (using the backtick operator) showing exactly what that construct expands into.

D.3 Internal compiler constructs

We now document a selection of most of the important internal language constructs currently used internally in the R compiler. These constructs are not meant to be used by the end-user in RCOMP programs. However, when testing and debugging the compiler, or for understanding how it works, it may be useful to try compiling code containing these lower-level constructs. Also, for writing standard libraries, direct use of lower-level constructs may enable better-optimized code than the compiler is currently capable of generating given only the highest-level constructs.

Generally, there is nothing in the current RCOMP architecture that actually prevents user programs from being written using any desired mixture of user-level constructs, intermediate- to low-level internal constructs, and target assembly code. However, users should be aware that if they do not stick to the user-level constructs, then their programs may not be portable across changes in the compiler internals and/or the target architecture.

D.3.1 Intermediate-level internal constructs

Of the internal compiler constructs, the following are still relatively high-level.

D.3.1.1 Mid-level variable manipulation

`<-` Bind.

Syntax: `(`*`var`*` <- `*`val`*`)`

Elements:

var — Variable to bind.

val — Expression whose value to bind it to.

Description:

Assuming the variable *var* is initially clear, bind the variable to the value obtained by evaluating the expression *val*.

`->`

Unbind.

Syntax: `(var -> val)`

Elements:

var — Variable to unbind.

val — Expression whose value to unbind it to.

Description:

Assuming the variable *var* initially contains the value that the expression *val* evaluates to, restore *var* to 0.

`with-regvars`

Declare register variables.

Syntax:

```
(with-regvars var-or-vars
        statement1
        statement2 ... )
```

Elements:

var-or-vars — A variable or list of variables to declare.

***statement*$_1$, *statement*$_2$, ...** — Statements to compile within the context of this declaration.

Description:

This mid-level construct declares a list of variables and forces them into registers, for greater efficiency of the body statements that refer to those variables. The variables are ensured to be declared only within the lexical body of this statement. They are not guaranteed to remain in registers throughout the entire body, only to be in registers intially.

`register`

Advise register allocation.

Syntax: `(register var-or-vars)`

Elements:

var-or-vars — A variable or list of variables.

Description:

Advise the compiler to move the listed variables into registers now, rather than later. This is considered to be advice to the compiler, intended to aid optimization. The compiler need not obey the advice, although the current implementation always does.

`scope`

Declare a variable around a body.

Syntax:

```
(scope var-or-vars
         statement1
         statement2 ... )
```

Elements:

var-or-vars — A variable or list of variables to declare.

statement$_1$, statement$_2$, ... — Statements to compile within the context of this declaration.

Description:

Declares the given list of variables around a body. The variables are created at the start of the body and destroyed at the end. Values of all variables are intially 0. The body is assumed to restore the values of all variables to 0 before it completes. If it does not, further program behavior will not obey the intended semantics! It is currently an error if any of the named variables already exist in the environment (currently there is no compiler support for multiple lexical nestings of scopes for variables with the same name).

D.3.1.2 Mid-level environment manipulation

`ensure-green`

Ensure that a body leaves the environment unchanged.

Syntax:

```
(ensure-green
         statement1
         statement2 ... )
```

Elements:

statement*$_1$*, statement*$_2$*, ... — Statements to compile within the context of this declaration.

Description:

Leaves the environment after executing the given body exactly the same as it was before the body. (At least, with regards to its location map.) This may generate code to move variables back to their original locations if they happened to be moved during the course of the body.

`with-location-map` Enforce a location map around a body.

Syntax:

```
(with-location-map locmapdesc
     statement1
     statement2 ... )
```

Elements:

locmapdesc — Description of a location map.

statement*$_1$*, statement*$_2$*, ... — Statements to compile within the context of this declaration.

Description:

A declaration that ensures that the environment at the start and end of the body maps variables to locations exactly as described by the given location map. This may trigger relocating variables around to be in the appropriate locations. If the set of variables in the current environment is not the same as that in the location map, currently a compiler error is generated.

`with-environment` Enforce an environment around a body.

Syntax:

```
(with-environment envdesc
     statement1
     statement2 ... )
```

Elements:

envdesc — Description of an environment.

***statement*$_1$, *statement*$_2$, …** — Statements to compile within the context of this declaration.

Description:

Just like `with-location-map`, except it takes an entire environment description (currently this is the same as an environment object) as its input, rather than just a location map description. Only affects the location map of the current environment, and not other properties of the environment.

`environment`

Enforce an environment.

Syntax: `(environment` *`envdesc`*`)`

Elements:

envdesc — Description of an environment.

Description:

Ensure that the environment immediately following this statement is equivalent to the given environment—that is, has the same location map. This may involve moving variables around to be in the appropriate locations. If the set of variables in the current environment is not the same as the set in the given environment, currently a compiler error is signaled.

`locmap`

Enforce a location map.

Syntax: `(locmap` *`locamapdesc`*`)`

Elements:

locmapdesc — Description of a location map.

Description:

Just like `environment`, but takes a location map description instead of a full environment.

D.3.1.3 Mid-level control-flow constructs

`infloop`

Infinite loop.

Syntax:

```
(infloop
    statement₁
    statement₂ ... )
```

Elements:

statement*$_1$*, statement*$_2$*, … — Statements to compile inside the loop.

Description:

Generates an "infinite loop" segment of code—if processing inside the loop runs sequentially off the end, it comes back to the beginning. If the infinite loop is encountered from the outside, we skip over it.

Actually, the present compilation of this construct does not actually enforce that the loop is infinite; entry/exit points could be inserted into the middle of the loop body.

`skip`

Unexecutable code segment.

Syntax:

```
(skip
    statement₁
    statement₂ ... )
```

Elements:

statement*$_1$*, statement*$_2$*, … — Statements to compile in the skipped section.

Description:

Generates a segment of code that should be skipped over if encountered; it should not be executed. This is primarily only useful for preventing static data objects from being executed. Really this is implemented exactly like `infloop`. But the connotation is different. In `infloop` we are trying to keep the PC on the *inside*, in `skip` we are trying to keep it on the *outside*. But in both cases, the PC actually just stays on whichever side it was originally. (Unless we cheat and construct brach-pairs connecting inside and outside.)

D.3.1.4 Mid-level branching constructs

`bcs-branch-pair` Pair of binary conditional switching branches.

Syntax:

```
(bcs-branch-pair
            toplabel (vara1 opa vara2)
            botlabel (varb1 opb varb2)
    statement1
    statement2 ... )
```

Elements:

toplabel — Label of the upper branch in the pair.

vara1, vara2 — Variables to compare at the top of the branch.

opa — Operation to compare with at the top of the branch.

botlabel — Label of the lower branch in the pair.

varb1, varb2 — Variables to compare at the bottom of the branch.

opb — Operation to compare with at the bottom of the branch.

statement$_1$***, statement***$_2$***,*** ... — Statements to compile in between the two branches.

Description:

This construct implements the common low-level reversible control-flow situation in which one has a pair of switching (that is, paired) conditional branches pointing to each other, at the top and bottom of a block of code, and the condition being tested is a binary (two-operand) function of two given variables. The operation may be any of the relational operators `=`, `<`, `>`, `<=`, `>=`, `!=` which have the expected C-like meanings of signed integer comparison.

Depending on how `bcs-branch-pair` is used, it can be used to implement either reversible `if` statements or loops. The code is potentially a loop if the second condition can succeed even if the first condition does not.

An important feature of `bcs-branch-pair` is that it ensures that the environment is conserved by the body, despite whatever juggling of registers is done inside the body. This is important because otherwise, we could not at compile time predict what the environment would be after the branch, because we would not know whether the branch would be taken or not.

`twin-us-branch` Pair of unconditional switching branches.

Syntax:

```
(twin-us-branch toplabel botlabel
    statement1
    statement2 ... )
```

Elements:

toplabel — Label of the upper branch in the pair.

botlabel — Label of the lower branch in the pair.

statement*₁*, statement*₂*, … — Statements to compile in between the two branches.

Description:

This construct implements the common low-level reversible control-flow situation in which one has a pair of switching (paired) unconditional branches pointing to each other, at the top and bottom of a block of code. Such a structure may surround a infinite loop, or the body of a subroutine whose entry/exit point is somewhere in its middle. If the structure is encountered from outside, we just jump over it. This construct knows that the environment after the branch pair is the same as the one before it.

D.3.1.5 Miscellaneous mid-level constructs

`with`

Execute body under some temporary condition.

Syntax:

```
(with tmp-statement
    statement1
    statement2 ... )
```

Elements:

tmp-statement — Temporary statement to do before the body and undo after the body.

statement*₁*, statement*₂*, … — Statements to do after doing and before undoing the given statement.

Description:

Given any statement *tmp-statement* that the compiler knows how to invert, performs that statement, then performs the body statements *statement*$_i$, then performs the inverse operation of the statement to undo its effects. Does not yet handle all possible varieties of *tmp-statement*.

`undo` Undo a given statement.

Syntax: `(undo statement)`

Elements:

statement — Statement to undo.

Description:

Execute a given statement in reverse, thereby undoing its effects.

D.3.2 Low-level constructs

D.3.2.1 Low-level manipulation of variables

`relocate` Relocate a variable in the environment.

Syntax: `(relocate var loc)`

Elements:

var — The variable to relocate.

loc — Specifier for the register or stack location to relocate to.

Description:

This construct relocates the given variable to a specific register or stack location. The *loc* is currently represented as a list, with either the form `(reg regno)`, where *regno* is the register number, 0–31, or `(stock offset)`, where *offset* is the stack offset, which should be a negative integer specifying the location relative to the current value of the stack pointer (which currently is register 1).

The relocation of a variable involves both changing its location assignment in the environment, and generating code that actually moves the variable's value from the old location to the new location.

If a different variable is already assigned to the given location, `relocate` moves it out of the way. (By exchanging it with the old location of *var* if there was one, or by moving it to a fresh location.)

`new-var-at` Create a new variable at a given location.

Syntax: `(new-var-at` *`varname location`*`)`

Elements:

varname — The name of the variable to create.

location — Location specifier to create the new variable at.

Description:

Create a new variable named *varname* in the current environment, and assign it to location *location*. If another variable was assigned to *location*, move that variable to some other location.

`vacate`

Ensure that a given location is unassigned.

Syntax: `(vacate` *`loc`*`)`

Elements:

loc — Specifier of location to vacate.

Description:

If a variable is assigned to the given location, move it to somewhere else. If there are any free registers, the variable is moved to one of them, otherwise it is moved to the next available location on the stack.

`get-in-register`

Force a variable into a register.

Syntax: `(get-in-register` *`var`*`)`

Elements:

var — The variable to registerify.

Description:

Force the given variable to relocate to a register if it isn't in one already. Picks an arbitrary free register for it to live in, or if no registers are free, boots the least-recently-moved register variable back out to the stack, and moves our variable there.

`add-to-env`

Add a variable to the environment.

Syntax: `(add-to-env` *`varname`*`)`

Elements:

varname — Name of variable to create.

Description:

Explicitly creates a variable of the given name in the current environment, initially unassigned to any particular location. It is currently an error if a variable with the given name already exists.

`tell-loc` Set the raw location of a variable.

Syntax: `(tell-loc var loc)`

Elements:

var — The variable whose location to set.

loc — The location to set it to.

Description:

Explicitly sets the location of the given variable in the current environment to the one given. Does not move the variable's contents.

`remove-var` Remove a variable from the environment.

Syntax: `(remove-var var)`

Elements:

var — The variable to remove.

Description:

Explicitly removes a variable from the current environment. Dangerous to subsequent program correctness if the contents of *var* are non-zero.

D.3.2.2 Raw manipulation of environments

`declare-green` Explicitly restore the environment after a body.

Syntax:

```
(declare-green
     statement₁
     statement₂ ... )
```

Elements:

***statement*$_1$*, statement*$_2$*,* ...** — Statements to compile within the context of this declaration.

Description:

Explicitly set the environment after the body completes back to what it was before the body. Does not move variables appropriately. This is a very low-level operation.

`declare-locmap`

Explicitly set the current location map.

Syntax: `(declare-locmap locmapdesc)`

Elements:

locmapdesc — Location map description.

Description:

Explicitly set the entire variable-location map of the current environment to the one given. Does not move contents of variables' locations appropriately. This is a very low-level operation.

`declare-environment`

Explicitly set the current environment.

Syntax: `(declare-environment envdesc)`

Elements:

envdesc — Environment description.

Description:

Explicitly set the entire current environment to the one given. Does not move contents of variables' locations appropriately. This is a very low-level operation.

D.3.2.3 Raw register/stack manipuation

This section describes constructs for direct manipulation (moving and exchanging contents) of register and stack locations.

`swaploc`

Swaps the contents of two register/stack locations.

Syntax: `(swaploc` loc_1 loc_2`)`

Elements:

***loc*$_1$ *loc*$_2$** — Specifiers for the two locations to swap.

Description:

Swaps the contents of the two specified locations. See `relocate` for the format of a location specifier. Note this function does not change variable location assignments, only the values of the specified locations.

`moveloc`

Move the contents of one location to another.

Syntax: `(moveloc` loc_1 loc_2`)`

Elements:

***loc*$_1$** — Source location.

***loc*$_2$** — Destination location.

Description:

Assuming loc_2 is empty, moves the contents of loc_1 to it, leaving loc_1 empty. If loc_2 is not empty, the effect will be different.

`exregstack`

Exchange register with stack location.

Syntax: `(exregstack` *`reg stackloc`*`)`

Elements:

reg — A register location specifier.

stackloc — A stack location specifier.

Description:

Exchanges the contents of a given register location with those of a stack location. Does not change the environment. The current implementation works fine but leads to some suboptimization. (See the code comment.)

`swapregs`

Exchange the contents of two registers.

Syntax: `(swapregs` *r1 r2*`)`

Elements:

r1*, *r2 — Specifiers of register locations to swap.

Description:

Swaps the contents of two registers. Despite the absence of direct hardware support for this operation in the current PISA spec, the current implementation cleverly swaps the two registers in-place anyway by performing a sequence of three XOR's of one register into another. I learned this particular trick from Charles Isbell's hand-written assembly-language programs for early versions of Pendulum.

`movereg`

Moves one register to another.

Syntax: `(movereg` *r1 r2*`)`

Elements:

r1 — Source register.

r2 — Destination register.

Description:

Assuming *r2* is initially empty, moves *r1* to *r2*, leaving *r1* empty. Implemented like `swapregs`, but takes advantage of the assumption that *r2* is empty to eliminate one of the 3 XORs. If *r2* is not empty, the outcome will not fit the intended semantics.

`with-stack-top`

Temporarily go to top of stack.

Syntax:

```
(with-stack-top
     statement₁
     statement₂ ... )
```

Elements:

> ***statement*$_1$*, statement*$_2$*,* . . .** — Statements to execute in the context of the adjusted stack pointer.

Description:

Execute the given body within a context where the stack pointer is reset to the top of the stack, that is, beyond all currently-allocated variables. This is used in the implementation of subroutine calls. The environment is not adjusted, so any entries in the environment that might have been referring to stack-allocated variables will be present but invalid in the body. Higher-level constructs that use `with-stack-top` need to cope with this.

`with-SP-adjustment`

Temporarily adjust stack pointer.

Syntax:

```
(with-SP-adjustment amt
          statement1
          statement2 ... )
```

Elements:

> ***amt*** — Literal amount by which to adjust stack pointer.
>
> ***statement*$_1$*, statement*$_2$*,* . . .** — Statements to execute in the context of the adjusted stack pointer.

Description:

Execute the body within a context where the stack pointer is temporarily adjusted by the explicit amount *amt*. The stack grows downwards in memory, so negative amounts correspond to moving towards the “top” (most short-lived) part of the stack.

D.3.2.4 Low-level control flow constructs

`_if`

Specialized if statement.

Syntax:

```
(_if (reg1 op reg2) then
          statement1
          statement2 ... )
```

Elements:

reg1, reg2 — Registers to compare.

op — Binary relation to use for comparison.

statement$_1$***, statement***$_2$***,*** ... — Statements to execute if the condition is true.

Description:

Like `if`, but the condition must be a simple binary relation (with no nested subexpressions) between two explicit registers *reg1* and *reg2*.

`_ifelse`

Specialized if/else statement.

Syntax: `(_ifelse` (*reg1 op* reg2) `then`
(*if-statement*$_1$
if-statement$_2$ ...)
(*else-statement*$_1$
else-statement$_2$ ...))

Elements:

reg1, reg2 — Registers to compare.

op — Binary relation to use for comparison.

if-statement$_1$***, if-statement***$_2$***,*** ... — Statements to execute if the condition is true.

else-statement$_1$***, else-statement***$_2$***,*** ... — Statements to execute if the condition is false.

Description:

Like `_if`, but there is a second "else" body that will be executed if the condition fails.

`gosub`

Low-level subroutine call.

Syntax: `(gosub` *subname*`)`

Elements:

subname — Name of subroutine to call.

Description:

Calls the subroutine of the given name. This is a low-level operation that doesn't know anything about stacks, arguments, or calling conventions. It just transfers control.

`rgosub`

Low-level reverse subroutine call.

Syntax: `(rgosub` *`subname`*`)`

Elements:

subname — Name of subroutine to call.

Description:

Like `gosub`, but runs the subroutine in reverse.

`portal`

Subroutine entry/exit point.

Syntax: `(portal` *`label`*`)`

Elements:

label — Name to give this portal.

Description:

Declares a subroutine entry/exit point to exist at this point in the program, with the name *label*. A caller can call *label* and have control transferred to this point. Then, if control loops back around to this point again, the portal will switch control back to the caller.

The portal currently works by using register 2 to store the information needed to return to the caller. So the value in this register needs to be preserved during the body of the subroutine.

D.3.2.5 Low-level branching constructs

`sbra-pair`

Low-level pair of switching unconditional branches.

Syntax:

```
(sbra-pair toplabel botlabel
        statement1
        statement2 ... )
```

Elements:

toplabel — Label of the upper branch in the pair.

botlabel — Label of the lower branch in the pair.

statement$_1$**,** ***statement***$_2$**,** … — Statements to compile in between the two branches.

Description:

Lower-level analogue of `twin-us-branch` that does not worry about maintaining the environment across all the branching.

`sbra` Switching branch.

Syntax: `(sbra` *`thislabel otherlabel`*`)`

Elements:

thislabel — Label for this branch.

otherlabel — Label for the other branch that we switch to.

Description:

Unconditional switching branch labeled *thislabel* that exchanges control between here and another such branch labeled *otherlabel*.

The semantics is: If we arrive at this statement from *otherlabel*, then continue in normal sequential execution. If we arrive at this statement sequentially, then branch to *otherlabel*. If we arrive at this statement some other way, the behavior is some other, nonsensical thing. So you can see that these "switching" branches have to come in pairs.

`sbez` Switching branch on equal to zero.

Syntax: `(sbez` *`thislabel var otherlabel`*`)`

Elements:

thislabel — Label for this branch.

var — Variable to test.

otherlabel — Label for the other branch that we switch to.

Description:

This is an example of a *conditional* switching branch. It tests *var* and if it is zero, it behaves like `sbra`, otherwise it is a no-op. If we branch into this statement but its condition does not succeed, the resulting behavior will be nonsensical.

`sbne`

Switching branch on not equal to zero.

Syntax: `(sbne` ***`thislabel var otherlabel`***`)`

Elements:

thislabel — Label for this branch.

var — Variable to test.

otherlabel — Label for the other branch that we switch to.

Description:

This is an example of a *conditional* switching branch. It tests *var* and if it is nonzero, it behaves like `sbra`, otherwise it is a no-op. If we branch into this statement but its condition does not succeed, the resulting behavior will be nonsensical.

`bcs-branch`

Binary conditional switching branch.

Syntax: `(bcs-branch (`*`var1 op var2`*`)` ***`thislabel otherlabel`***`)`

Elements:

var1, var2 — Variables to compare.

op — Operation to compare with.

thislabel — Label for this branch.

otherlabel — Label for the other branch that we switch to.

Description:

This is a general two-operand conditional switching branch. It compares the two variables using the operation *op*, which may be any of the relational operators `=`, `<`, `>`, `<=`, `>=`, `!=`, which have the expected C-like meanings of signed integer comparison. If the comparison succeeds, it behaves like `sbra`, otherwise it is a no-op. If we branch into this statement but its condition does not succeed, the resulting behavior will be nonsensical.

`bc-branch`

Non-switching binary conditional branch.

Syntax: `(bc-branch (var1 op var2) destlabel)`

Elements:

var1, var2 — Variables to compare.

op — Operation to compare with.

destlabel — Label of destination.

Description:

This is like `bcs-branch` except that it is not a switching branch; that is, it does not branch to a location that explicitly refers back to it. Instead, the desination will probably be a subroutine entry/exit point which will save away the branch register to allow returning to this location later.

`rbc-branch`

Register binary conditional branch.

Syntax: `(rbc-branch (r1 op r2) destlabel)`

Elements:

r1, r2 — Registers to compare.

op — Operation to compare with.

destlabel — Label of destination.

Description:

Like `bc-branch` except that its arguments must be explicit registers, not variables.

D.3.2.6 Miscellaneous low-level constructs

`_with`

Specialized version of `with`.

Syntax:

```
(_with (var <- val)
       statement1
       statement2 ... )
```

Elements:

var — Variable name to bind.

val — Expression whose value to bind it to.

$statement_1$, $statement_2$, … — Statements to do after doing and before undoing the given statement.

Description:

A version of `with` in which the "statement" to be done before the body (and undone after it) must be a variable-binding. The `_with` construct is more efficient than the general version of `with` for this case, because the *val* expression is not evaluated as many times. However, `_with` may use an amount of temporary space linear in the length of the *val* expression during the course of the body, whereas `with` does not.

---

withargs

Prepare arguments as for a subroutine call.

Syntax:

```
(withargs arg-list
          statement1
          statement2 ... )
```

Elements:

arg-list — List of expressions for the arguments.

$statement_1$, $statement_2$, … — Statements to do in the context of having prepared the arguments.

Description:

Given a list of "argument" expressions, temporarily places the values of those expressions in the canonical locations where the arguments to a subroutine should go, and compiles the body in that context. This is part of the caller side of the current subroutine calling convention.

D.4 Compiler LISP source code

In this section, we give a verbatim listing of the most recent version of the RCOMP source. This is composed of the following files:

`loader.lisp` — §D.4.1, p. 335. This loads up all the parts of the RCOMP program in an appropriate order.

`util.lisp` — §D.4.2, p. 335. Defines general-purpose utilty functions and macros that we use.

`infrastructure.lisp` — §D.4.3, p. 336. Defines our macro-expansion like facility for defining how to compile language constructs.

`location.lisp` — §D.4.4, p. 341. Defines some functions for working with objects that describe a variable's location in the register file or stack.

`environment.lisp` — §D.4.5, p. 342. Defines the environment objects which map variables to their locations.

Files defining construct-expansion "macros":

- `regstack.lisp` — §D.4.6, p. 346. Defines low-level constructs for direct manipulation of registers and the stack.
- `variables.lisp` — §D.4.7, p. 347. Defines high- to low-level constructs for manipulation of variables in variable assignments (environments).
- `branches.lisp` — §D.4.8, p. 350. Constructs providing intermediate- and low-level support for various kinds of branch structures for control-flow.
- `expression.lisp` — §D.4.9, p. 354. Constructs and low-level functions for expanding nested expressions.
- `clike.lisp` — §D.4.10, p. 359. Defines constructs for various user-level user-level C-like operators.
- `print.lisp` — §D.4.11, p. 362. Defines a few very simple constructs for producing output.
- `controlflow.lisp` — §D.4.12, p. 362. Defines user-level to intermediate-level control flow constructs such as conditionals and looping.
- `subroutines.lisp` — §D.4.13, p. 364. Provides high and low level support for subroutines.
- `staticdata.lisp` — §D.4.14, p. 367. Defines constructs for defining static data objects. Currently this is the only way to provide input to a program.
- `program.lisp` — §D.4.15, p. 367. Defines very high-level constructs for wrapping around the entire program.
- `library.lisp` — §D.4.16, p. 368. Defines constructs that expand into code for standard subroutine libraries. Currently the library is very minimal.

`files.lisp` — §D.4.17, p. 369. Provides support for reading the source code to compile from a file.

`test.lisp` — §D.4.18, p. 370. Miscellaneous functions and R code fragments for exercising the compiler. Some of these may be obsolete.

Let us now present the code.

D.4.1 `loader.lisp`

This very simple file just loads up all the Common Lisp files that make up Rcomp. It is completely devoid of any sophisticated options. Perhaps someday we should use one of the popular CL system-definition facilities instead.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;----------------------------------------------------------------------
;;; This is the loader for the reversible compiler system.  Currently all
;;; files are just in the USER package.

;; Load up the system.
(load "util")            ; General-purpose utilities.
(load "infrastructure")  ; Mechanism for defining and compiling constructs.
(load "location")        ; Describing locations where variables are stored.
(load "environment")     ; Mapping variables to their locations.
(load "regstack")        ; Direct manipulation of registers and the stack.
(load "variables")       ; Creating, destroying, moving variables.
(load "branches")        ; Branches, branch pairs, labels.
(load "expression")      ; Binding variables to multiply-nested expressions.
(load "clike")           ; C-like assignment-operator statements.
(load "print")           ; Data output.
(load "controlflow")     ; High-level conditionals & loops.
(load "subroutines")     ; Support for subroutine calls.
(load "staticdata")      ; Static data definitions.
(load "program")         ; Highest-level constructs.
(load "library")         ; Standard library of R routines.
(load "files")           ; File compiler.

(load "test")            ; Example programs.

;; Command to reload the system.
(defun l () (load "loader"))
```

D.4.2 `util.lisp`

Defines some completely non-application-specific Lisp functions and macros that we use in Rcomp.

```
;;; -*- Package: user -*-
(in-package "USER")
;;;----------------------------------------------------------------------
;;; General utilities.

;;;----------------------------------------------------------------------
;;; Some abbreviations for Common Lisp entities.
```

```
;; Don't you agree that MULTIPLE-VALUE-BIND's name is too long?
(defmacro mvbind (&rest args)
  `(multiple-value-bind . ,args))

;; Same here.
(defmacro dbind (&rest args)
  `(destructuring-bind . ,args))

;;;----------------------------------------------------------------------
;;; Boolean shtuff.  Silly, but hey.

(defconstant true t)
(defconstant false nil)

(defmacro true! (&rest places)
  `(setf . ,(mapcan #'(lambda (place) (list place true)) places)))
(defmacro false! (&rest places)
  `(setf . ,(mapcan #'(lambda (place) (list place false)) places)))

;; Convert an arbitrary object to a true-false value.
(defun true? (obj) (if obj true false))
(defun false? (obj) (eq obj false))

;;;----------------------------------------------------------------------
;;; List manipulation.

;; REPL - Replace first occurrence.  FUNC is called on each item of LIST in
;; succession until it returns non-NIL, at which point a new list is
;; returned in which the guilty item is replaced by the value which was
;; returned by FUNC.  The new list shares its tail with the old.  If FUNC
;; never returns non-nil then a copy of LIST is returned.
(defun repl (list func)
  (if list
      (let ((v (funcall func (car list))))
        (if v (cons v (cdr list))
          (cons (car list) (repl (cdr list) func))))))

;; In this version of REPL, FUNC is passed not only each item of LIST,
;; but also the item's index (as per NTH or ELT).
(defun repl2 (list func &optional (firstindex 0))
  (labels
      ((helper (list index func)
          (if list
              (let ((v (funcall func (car list) index)))
                (if v (cons v (cdr list))
                  (cons (car list) (helper (cdr list) (1+ index) func)))))))
    (helper list firstindex func)))
```

D.4.3 `infrastructure.lisp`

This file is the core of our macro-style compilation architecture. The expansion of any given language construct is defined using the `defconstruct` macro, defined below.

The core function that does the work of compilation is `rcomp`. It recursively expands the macro definitions, while keeping track of the environment, until the process bottoms out in statements that cannot be further expanded.

```
;;; -*- Package: user -*-
```

```
(in-package "USER")
;;;----------------------------------------------------------------------
;;; Compilation infrastructure.

;; Given an object, return non-nil IFF it could possibly be an
;; infix-operator statement.
(defun infix-form? (form)
  (and (listp form)
       (>= (length form) 2)
       (symbolp (second form))
       (get (second form) 'is-infix)))

;; Given a form, get it into the canonical form where the operator is
;; first.
(defun canonicalize (form)
  (if (infix-form? form)
      `(,(second form) ,(first form) . ,(cddr form))
    form))

;; Given an object, if it's an operator (construct) symbol, return
;; its definition.
(defun definition (operator)
  (and (symbolp operator)
       (get operator 'construct-definition)))

;; Given an operator (construct symbol), return the opposite operator.
;; (Which will undo the effect of the given operator.)
(defun opposite (operator)
  (get operator 'opposite))

;; Given an object, return non-NIL iff it may potentially be a
;; single form statement (not a label atom) with a definition.
(defun statement? (form)
  (and (listp form)
       (not (null form))
       (or (definition (car form))
           (infix-form? form))))

;; Guess whether an object may be a list of statements/primitives.
(defun list-of-statements? (obj)
  (and (listp obj)
       (not (statement? obj))
       (not (null obj))
       (statement? (car obj))))

;; DEFCONSTRUCT - Define how a particular construct is to be compiled.
;; Given a construct name symbol CNAME, lambda list LAMBDA-LIST, and body
;; statements BODY, define CNAME to be a reversible language construct with
;; structure given by LAMBDA-LIST and compilation generated by the BODY.
;; During compilation the BODY gets executed with the variables mentioned
;; in the LAMBDA-LIST bound to corresponding parts of the item to be
;; compiled, and with the variable ENV bound to the variable-location
;; environment in effect at the start of the statement.  The body should
;; return 2 values: the first is a list of statements to which this
;; statement is equivalent.  The second value indicates the environment in
;; effect after the given statement(s).  It may be NIL meaning that the
;; source as a high-level statement does not affect the environment after
;; the statement, although the compiled lower-level statements might.

(defmacro defconstruct (cname lambda-list &body body)
  (let ((opposite
```

```
         (if (eq (car body) :opposite)
             (prog1
               (cadr body)
               (setf body (cddr body)))
           cname)))
    `(setf (get ',cname 'opposite) ',opposite
           (get ',cname 'construct-definition)
           #'(lambda (args env)
               (let ((form (cons ',cname args)))
                 (destructuring-bind ,lambda-list args
                   . ,body))))))

(defmacro definfix ((leftarg opname &rest rightargs) &body body)
  `(progn
     (defconstruct ,opname (,leftarg . ,rightargs)
       . ,body)
     (true! (get ',opname 'is-infix))))

;; Given a statement or a list of statements to compile and an optional
;; initial environment (which defaults to the empty environment), return an
;; equivalent list of compiled statements and the environment in effect
;; after them.
(defun rcomp (source &optional startenv)
  (when (null startenv) (setf startenv (empty-env)))
  (setf source (canonicalize source))
  (cond
   ((null source)
    (values source startenv))
   ((statement? source)
    ;; Source is a single non-label statement with a definition.
    (let ((def (definition (first source))))
      (mvbind (compiled endenv)
              ;; Compile it once.
              (funcall def (cdr source) startenv)
         (mvbind (recomp reenv)
                 ;; Try compiling it further.
                 (rcomp compiled startenv)
            (values recomp
                    (or endenv reenv))))))
   ((form-list? source)
    ;; Source is a list of statements.
    (mvbind (firstcomp firstendenv)
            ;; Compile first statement.
            (rcomp (first source) startenv)
       (mvbind (restcomp restendenv)
               ;; Compile remaining statements in environment from
               ;; first statement.
               (rcomp (rest source) firstendenv)
          (values (if (listp firstcomp)
                      (append firstcomp restcomp)
                    (cons firstcomp restcomp))
                  restendenv))))
   (t
    ;; In all other cases just compile the source to itself
    ;; and leave the environment unchanged.
    (values (list source) startenv))))

;;
;; This version of RCOMP, for debugging purposes, prints out the entire
;; state of the partially-compiled program after each individual code
;; transformation.
```

```
;;
;; WHOLE represents the entire current state of the compilation,
;; represented as a cons cell whose CDR is the current partially-compiled
;; source, which MUST be a LIST of statements, not a single statement.
;; POINTER is a pointer to the cons cell whose CDR is the part of the
;; source that remains to be compiled.  In general, the CAR of this CDR
;; will be an ENV statement giving the current environment.
;;
(defun rcomp-repl (whole &optional (pointer whole))
  (myprint (cdr whole)
           (if (eq (caadr pointer) 'env)
               (cddr pointer)
             (cdr pointer)))            ;Print thuh whole shebang.
  (format t "~&Ready: ")
  ;; (clear-input) (finish-output) ;These don't seem to work right.
  ;; (read-line)
  (let ((source (cdr pointer))          ;Remaining source to compile.
        startenv)
    (if (and (listp source)             ;List of statements.
             (listp (car source))       ;Non-label statement.
             (eq (caar source) 'env))   ;Special (ENV <env>) statement.
        (setf startenv (cadar source))  ;Get the <env>.
      (progn
        ;; Invent an ENV statement and insert it.
        (format t "~&Default environment.~%")
        (setf startenv (empty-env))
        (setf (cdr pointer)             ;Alter our object as follows.
              `((env ,startenv)
                . ,source))
        (myprint (cdr whole) (cddr pointer))))
    ;; Now STARTENV is the current env, and current source obj is just
    ;; after the initial env statement.
    (setf source (cddr pointer))
    ;; If no statements left to compile, we're done.
    (when (null source)
      (return-from rcomp-repl whole))
    (let ((form (car source)))
      ;; From here on we approximately mirror structure of RCOMP.
      ;; If first form is an infix form, canonicalize it.
      (when (infix-form? form)
        ;(format t "~&Canonicalize.~%")
        (setf form (canonicalize form)
              (car source) form)
        ;(myprint (cdr whole))
        )
      (cond
       ;; If first item is label: do nothing with it.
       ((atom form)
        (format t "~&Label.~%")
        (setf (cdr pointer) `(,form
                              (env ,startenv)
                              . ,(cdr source)))
        (rcomp-repl whole (cdr pointer)))
       (t ;; Else first item is a non-label STATEMENT.
        (let ((first (first form)))
          (if (symbolp first)
              ;; Assume source code is a single statement, FIRST is the symbol
              ;; naming the statement type, for dispatching.
              (if (eq first 'env)
                  (progn
                    (format t "~&Environment override.~%")
```

```
                    (setf (cdr pointer) source)
                    (rcomp-repl whole pointer))
                (let ((def (definition first)))
                  (if (null def)
                      ;; No definition for this.  Assume it's a final
                      ;; assembly instruction and doesn't change the
                      ;; environment.
                      (progn
                        (format t "~&Final.~%")
                        (setf (cdr pointer) `(,form
                                              (env ,startenv)
                                              . ,(cdr source)))
                        (rcomp-repl whole (cdr pointer)))
                    ;; OK, we do have a definition for it.
                    (mvbind (compiled endenv)
                            ;; Call the transformer function.
                            (funcall def (cdr form) startenv)
                            ;; Insert result.
                            (format t "~&Expand ~s.~%" first)
                            (if (and endenv (null compiled))
                                (setf (cdr pointer)
                                      `((env ,endenv)
                                        . ,(cdr source)))
                              (setf (cddr pointer)
                                    (if endenv
                                        `(,@(if compiled
                                                (if (not (form-list? compiled))
                                                    (list compiled)
                                                  compiled))
                                            (env ,endenv)
                                            . ,(cdr source))
                                      `(,@(if compiled
                                              (if (not (form-list? compiled))
                                                  (list compiled)
                                                compiled))
                                          . ,(cdr source)))))
                            ;; Try compiling same thing again.
                            (rcomp-repl whole pointer)))))
            ;; The first item isn't a symbol so assume it's a statement
            ;; and treat the form as a list of statements.
            (progn
              (setf source (append form (cdr source))
                    (cddr pointer) source)
              (format t "~&Insert statement list.~%")
              (rcomp-repl whole pointer))))))))

;; The user-level compiler-debugging routine.
(defun rcomp-debug (source)
  (rcomp-repl (cons nil (list source))))

(defun rcd (source)
  (rcomp-debug source))

;; non-nil iff obj could be a list of forms (not incl. label syms)
(defun form-list? (obj)
  (and (listp obj)
       (not (and (car obj) (symbolp (car obj))))
       (not (and (cadr obj) (symbolp (cadr obj))))))

;; Expand the source code to its compilation once, but not
;; recursively.  This is for debugging.
```

```
(defun expand1 (source &optional env)
  (let ((def (get (first source) 'construct-definition)))
    (if (null def)
        (values (list source) env)
      (funcall def (cdr source) env))))

;; For testing.
(defun myprint (code &optional pointer)
  #|(format t "~&~@
             --------~@
             Program:~@
             --------~%")|#
  (format t "~&~%")
  (if (list (car code))
      (dolist (s code)
        (if (eq s (car pointer)) (format t "==>"))
        (cond
         ((atom s) ;Interpret atoms as labels.
          (format t "~s:~15T" s))
         ((and (symbolp (car s))
               (not (get (car s) 'construct-definition))
               (not (eq (car s) 'env))
               (not (and (symbolp (cadr s))
                         (get (cadr s) 'construct-definition))))
          (format t "~16T")
          (dolist (w s)
            (if (register? w)
                (format t "$~s " (cadr w))
              (format t "~:w " w)))
          (format t "~%"))
         (t
          (format t "~16T~:w~%" s))))
    (pprint code))
  (format t "~&~%")
  (values))

(defun rc (source &optional startenv)
  (mvbind (prog env)
          (rcomp source startenv)
          (myprint prog)
          (format t "~&~%Final environment:")
          (print env))
  (values))
```

D.4.4 `location.lisp`

Functions for working with "location" objects which give the locations of variables. Currently these are just implemented as simple list structures.

```
;;; -*- Package: user -*-
(in-package "USER")
;;; --------------------------------------------------------------------
;;; A LOCATION object indicates where a variable is stored.
;;;
;;; The current implementation uses list structures.  If a <LOCATION> is
;;; NIL, then the variable exists in the environment but has no storage
;;; location (and is therefore also unbound).  If a <LOCATION> is (REG
;;; <regno>) then the variable is located in register number <REGNO>.  If
```

```
;;; <LOCATION> is (STACK <offset>) then the variable is located on the
;;; stack at the address SP+<OFFSET>, where SP is the current value of the
;;; stack pointer register.
;;; ---------------------------------------------------------------------

;; Return non-nil iff the object OBJ is a register location.
(defun register? (obj)
  (and (listp obj) (cdr obj) (null (cddr obj))
       (eq (car obj) 'reg)))

(defun stackloc? (obj)
  (and (listp obj) (cdr obj) (null (cddr obj))
       (eq (car obj) 'stack)))

;; Return non-nil iff the object OBJ is a null location (meaning the
;; location of a variable that is not located anywhere).
(defun null-loc? (obj)
  (null obj))

;; Return the given stack location's offset from the current stack
;; pointer.
(defun offset (stackloc)
  (cadr stackloc))

(defun location? (obj)
  (or (register? obj) (stackloc? obj)))
```

D.4.5 environment.lisp

Defines environment objects, which keep track of variables' location assignments. This is currently implemented as a full-fleged CLOS object.

```
;;; -*- Package: user -*-
(in-package "USER")
;;; =====================================================================
;;; This file defines the interface to and implementation of ENVIRONMENT
;;; objects.  An environment object determines what variables are present
;;; in the R environment at a given point in the program, and where they
;;; are stored.  The environment also maintains identifiers for static
;;; objects.  This file provides the programmer's interface to environment
;;; objects, and should be used in lieu of manipulating the underlying
;;; structures directly.  This is intended to reduce errors and allow
;;; environments to be reimplemented at a later time.
;;; =====================================================================

(defun empty-locmap () '())

(defclass environment ()
  ((variable-locations
    :type list ;More specifically an ALIST from identifiers to locations.
    :initform (empty-locmap)
    :initarg :locmap
    :accessor locmap
    :documentation "An ALIST of the form ((<var1> . <location1>) ...).
       Each VAR is a symbol, and only appears once in the alist.  The
       variables that have most recently been created or moved appear at
       the front of the alist.")
   (static-value-identifiers
    :type list
```

```
    :initform nil
    :initarg :staticvals
    :accessor staticvals
    :documentation "A list of identifier symbols that denote static
      data values permanently located in memory.")
   (static-array-identifiers
    :type list
    :initform nil
    :initarg :staticarrays
    :accessor staticarrays
    :documentation "A list of identifier symbols denoting static arrays.")
   )
  (:documentation "An environment specifies the meanings of identifiers at
     a given point during the compilation of a program."))

(defmacro make-environment (&rest args)
  `(make-instance 'environment . ,args))

(defun empty-env ()
  (make-environment))

(defun copy-environment (env)
  (make-environment :locmap (copy-alist (locmap env))
                    :staticvals (copy-list (staticvals env))
                    :staticarrays (copy-list (staticarrays env))))

(defmethod env-to-list ((env environment))
  `(:locmap ,(locmap env)
    :staticvals ,(staticvals env)
    :staticarrays ,(staticarrays env)))

(defmethod print-object ((env environment) stream)
  (write (env-to-list env) :stream stream))

;; Return an environment like ENV, but with VAR bound to location LOC
;; in the location map.
(defmethod set-loc (var loc (env environment))
  (setf env (copy-environment env)
        (locmap env) `((,var .,loc).,(remove (assoc var (locmap env))
                                             (locmap env))))
  env)

;; Return an environment that is just like the given environment ENV but
;; with the variable VAR removed from the location map.
(defmethod remove-var (var (env environment))
  (setf env (copy-environment env)
        (locmap env) (remove (assoc var (locmap env)) (locmap env)))
  env)

;; Return non-nil iff the variable VAR exists in the environment ENV.
(defmethod defined-in-env? (var (env environment))
  (assoc var (locmap env)))

;; Return the location of variable VAR in environment ENV, or nil if VAR
;; does not exist in the environment.  This is not guaranteed to be
;; distinct from the null location.  (The DEFINED-IN-ENV function can be
;; used to distinguish the two cases.)
(defmethod location (var (env environment))
  (cdr (assoc var (locmap env))))

;; Return the variable stored at the given location in the given
```

```
;; environment, or nil if none.
(defmethod var-at-loc (loc (env environment))
  (car (rassoc loc (locmap env) :test #'equal)))

;; Return non-nil iff the two environments E1 and E2 contain the exact same
;; set of variables.
(defmethod equal-vars? ((e1 environment) (e2 environment))
  (let ((answer t))
    (dolist (v (append (mapcar #'car (locmap e1)) (mapcar #'car (locmap e2))))
        (if (not (and (assoc v (locmap e1)) (assoc v (locmap e2))))
            (setf answer nil)))
    answer))

;; Return non-nil iff the two environments E1 and E2 are equivalent, in the
;; sense that they have the same variables and assign them to the same
;; locations.
(defmethod equal-env? ((e1 environment) (e2 environment))
  (let ((answer t))
    (dolist (v (append (mapcar #'car (locmap e1)) (mapcar #'car (locmap e2))))
      (if (not (equal (assoc v (locmap e1)) (assoc v (locmap e2))))
          (setf answer nil)))
    answer))

;; Return the first variable in environment E1 that is not located in the
;; same place in environment E2.
(defmethod first-misloc ((e1 environment) (e2 environment))
  (dolist (v (mapcar #'car (locmap e1)))
    (if (not (equal (cdr (assoc v (locmap e1))) (cdr (assoc v (locmap e2)))))
        (return v))))

;; Return the lowest-numbered available register location in the given
;; environment.  Available means not containing any variable.  Registers 0
;; and 1 are never available because 0 is reserved to contain 0 and 1 is
;; reserved to be the stack pointer.  Returns nil if no registers are
;; available.
(defmethod next-avail-reg ((env environment))
  (let ((i 2))
    (loop
     (if (not (rassoc `(reg ,i) (locmap env) :test #'equal))
         (return `(reg ,i)))
     (incf i)
     (if (= i 32) (return)))))

;; Return the first stack location ABOVE THE CURRENT STACK
;; POINTER that is available (doesn't contain a variable) in the given
;; environment.  The stack grows down, so ABOVE means MORE NEGATIVE THAN.
(defmethod next-avail-stack ((env environment))
  (let ((i -1))
    (loop
     (if (not (rassoc `(stack ,i) (locmap env) :test #'equal))
         (return `(stack ,i)))
     (decf i))))

;; Return a variable in the given environment that was least recently
;; created or moved.
(defmethod least-recently-moved ((env environment))
  ;; Currently this information is maintained by putting newly-created or
  ;; moved variables on the front of the alist, so we just return the last
  ;; variable on the list.
  (car (nth (1- (length (locmap env))) (locmap env))))
```

```
;; Return non-nil if VAR is located in a register in environment ENV.
(defmethod in-register? (var (env environment))
  (register? (location var env)))

;; Return the index of the topmost stack location at SP or below that
;; has a variable in it.  In other words, where is
;; the top of the stack in relation to the current stack pointer.
(defmethod top-of-stack ((env environment))
  (let ((h 0))
    (dolist (loc (locmap env))
      (if (eq (cadr loc) 'stack)
          (if (< (caddr loc) h)
              (setf h (caddr loc)))))
    h))

(defmethod add-staticval (name (env environment))
  (setf env (copy-environment env)
        (staticvals env) (cons name (staticvals env)))
  env)

(defmethod add-staticarray (name (env environment))
  (setf env (copy-environment env)
        (staticarrays env) (cons name (staticarrays env)))
  env)

(defmethod static-id? (obj (env environment))
  (and (symbolp obj)
       (or (member obj (staticvals env))
           (member obj (staticarrays env)))))

(defmethod dynamic-var? (obj (env environment))
  (and (symbolp obj)
       (not (static-id? obj env))))

(defmethod static-array? (obj (env environment))
  (member obj (staticarrays env)))

(defun negated-sym? (obj)
  (and (symbolp obj)
       (eq #\- (elt (symbol-name obj) 0))))

(defun positive-of (negsym)
  (intern (subseq (symbol-name negsym) 1)))

(defmethod literal? (obj (env environment))
  (or (numberp obj)
      ;; Static array identifiers are literals because they stand for their
      ;; addresses, and are left in the form of symbols which are processed
      ;; directly by the assembler.
      (static-array? obj env)
      ;; If FOO is a static array, -FOO is a literal also.
      (and (negated-sym? obj)
           (static-array? (positive-of obj) env))
      ;; Expression is the address of a static-val.
      (static-val-addr? obj env)))

(defmethod static-val? (obj (env environment))
  (member obj (staticvals env)))

(defmethod static-val-addr? (obj (env environment))
  (and (listp obj)
```

```
      (eq (car obj) '&)
      (null (cddr obj))
      (static-val? (second obj) env)))
```

D.4.6 regstack.lisp

This file defines low-level constructs for directly manipulating registers and the stack.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;--------------------------------------------------------------------
;;; Register/stack manipulation.

(defconstruct relocate (var loc)
  (let ((oldloc (location var env)))
    (if (not (equal oldloc loc))        ;If not already there.
        (if (null oldloc)
            `((vacate ,loc)
              (tell-loc ,var ,loc))
          (if (null loc)
              `(tell-loc ,var ,loc)
            (let ((oldv (var-at-loc loc env)))
              (if oldv
                  ;; If new location occupied, swap.
                  `((swaploc ,oldloc ,loc)
                    (tell-loc ,var ,loc)
                    (tell-loc ,oldv ,oldloc))
                ;; Not occupied, just move.
                (if (null-loc? oldloc)
                    `(tell-loc ,var ,loc)
                  `((moveloc ,oldloc ,loc)
                    (tell-loc ,var ,loc))))))))))

;;; loc1 and loc2 should be register or stack locations.
(defconstruct swaploc (loc1 loc2)
  (if (and (register? loc1)
           (register? loc2))
      `(swapregs ,loc1 ,loc2)
    (if (register? loc1)
        `(exregstack ,loc1 ,loc2)
      (if (register? loc2)
          `(exregstack ,loc2 ,loc1)
        ;; We need a temporary register to facilitate the stack exchange;
        ;; we choose reg. 31 for no particular reason.  The net change to
        ;; it is nil.  This all works but could probably be made
        ;; considerably more efficient.
        `((exregstack (reg 31) ,loc1)
          (exregstack (reg 31) ,loc2)
          (exregstack (reg 31) ,loc1))))))

;; Assuming loc2 is clear, move loc1 to it.
(defconstruct moveloc (loc1 loc2)
  (if (and (register? loc1)
           (register? loc2))
      `(movereg ,loc1 ,loc2)
    (if (register? loc1)
        `(exregstack ,loc1 ,loc2)
      (if (register? loc2)
```

```
          `(exregstack ,loc2 ,loc1)
        ;; We need a temporary register to facilitate the stack exchange;
        ;; we choose reg. 31 for no particular reason.  The net change to
        ;; it is nil.  This all works but could probably be made
        ;; considerably more efficient.
        `((exregstack (reg 31) ,loc1)
          (exregstack (reg 31) ,loc2)
          (exregstack (reg 31) ,loc1))))))

;; Would save a lot of ADDI instructions if I changed this to modify the
;; stack pointer before but not after; and instead change the environment
;; to reflect correct new variable locations and amount of stack adjustment
;; from original.  But perhaps it would be better to leave the stack
;; pointer alone and get rid of adjacent ADDIs via a later peephole
;; optimization or something.
(defconstruct exregstack (reg stackloc)
  `(with-SP-adjustment ,(offset stackloc)
        (exch ,reg (reg 1))))

;; Given a register, push its contents onto the stack.  Not currently used.
(defconstruct push (reg)
  `((exch ,reg (reg 1)) ;; <-- Convention: r1 is stack pointer.
    (++ (reg 1))))

;;;-----------------------------------------------------------------------
;;; Pure register manipulation.

;; swapregs R1 R2 - Given two registers, swap their contents.
(defconstruct swapregs (r1 r2)
  ;; Implementation for architectures that support XOR'ing regs, but not
  ;; swapping regs directly.  Another way uses +=, -=, and NEG but takes 4
  ;; instructions.
  `((,r1 ^= ,r2)
    (,r2 ^= ,r1)
    (,r1 ^= ,r2)))

;; Fast way to move r1 to r2 when r2 is known to be empty!  Otherwise
;; behavior is "undefined" (actually in this implementation r1 gets r2, but
;; r2 ends up with r2^r1).
(defconstruct movereg (r1 r2)
  `((,r2 ^= ,r1)
    (,r1 ^= ,r2)))
```

D.4.7 variables.lisp

Defines high- to low-level constructs for manipulation of variables in variable assignments (environments).

```
;;; -*- Package: user -*-
(in-package "USER")
;;; -----------------------------------------------------------------------
;;; Constructs for variable-environment manipulation (creating/destroying
;;; variables, changing their locations, etc.)
;;; -----------------------------------------------------------------------

;;;-----------------------------------------------------------------------
;;; User-level constructs.

;; Create a new variable VAR, bind it to VAL, execute the BODY, and then
```

```
;; unbind it from VAL and get rid of it.  VAL may be an expression, but it
;; must evaluate to whatever value VAR actually has at the end of the BODY,
;; or else all bets are off!
;;
;; 6/3/97 - Now LET is slightly more general---VAR can be put into a
;; relationship with VAL in any of a number of ways... <-, <->, ^=, +=...
;; <-, ^=, += are all equivalent given that SCOPE forces VAR to initially
;; be zero, but <-> is different... It sets VAR by swapping it with VAL,
;; which obviously must be a location of some sort.  Afterwards VAR is
;; restored to zero by swapping it back.  Note that in this case, if VAR is
;; not left at zero by the BODY, this is fine and results in VAL being
;; side-effected.  In other words, this kind of LET is effectively
;; temporarily giving VAL a new name which pulls it into a register if say
;; it was originally an array entry.  Another kind of operation (not yet
;; defined) would have VAL be a variable and assign VAR to be truly a
;; synonym for that exact same variable.

(defconstruct let ((var ~ val) &body body)
  (if (eq ~ '<-)
      `(scope ,var
          (_with (,var <- ,val)
                 . ,body))
    `(scope ,var
        (with (,var ,~ ,val)
              . ,body))))

;; Declare some vars that should be allocated register locations as soon
;; as they are created.
(defconstruct with-regvars (var-or-vars &body body)
  `(scope ,var-or-vars
     (register ,var-or-vars)
     . ,body))

;; User-level hint to compiler: put the following variables in registers
;; now rather than later.
(defconstruct register (var-or-vars)
  (let ((varlist (if (listp var-or-vars) var-or-vars (list var-or-vars))))
    (mapcar #'(lambda (var) `(get-in-register ,var)) varlist)))

;;;----------------------------------------------------------------------
;;; Intermediate-level constructs.  Not recommended for casual users.

(defconstruct with-location-map (locmapdesc &body body)
  `((locmap ,locmapdesc)
    ,@body
    (locmap ,locmapdesc)))

;; WITH-ENVIRONMENT envdesc body - Ensure that the environment, as far as
;; location maps go, is equivalent to the one specified by environment
;; description ENVDESC, both at the beginning and at the end of the body.
(defconstruct with-environment (envdesc &body body)
  `((environment ,envdesc)
    ,@body
    (environment ,envdesc)))

;; ENVIRONMENT envdesc - Ensure that the environment is equivalent to the
;; one specified by environment description ENVDESC.  ENVDESC must describe
;; an environment object.  Currently the only supported kind of description
;; is an environment object itself.  Environments are equivalent if they
;; have the same variables in the same locations.  ENVIRONMENT will move
;; variables around as necessary to make the environments match, but it
```

```
;; will not create or destroy any variables.  If the environments cannot be
;; made to match, currently a compiler error is generated.
(defconstruct environment (envdesc)
  (if (equal-env? env envdesc)
      (values '() envdesc)  ;Tell RCOMP the requested form of the description.
    (if (equal-vars? env envdesc)       ;Do the envs have the same variables?
        ;; Relocate the first mis-located variable, and try again.
        (let ((v (first-misloc env envdesc)))
          (if (null (location v envdesc))
              `(environment ,(set-loc v (location v env) envdesc))
            `((relocate ,v ,(location v envdesc))
              (environment ,envdesc))))
      (error "Environments ~s and ~s don't match." env envdesc))))

(defconstruct locmap (locmapdesc)
  (setf env2 (copy-environment env)
        (locmap env2) locmapdesc)
  `(environment ,env2))

(defconstruct declare-locmap (locmapdesc)
  (setf env (copy-environment env)
        (locmap env) locmapdesc)
  (values nil env))

(defconstruct declare-environment (envdesc)
  (setf env (copy-environment envdesc))
  (values nil env))

;; The given variable should exist throughout the body of the scope, no
;; more, no less.  The body must leave the variable clear, or else!
(defconstruct scope (var &body body)
  (let ((vlist (if (listp var) var (list var))))
    `(with ,(mapcar #'(lambda (var) `(add-to-env ,var)) vlist)
           ,@body))
  ;; Note danger if var is not actually clear at end of BODY!
  )

;; ENSURE-GREEN enforces environmental correctness -- it leaves the
;; environment just the way it found it. (With regards to its location map.)
(defconstruct ensure-green (&body body)
  `(,@body
    (locmap ,(locmap env))))

;; DECLARE-GREEN declares that the environment in effect when the body
;; is entered will necessarily be in effect when it ends.  Don't use this
;; when it isn't true!
(defconstruct declare-green (&body body)
  (values body env))

;; Make the LOCATION be clear, and associate it with a new variable VARNAME.
(defconstruct new-var-at (varname location)
  (if (defined-in-env? varname env)
      (error "Variable ~s is already in the environment!" varname)
    `((vacate ,location)
      (tell-loc ,varname ,location))))

;; Make a particular location LOC become available (empty, and no variable
;; assigned to it).
(defconstruct vacate (loc)
  (let ((v (var-at-loc loc env)))
    (if v
```

```
        ;; Location is occupied by variable V.
        (let ((reg (next-avail-reg env)))
          ;; If any registers are available, move V there.
          (if reg `(relocate ,v ,reg)
            ;; Else move V to the next available stack location.
            (let ((s (next-avail-stack env)))
              (if s `(relocate ,v ,s))))))))

;; Arrange for the given variable, which should already be present in the
;; environment, to be located in a register (instead of on the stack).
(defconstruct get-in-register (var)
  (if (symbolp var)
      (let ((l (location var env)))
        (if (not (register? l))         ;If it's not already in a register,
            (let ((reg (next-avail-reg env)))
              (if reg       ;If there is a register avaiable, put it there.
                  (if (null-loc? l)
                      `(tell-loc ,var ,reg)
                    `(relocate ,var ,reg))
                ;; Else boot out the least-recently moved variable.
                (let* ((victim (least-recently-moved env))
                       (loc (location victim env)))
                  (if (null-loc? l)
                      `((vacate ,loc)
                        (tell-loc ,var ,loc))
                    `(relocate ,var ,loc)))))))))

;;;---------------------------------------------------------------------
;;; Primitive environment-manipulation constructs.

;; Create the given variable in the environment, but don't give it a
;; location quite yet.
(defconstruct add-to-env (var) :opposite remove-var
  (if (defined-in-env? var env)
      (error "Variable ~s already exists!" var)
    (values '() (set-loc var nil env))))

;; Change the current environment to have the location of variable VAR as
;; being LOC.  Generates no code.  This construct is dangerous if the old
;; location of VAR has a non-zero runtime value, and is not associated with
;; any other variable.
(defconstruct tell-loc (var loc)
  (values '() (set-loc var loc env)))

;; Assuming that a variable is empty, remove it from the environment!
;; (Danger, Will Robinson!)  This causes grave problems if the runtime
;; value of the variable is not zero.  But currently we generate no runtime
;; code to notice that condition, so watch out!
(defconstruct remove-var (varname) :opposite add-to-env
  (values '() (remove-var varname env)))
```

D.4.8 `branches.lisp`

Constructs providing intermediate- and low-level support for various kinds of branch structures for control-flow.

```
;;; -*- Package: user -*-
(in-package "USER")
```

```
;;;-------------------------------------------------------------------
;;; Support for various kinds of branches.
;;; These constructs are not intended to appear in source code,
;;; but are rather used to implement higher-level control-flow
;;; constructs.
;;;-------------------------------------------------------------------

;;;-------------------------------------------------------------------
;;; Relatively high-level branch constructs.

;; paired binary conditional switching branches.  The body in between the
;; two branches must conserve the environment or else we won't have a
;; definite compile-time idea of the environment after the branch pair
;; because we don't know whether the branch succeeds or fails at run-time
;; or not.  Note the vars must really be variables and not literals or
;; registers.
(defconstruct bcs-branch-pair (toplab (vara1 ~a vara2)
                               botlab (varb1 ~b varb2)
                               &body body)
  `(;; All the variables involved need to be in registers before we start.
    (get-in-register ,vara1)
    (get-in-register ,vara2)
    (get-in-register ,varb1)
    (get-in-register ,varb2)
    (bcs-branch (,vara1 ,~a ,vara2) ,toplab ,botlab)
    ;; Since the vars were already in registers, BCS-BRANCH will not have
    ;; modified the environment after the branch point.
    (ensure-green ;Complain if the body doesn't clean up after itself.
     ,@body)
    ;; Due to the above GET-IN-REGISTERs and the ENSURE-GREEN, the b
    ;; variables will already be in registers here, so this BCS-BRANCH will
    ;; not need to modify the environment at all from the current one,
    ;; which is identical to the one just before the branch point.
    (bcs-branch (,varb1 ,~b ,varb2) ,botlab ,toplab)))

;; (The following construct is currently not used.)  Twin (meaning with
;; identical tests and variables) binary-operator conditional switching
;; branches.  The body in between the two branches must conserve the
;; environment or else we won't have a definite compile-time idea of the
;; environment after the branch pair because we don't know whether the
;; branch succeeds or fails at run-time or not.
(defconstruct twin-bcs-branch ((var1 ~ var2) toplab botlab &body body)
  `((get-in-register ,var1)
    (get-in-register ,var2)
    (bcs-branch (,var1 ,~ ,var2) ,toplab ,botlab)
    ;; bcs-branch had better not itself change the environment
    ;; after it branches!
    (ensure-green
     ,@body)
    (bcs-branch (,var1 ,~ ,var2) ,botlab ,toplab)))

;; Twin unconditional switching branches.
;; Now that the environment contains information other than
;; the locations of run-time variables, does it make
;; sense for it to be completely green?  Declaring static variables
;; inside the body might be expected to be able to affect the
;; outside world.  Or, maybe it shouldn't.  Haven't decided yet.
(defconstruct twin-us-branch (toplab botlab &body body)
  ;; TWIN-US-BRANCH is GREEN on the outside because encountering it from
  ;; the outside you just jump over it, and the environment doesn't change.
  `(declare-green
```

```
    (sbra-pair ,toplab ,botlab . ,body))
  ;; We have no idea what the environment is inside the first bra, though
  ;; (cuz we might slip into the middle as a subroutine call), so we can't
  ;; put an ensure-green or anything in there.  The environment in effect
  ;; at this point, which just leeches in from above, is usually wrong.
  ;; Still, the body needs to be careful that whatever environment is in
  ;; effect at its end is the same as the one it assumes at its top.  But
  ;; this is a job for a higher level to worry about.
  )

;;;----------------------------------------------------------------------
;;; lower-level branch constructs.

;;; Note to self: I think that the way branches and environments
;;; interact may be incorrect.  If the variables mentioned in the
;;; branch are not in registers, then the environment needs to change
;;; to allow the branch to be done---so the environment declared
;;; at the place we're branching to may be wrong.  Need to go thru
;;; and fix carefully.  Really, need a more sophisticated approach
;;; to how environments are kept track of during compilation of
;;; complex control-flow structures.

;; This low-level thing doesn't worry about environments at all.
;; That's a job for higher-level dudes that use it.
(defconstruct sbra-pair (toplab botlab &body body)
  `((sbra ,toplab ,botlab)
    ,@body
    (sbra ,botlab ,toplab)))

;;
;; SBRA: Switching branch (unconditional).  Branch is
;; from label thislab to otherlab.
;;
;; Semantics is: if we branch to this statement from
;; otherlab, then continue forwards normally.  If we
;; arrive at this statement normally, then branch to
;; otherlab.  If we arrive at this statement some other
;; way, results are undefined.
;;
(defconstruct sbra (thislab otherlab)
  `((label ,thislab)  ;Convention: label precedes labeled statement.
    (bra ,otherlab))) ;Assume that the hardware gives the semantics we want.

(defconstruct sbez (thislab var lab)
  ;; There's no built-in BEZ, so we reserve reg $0 to be zero
  ;; so we can just use BEQ instead.
  `(bcs-branch (,var = (reg 0)) ,thislab ,lab))

(defconstruct sbnz (thislab var lab)
  ;; There's no built-in BNZ, so we reserve reg $0 to be zero
  ;; so we can just use BNE instead.
  `(bcs-branch (,var != (reg 0)) ,thislab ,lab))

(defconstruct bcs-branch ((var1 ~ var2) thislab otherlab)
  (cond
   ;; If the variables aren't in registers, get them there.  The only thing
   ;; is, I'm not sure it makes sense to do any environment manipulation at
   ;; this level because it makes it difficult for the higher level stuff
   ;; to ensure thtat the environment is consistent at both ends of the
   ;; actual branch point.
   ((and (member ~ '(= !=)) (numberp var2) (zerop var2))
```

```
    ;; To compare for equality with zero, compare with $0 (reserved for 0).
    `(bcs-branch (,var1 ,~ (reg 0)) ,thislab ,otherlab))
   ((and (or (register? var1)
             (in-register? var1 env)
             (eql var1 0))
         (or (register? var2)
             (in-register? var2 env)
             (eql var2 0)))
    `((label ,thislab)
      ;; This assumes that BC-BRANCH has the right switching sort of
      ;; semantics, and that it doesn't insert any instructions before the
      ;; actual branch.
      (bc-branch (,var1 ,~ ,var2) ,otherlab)))
   ;; Does this make sense?
   ((and (symbolp var1) (defined-in-env? var1 env)
         (not (in-register? var1 env)))
    `((get-in-register ,var1)
      (bcs-branch (,var1 ,~ ,var2) ,thislab ,otherlab)))
   ((and (symbolp var2) (defined-in-env? var2 env)
         (not (in-register? var2 env)))
    `((get-in-register ,var2)
      (bcs-branch (,var1 ,~ ,var2) ,thislab ,otherlab)))
   ((and (symbolp var1) (not (defined-in-env? var1 env)))
    `((add-to-env ,var1)
      (bcs-branch (,var1 ,~ ,var2) ,thislab ,otherlab)))
   ((and (symbolp var2) (not (defined-in-env? var2 env)))
    `((add-to-env ,var2)
      (bcs-branch (,var1 ,~ ,var2) ,thislab ,otherlab)))
   ;; I could go on and handle expressions and literals as well, but again
   ;; I'm concerned that this isn't the right level.
   ))

;; binary conditional branch
(defconstruct bc-branch ((var1 ~ var2) destlab)
  (cond
   ((and (or (register? var1) (eql var1 0))
         (or (register? var2) (eql var2 0)))
    `(rbc-branch (,var1 ,~ ,var2) ,destlab))
   ;; If the variables are in registers, just look at the registers.
   ((and (symbolp var1) (in-register? var1 env))
    `(bc-branch (,(location var1 env) ,~ ,var2) ,destlab))
   ((and (symbolp var2) (in-register? var2 env))
    `(bc-branch (,var1 ,~ ,(location var2 env)) ,destlab))
   ;; We can't really do any environment manipulation at this level
   ;; because the instructions manipulated will come between us and the
   ;; label inserted by BCS-BRANCH.
   (t
    (error "BC-BRANCH can only cope with variables residing in registers!")
    )))

;;;----------------------------------------------------------------------
;;; Branching primitives.

;; A label is a tag that gives an address in code that is a target
;; for branching.
(defconstruct label (labname)
  labname)

(defconstruct bez (reg lab)
  ;; There's no built-in BEZ, so we reserve reg $0 to be zero
  ;; so we can just use BEQ instead.
```

```
  `(rbc-branch (,reg = (reg 0)) ,lab))

(defconstruct bnz (reg lab)
  ;; Similar, use BNE.
  `(rbc-branch (,reg != (reg 0)) ,lab))

(defparameter *bc-branch-instructions*
  '((!= . bne) (= . beq)))

(defparameter *zerocmp-branch-instructions*
  '((>= . bgez) (<= . blez) (> . bgtz) (< . bltz)))

;; "register binary conditional branch"
;; tests whether two registers satisfy some relation ~ and if so
;; branch to DESTLAB.
(defconstruct rbc-branch ((reg1 ~ reg2) destlab)
  (cond
   ((assoc ~ *bc-branch-instructions*)
    `(,(cdr (assoc ~ *bc-branch-instructions*))
      ,reg1 ,reg2 ,destlab))
   ((and (eql reg2 0)
         (assoc ~ *zerocmp-branch-instructions*))
    `(,(cdr (assoc ~ *zerocmp-branch-instructions*))
      ,reg1 ,destlab))))
```

D.4.9 expression.lisp

Constructs and low-level functions for expanding nested expressions.

```
;;; -*- Package: user -*-
(in-package "USER")

;; This version of WITH allows any temporary effect, not just
;; variable binding, to be done and undone around the body.
(defconstruct with (statement &body body)
  `(,statement
    ,@body
    (undo ,statement)))

;; Given any statements, do their reverse (undoing their effects).  Don't
;; depend too much on this always working yet.
(defconstruct undo (&rest statements)
  (unless (null statements)
    (dbind (first . rest) statements
      (if (list-of-statements? first)
          `(undo ,@first . ,rest)
        (let ((statement (canonicalize first)))
          `((undo . ,rest)
            (,(opposite (first statement)) . ,(rest statement))))))))

;; Maps "expanding" binary operators to their do/undo statements.
(defparameter *expanding*
  '((& ^=&) (<< ^=<<) (>> ^=>>) (* +=* -=*) (*/ +=*/ -=*/) (_ <-_ ->_)))
(defun forw (binop)
  (cadr (assoc binop *expanding*)))
(defun revers (binop)
  (or (caddr (assoc binop *expanding*))
      (cadr (assoc binop *expanding*))))
```

```
(defun expression? (obj)
  (and (listp obj)
       (not (location? obj))
       (not (statement? obj))))

;; This version of with doesn't evaluate expressions as many times.
;; but uses up linear space during body.  It only handles <- (bind) type
;; statements though.
(defconstruct _with ((var <- val) &body body)
  (cond
   ((or (not (expression? val))
        (literal? val env))
    `(with (,var <- ,val)
           . ,body))
   ((null (cddr val)) ; No more than 2 words in value expression.
    (cond
     ((eq (first val) '*)
      ;; These expansions are a bit questionable because what if the body
      ;; tries to look at the dereferenced value also?  It will see 0 (or
      ;; whatever was in VAR) instead.  But the alternative, of introducing
      ;; yet another temporary and swapping the contents back before doing
      ;; the body, seems too inefficient.
      (if (or (register? (second val))
              (dynamic-var? (second val) env))
          `((,var <->* ,(second val))
            ,@body
            (,var <->* ,(second val)))
        (let ((tv (gentemp)))
          `(_with (,tv <- ,(second val))
              `((,var <->* ,tv)
                ,@body
                (,var <->* ,tv))))))))
   (t
    (dbind (a1 ~ a2) val  ;But what about other expressions?
      (let ((rb (revers ~))
            (fb (forw ~)))
        (cond
         ((and (numberp a1) (numberp a2))
          `((,var <- ,(funcall ~ a1 a2)) ;Warning: this is too simplistic.
            ,@body
            (,var -> ,(funcall ~ a1 a2))))
         ((eq ~ '+)
          `(with ((,var += ,a1)
                  (,var += ,a2))
            ,@body))
         ((and (eq ~ '+) (not (expression? a1)))
          `(_with (,var <- ,a2)
              (with (,var += a1)
                 ,@body)))
         ((and (eq ~ '+) (not (expression? a2)))
          `(_with (,var <- ,a1)
              (with (,var += ,a2)
                 ,@body)))
         ((and (eq ~ '-) (not (expression? a2)))
          `(_with (,var <- ,a1)
              (with (,var -= ,a2)
                 ,@body)))
         ((and (expression? a1) (expression? a2))
          (let ((tv1 (gentemp))
                (tv2 (gentemp)))
            `(let (,tv1 <- ,a1)
```

```
               (let (,tv2 <- ,a2)
                 (,var ,fb ,tv1 ,tv2)
                 ,@body
                 (,var ,rb ,tv1 ,tv2)))))
         ((expression? a1)
          (let ((tv (gentemp)))
            `(let (,tv <- ,a1)
               (,var ,fb ,tv ,a2)
               ,@body
               (,var ,rb ,tv ,a2))))
         ((expression? a2)
          (let ((tv (gentemp)))
            `(let (,tv <- ,a2)
               (,var ,fb ,a1 ,tv)
               ,@body
               (,var ,rb ,a1 ,tv))))
         (t
          `((,var ,fb ,a1 ,a2)
            ,@body
            (,var ,rb ,a1 ,a2)))))))))

;;;----------------------------------------------------------------------
;;; Variable binding and unbinding.  For most purposes this takes the
;;; place of assignment.

;; Semantics of BIND: assuming that VAR is already clear,
;; set it to VAL.
(definfix (var <- val) :opposite ->
  ;; Implemented using +=, but ^= would also work.
  (cond
   ((or (symbolp val) (numberp val) (register? val) (literal? val env))
    `(,var += ,val))
   ((expression? val)
    ;; Binary expression.
    (destructuring-bind (a1 ~ a2) val   ;What about other syntaxes?
      (cond
       ((and (numberp a1) (numberp a2))
        `(,var += ,(funcall ~ a1 a2)))  ; Really too simplistic.
       ((eq ~ '+)
        `((,var += ,a1)
          (,var += ,a2)))
       ((eq ~ '-)
        `((,var += ,a1)
          (,var -= ,a2)))
       ((eq ~ '^)
        `((,var ^= ,a1)
          (,var ^= ,a2)))
       ((eq ~ '*)
        `(,var +=* ,a1 ,a2))
       ((assoc ~ *expanding*)
        `(,var ,(forw ~) ,a1 ,a2))
       ((extract form env :lvalues 1))
       (t
        (error "Don't know how to compile ~s." form)))))))

;; Assuming that VAR=VAL, restore it to zero.
(definfix (var -> val) :opposite <-
  (cond
   ((or (symbolp val) (numberp val) (register? val) (literal? val env))
    `(,var -= ,val))
   ((expression? val)
```

```
    ;; Binary expression.
    (destructuring-bind (a1 ~ a2) val
      (cond
       ((and (numberp a1) (numberp a2))
        `(,var -= ,(funcall ~ a1 a2)))  ; Really too simplistic.
       ((eq ~ '+)
        `((,var -= ,a1)
          (,var -= ,a2)))
       ((eq ~ '-)
        `((,var -= ,a1)
          (,var += ,a2)))
       ((eq ~ '^)
        `((,var ^= ,a1)
          (,var ^= ,a2)))
       ((eq ~ '*)
        `(,var -=* ,a1 ,a2))
       ((assoc ~ *expanding*)
        ;; Use the appropriate reverse op if different from forward one.
        `(,(revers ~) ,var ,a1 ,a2))
       ((extract form env :lvalues 1))
       (t
        (error "Don't know how to compile ~s." form))))))


;;; --------------------------------------------------------------------
;;; New thingy.  constructs all use this same function EXTRACT to
;;; automatically replace located variables with their locations, move
;;; stack variables into registers before operating on them, create
;;; temporary variables for subexpressions and compute their values.

;; EXTRACT - you give it a form, and it returns code that
;; creates appropriate temporary stuff around it and gets variables
;; in registers together with a reduced version of the original
;; form.
;;    RELEVANT-TERMS is a list of the indices (as per NTH or ELT) of those
;; terms that are candidates for expanding.  If not provided, all terms are
;; considered fair game.
;;    LVALUES is similarly the index of or a list of the indices of
;; a term or terms that are considered to be "lvalues", that is,
;; "destinations" where the value of the term is changed by the
;; statement.  Anything in LVALUES is automatically also a candidate
;; for expanding.
;;    The indices in both RELEVANT-TERMS and LVALUES refer to the
;; indices the terms have *after* any canonicalization.
;;    NIL is returned if EXTRACT can't do anything.

(defun extract (form env &key lvalues (relevant-terms
                                       (labels ((countlist (n)
                                                 (if (>= n 0)
                                                     (cons n
                                                       (countlist
                                                        (1- n))))))
                                         (countlist (length form)))))
  (setf form (canonicalize form));So we can forget about infix.
  (labels ((lvalue? (index)
             "Return non-NIL if the given index is the index of
              a term that is an LVALUE (modifiable term)."
             (or (eql index lvalues)
                 (and (listp lvalues)
                     (member index lvalues))))
           (candidate? (index)
```

```
         "Return non-NIL if the given index is the index of a
          term that is a candidate for expansion."
         (or (member index relevant-terms)
             (lvalue? index))))
;; First we locate the first term that is an expression or
;; a literal, and convert it into a temporary variable.
(let (before)
  (setf (cdr form)
        (repl2 (cdr form)
              #'(lambda (term index)
                  (if (and (candidate? index)
                           (or (expression? term)
                               (literal? term env)))
                      (let ((tv (gentemp)))
                        (if (lvalue? index)
                            (setf before `(,tv <-> ,term))
                          (setf before `(,tv <- ,term)))
                        tv)))
              1))
  (when before
    (return-from extract `(let ,before ,form))))
;; If any term is a static value identifier, wrap the statement in
;; a binding of a temporary to the value's address and replace the
;; term with a dereferencing expression.
(let (before)
  (setf (cdr form)
        (repl2 (cdr form)
              #'(lambda (term index)
                  (if (and (candidate? index)
                           (static-val? term env))
                      (let ((tv (gentemp)))
                        (setf before `(,tv <- (& ,term)))
                        `(* ,tv))))
              1))
  (when before
    (return-from extract `(let ,before ,form))))
;; Now look for variables and make sure they are in the environment.
;; If not, add them, but don't remove them afterwards.  This lets
;; user refrain from explicitly adding variables although he will
;; still have to get rid of them manually.
(let ((index 1))
  (dolist (term (cdr form))
    (when (and (dynamic-var? term env)
               (candidate? index)
               (not (defined-in-env? term env)))
      (return-from extract
        `((add-to-env ,term)
          ,form)))
    (incf index)))
;; Next, get any mentioned variables into registers.
(let ((index 1))
  (dolist (term (cdr form))
    (when (and (dynamic-var? term env)
               (candidate? index)
               (not (in-register? term env)))
      (return-from extract
        `((get-in-register ,term)
          ,form)))
    (incf index)))
;; Finally, replace variables with the registers they're in.
(let (found?)
```

```
     (setf (cdr form)
          (repl2 (cdr form)
               #'(lambda (term index)
                   (when (and (dynamic-var? term env)
                            (candidate? index)
                            (in-register? term env))
                     (true! found?)
                     (location term env)))
               1))
     (if found? form))))              ;End function EXTRACT.
```

D.4.10 `clike.lisp`

Defines constructs for various user-level user-level C-like operators.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;----------------------------------------------------------------------
;;; C-like assignment statements.

(definfix (var ++)
  `(,var += +1))

;; "-" statement: negate the given lvalue in place.
(defconstruct - (var)
  (cond
   ((register? var)
    `(neg ,var))
   (t
    (extract form env :lvalues 1))))

(definfix (var <=< val)
  (cond
   ((and (numberp val) (zerop val))
    '())
   ((and (register? var) (static-val-addr? val env))
    `(rl ,var ,(second val)))
   ((and (register? var) (literal? val env))
    `(rl ,var ,val))
   ((and (register? var) (register? val))
    `(rlv ,var ,val))
   (t
    (extract form env
             :relevant-terms (if (not (literal? val env)) '(2))
             :lvalues 1))))

(definfix (var >=> val)
  (cond
   ((and (numberp val) (zerop val))
    '())
   ((and (register? var) (static-val-addr? val env))
    `(rr ,var ,(second val)))
   ((and (register? var) (literal? val env))
    `(rr ,var ,val))
   ((and (register? var) (register? val))
    `(rrv ,var ,val))
   (t
```

```
    (extract form env
             :relevant-terms (if (not (literal? val env)) '(2))
             :lvalues 1))))

(definfix (var += val) :opposite -=
  (cond
   ((and (numberp val) (zerop val))
    '()) ;Optimization: don't add 0
   ((and (register? var) (static-val-addr? val env))
    `(addi ,var ,(second val)))
   ((and (register? var) (literal? val env))
    `(addi ,var ,val))
   ((and (register? var) (register? val))
    `(add ,var ,val))
   (t
    (extract form env
             :relevant-terms (if (not (literal? val env)) '(2))
             :lvalues 1))))

(definfix (var -= val) :opposite +=
  (cond
   ((and (register? var) (numberp val))
    `(,var += ,(- val)))                ;No SUBI instruction.
   ((and (register? var) (static-array? val env))
    `(,var += ,(intern (concatenate 'string "-" (symbol-name val)))))
   ((and (register? var) (static-val-addr? val env))
    `(addi ,var ,(intern (concatenate 'string "-"
                                      (symbol-name (second val))))))
   ((and (register? var)
         (literal? val env)
         (negated-sym? val))
    `(,var += ,(positive-of val)))
   ((and (register? var) (register? val))
    `(sub ,var ,val))
   (t
    (extract form env
             :relevant-terms (if (not (literal? val env)) '(2))
             :lvalues 1))))

;; Very much like +=.  More abstraction?
(definfix (var ^= val)
  (cond
   ((and (numberp val) (zerop val))
    '()) ;Optimization: don't add 0
   ((and (register? var) (numberp val))
    `(xori ,var ,val))
   ((and (register? var) (register? val))
    `(xor ,var ,val))
   (t
    (extract form env
             :relevant-terms (if (not (numberp val)) '(2))
             :lvalues 1))))

(definfix (dest ^=& src1 src2)
  (cond
   ((and (register? dest) (register? src1) (register? src2))
    `(andx ,dest ,src1 ,src2))
   (t
    (extract form env :lvalues 1))))

(definfix (dest ^=<< src1 src2)
```

```
  (cond
   ((and (register? dest) (register? src1) (register? src2))
    `(sllvx ,dest ,src1 ,src2))
   (t
    (extract form env :lvalues 1))))

;; This is a logical shift right.
(definfix (dest ^=>> src1 src2)
  (cond
   ((and (register? dest) (register? src1) (register? src2))
    `(srlvx ,dest ,src1 ,src2))
   (t
    (extract form env :lvalues 1))))

;; (base _ offset) gets transformed into this, where dest is
;; some temporary register.
(definfix (dest <->_ base offset)
  ;; THIS IS WRONG!  Extracting before expanding runs the danger
  ;; that the variable assignments used during the extraction
  ;; could be invalidated during the body of the WITH, I think 6/3/97
  (or (extract form env :lvalues 1)
      `(with (,base += ,offset)
          (,dest <->* ,base))))

(definfix (dest <-_ base offset) :opposite ->_
  `(,dest <-* (,base + ,offset)))

(definfix (dest ->_ base offset) :opposite <-_
  `(,dest ->* (,base + ,offset)))

(definfix (dest <-* ptr)
  ;; THIS IS WRONG!  Extracting before expanding runs the danger
  ;; that the variable assignments used during the extraction
  ;; could be invalidated during the header of the LET.
  (or
   (extract form env :lvalues 1)
   (let ((tv (gentemp)))
     `(let (,tv <->* ,ptr)
        (,dest <- ,tv)))))

(definfix (dest ->* ptr)
  (or
   (extract form env :lvalues 1)
   (let ((tv (gentemp)))
     `(let (,tv <->* ,ptr)
        (,dest -> ,tv)))))

(definfix (left <->* rightptr)
  (cond
   ((extract form env :lvalues 1))
   ((and (register? left) (register? rightptr))
    `(exch ,left ,rightptr))))

(definfix (var +=* val1 val2) :opposite -=*
  `(call mult ,var ,val1 ,val2))

(definfix (var -=* val1 val2) :opposite +=*
  `(rcall mult ,var ,val1 ,val2))

(definfix (var +=*/ val1 val2) :opposite -=*/
  `(call _smf ,val1 ,val2 ,var)) ;Note dest is last.
```

```
(definfix (var -=*/ val1 val2) :opposite +=*/
  `(rcall _smf ,val1 ,val2 ,var))

(definfix (left <-> right)
  (cond
   ((and (location? left) (location? right))
    `(swaploc ,left ,right))
   ((and (listp right)
         (>= (length right) 3)
         (dbind (base ~ offset) right
           (if (eq ~ '_)
               `(,left <->_ ,base ,offset)))))
   ((and (listp left)
         (>= (length left) 3)
         (dbind (base ~ offset) left
           (if (eq ~ '_)
               `(,right <->_ ,base ,offset)))))
   ((extract form env :lvalues '(1 2)))))

(defun << (a b)
  (ash a b))
```

D.4.11 `print.lisp`

Defines a few very simple constructs for producing output.

```
;;; -*- Package: user -*-
(in-package "USER")

(defconstruct printword (val)
  `((rawprint 0)
    (rawprint ,val)))

(defconstruct println ()
  `((rawprint 1)))

(defconstruct rawprint (val)
  (cond
   ((register? val)
    `(output ,val))
   (t
    (extract form env))))
```

D.4.12 `controlflow.lisp`

Defines user-level to intermediate-level control flow constructs such as conditionals and looping.

```
;;; -*- Package: user -*-
(in-package "USER")

;;; ---------------------------------------------------------------------
;;; High-level control flow constructs suitable for user use.

;; if CONDEXPR then
```

```
;;    BODY...
;; [else BODY2...]
;;
;; CONDEXPR is evaluated; if result is nonzero body is executed.  In either
;; case, CONDEXPR is then evaluated in reverse.  The value should be the
;; same whether or not BODY was executed, or else behavior undefined.

;; NOTE 6/26/97: IF and all its subsidiary branching constructs need to be
;; completely cleaned up and reorganized.  One big thing is that code for
;; computing EXPR in a condition expression of a form like (EXPR > 0) needs
;; to be wrapped around the entire IF.  And things like (EXPR1 > EXPR2)
;; need to be transformed into ((EXPR1 - EXPR2) > 0).

(defconstruct if (condexpr then &body body)
  (cond
   ((member 'else body)
    (let ((ifpart (subseq body 0 (position 'else body)))
          (elsepart (subseq body (1+ (position 'else body)))))
      `(ifelse ,condexpr ,ifpart ,elsepart)))
   ((and (listp condexpr)
         (null (cdddr condexpr))
         (member (second condexpr) '(= != > <= < >=)))
    `(_if ,condexpr then . ,body))
   (t
    ;; The current version is appropriate only for testing an arbitrary
    ;; value to see if it is non-zero.  For other kinds of conditions,
    ;; other implementations would be more appropriate.
    (if (symbolp condexpr)
        `(_if (,condexpr != 0) then . ,body)
      (let ((tv (gentemp)))
        `(let (,tv <- ,condexpr)
           (_if (,tv != 0) then . ,body)))))))

;; for VAR = START to END
;;    BODY...
;;
;; Semantics: START and END are expressions.  They are each evaluated once
;; forwards at the beginning of the loop, and once in reverse at the end of
;; the loop.  They should return the same value both times.
;;
;; VAR is a fresh variable whose scope is the BODY.  It is set to START,
;; and then the BODY is executed.  If VAR is ever END after executing the
;; body, then the construct immediately terminates.  Otherwise, VAR is
;; incremented by 1 and the BODY is executed again.  START may be equal to
;; END, in which case the BODY is executed exactly once.  If the values of
;; START and END ever change during evaluation, or if BODY ever sets VAR to
;; START-1, the behavior of the entire program becomes undefined.

(defconstruct for (var = start wordto end &body body)
  (let ((top (gentemp "_FORTOP"))       ;Loop entry point.
        (bot (gentemp "_FORBOT"))       ;Bottom of loop.
        (stvar (gentemp "_FORSTART"))
        (endvar (gentemp "_FOREND")))   ;Loop boundary values.
    `(let (,stvar <- ,start)
          (let (,endvar <- (,end + 1))
               (scope ,var
                      (,var <- ,stvar)
                      ;; The loop itself.
                      (bcs-branch-pair ,top (,var != ,stvar)
                                       ,bot (,var != ,endvar)
                        ,@body
```

```
                      (,var ++))
                    (,var -> ,endvar))))))

;;;---------------------------------------------------------------------
;;; Medium-level control flow constructs not intended
;;; for direct user use.

;; InfLoop: Unconditional branch from bottom of body back to top.
;; Also, if we hit it from outside we jump over it.
(defconstruct infloop (&body body)
  `(twin-us-branch ,(gentemp "_LOOPTOP") ,(gentemp "_LOOPBOT")
                   . ,body))

(setf (get '=  'opposite) '!=
      (get '!= 'opposite) '=
      (get '>  'opposite) '<=
      (get '<= 'opposite) '>
      (get '<  'opposite) '>=
      (get '>= 'opposite) '<
      )

;;; _if (VAR ~ VAL) then BODY
(defconstruct _if ((reg1 ~ reg2) then &body body)
  (let ((l1 (gentemp "_IFTOP"))
        (l2 (gentemp "_IFBOT")))
    `(twin-bcs-branch (,reg1 ,(opposite ~) ,reg2) ,l1 ,l2
        ;; The body sure better not change whether var=0.
        ,@body)))

(defconstruct _ifelse ((reg1 ~ reg2) ifstuff elsestuff)
  (let ((iftop (gentemp "_IFTOP"))
        (ifbot (gentemp "_IFBOT"))
        (elsetop (gentemp "_ELSETOP"))
        (elsetop (gentemp "_ELSEBOT")))
    `((bcs-branch (,reg1 ,(opposite ~) ,reg2) ,iftop ,elsetop)
      (ensure-green . ,ifstuff)
      (sbra ,ifbot ,elsebot)
      (sbra ,elsetop ,iftop)
      (ensure-green . ,elsestuff)
      (bcs-branch (,reg1 ,~ ,reg2) ,elsebot ,ifbot))
    ))
```

D.4.13 subroutines.lisp

Provides high and low level support for subroutines.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;---------------------------------------------------------------------
;;; Subroutine calling support.
;;;---------------------------------------------------------------------

;;;---------------------------------------------------------------------
;;; User-level constructs.

;; Defsub: Implements subroutine entry/return conventions.
(defconstruct defsub (subname arglist &body body)
```

```
  (let ((bodyenv (entryenv arglist env)))
  ;; We wrap it in a branch pair so that if we encounter it from the
  ;; outside we jump over it, and if it runs off its end it comes back to
  ;; the beginning.  This latter behavior facilitates calling a subroutine
  ;; with a single switching-branch to its entry/exit point.
  `((twin-us-branch ,(gentemp "_SUBTOP") ,(gentemp "_SUBBOT")
      ;; At start and end of body, environment is as according
      ;; to subroutine calling convention.
      (declare-environment ,bodyenv)
      (portal ,subname) ;Entry/exit point.
      ,@body
      (environment ,bodyenv)))))

;;;----------------------------------------------------------------------
;;; Subroutine calling convention support.

;; NOTE: Currently this does not work right for more than 29 arguments
;; (i.e. when some args need to go on the top of the stack instead of
;; in registers!).
(defconstruct call (subname &rest actualargs)
  `(withargs ,actualargs
     (with-stack-top
        (gosub ,subname))))

(defconstruct rcall (subname &rest actualargs)
  `(withargs ,actualargs
     (with-stack-top
        (rgosub ,subname))))

(defconstruct with-stack-top (&body body)
  (let ((offset (top-of-stack env)))
    `(with-sp-adjustment ,offset
        . ,body)))

(defconstruct with-SP-adjustment (amt &body body)
  ;; AMT must be a literal number.
  `(((reg 1) += ,amt)
    ,@body
    ((reg 1) -= ,amt)))

;; Call a subroutine at a low level with no mention of arguments.
(defconstruct gosub (subname) :opposite rgosub
  ;; A switching branch to the SWAPBRN in the portal should do the trick.
  `(bra ,subname))

(defconstruct rgosub (subname) :opposite gosub
  `(rbra ,subname))

;; Portal: Entry/exit point of a subroutine.
(defconstruct portal (label)
  `((label ,label)
    ;; We always use register $2 for storing our subroutine offsets, by
    ;; convention.
    (swapbr (reg 2))  ;retvar<->BR
    (neg (reg 2))     ;retvar = -retvar
    ))

;;; WITHARGS below needs some work.  It currently can only prepare the 29
;;; arguments that we can fit into registers.  I was intending that if
;;; there are more arguments they should be passed on the stack.  This is
;;; not too hard, but I'm not sure it's worth it.
```

```
;; Prepare arguments in conventional locations as for a subroutine call.
(defconstruct withargs (actualargs &body body)
  (let ((result `((vacate (reg 2))
                  . ,body))
        (r 1))
    (dolist (a actualargs)
      (if (and (symbolp a) (not (static-id? a env)))
          (if (defined-in-env? a env)
              (push `(relocate ,a ,(argno-to-location r)) result)
            (setf result
                  `((new-var-at ,a ,(argno-to-location r))
                    ,@result
                    (remove-var ,a))))
        (setf result
              (let ((tv (gentemp)))
                `((new-var-at ,tv ,(argno-to-location r))
                  ;; This prevents evaluating A from causing
                  ;; earlier-placed registers to change their locations.
                  ;; 6/3/97- But ENSURE-GREEN really enforces more than
                  ;; just this, unfortunately.
                  (ensure-green
                   (,tv <- ,a))
                  ,@result
                  (,tv -> ,a)
                  ;;^-DANGER! Assumes subroutine didn't change value of A.
                  (remove-var ,tv)))))
      (incf r))
    result))

;; On subroutine entry/exit, the environment contains:
;; A return-address variable located in register 2.
;; All the arguments in registers 3,4,... until we run out,
;; and then stack locations -1,-2,... (below top of stack).
;; If not all the registers were used for arguments, then
;; there is a variable for each unused one (above 2), used
;; to ensure that all these other registers are restored to
;; their original state upon exit.
(defun entryenv (arglist origenv)
  (let (locmap (r 0))
    (dolist (a (cons '_RET arglist))
      (push
       `(,a . ,(argno-to-location r))
       locmap)
      (incf r))
    (setf r (+ r 2))
    (if (<= r 31)
        (loop
         (push `(,(intern (concatenate 'string "_R" (princ-to-string r)))
                 reg ,r) locmap)
         (incf r)
         (if (> r 31) (return))))
    (setf locmap (reverse locmap))
    (let ((env (copy-environment origenv)))
      (labels
          ((is-arg? (name) (member name arglist)))
        (setf (locmap env) locmap
              (staticvals env) (remove-if #'is-arg? (staticvals env))
              (staticarrays env) (remove-if #'is-arg? (staticarrays env))))
      env)))
```

```
;; Convert an argument number (0 and up) to a location (reg <regno>)
;; or (stack <offset>).  Argument 0 is the return address.
(defun argno-to-location (argno)
  (if (<= argno 29)
      `(reg ,(+ argno 2))
    `(stack ,(- 29 argno))))
```

D.4.14 `staticdata.lisp`

Defines constructs for defining static data objects. Currently this is the only way to provide input to a program.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;--------------------------------------------------------------------
;;; Constructs for declaring static data.

;; Define NAME to refer to a static word of data in memory
;; of value VALUE.
(defconstruct defword (name value)
  `(skip
    (staticval ,name)
    (label ,name)
    (dataword ,value)))

(defconstruct defarray (name &rest elements)
  `(skip
    (staticarray ,name)
    (label ,name)
    . ,(mapcar #'(lambda (elem) `(dataword ,elem)) elements)))

;;;--------------------------------------------------------------------

;; VALUE is a word of data that should be included at
;; this point in the program in literal form.
(defconstruct dataword (value)
  `(data ,value))

(defconstruct staticval (name)
  (values nil (add-staticval name env)))

(defconstruct staticarray (name)
  (values nil (add-staticarray name env)))

;; If the flow of control gets to code surrounded by SKIP it will skip over
;; the contents without executing them.
(defconstruct skip (&body body)
  ;; Implemented by an unconditional branch pair around the body.
  `(sbra-pair ,(gentemp "_PRESKIP") ,(gentemp "_POSTSKIP")
            . ,body))
```

D.4.15 `program.lisp`

Defines very high-level constructs for wrapping around the entire program.

```
;;; -*- Package: user -*-
(in-package "USER")

;;----------------------------------------------------------------------
;;; Highest-level constructs.

(defconstruct defmain (progname &body body)
  `(;; Always include the standard library of subroutines.
    (standard-library)
    ;; We surround the whole program with a branch pair because I don't
    ;; think that our current idea of START/FINISH boundary instructions
    ;; can be non-noops on the real machine without dissipation. This
    ;; also skips over main if control somehow comes down from above.
    (twin-us-branch _MAINTOP _MAINBOT
      ;; Execution starts and ends with exactly 0 dynamic variables.
      (with-location-map ,(empty-locmap)
        (declare-startpoint ,progname)
        ;; To begin execution, the PC should initally be set to this label.
        (label ,progname)
        (start)
        ,@body
        (finish)))))

;;----------------------------------------------------------------------

;; Defprog: a whole program with subroutines and a main routine.
;; Now deprecated in favor of defsub + defmain (6/26/97).
(defconstruct defprog (progname subs &body main)
  `(;; Always include the standard library of subroutines.
    (standard-library)
    ;; Include user subroutines.
    ,@subs
    ;; We surround the whole program with a branch pair because I don't
    ;; think that our current idea of START/FINISH boundary instructions
    ;; can be implemented without dissipation.
    (twin-us-branch _MAINTOP _MAINBOT
     ;; Execution starts and ends with exactly nothing in the environment.
     (with-location-map ,(empty-locmap)
       (declare-startpoint ,progname)
       ;; To begin execution, the PC should initally be set to this label.
       (label ,progname)
       (start)
       ,@main
       (finish)))))

;;----------------------------------------------------------------------

(defconstruct declare-startpoint (labname)
  `(.start ,labname))
```

D.4.16 library.lisp

Defines constructs that expand into code for standard subroutine libraries. Currently the library is very minimal.

```
;;; -*- Package: user -*-
(in-package "USER")

(defconstruct standard-library ()
```

```
  ;; List of standardly available subroutines.
  '((def-smf)))

;; Define the simplest normal integer multiplication function.
;; This adds the low word of the product of unsigned integers
;; M1 and M2 into PROD.

(defconstruct def-mult ()
  '(defsub mult (m1 m2 prod)
    ;; Use grade-school algorithm.
    (for pos = 0 to 31                  ; For each of the 32 bit-positions,
       (if (m1 & (1 << pos)) then       ;   if that bit of m1 is 1, then
         (prod += (m2 << pos))))))      ;     add m2, shifted over to that
                                        ;       position, into p.

;; Define the signed multiplication-by-fraction function, which takes
;; two signed integers M1 and M2, and adds the high word of their true
;; integer product into PROD.  This is like multiplying M1 by M2 when
;; M2 is considered to represent a fraction with numerator explicit and
;; denominator 2^31.
;;
;; This version of the function was tested in C and seemed to work
;; satisfactorily, although more testing is needed.  Need to compare
;; with upper bits of true (64-bit-wide) product. 6/26/97

(defconstruct def-smf ()
  '(defsub _smf (m1 m2 prod)
     (with-regvars (m1p m2p mask shifted bit p)
       (with ((mask <- 1)
              (m1p <- m1) (if (m1 < 0) then (- m1p))
              (m2p <- m2) (if (m2 < 0) then (- m2p)))
          (mask <=< 31)
          (for position = 1 to 31
             (mask >=> 1)
             (with (bit <- (m1p & mask))
                (if bit then
                  (with (shifted <- (m2p >> position))
                     (p += shifted)))))
          (if (m1 < 0) then (- p))
          (if (m2 < 0) then (- p))
          (prod += p)))))
```

D.4.17 `files.lisp`

Provides support for reading the source code to compile from a file.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;----------------------------------------------------------------------
;;; File compilation code.

(defun rcompile-file (filename &key debug)
  (let (source)
    (with-open-file (stream filename)
       (loop
        (let ((next-form
               (read stream nil :eof)))
```

```
          (cond
           ((eq next-form :eof)
            (setf source (reverse source))
            (return))
           (t
            (push next-form source))))))
    (format t "~&Source:~%")
    (myprint source)
    (if debug
        (rcomp-debug source)
      (rc source))))
```

D.4.18 test.lisp

Miscellaneous functions and program fragments for exercising the compiler. Some of these may be obsolete.

```
;;; -*- Package: user -*-
(in-package "USER")
;;;-----------------------------------------------------------------------
;;; Testing code.

(defun test-sch ()
  (rcompile-file "sch.r"))

(defun test-sch-debug ()
  (rcompile-file "sch.r" :debug t))

(defparameter *test*
  '(defprog example1
     ()
     (let p = (3 * 5))))

(defparameter *test2*
  '(defprog example-program-2 ()
     (call mult 3 5 p)))

(defun test ()
  (rc *test*))

;; My original very simple MULT routine.
(defparameter *mult-orig*
  '(defsub mult (m1 m2 prod)
      ;; Use grade-school algorithm.
      (for pos = 0 to 31              ; For each of the 32 bit-positions,
         (if (m1 & (1 << pos)) then   ;   if that bit of m1 is 1, then
            (prod += (m2 << pos)))))  ;     add m2, shifted over to that
                                      ;       position, into p.

;; I'd like this code to compile to produce exactly my hand-compiled-
;; and -optimized version of the MULT routine.  Currently, it won't though.
(defparameter *mult-opt*
  '(defsub mult-opt (m1 m2 product)
     (with-register-vars ((limit = 32) (mask = 1) shifted bit position)
        (for position = 0 until limit
           (with (bit <- (m1 & mask))
              (if bit then
                 (with (shifted <- (m2 << position))
                    (product += shifted))))
```

```
           (mask <=< 1))))
  )

;; Hand-compiled, optimized multiply routine.  We could actually
;; include this in programs if we want.
(defparameter *mult-hand*
  '(alloctop
    (bra   allocbot)
    alloc4
    (swbrn $2)     ; This sub-subroutine frees
    (addi  $1  +1) ; 4 registers for use in the
    (exch  $31 $1) ; MULT subroutine.  It leaves
    (addi  $1  +1) ; the stack pointer pushed
    (exch  $30 $1) ; above, but we don't mind.
    (addi  $1  +1)
    (exch  $29 $1)
    (addi  $1  +1)
    (exch  $28 $1)
    allocbot
    (bra   alloctop)
    ;; This subroutine's arguments are in registers $3, $4, and $5.
    subtop
    (bra   subbot)              ; MULT top.
    mult
    (swbrn $2)                  ; Subroutine entry/exit point.
    (exch  $2 $1)               ; Push return address.
    (bra   alloc4)              ; Allocate 4 registers ($28-$31).
    (addi  $31 32)              ; limit <- 32
    (addi  $2 1)                ; mask <- 1
    fortop
    (bne   $30 $0 forbot)       ; unless (position != 0) do
    (andx  $28 $3 $2)           ;     bit <- m1&mask
    iftop
    (beq   $28 $0 ifbot)        ;     if (bit != 0) then
    (sllvx $29 $4 $30)          ;        shifted <- m2<<position
    (add   $5 $29)              ;        product += shifted
    (sllvx $29 $4 $30)          ;        shifted -> m2<<position
    ifbot
    (beq   $28 $0 iftop)        ;     end if
    (andx  $28 $3 $2)           ;     bit -> m1&mask
    (rl    $2 1)                ;     mask <=< 1 (rotate left by 1)
    (addi  $30 +1)              ;     i++
    forbot
    (bne   $30 $31 fortop)      ; and repeat while (position != limit).
    (sub   $30 $31)             ; position -> limit
    (addi  $2 -1)               ; mask -> 1
    (addi  $31 -32)             ; limit -> 32
    (rbra  alloc4)              ; Deallocate 4 registers ($28-$31).
    (exch  $2 $1)               ; Pop return address.
    subbot
    (bra   subtop)              ; MULT bottom.
    ))

(defparameter *mult-frac*
  ;; Like mult, but interprets the multiplier (1st arg) to be a
  ;; number between 1 and -1, on a scale where 2^31 = 1, -2^31 = -1.
  '(defsub mult-frac (m1 m2 prod)
      (for pos = 0 to 31
         (if (m1 & (1 << pos)) then
            (prod += (m2 >> (31 - pos)))))))
```

```
;;; Interesting question: does MULT-FRAC yield the same result
;;; independently of the order of m1 and m2?  I know the original
;;; MULT did.  I think it does.

;; This optimized version the same number of instructions as mult-opt.
;; Put this in the standard library?
(defparameter *mult-frac-opt*
  '(defsub mult-frac-opt (m1 m2 product)
     (with-registers ((limit = 32) (mask = 1) shifted bit position)
        (for position = 0 until limit
           (mask >=> 1)
           (with (bit <- (m1 & mask))
              (if bit then
                 (with (shifted <- (m2 >> position))
                    (product += shifted))))
           )))
  )

;;; Now, what to do if integers are signed?
;;; What to do: for actual multiplication, only look at bits
;;; 30-0 of multiplier.  Branch on bit 31.  If 1, then
;;; subtract shifted multiplicand from product instead of
;;; adding it.

(defparameter *signed-mult-frac-opt*
  '(defsub signed-mult-frac-opt (m1 m2 product)
     (with-registers ((limit = 32) mask shifted bit position)
        (mask <- (1 << 31))
        (for position = 1 until limit
           (mask >=> 1)
           (with (bit = (m1 & mask))
              (if bit then
                 (with (shifted = (m2 >> position))
                    (if (m2 > 0)
                        (product += shifted)
                      else
                        (product -= shifted))))))
        (mask -> 1))))


;; This will be the first version of SIGNED-MULT-FRAC to actually
;; be compilable. (NOT. -6/26)
(defparameter *smf-first*
  '(defsub smf-first (m1 m2 prod)
     (with-regvars (mask shifted bit)
       (with (mask <- 1)
          (mask <=< 31)
          (for position = 1 to 31
             (mask >=> 1)               ;Rotate right by 1 bit
             (with (bit <- (m1 & mask))
                (if bit then
                  (with (shifted <- (m2 >> position))
                      (if (m2 > 0)
                          (prod += shifted)
                        else
                          (prod -= shifted))))))))))

;; OK, now THIS version is the new target. 6/26
(defparameter *smf-new*
  '(defsub smf-new (m1 m2 prod)
     (with-regvars (m1p m2p mask shifted bit p)
```

```
       (with ((mask <- 1)
              (m1p <- m1) (if (m1 < 0) then (- m1p))
              (m2p <- m2) (if (m2 < 0) then (- m2p)))
          (mask <=< 31)
          (for position = 1 to 31
             (mask >=> 1)               ;Rotate right by 1 bit
             (with (bit <- (m1 & mask))
                (if bit then
                  (with (shifted <- (m2 >> position))
                     (p += shifted)))))
          (if (m1 < 0) then (- p))
          (if (m2 < 0) then (- p))
          (prod += p)))))

;;; Stuff for Schroedinger program.
;; Point function.
(defparameter *pfunc*
  '(defsub pfunc (dest src i alphas epsilon)
      ((dest _ i) += (((alphas _ i) << 1) */ (src _ i)))
      ((dest _ i) -= (epsilon */ (src _ ((i + 1) & 127))))
      ((dest _ i) -= (epsilon */ (src _ ((i - 1) & 127)))))
  )

;; Wavefunction step.
(defparameter *schstep*
  '(defsub schstep (psiR psiI alphas epsilon)
      (for i = 0 until 128              ;For each point i,
         (call pfunc psiR psiI i))      ; psiR[i] += func(psiI,i)
      (for i = 0 until 128              ;For each point i,
         (rcall pfunc psiI psiR i)))    ; psiI[i] -= func(psiR,i)
  )

;; Data for the schroedinger simulation:
;; Epsilon, alphas, and psis.
(defparameter *schdata*
  '((array epsilon 203667001)
    (array alphas
           458243442 456664951 455111319 453582544 452078627 450599569
           449145369 447716027 446311542 444931917 443577149 442247239
           440942188 439661994 438406659 437176182 435970563 434789802
           433633899 432502854 431396668 430315339 429258869 428227257
           427220503 426238607 425281569 424349389 423442068 422559605
           421701999 420869252 420061363 419278332 418520159 417786845
           417078388 416394790 415736049 415102167 414493143 413908977
           413349669 412815220 412305628 411820895 411361019 410926002
           410515843 410130542 409770099 409434515 409123788 408837920
           408576909 408340757 408129463 407943027 407781450 407644730
           407532868 407445865 407383720 407346432 407334003 407346432
           407383720 407445865 407532868 407644730 407781450 407943027
           408129463 408340757 408576909 408837920 409123788 409434515
           409770099 410130542 410515843 410926002 411361019 411820895
           412305628 412815220 413349669 413908977 414493143 415102167
           415736049 416394790 417078388 417786845 418520159 419278332
           420061363 420869252 421701999 422559605 423442068 424349389
           425281569 426238607 427220503 428227257 429258869 430315339
           431396668 432502854 433633899 434789802 435970563 437176182
           438406659 439661994 440942188 442247239 443577149 444931917
           446311542 447716027 449145369 450599569 452078627 453582544
           455111319 456664951)
    ;; This is the shape of the initial wavefunction.
    (array psis
```

```
           2072809 3044772 4418237 6333469 8968770 12546502 17338479
           23669980 31921503 42527251 55969298 72766411 93456735 118573819
           148615999 184009768 225068513 271948808 324607187 382760978
           445857149 513053161 583213481 654924586 726530060 796185813
           861933650 921789572 973841548 1016350163 1047844835 1067208183
           1073741824 1067208183 1047844835 1016350163 973841548 921789572
           861933650 796185813 726530060 654924586 583213481 513053161
           445857149 382760978 324607187 271948808 225068513 184009768
           148615999 118573819 93456735 72766411 55969298 42527251 31921503
           23669980 17338479 12546502 8968770 6333469 4418237 3044772
           2072809 1393998 926112 607804 394060 252382 159681 99804 61622
           37586 22647 13480 7926 4604 2642 1497 838 463 253 136 73 38 20
           10 5 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
           0 0 0 0 0 0 0 0 0))
  )


(defparameter *sch*
  '((DEFWORD EPSILON 203667001)
    (DEFARRAY ALPHAS 458243442 456664951 455111319 453582544
      452078627 450599569 449145369 447716027 446311542 444931917
      443577149 442247239 440942188 439661994 438406659 437176182
      435970563 434789802 433633899 432502854 431396668 430315339
      429258869 428227257 427220503 426238607 425281569 424349389
      423442068 422559605 421701999 420869252 420061363 419278332
      418520159 417786845 417078388 416394790 415736049 415102167
      414493143 413908977 413349669 412815220 412305628 411820895
      411361019 410926002 410515843 410130542 409770099 409434515
      409123788 408837920 408576909 408340757 408129463 407943027
      407781450 407644730 407532868 407445865 407383720 407346432
      407334003 407346432 407383720 407445865 407532868 407644730
      407781450 407943027 408129463 408340757 408576909 408837920
      409123788 409434515 409770099 410130542 410515843 410926002
      411361019 411820895 412305628 412815220 413349669 413908977
      414493143 415102167 415736049 416394790 417078388 417786845
      418520159 419278332 420061363 420869252 421701999 422559605
      423442068 424349389 425281569 426238607 427220503 428227257
      429258869 430315339 431396668 432502854 433633899 434789802
      435970563 437176182 438406659 439661994 440942188 442247239
      443577149 444931917 446311542 447716027 449145369 450599569
      452078627 453582544 455111319 456664951)
    (DEFARRAY PSIR 2072809 3044772 4418237 6333469 8968770 12546502
      17338479 23669980 31921503 42527251 55969298 72766411 93456735
      118573819 148615999 184009768 225068513 271948808 324607187
      382760978 445857149 513053161 583213481 654924586 726530060
      796185813 861933650 921789572 973841548 1016350163 1047844835
      1067208183 1073741824 1067208183 1047844835 1016350163
      973841548 921789572 861933650 796185813 726530060 654924586
      583213481 513053161 445857149 382760978 324607187 271948808
      225068513 184009768 148615999 118573819 93456735 72766411
      55969298 42527251 31921503 23669980 17338479 12546502 8968770
      6333469 4418237 3044772 2072809 1393998 926112 607804 394060
      252382 159681 99804 61622 37586 22647 13480 7926 4604 2642
      1497 838 463 253 136 73 38 20 10 5 2 1 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    (DEFARRAY PSII 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0)
    (DEFSUB PFUNC (DEST SRC I ALPHAS EPSILON)
```

```
      ((DEST _ I) += ((ALPHAS _ I) */ (SRC _ I)))
      ((DEST _ I) -= (EPSILON */ (SRC _ ((I + 1) & 127))))
      ((DEST _ I) -= (EPSILON */ (SRC _ ((I - 1) & 127)))))
    (DEFSUB SCHSTEP (PSIR PSII ALPHAS EPSILON)
      (FOR I = 0 TO 127 (CALL PFUNC PSIR PSII I))
      (FOR I = 0 TO 127 (RCALL PFUNC PSII PSIR I)))
    (DEFSUB PRINTWAVE (WAVE)
      (FOR I = 0 TO 127 (PRINTWORD (WAVE _ I))) (PRINTLN))
    (DEFMAIN SCHROED
      (FOR I = 1 TO 50
           (FOR J = 1 TO 20 (CALL SCHSTEP PSIR PSII ALPHAS EPSILON))
           (CALL PRINTWAVE PSIR) (CALL PRINTWAVE PSII)))))

(defparameter *sch-frag1*
  '((DEFWORD EPSILON 203667001)
    (DEFARRAY ALPHAS 458243442 456664951)
    (DEFARRAY PSIR 2072809 3044772)
    (DEFARRAY PSII 0 0)
    (DEFSUB PFUNC (DEST SRC I ALPHAS EPSILON)
      ((DEST _ I) += ((ALPHAS _ I) */ (SRC _ I)))
      ((DEST _ I) -= (EPSILON */ (SRC _ ((I + 1) & 127))))
      ((DEST _ I) -= (EPSILON */ (SRC _ ((I - 1) & 127)))))
    (DEFSUB SCHSTEP (PSIR PSII ALPHAS EPSILON)
      (FOR I = 0 TO 127 (CALL PFUNC PSIR PSII I))
      (FOR I = 0 TO 127 (RCALL PFUNC PSII PSIR I)))
    (DEFSUB PRINTWAVE (WAVE)
      (FOR I = 0 TO 127 (PRINTWORD (WAVE _ I))) (PRINTLN))
    (DEFMAIN SCHROED
      (FOR I = 1 TO 50
           (FOR J = 1 TO 20 (CALL SCHSTEP PSIR PSII ALPHAS EPSILON))
           (CALL PRINTWAVE PSIR) (CALL PRINTWAVE PSII)))))

(defparameter *sch-frag2*
  '((DEFWORD EPSILON 203667001)
    (DEFARRAY ALPHAS 458243442)
    (DEFARRAY PSIR 2072809)
    (DEFARRAY PSII 0)
    (CALL SCHSTEP PSIR PSII ALPHAS EPSILON)))

(defparameter *sch-frag3*
  '((DEFWORD EPSILON 203667001)
    (DEFARRAY ALPHAS 458243442)
    (CALL SCHSTEP ALPHAS EPSILON)))
```

Appendix E

Reversible Schrödinger wave simulation

This appendix gives the complete code for SCH, the reversible simulator of the Schrödinger wave equation that was mentioned in §9.5.6 (p. 241), and in §C.5 (p. 309). We give the C, R, and PISA versions of the program.

E.1 Derivation of discrete update rule

Here we derive a naive approximate method for simulating Schrödinger's equation in discrete, reversible fashion. Later we will derive some alternatives.

A bit of history. Most of the following is original work, except that the final key idea, for making the simulation exactly reversible, is something that I learned about from Margolus in personal discussions. This trick apparently originated with Fredkin and Barton in 1975, in the context of their own work (in collaboration with Richard Feynman) on a discrete reversible Schrödinger equation update rule. (I was not aware of this work when I reinvented part of it in the early versions of my own simulation.) The story of the serendipitous discovery of this trick is told in Fredkin 1999 [75], which also describes their version of the discrete rule in some detail.

Now, let us begin our derivation. We start with the full general form of the wave equation when expressed in a state space in which the eigenstates correspond to particle positions. It is possible to describe a system's state space in many other ways, but this way seems most straightforward and natural to those not immersed in quantum physics. It also lends itself to visualization of the wave function on a graphics display.

Here is the Schrödinger equation in its full glory:

$$-\frac{\hbar^2}{2}\sum_{j=0}^{N-1}\frac{1}{m_j}\frac{\partial^2}{\partial x_j^2}\Psi(\vec{x},t)+\mathcal{V}(\vec{x},t)\Psi(\vec{x},t)=i\hbar\frac{\partial}{\partial t}\Psi(\vec{x},t).$$

It requires some explanation for the general reader. $\hbar$ is Planck's constant over 2π, $1.055\times10^{-34}\mathrm{J}\cdot\mathrm{s}$. N is the number of positional degrees of freedom, $3n$ for n particles in 3 dimensions. m_j is the mass associated with the jth degree of freedom, *e.g.* in 3 dimensions the mass of particle $\lfloor j/3\rfloor$. $\vec{x}$ is a vector of all particle position coordinates, and x_j is particular position coordinate. t is time. $\mathcal{V}$ is a potential energy function which is a function of the positions of all particles and optionally (if representing a time-dependent potential) the time. Ψ is the wave function itself, a function of the positions of all particles and of time; its value is generally a complex number with both real and imaginary parts, $\Psi=\Re\Psi+i\Im\Psi$. The imaginary unit $i=\sqrt{-1}$ appears on the right-hand side of the equation. The magnitude of $\Psi(\vec{x},t)$, that is, $\Re^2\Psi+\Im^2\Psi$, is, when normalized, thought of as the probability density of finding the system in state $\vec{x}$ at time t.

The important point to note, from the perspective of a dynamical simulation, is that the equation describes how the wave function evolves over time, via the partial derivative with respect to time that appears on the right side of the equation.

Now that we've seen the full form of the equation, let's proceed to strip it down to a slightly simpler form. First, we'll get rid of the summation over the $\vec{x}$ vector; since we are restricting ourselves for the moment to one particle in one dimension, there is only one coordinate x, and only one mass m.

$$-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\Psi(x,t)+\mathcal{V}(x,t)\Psi(x,t)=i\hbar\frac{\partial}{\partial t}\Psi(x,t).$$

Next, rather than writing the (x,t) arguments to Ψ and $\mathcal{V}$ repeatedly, let them be understood.

$$-\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2}+\mathcal{V}\Psi=i\hbar\frac{\partial\Psi}{\partial t}.$$

Next, we rewrite partial derivative operators $\partial/\partial v$ as d_v/dv where d_v is a differential operator meaning "the infinitesimal change in the given quantity when variable v changes by an infinitesimal amount dv and everything else is held constant".

$$-\frac{\hbar^2}{2m}\frac{d_x^2\Psi}{dx^2}+\mathcal{V}\Psi=i\hbar\frac{d_t\Psi}{dt}.$$

This notational change will allow us to extricate the differentials in the numerator and denominator of a partial derivative operator from each other.

Now, we solve for $d_t\Psi$, the t subscript reminding us that this is the $d\Psi$ that is associated with dt in the original expression $\partial\Psi/\partial t$.

$$d_t\Psi = \frac{dt}{i\hbar}\left(-\frac{\hbar^2}{2m}\frac{d_x^2\Psi}{dx^2} + \mathcal{V}\Psi\right). \tag{E.1}$$

This equation gives us the infinitesimal change in $\Psi(x,t)$ at a point x as time increases by an infinitesimal amount dt and the point x is held constant. This is the most direct statement of how Ψ evolves over time.

Now we come to our first approximation. So far we have treated time and space as continuous: dt means an infinitesimally small amount of time, and dx an infinitesimally small distance in space. However, in our simulation we will *discretize* time and space into points with a minimum, non-infinitesimal distance between them, so that we only have to represent the state of the wave function at a finite number of points, and so that when we advance to the "next" time, there will be a non-infinitesimal change in the wave function state. We indicate this conceptual change by replacing d with Δ, Δx meaning the distance between our discrete points, Δt the distance between our discrete times, and $\Delta_t\Psi$ the change in $\Psi(x)$ in a time Δt.

$$\Delta_t\Psi \approx \frac{\Delta t}{i\hbar}\left(-\frac{\hbar^2}{2m}\frac{\Delta_x^2\Psi}{\Delta x^2} + \mathcal{V}\Psi\right). \tag{E.2}$$

The earlier equation with the differentials (equation E.1) expresses the fact that the two sides of this equation with the deltas approach equality in the limit, as Δx and Δt approach zero. But hereafter in our derivation, we will treat the two sides as being equal even though the deltas are not zero; that is our approximation.

We believe we can prove that this approximation is correct to second order, i.e., that as long as the real rate of change of Ψ is small during the entire interval Δt, and if the minimum wavelength in the represented wave is significantly longer than Δx, then the difference between our computed $\Delta\Psi$ and the actual $\Delta\Psi$ will improve quadratically, proportionately to the square of $\Delta t/\Delta x^2$, as this ratio is decreased.

Now, before we can make further progress, we need to decide how to evaluate the expression $\Delta_x^2\Psi$ at particular points x. In general, by $\Delta_v q$ we mean the change in quantity q, a function of v, when v changes by Δv. However, how are we to define $\Delta_v q(v_k)$ for a particular point v_k? One symmetrical answer is that it is simply $q(v_k + \Delta v/2) - q(v_k - \Delta v/2)$.

Applying this to our particular case, we find (omitting still the ever-present t

argument):

$$\begin{aligned}
\Delta_x^2\Psi(x) &= \Delta_x\Psi(x+\frac{\Delta x}{2}) - \Delta_x\Psi(x-\frac{\Delta x}{2}) \\
&= (\Psi(x+\Delta x)-\Psi(x)) - (\Psi(x)-\Psi(x-\Delta x)) \\
&= \Psi(x+\Delta x)+\Psi(x-\Delta x)-2\Psi(x) \\
&= 2\left(\frac{\Psi(x+\Delta x)+\Psi(x-\Delta x)}{2}-\Psi(x)\right).
\end{aligned}$$

And plugging this into the equation E.2 (considered as an equality) we get:

$$\begin{aligned}
&\Delta_t\Psi(x) = \\
&\quad \frac{\Delta t}{i\hbar}\left(\frac{-\hbar^2}{\Delta x^2 m}\left(\frac{\Psi(x+\Delta x)+\Psi(x-\Delta x)}{2}-\Psi(x)\right)+\mathcal{V}(x)\Psi(x)\right).
\end{aligned}$$

However, rather than sprinkling things like $x+\Delta x$ throughout our equations from here on, let us take it as given that Ψ is always evaluated at points that are separated by integer multiples of Δx, and so we can treat Ψ as a vector with elements indexed by integers k. So hereafter we will replace $\Psi(x)$ by Ψ_k, $\Psi(x+\Delta x)$ by Ψ_{k+1}, *etc.*, and similarly for $\mathcal{V}(x)$ as well.

$$\Delta_t\Psi_k = \frac{\Delta t}{i\hbar}\left(-\frac{\hbar^2}{\Delta x^2 m}\left(\frac{\Psi_{k+1}+\Psi_{k-1}}{2}-\Psi_k\right)+\mathcal{V}_k\Psi_k\right).$$

Now, let's play around with this expression algebraically, and collect together the terms involving Ψ_k:

$$\begin{aligned}
\Delta_t\Psi_k &= -\frac{\Delta t\hbar^2}{i\hbar\Delta x^2 m}\left(\frac{\Psi_{k+1}+\Psi_{k-1}}{2}-\Psi_k\right)+\frac{\Delta t\mathcal{V}_k}{i\hbar}\Psi_k \\
&= -\frac{\Delta t\hbar}{i\Delta x^2 m}\left(\frac{\Psi_{k+1}+\Psi_{k-1}}{2}\right)+\frac{\Delta t\hbar}{i\Delta x^2 m}\Psi_k+\frac{\Delta t\mathcal{V}_k}{i\hbar}\Psi_k \\
&= -\frac{\Delta t\hbar}{2i\Delta x^2 m}(\Psi_{k+1}+\Psi_{k-1})+\frac{\Delta t}{i}\left(\frac{\hbar}{\Delta x^2 m}+\frac{\mathcal{V}_k}{\hbar}\right)\Psi_k \\
&= \frac{i\Delta t\hbar}{\Delta x^2 m}(\Psi_{k+1}+\Psi_{k-1})-i\Delta t\left(\frac{\hbar}{\Delta x^2 m}+\frac{\mathcal{V}_k}{\hbar}\right)\Psi_k.
\end{aligned}$$

Next, to make the expression more concise, we introduce the following new quantities to use as abbreviations:

$$\epsilon = \frac{\hbar\Delta t}{m\Delta x^2},\ \omega_k = \frac{\mathcal{V}_k}{\hbar},\ \alpha_k = \epsilon + \omega_k\Delta t,$$

and rewrite our formula in terms of them:

$$\Delta_t \Psi_k = \frac{i\epsilon}{2}(\Psi_{k+1} + \Psi_{k-1}) - i\alpha_k \Psi_k.$$

But now, this is all just giving us $\Delta_t\Psi$—the change in Ψ over a time Δt. How exactly will this let us calculate Ψ at some future time given Ψ at the current time?

Well, using our earlier definition for Δ, we can expand the left hand side of the equation as follows (reintroducing t as an explicit argument):

$$\Psi_k(t + \frac{\Delta t}{2}) - \Psi_k(t - \frac{\Delta t}{2}) = \frac{i\epsilon}{2}(\Psi_{k+1}(t) + \Psi_{k-1}(t)) - i\alpha_k \Psi_k(t).$$

If we solve this for $\Psi_k(t + \Delta t/2)$, we get

$$\Psi_k(t + \frac{\Delta t}{2}) = \Psi_k(t - \frac{\Delta t}{2}) + \frac{i\epsilon}{2}(\Psi_{k+1}(t) + \Psi_{k-1}(t)) - i\alpha_k \Psi_k(t),$$

and if we now replace Δt everywhere with $2\Delta t$ (a change which does not matter since Δt was an arbitrarily chosen value already), including within the definitions of ϵ and α_k, we get

$$\Psi_k(t + \Delta t) = \Psi_k(t - \Delta t) + i\epsilon(\Psi_{k+1}(t) + \Psi_{k-1}(t)) - 2i\alpha_k \Psi_k(t).$$

Now for conciseness we replace $(t + \Delta t)$ as an argument with $s + 1$ as a subscript, similarly to what we did earlier when we replaced $(x + \Delta x)$ with $k + 1$, and obtain

$$\Psi_{k,s+1} = \Psi_{k,s-1} + i\left(\epsilon(\Psi_{k+1,s} + \Psi_{k-1,s}) - 2\alpha_k \Psi_{k,s}\right).$$

Well, there are a couple of important things to note about this equation. First rather than deriving the state of the wave at the next time step $s + 1$ from the state at step s, the equation requires using the states at the two previous steps s and $s - 1$. It's a second order difference equation, and it's of the form that Fredkin has shown to always be totally reversible, given a numeric representation that supports a reversible addition operation.

The second important point is that although the equation is complex, the real part of Ψ_{s+1} depends only on the real part of Ψ_{s-1} and the imaginary part of Ψ_s. Specifically:

$$\Re\Psi_{k,s+1} = \Re\Psi_{k,s-1} + \Im\left(2\alpha_k \Psi_{k,s} - \epsilon(\Psi_{k+1,s} + \Psi_{k-1,s})\right) \tag{E.3}$$

and similarly, the imaginary part of Ψ_{s+1} depends only on the imaginary part of Ψ_{s-1} and the real part of Ψ_s:

$$\Im\Psi_{k,s+1} = \Im\Psi_{k,s-1} - \Re\left(2\alpha_k \Psi_{k,s} - \epsilon(\Psi_{k+1,s} + \Psi_{k-1,s})\right) \tag{E.4}$$

Therefore, if we consider the real components of Ψ at all the even-numbered steps $s = 2n$ (for n an integer), and the imaginary components of Ψ at all odd-numbered steps $s = 2n + 1$, we see that the evolution of those components is totally self-contained, and we can ignore the real components at odd times, and the imaginary components at even times, and thus work with only a single real number at each time step. Let us do so, and define, for integers n, the real quantities

$$\begin{aligned} \mathcal{X}_{k,n} &\equiv \Re\Psi_{k,2n} \\ \mathcal{Y}_{k,n} &\equiv \Im\Psi_{k,2n+1}. \end{aligned}$$

Then we can rewrite the equations (E.3 and E.4) above as two equations:

$$\begin{aligned} \mathcal{X}_{k,n+1} &= \mathcal{X}_{k,n} + f_k(\vec{\mathcal{Y}}_n) \\ \mathcal{Y}_{k,n+1} &= \mathcal{Y}_{k,n} - f_k(\vec{\mathcal{X}}_{n+1}), \end{aligned} \tag{E.5}$$

where (for $\mathcal{Q}$ being either $\mathcal{X}$ or $\mathcal{Y}$) by $\vec{\mathcal{Q}}_n$ we mean the vector of all values $\mathcal{Q}_{k,n}$, and where (now omitting the n subscript):

$$f_k(\vec{\mathcal{Q}}) = 2\alpha_k \mathcal{Q}_k - \epsilon(\mathcal{Q}_{k+1} + \mathcal{Q}_{k-1}).$$

The update rule (E.5) lends itself to a particularly simple pseudo-code implementation given reversible addition/subtraction instructions, as are the C language's `+=` and `-=` operators when performed on integers:

$$\begin{aligned} \vec{\mathcal{X}} &\mathrel{+}= \lfloor f(\vec{\mathcal{Y}}) \rfloor \\ \vec{\mathcal{Y}} &\mathrel{-}= \lfloor f(\vec{\mathcal{X}}) \rfloor, \end{aligned}$$

where the absence of the k subscript is intended to suggest application of each operation to every element of the given vector. The n is implicit, and is no longer needed; each value of $\vec{\mathcal{X}}$ that is computed will implicitly represent the value of $\vec{\mathcal{X}}$ two time steps beyond the previous value that was computed, and each value of $\vec{\mathcal{Y}}$ that is computed will implicitly represent the value of $\vec{\mathcal{Y}}$ one time step beyond the previous value of $\vec{\mathcal{X}}$, and two time steps beyond the previously computed $\vec{\mathcal{Y}}$.

This process of updating $\vec{\mathcal{X}}$ and $\vec{\mathcal{Y}}$, can then be exactly undone by:

$$\begin{aligned} \vec{\mathcal{Y}} &\mathrel{+}= \lfloor f(\vec{\mathcal{X}}) \rfloor \\ \vec{\mathcal{X}} &\mathrel{-}= \lfloor f(\vec{\mathcal{Y}}) \rfloor. \end{aligned}$$

So the above analysis gives us a method of reversibly updating two arrays, representing the real and imaginary parts of Ψ at successive times. Note that this does not tell us both components of Ψ at either particular time, but if Ψ is changing gradually, as it will be if Δt is sufficiently small, then the two components taken together will be a fairly accurate representation of Ψ's complex value at either of the times.

We note that if there is actually only one spatial point $k = 0$, so that the "neighboring" points $k+1$ and $k-1$, are actually, in modulus 1 arithmetic, the same point as k itself, then the above algorithm actually reduces to (letting $X = \mathcal{X}_0, Y = \mathcal{Y}_0$):

$$\begin{aligned} X \quad &+= \quad \lfloor 2\omega_0 \Delta t Y \rfloor \\ Y \quad &-= \quad \lfloor 2\omega_0 \Delta t X \rfloor, \end{aligned}$$

which is an approximate circle-drawing algorithm, with the X's and Y's giving the coordinates of points on (or near) a circle. It is essentially the same algorithm as that described by Margolus in [115], §2.8.2, pp. 82–84, with our energy-based quantity $2\omega_0 \Delta t = 2\mathcal{V}_0 \Delta t/\hbar$ representing a quantity which can be thought of as being approximately the amount of change in the (X, Y) vector per $2\Delta t$ time, as a fraction of its length. This quantity takes the place of the $2\sin(\omega)$ quantity which plays the same role in Margolus's equation, determining there the angular change (in radians) of the point (x_t, y_t) across a span of 2 time steps. Note that if $\Delta t = 1$ and $\omega \approx 0$, then $2\omega\Delta t \approx 2\sin(\omega)$.

Improved algorithms. The above algorithm has the flaw that if Δt is too large, so that ϵ and all α_ks do not all remain small numbers, then the resulting evolution will be far from the sort of unitary, norm-preserving operation that we would like to have.

The dynamical system that Margolus originally described did not suffer from this problem, since his was based on an iteration $z_{t+1} = e^{i\omega} z_t$ that was exact even if ω was large. We would really prefer to have a way of stepping through the Schrödinger equation that benefited from a similar property.

I have produced other, slightly more sophisticated versions of the above algorithm, which attempt to do a better job of preserving unitarity, at least, even when Δt is large.

Actually I think no algorithm based on discrete spatial simulation will work in the case where the potential energy function that is imposed on the system leads to a particle acquiring a momentum that corresponds to a wavelength that is small compared to the separation between points. However, as long as the separation between points is much smaller than the shortest wavelength that ever appears, it should be possible to construct a simulation that is reasonably accurate even when fairly large amounts of time are jumped over in a single step.

I could at this point go on and discuss in much more detail all my different variations of this simulation, and their pros and cons, but at this point it seems to be a low priority. For now, suffice it to say that I have a 100 percent reversible technique for simulating the evolution of the Schrödinger wave function in this simple system, it is accurate as a first order approximation (although I have not here taken the space to formalize and prove that assertion), and empirical demonstrations (on a normal computer) verify that the simulation behaves quite well in a variety of simple test cases involving different initial position distributions, velocities, and potential energy functions. Phenomena such as tunneling and interference have been observed. Total probability is nearly conserved. And total reversibility has been experimentally validated.

For Fredkin and Barton's update rule, which is essentially the same as ours, Richard Feynman apparently discovered that there *is* is a definition of total probability that is *exactly* conserved by the update rule [75]. In the context of our discussion, the corresponding definition of the exactly conserved probability over time steps n would be: $P_n = \vec{\mathcal{X}}_n^2 - \vec{\mathcal{Y}}_{n-1} \cdot \vec{\mathcal{Y}}_{n+1}$. We just recently learned of Feynman's invariant, and we have not yet checked our own update rule to make sure that it works.

We now show how to port the above algorithm to a reversible processor. Updating the state need involve only integer addition and multiplication. (Pendulum does not currently support a built-in multiplication operation, so I had to implement multiplication as a subroutine, which was easy to write in my high-level language.)

I believe this program, as it stands, constitutes an interesting demonstration of a significant reversible program in our reversible language, and also demonstrates the ability of a totally reversible program written in our language to simulate physics without incurring an asymptotically increasing need for storage.

E.2 Reversible C implementation

I have several different versions of my C program for simulating the Schrödinger equation. The following version, `schii.c`, is an exactly-reversible version that uses only integer arithmetic, and thus was the basis for the version of the program to run on Pendulum, since we have not created any floating-point support for Pendulum as of yet.

Large parts of this program are simply concerned with drawing the graphics display using the X window system, and are therefore uninteresting from the point of view of the simulation technique itself. We have tried to isolate most of the graphics code into a section at the bottom of the program.

The user interface to the program is currently very minimal. Any key press prints out current statistics about the wave function. Any mouse button press exits the program. To change any parameters of the simulation, one must edit the appropriate constant and recompile the program. Fortunately, the program is short enough so that this does not take very long.

The key functions in the program are: `signed_mult_frac()`, which is the integer multiplication routine for integers taken as representing fractions between -1 and 1, `function()`, which computes the appropriate function at a given point that gives the amount by which a component of the wave vector should be changed at that point, and `step_forwards()` and `step_back()`, which perform a state update in the given time direction. These functions contain the core functionality which we ported to R and Pendulum assembly.

```
/* SCHII: Like SCHI2 except uses only integer multiplication in the main loop.
   SCHI2 kept its state in integers but calculating amounts to add
   using floating-point math.

   This will be the model for my Pendulum implementation.
   */

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

/* Physical constants. We'll use MKS (m,kg,s,nt,J,coul...) units. */
#define planck_h (6.626e-34) /* Planck's constant, in Joule-seconds */
#define light_c (2.998e8) /* Speed of light in meters/second */
static const double
  hbar = planck_h/(2*M_PI),/* h/2*pi, also in J-sec */
  elec_m = 9.109e-31,/* Electron rest mass, in kg */
  elec_q = 1.602e-19,/* Electron charge (abs.value) in Coulombs */
  coul_const = 8.988e9; /* 1/4*pi*epsilon_0 (Coulomb's law constant)
   in nt-m^2/coul^2. */

/* Parameters of simulation.                                       o  */
#define space_width (1e-10) /* Width of sim space in meters: 1 A. */
#define num_pts (128) /* Number of discrete space points in sim. */
static const double
  sim_dx = space_width/num_pts, /* delta btw. pts, in meters. */
  sim_dt = 5e-22,/* Simulated time per step, in secs. */
  init_vel = light_c*0.0,/* Initial velocity in m/s */
  initial_mu = -space_width/4,/* initial mean electron pos, rel. to ctr. */
  initial_sigma = space_width/20; /* width of initial hump. */

/* For holding some arrays of size num_pts, in real/imaginary pairs. */
static double
   *energies,/* Real potential energies at points. */
   *on_real,*on_imag,
   *off_real,*off_imag,/* On/off diagonal matrix elements, real/imag */
   *Psi_real,*Psi_imag; /* Real at t, imag at t+1/2. */

/* Now, we will use Psi_real and Psi_imag only for translation between
   the internal, integer form, and how it is used externally.  Here is
   the real wave function: */

static int *psiR,*psiI; /* Real and imaginary integer wave function. */
static double scaleFactor; /* The value, in the integer range, that a real
   value of 1.0 translates to. */

static int n_steps = 0; /* Number of iterations done so far. */
static double total_t = 0; /* Total simulated time so far. */
```

```
static int max_steps = 10000; /* go a million iterations before reversing */
static int direction = 0; /* forwards */

#define STEPS_PER_SHOT 20

typedef enum energ_funcs {
  neg_gaus=0, pos_gaus, inv_cutoff, parabolic, const_nonzero, const_zero,
  step_barrier
} energ_func_id;

static energ_func_id which_potential = parabolic;

static const char *energ_func_strs[] = {
  "Negative Gaussian potential well",
  "Positive Gaussian potential bump",
  "Inverse distance well with cutoff",
  "Parabolic well for harmonic oscillator",
  "Constant, non-zero energy level",
  "Constant, zero energy",
  "A step barrier to tunnel through"
};

void print_sim_params() {
  printf("\n");
  printf("SCHROEDINGER SIMULATOR PARAMETERS\n");
  printf("---------------------------------\n");
  printf("\n");
  printf("Width of simulated space is %g meters (%g light-seconds).\n",
 space_width, space_width/light_c);
  printf("Simulating %d discrete points in space.\n", num_pts);
  printf("Distance between points: %g m (%g ls).\n",sim_dx,
 sim_dx/light_c);
  printf("Time per simulation step: %g secs (light dist: %g m)\n",
 sim_dt, sim_dt*light_c);
  printf("Initial electron position: mu=%g m, sigma=%g m.\n",
 initial_mu, initial_sigma);
  printf("Using potential energy function %d: %s.\n", which_potential,
 energ_func_strs[which_potential]);
  printf("Initial electron velocity = %g m/s (%g c).\n",
 init_vel, init_vel/light_c);
  printf("Number of steps to go before reversing: %d.\n",max_steps);
  printf("\n");
}

static double *init_real,*init_imag;

void print_stats() {
  /* Calculate and print some stats of the wavefunction. */
  int this = n_steps&1;
  int i;
  double total_p = 0;
  double mom_real = 0, mom_imag = 0,
    potential = 0,
    kin_real = 0, kin_imag = 0,
    energ_real = 0, energ_imag = 0;
  double dPsi2_real, dPsi2_imag;
  double diff=0;

  for(i=0;i<num_pts;i++){
    int next = (i+1)%num_pts,
```

```
      prev = (i-1+num_pts)%num_pts;
    double real = Psi_real[i],
      imag = Psi_imag[i];
    double pd = real*real+imag*imag;
    total_p += pd;

    dPsi2_real = Psi_real[next] - Psi_real[prev];
    dPsi2_imag = Psi_imag[next] - Psi_imag[prev];
    mom_real += real*dPsi2_imag - imag*dPsi2_real;
    mom_imag -= real*dPsi2_real + imag*dPsi2_imag;

    potential += pd*energies[i];
  }
  mom_real *= hbar/(2*sim_dx);
  mom_imag *= hbar/(2*sim_dx);

  for(i=0;i<num_pts;i++){
    double dr = Psi_real[i] - init_real[i];
    double di = Psi_imag[i] - init_imag[i];
    diff += dr*dr + di*di;
  }
  diff /= num_pts;

  printf("Cycle = %d, t = %g. Total P = %g.\n",n_steps,total_t,total_p);
  printf("   Mean squared diff. from init. state = %g. (RMS = %g)\n",
 diff, sqrt(diff));
  printf("   Momentum = (%g + i %g) kg m/s.\n",mom_real,mom_imag);
  printf("   Velocity = (%g + i %g) m/s.\n",
 mom_real/elec_m,mom_imag/elec_m);
  printf("            = (%g + i %g) c.\n",
 mom_real/(elec_m*light_c),mom_imag/(elec_m*light_c));
  printf("  Potential = %g J (%g eV)\n", potential, potential/1.602e-19);
}

/* Gaussian (normal) distribution, non-normalized. */
double normal(double x,double mu,double sigma)
{
  double d = (x-mu)/sigma;
  return exp(-0.5*d*d);
}

double *malloc_doubles(){
  return (double *)calloc(num_pts,sizeof(double));
}

int *malloc_ints(){
  return (int *)calloc(num_pts,sizeof(int));
}

/* x should now be a real position in space */
double energy(double x){
  switch(which_potential){
  case neg_gaus:
    /* Negative Gaussian potential well. */
    return - 3e-15 * normal(x,0,space_width/10);
  case pos_gaus:
    /* Positive Gaussian potential bump. */
    return 5e-15 * normal(x,0,space_width/10);
  case inv_cutoff:
    /* Well where energy drops with inverse distance from center, down
       to a cutoff threshold. */
```

```
    {
      double ax = (x<0?-x:x);
      double p;
      p = - (coul_const*(elec_q*elec_q)/ax);
      if (p > -4e-14) {
return p;
      } else {
return -4e-14;
      }
    }
  case parabolic:
    /* Parabolic well for harmonic oscillator. */
    return x*x*1e+6;
  case const_nonzero:
    /* Constant, nonzero energy.  Apparent wave rotation rate differs
       from zero-energy case.*/
    return -24e-15;
  case const_zero:
    /* Constant, zero energy. */
    return 0;
  case step_barrier:
    /* A step barrier through which to tunnel. */
    if (x < space_width * 0.15) {
      return 0;
    } else if (x < space_width * 0.18) {
      return 1e-15;
    } else {
      return 0;
    }
  }
  return 0;
}

static double epsilon;
static double *alphas;
static int *alphasi;
static int epsiloni;

void cache_energies() {
  int i;
  /* epsilon: an angle that roughly indicates how much of a
     point's amplitude gets spread to its neighboring points per time
     step.  A function of sim dt and dx parameters only. */
  epsilon = hbar*sim_dt/(elec_m*sim_dx*sim_dx);
  epsiloni = (int)((unsigned)(0x80000000) * epsilon);
  printf("Sim epsilon angle: %g radians (%g of a circle).\n",
 epsilon, epsilon/(2*M_PI));
  printf("Integer epsilon: %d\n",epsiloni);
  energies = malloc_doubles();
  on_real = malloc_doubles();
  on_imag = malloc_doubles();
  off_real = malloc_doubles();
  off_imag = malloc_doubles();
  alphas = malloc_doubles();
  alphasi = malloc_ints();
  printf("Integer alphas:\n");
  for(i=0;i<num_pts;i++){
    double x = (i - num_pts/2.0)*sim_dx; /* Position in space. */
    double alpha;
    energies[i] = energy(x);
    /* alpha: at this particular point, what's the absolute phase rotation
```

```
       angle for on-diagonal.  Energy makes a contribution. */
    alpha = epsilon + energies[i]*sim_dt/hbar;
    /* A = exp(i*Atheta) */
    on_real[i] = cos(epsilon) * cos(-alpha);
    on_imag[i] = cos(epsilon) * sin(-alpha);
    /* What's the phase rotation for off-diag (neighbors). */
    off_real[i] = sin(epsilon) * cos(M_PI/2 - alpha);
    off_imag[i] = sin(epsilon) * sin(M_PI/2 - alpha);

    alphas[i] = alpha;
    alphasi[i] = (int)((unsigned)(0x80000000)*alphas[i]*2);
    printf("%d ",alphasi[i]);
  }
  printf("\n");
}

void init_wave() {
  double *probs = malloc_doubles();
  int i;
  double tprob;
  double lambda;
  Psi_real = malloc_doubles();
  Psi_imag = malloc_doubles();
  psiR = malloc_ints();
  psiI = malloc_ints();
  init_real = malloc_doubles();
  init_imag = malloc_doubles();
  tprob = 0;
  for(i=0;i<num_pts;i++){
    double x = (i - num_pts/2.0)*sim_dx;
    /* Unnormalized initial probability of finding electron here. */
    double p = normal(x,initial_mu,initial_sigma);
    probs[i] = p;
    tprob += p;
  }
  for(i=0;i<num_pts;i++){
    probs[i] /= tprob;
  }
  lambda = planck_h/(elec_m*init_vel);
  printf("Initial de Broglie wavelength is %g m (%g ls).\n",
 lambda,lambda/light_c);
  for(i=0;i<num_pts;i++){
    double x = (i - num_pts/2.0)*sim_dx;
    Psi_real[i] = sqrt(probs[i]) * cos(x/lambda*2*M_PI);
    Psi_imag[i] = sqrt(probs[i]) * sin(x/lambda*2*M_PI);
  }
  {
    double maxval = 0;
    double absval;
    for(i=0;i<num_pts;i++){
      absval = Psi_real[i];
      absval = (absval<0)?-absval:absval;
      if (absval>maxval) maxval=absval;
      absval = Psi_imag[i];
      absval = (absval<0)?-absval:absval;
      if (absval>maxval) maxval=absval;
    }
    printf("The maximum absolute value initially is: %g.\n",maxval);
    /* We'll scale our integers so that they can hold values up to
       almost twice the largest initial value before they incur
       an overflow. */
```

```
    scaleFactor = (1<<30)/maxval;
    printf("Therefore the scale factor will be: %g.\n",scaleFactor);
  }
  /* Convert to integers. */
  printf("Integer psis:\n");
  for(i=0;i<num_pts;i++){
    psiR[i] = Psi_real[i]*scaleFactor;
    psiI[i] = Psi_imag[i]*scaleFactor;
    printf("%d+%di ",psiR[i],psiI[i]);
  }
  printf("\n");
  /* Convert back to doubles for convenience. */
  for(i=0;i<num_pts;i++){
    Psi_real[i] = init_real[i] = psiR[i]/scaleFactor;
    Psi_imag[i] = init_imag[i] = psiI[i]/scaleFactor;
  }
}

void sim_init () {
  print_sim_params();
  cache_energies();
  init_wave();
  print_stats();
}

int signed_mult_frac(int m1,int m2)
{
  int pos,prod=0;
  unsigned int mask = 1<<31;
  int m1p=m1,m2p=m2;
  if (m1<0) m1p = -m1p;
  if (m2<0) m2p = -m2p;
  for(pos=1;pos<32;pos++){
    mask >>= 1;
    if (m1p&mask)
      prod += m2p>>pos;
  }
  if (m1<0) prod = -prod;
  if (m2<0) prod = -prod;
  return prod;
}

double function(int *vec,int i){
  int j,k;
  j = i+1; if (j==-1) j=num_pts-1; else if (j==num_pts) j=0;
  k = i-1; if (k==-1) k=num_pts-1; else if (k==num_pts) k=0;
  return
    signed_mult_frac(alphasi[i],vec[i])
    - signed_mult_frac(epsiloni,vec[j])
    - signed_mult_frac(epsiloni,vec[k]);
}

void step_forwards() {
  int i;
  for(i=0;i<num_pts;i++)
    psiR[i] += (int)(function(psiI,i));
  for(i=0;i<num_pts;i++)
    psiI[i] -= (int)(function(psiR,i));
  for(i=0;i<num_pts;i++){
    Psi_real[i] = psiR[i]/scaleFactor;
    Psi_imag[i] = psiI[i]/scaleFactor;
```

```
  }
  n_steps++;
  total_t+=sim_dt;
}

void step_back() {
  int i;
  n_steps--;
  for(i=0;i<num_pts;i++)
    psiI[i] += (int)(function(psiR,i));
  for(i=0;i<num_pts;i++)
    psiR[i] -= (int)(function(psiI,i));
  for(i=0;i<num_pts;i++){
    Psi_real[i] = psiR[i]/scaleFactor;
    Psi_imag[i] = psiI[i]/scaleFactor;
  }
  total_t-=sim_dt;
}

void sim_step() {
  if (direction == 0) {
    step_forwards();
  } else {
    step_back();
  }
  if (direction == 1 && n_steps == 0) {
    printf("\nPresumably back to initial state.\n");
    print_stats();
    printf("Now turning and going forwards.\n");
    direction = 0;
  } else if (direction == 0 && n_steps == max_steps) {
    printf("\nCompleted %d steps.\n", max_steps);
    print_stats();
    direction = 1;
    printf("Now turning around and going backwards.\n");
  }
}

/*--------------------------------------------------------------------*/
/* Graphics. */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include "icon.bitmap"
#define BITMAPDEPTH 1
static Display *display;
static int screen;

static double *prev_real, *prev_imag;

static double maxv;
static double maxp;

void init_graphics() {
  int i;
  int this = n_steps&1;
  prev_real = malloc_doubles();
  prev_imag = malloc_doubles();
  maxv = 0;
  maxp = 0;
```

```
  for(i=0;i<num_pts;i++){
    double absv;
    absv = Psi_real[i];
    absv = (absv<0)?-absv:absv;
    if (absv > maxv) maxv = absv;
    absv = Psi_imag[i];
    absv = (absv<0)?-absv:absv;
    if (absv > maxv) maxv = absv;
    absv = Psi_real[i]*Psi_real[i]
      + Psi_imag[i]*Psi_imag[i];
    if (absv > maxp) maxp = absv;
  }
}

void draw_graphics(win,gc,window_width,window_height,gce,gcreal,gcimag)
     Window win;
     GC gc;
     unsigned window_width, window_height;
     GC gce,gcreal,gcimag;
{
  double ght = window_height/2; /* How much Y space per graph */
  unsigned c1 = ght/2; /* origin y for top graph */
  unsigned c2 = window_height; /* origin y for bottom graph */
  int this = n_steps&1; /* which Psi is current */
  int i;

  for(i=0;i<num_pts;i++){
    unsigned x = i*window_width/num_pts;
    double prevReal = prev_real[i],
      prevImag = prev_imag[i],
      thisReal = Psi_real[i],
      thisImag = Psi_imag[i];
    double prevProb = prevReal*prevReal + prevImag*prevImag,
      thisProb = thisReal*thisReal + thisImag*thisImag;
    int prY = (int)((prevReal/maxv)*ght*0.5);
    int piY = (int)((prevImag/maxv)*ght*0.5);
    int ppY = (int)((prevProb/maxp)*ght);
    int trY = (int)((thisReal/maxv)*ght*0.5);
    int tiY = (int)((thisImag/maxv)*ght*0.5);
    int tpY = (int)((thisProb/maxp)*ght);

    /* Similar to above calculation, but for point next to us on the right.*/
    unsigned j = (i+1)%num_pts;
    unsigned xj = (i+1)*window_width/num_pts;
    double RprevReal = prev_real[j],
      RprevImag = prev_imag[j],
      RthisReal = Psi_real[j],
      RthisImag = Psi_imag[j];
    double RprevProb = RprevReal*RprevReal + RprevImag*RprevImag,
      RthisProb = RthisReal*RthisReal + RthisImag*RthisImag;
    int RprY = (int)((RprevReal/maxv)*ght*0.5);
    int RpiY = (int)((RprevImag/maxv)*ght*0.5);
    int RppY = (int)((RprevProb/maxp)*ght);
    int RtrY = (int)((RthisReal/maxv)*ght*0.5);
    int RtiY = (int)((RthisImag/maxv)*ght*0.5);
    int RtpY = (int)((RthisProb/maxp)*ght);

    /* Erase old Psi at this pos. */
    XDrawLine(display,win,gce,x,c1-prY,xj,c1-RprY);
    XDrawLine(display,win,gce,x,c1-piY,xj,c1-RpiY);
    /* Draw new Psi. */
```

```
    XDrawLine(display,win,gcreal,x,c1-trY,xj,c1-RtrY);
    XDrawLine(display,win,gcimag,x,c1-tiY,xj,c1-RtiY);
    /* Erase old prob and draw new. */
    XDrawLine(display,win,gce,x,c2-ppY,xj,c2-RppY);
    XDrawLine(display,win,gc,x,c2-tpY,xj,c2-RtpY);
    /* Draw energy function. */
    XDrawPoint(display,win,gc,x,(int)(c2-(ght*0.5)-ght*energies[i]*1e14));
  }
  for(i=0;i<num_pts;i++){
    prev_real[i] = Psi_real[i];
    prev_imag[i] = Psi_imag[i];
  }
  XFlush(display);
}

/*----------------------------------------------------------------------*/
/* Pretty much everything below here is uninteresting X interfacing
   stuff. */

get_GC(Window win, GC *gc, XFontStruct *font_info, int foo) {
  unsigned long valuemask = 0; /* ignore XGCvalues and use defaults */
  XGCValues values;
  unsigned int line_width = 1;
  int line_style = LineSolid;
  int cap_style = CapButt;
  int join_style = JoinRound;
  int dash_offset = 0;
  static char dash_list[] = {
    12, 24 };
  int list_length = 2;

  /* Create default graphics context */
  *gc = XCreateGC(display,win,valuemask,&values);

  /* specify font */
  XSetFont(display,*gc,font_info->fid);

  {
    XColor sdr,edr;
  if (foo == 1) {
    XSetForeground(display,*gc,WhitePixel(display,screen));
  } else if (foo == 0) {
    XSetForeground(display,*gc,BlackPixel(display,screen));
  } else if (foo == 2) {
    XAllocNamedColor(display,DefaultColormap(display,screen),"cyan",
&sdr,&edr);
    XSetForeground(display,*gc,edr.pixel);
  } else if (foo == 3) {
    XAllocNamedColor(display,DefaultColormap(display,screen),"yellow",
&sdr,&edr);
    XSetForeground(display,*gc,edr.pixel);
  }}

  /* set line attributes */
  XSetLineAttributes(display,*gc,line_width,line_style,cap_style,
     join_style);

  /* set dashes to be line_width in length */
  XSetDashes(display,*gc,dash_offset,dash_list,list_length);
}
```

```
load_font(XFontStruct **font_info) {
  char *fontname = "9x15";

  /* Access font */
  if ((*font_info = XLoadQueryFont(display,fontname)) == NULL) {
    fprintf(stderr,"Basic: Cannot open 9x15 font\n");
    exit(-1);
  }
}

int main(argc,argv)
     int argc;
     char **argv;
{
  Window win;
  unsigned width, height; /* window size */
  int x = 0, y = 0; /* window position */
  unsigned border_width = 4; /* border four pixels wide */
  unsigned display_width, display_height;
  char *window_name = "Schroedinger Wave Simulator";
  char *icon_name = "schroed";
  Pixmap icon_pixmap;
  XSizeHints size_hints;
  XEvent report;
  GC gc,gce,gcreal,gcimag;
  XFontStruct *font_info;
  char *display_name = NULL;
  int i;

  /* connect to X server */
  if ( (display=XOpenDisplay(display_name)) == NULL ) {
    fprintf(stderr,
    "cannot connect to X server %s\n",
    XDisplayName(display_name));
    exit(-1);
  }

  sim_init();
  init_graphics();

  /* get screen size from display structure macro */
  screen = DefaultScreen(display);

  display_width = DisplayWidth(display,screen);
  display_height = DisplayHeight(display,screen);

  /* size window with enough room for text */
  width = display_width/3, height = display_height/3;

  /* create opaque window */
  win = XCreateSimpleWindow(display,RootWindow(display,screen),
    x,y,width,height,border_width,
    WhitePixel(display,screen),
    BlackPixel(display,screen));

  /* Create pixmap of depth 1 (bitmap) for icon */
  icon_pixmap = XCreateBitmapFromData(display, win, icon_bitmap_bits,
     icon_bitmap_width,
     icon_bitmap_height);

  /* initialize size hint property for window manager */
```

```
  size_hints.flags = PPosition | PSize | PMinSize;
  size_hints.x = x;
  size_hints.y = y;
  size_hints.width = width;
  size_hints.height = height;
  size_hints.min_width = 175;
  size_hints.min_height = 125;

  /* set properties for window manager (always before mapping) */
  XSetStandardProperties(display,win,window_name,icon_name,
 icon_pixmap,argv,argc,&size_hints);

  /* Select event types wanted */
  XSelectInput(display,win, ExposureMask | KeyPressMask |
       ButtonPressMask | StructureNotifyMask);

  load_font(&font_info);

  /* create GC for text and drawing */
  get_GC(win, &gc, font_info, 1);
  get_GC(win, &gce, font_info, 0);
  get_GC(win, &gcreal, font_info, 2);
  get_GC(win, &gcimag, font_info, 3);

  /* Display window */
  XMapWindow(display,win);

  while (1) { /* Event loop. */
    int i;
    XNextEvent(display,&report);
    switch(report.type) {
    case Expose:
      /* get rid of all other Expose events on the queue */
      while (XCheckTypedEvent(display, Expose, &report));
      draw_graphics(win, gc, width, height, gce, gcreal, gcimag);
      for(i=0;i<STEPS_PER_SHOT;i++)
sim_step();
      /*print_stats();
sleep(1);*/
      XClearArea(display,win,0,0,1,1,1);
      break;
    case ConfigureNotify:
      width = report.xconfigure.width;
      height = report.xconfigure.height;
      XClearArea(display,win,0,0,width,height,1);
      break;
    case KeyPress:
      print_stats();
      break;
    case ButtonPress:
      XUnloadFont(display,font_info->fid);
      XFreeGC(display,gc);
      XFreeGC(display,gce);
      XFreeGC(display,gcreal);
      XFreeGC(display,gcimag);
      XCloseDisplay(display);
      exit(1);
    default:
      break;
    }
  }
```

```
  return 0;
}
```

E.3 Source code in R language

The following is the complete source code for the Schrödinger simulation as ported into the R programming language, except for the multiplication routine, which the compiler provides as a standard library function. See the `def-smf` construct in §D.4.16 (p. 368) for the R source for the multiplication subroutine.

Note that the initial wavefunction state is provided in the form of a static data array, so that we do not have to port the trigonometric functions that were used to generate the initial state in the C version of the program. Also note that we did not bother to port the graphics code. Output is instead provided in a raw form which is parsed, displayed, and compared with the original C program's output by a separate program which is wrapped around the Pendulum simulator.

```
;;;----------------------------------------------------------------------
;;;
;;;  Schroedinger Wave Equation simulation program.
;;;  The first major test of R (the reversible language)!
;;;
;;;  Current status: More compiler work needed. 6/12/97.
;;;
;;;----------------------------------------------------------------------

;;; Currently all data must come before the code that uses it, so that the
;;; compiler will recognize these identifiers as names of static data items
;;; rather than as dynamic variables.

;; epsilon = hbar*dt/m*dx^2.  DX=7.8125e-13m, DT=5e-22s
(defword epsilon 203667001) ; 0.0948398 radians.
;; Parabolic potential well with 128 points.
(defarray alphas
  458243442 456664951 455111319 453582544 452078627 450599569 449145369
  447716027 446311542 444931917 443577149 442247239 440942188 439661994
  438406659 437176182 435970563 434789802 433633899 432502854 431396668
  430315339 429258869 428227257 427220503 426238607 425281569 424349389
  423442068 422559605 421701999 420869252 420061363 419278332 418520159
  417786845 417078388 416394790 415736049 415102167 414493143 413908977
  413349669 412815220 412305628 411820895 411361019 410926002 410515843
  410130542 409770099 409434515 409123788 408837920 408576909 408340757
  408129463 407943027 407781450 407644730 407532868 407445865 407383720
  407346432 407334003 407346432 407383720 407445865 407532868 407644730
  407781450 407943027 408129463 408340757 408576909 408837920 409123788
  409434515 409770099 410130542 410515843 410926002 411361019 411820895
  412305628 412815220 413349669 413908977 414493143 415102167 415736049
  416394790 417078388 417786845 418520159 419278332 420061363 420869252
  421701999 422559605 423442068 424349389 425281569 426238607 427220503
  428227257 429258869 430315339 431396668 432502854 433633899 434789802
  435970563 437176182 438406659 439661994 440942188 442247239 443577149
```

```
  444931917 446311542 447716027 449145369 450599569 452078627 453582544
  455111319 456664951)

;; This is the shape of the initial wavefunction; amplitude doesn't matter.
;; Real part.
(defarray psiR 2072809 3044772 4418237 6333469 8968770 12546502 17338479
  23669980 31921503 42527251 55969298 72766411 93456735 118573819 148615999
  184009768 225068513 271948808 324607187 382760978 445857149 513053161
  583213481 654924586 726530060 796185813 861933650 921789572 973841548
  1016350163 1047844835 1067208183 1073741824 1067208183 1047844835
  1016350163 973841548 921789572 861933650 796185813 726530060 654924586
  583213481 513053161 445857149 382760978 324607187 271948808 225068513
  184009768 148615999 118573819 93456735 72766411 55969298 42527251
  31921503 23669980 17338479 12546502 8968770 6333469 4418237 3044772
  2072809 1393998 926112 607804 394060 252382 159681 99804 61622 37586
  22647 13480 7926 4604 2642 1497 838 463 253 136 73 38 20 10 5 2 1 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
;;Imaginary part.
(defarray psiI 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)

;; This subroutine updates one of the two waves, using the other.
(defsub halfstep (dest src)
  (let (e <- epsilon)
    (for i = 0 to 127
       (let (d <-> (dest _ i))
          (d += ((alphas _ i) */ (src _ i)))
          (d -= (e */ (src _ ((i + 1) & 127))))
          (d -= (e */ (src _ ((i - 1) & 127))))))))

;; Print the current wave to the output stream.
(defsub printwave (wave)
  (for i = 0 to 127
       (printword (wave _ i)))
  (println))

;; Main program, goes by the name of SCHROED.
(defmain schroed
  (for i = 1 to 1000 ;Enough time for electron to fall to well bottom.
     ;; Take turns updating the two components of the wave.
     (call halfstep psiR psiI)
     (rcall halfstep psiI psiR)
     ;; Print both wave components.
     (call printwave psiR)
     (call printwave psiI)))
```

E.4 Compiled PISA code

The following is the exact PISA assembly code output that was produced from the above input file by the R compiler Rcomp we listed in ch. D. It consists of 830 machine words (395 of data, 435 of program). We will not review this code in detail here. However, when it was executed under our Pendulum virtual machine PendVM (a C program written by Matt DeBergalis), it was found to produce exactly identical output, on every step, to that of the original C version of the program, listed earlier, thus validating the correctness of Rcomp and PendVM (at least for this program). And when execution was stopped and reversed at any point, the processor state returned exactly to the orignal state at the start of the program (validating PendVM's guarantee of reversibility).

```
;; pendulum pal file
_PRESKIP395:    BRA _POSTSKIP396
EPSILON:        DATA 203667001
_POSTSKIP396:   BRA _PRESKIP395
_PRESKIP397:    BRA _POSTSKIP398
ALPHAS:         DATA 458243442
                DATA 456664951
                DATA 455111319

                ⋮  (122 intervening data statements elided)

                DATA 453582544
                DATA 455111319
                DATA 456664951
_POSTSKIP398:   BRA _PRESKIP397
_PRESKIP399:    BRA _POSTSKIP400
PSIR:           DATA 2072809
                DATA 3044772
                DATA 4418237

                ⋮  (122 intervening data statements elided)

                DATA 0
                DATA 0
                DATA 0
_POSTSKIP400:   BRA _PRESKIP399
_PRESKIP401:    BRA _POSTSKIP402
PSII:           DATA 0
                DATA 0
                DATA 0

                ⋮  (122 intervening data statements elided)

                DATA 0
                DATA 0
                DATA 0
_POSTSKIP402:   BRA _PRESKIP401
_SUBTOP403:     BRA _SUBBOT404
HALFSTEP:       SWAPBR $2
                NEG $2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
                ADDI $31 EPSILON
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                EXCH $30 $31
                ADDI $1 -3
```

```
                EXCH $29 $1
                ADDI $1 3
                ADD $29 $30
                EXCH $30 $31
                ADDI $31 -EPSILON
                ADDI $30 128
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                ADD $31 $28
_FORTOP407:     BNE $31 $28 _FORBOT408
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADD $3 $31
                EXCH $27 $3
                SUB $3 $31
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $26 ALPHAS
                ADD $26 $31
                ADDI $1 -7
                EXCH $25 $1
                ADDI $1 7
                ADDI $1 -8
                EXCH $24 $1
                ADDI $1 8
                EXCH $24 $26
                ADD $25 $24
                EXCH $24 $26
                SUB $26 $31
                ADDI $26 -ALPHAS
                ADD $24 $4
                ADD $24 $31
                ADDI $1 -9
                EXCH $23 $1
                ADDI $1 9
                EXCH $23 $24
                ADD $26 $23
                EXCH $23 $24
                SUB $24 $31
                SUB $24 $4
                XOR $23 $5
                XOR $5 $23
                XOR $26 $4
                XOR $4 $26
                XOR $26 $4
                XOR $25 $3
                XOR $3 $25
                XOR $25 $3
                XOR $24 $2
                XOR $2 $24
                ADDI $1 -9
                BRA SMF
                ADDI $1 9
                ADD $27 $5
                ADDI $1 -9
                RBRA SMF
                ADDI $1 9
                ADD $2 $26
```

```
ADD $2 $31
ADDI $1 -10
EXCH $22 $1
ADDI $1 10
EXCH $22 $2
SUB $4 $22
EXCH $22 $2
SUB $2 $31
SUB $2 $26
ADDI $2 ALPHAS
ADD $2 $31
EXCH $4 $2
SUB $3 $4
EXCH $4 $2
SUB $2 $31
ADDI $2 -ALPHAS
ADD $2 $31
ADDI $2 1
ADDI $3 127
ANDX $4 $2 $3
ADDI $3 -127
ADD $3 $26
ADD $3 $4
EXCH $22 $3
ADD $5 $22
EXCH $22 $3
SUB $3 $4
SUB $3 $26
XOR $3 $5
XOR $5 $3
XOR $3 $4
XOR $4 $3
XOR $3 $4
XOR $29 $3
XOR $3 $29
XOR $29 $3
XOR $22 $2
XOR $2 $22
ADDI $1 -10
BRA SMF
ADDI $1 10
SUB $27 $5
ADDI $1 -10
RBRA SMF
ADDI $1 10
ADD $2 $26
ADD $2 $29
ADDI $1 -11
EXCH $21 $1
ADDI $1 11
EXCH $21 $2
SUB $4 $21
EXCH $21 $2
SUB $2 $29
SUB $2 $26
ADDI $2 127
ANDX $29 $22 $2
ADDI $2 -127
ADDI $22 -1
SUB $22 $31
ADD $2 $31
```

```
ADDI $2 -1
ADDI $4 127
ANDX $5 $2 $4
ADDI $4 -127
ADD $4 $26
ADD $4 $5
EXCH $22 $4
ADD $21 $22
EXCH $22 $4
SUB $4 $5
SUB $4 $26
XOR $4 $5
XOR $5 $4
XOR $21 $4
XOR $4 $21
XOR $21 $4
XOR $22 $2
XOR $2 $22
ADDI $1 -11
BRA SMF
ADDI $1 11
SUB $27 $5
ADDI $1 -11
RBRA SMF
ADDI $1 11
ADD $2 $26
ADD $2 $21
EXCH $29 $2
SUB $4 $29
EXCH $29 $2
SUB $2 $21
SUB $2 $26
ADDI $2 127
ANDX $21 $22 $2
ADDI $2 -127
ADDI $22 1
SUB $22 $31
ADD $25 $31
EXCH $27 $25
SUB $25 $31
ADDI $31 1
ADDI $1 -11
EXCH $21 $1
ADDI $1 11
XOR $29 $3
XOR $3 $29
ADDI $1 -10
EXCH $22 $1
ADDI $1 10
XOR $2 $24
XOR $24 $2
XOR $3 $25
XOR $25 $3
XOR $4 $26
XOR $26 $4
XOR $5 $23
XOR $23 $5
ADDI $1 -9
EXCH $23 $1
ADDI $1 9
ADDI $1 -8
```

```
                EXCH $24 $1
                ADDI $1 8
                ADDI $1 -7
                EXCH $25 $1
                ADDI $1 7
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
_FORBOT408:     BNE $31 $30 _FORTOP407
                SUB $31 $30
                ADDI $30 -128
                ADDI $28 EPSILON
                EXCH $30 $28
                SUB $29 $30
                EXCH $30 $28
                ADDI $28 -EPSILON
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
_SUBBOT404:     BRA _SUBTOP403
_SUBTOP444:     BRA _SUBBOT445
PRINTWAVE:      SWAPBR $2
                NEG $2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
                ADDI $31 128
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADD $30 $29
_FORTOP446:     BNE $30 $29 _FORBOT447
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                OUTPUT $28
                ADD $28 $3
                ADD $28 $30
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                EXCH $26 $28
                ADD $27 $26
```

```
                EXCH $26 $28
                SUB $28 $30
                SUB $28 $3
                OUTPUT $27
                ADD $26 $3
                ADD $26 $30
                EXCH $28 $26
                SUB $27 $28
                EXCH $28 $26
                SUB $26 $30
                SUB $26 $3
                ADDI $30 1
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
_FORBOT447:     BNE $30 $31 _FORTOP446
                SUB $30 $31
                ADDI $31 -128
                ADDI $29 1
                OUTPUT $29
                ADDI $29 -1
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
_SUBBOT445:     BRA _SUBTOP444
_SUBTOP457:     BRA _SUBBOT458
SMF:            SWAPBR $2
                NEG $2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $29 1
                ADD $31 $3
_IFTOP459:      BGEZ $3 _IFBOT460
```

```
                NEG $31
_IFBOT460:      BGEZ $3 _IFTOP459
                ADD $30 $4
_IFTOP461:      BGEZ $4 _IFBOT462
                NEG $30
_IFBOT462:      BGEZ $4 _IFTOP461
                RL $29 31
                ADDI $1 -7
                EXCH $25 $1
                ADDI $1 7
                ADDI $25 1
                ADDI $1 -8
                EXCH $24 $1
                ADDI $1 8
                ADDI $24 32
                ADDI $1 -9
                EXCH $23 $1
                ADDI $1 9
                ADD $23 $25
_FORTOP463:     BNE $23 $25 _FORBOT464
                RR $29 1
                ANDX $27 $31 $29
_IFTOP467:      BEQ $27 $0 _IFBOT468
                SRLVX $28 $30 $23
                ADD $26 $28
                SRLVX $28 $30 $23
_IFBOT468:      BEQ $27 $0 _IFTOP467
                ANDX $27 $31 $29
                ADDI $23 1
_FORBOT464:     BNE $23 $24 _FORTOP463
                SUB $23 $24
                ADDI $24 -32
                ADDI $25 -1
_IFTOP469:      BGEZ $3 _IFBOT470
                NEG $26
_IFBOT470:      BGEZ $3 _IFTOP469
_IFTOP471:      BGEZ $4 _IFBOT472
                NEG $26
_IFBOT472:      BGEZ $4 _IFTOP471
                ADD $5 $26
_IFTOP473:      BGEZ $4 _IFBOT474
                NEG $30
_IFBOT474:      BGEZ $4 _IFTOP473
                SUB $30 $4
_IFTOP475:      BGEZ $3 _IFBOT476
                NEG $31
_IFBOT476:      BGEZ $3 _IFTOP475
                SUB $31 $3
                ADDI $29 -1
                ADDI $1 -9
                EXCH $23 $1
                ADDI $1 9
                ADDI $1 -8
                EXCH $24 $1
                ADDI $1 8
                ADDI $1 -7
                EXCH $25 $1
                ADDI $1 7
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
```

```
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
_SUBBOT458:     BRA _SUBTOP457
_MAINTOP:       BRA _MAINBOT
                .START SCHROED
SCHROED:        START
                ADDI $2 1
                ADDI $3 1001
                ADD $4 $2
_FORTOP477:     BNE $4 $2 _FORBOT478
                XOR $5 $4
                XOR $4 $5
                ADDI $4 PSII
                XOR $6 $3
                XOR $3 $6
                ADDI $3 PSIR
                XOR $7 $2
                XOR $2 $7
                BRA HALFSTEP
                ADDI $3 -PSIR
                ADDI $4 -PSII
                ADDI $4 PSIR
                ADDI $3 PSII
                RBRA HALFSTEP
                ADDI $3 -PSII
                ADDI $4 -PSIR
                ADDI $3 PSIR
                BRA PRINTWAVE
                ADDI $3 -PSIR
                ADDI $3 PSII
                BRA PRINTWAVE
                ADDI $3 -PSII
                ADDI $5 1
                XOR $2 $7
                XOR $7 $2
                XOR $3 $6
                XOR $6 $3
                XOR $4 $5
                XOR $5 $4
_FORBOT478:     BNE $4 $3 _FORTOP477
                SUB $4 $3
                ADDI $3 -1001
                ADDI $2 -1
                FINISH
_MAINBOT:       BRA _MAINTOP
```

Appendix F

Units, Constants, and Notations

This appendix simply lists various fundamental units, physical constants, and notations used throughout the text, for easy reference.

Table F.1 shows the names, abbreviations, and values of the standard order-of-magnitude prefixes for units of measurement.

Table F.2 shows the various base units of measurement referred to in this thesis. Any of these of course may also appear together with any of the prefixes listed in table F.1.

Table F.3 shows the fundamental physical constants we use. The Planck length L_{P} is listed in the units table (table F.2), but it may also be considered to be a fundamental physical constant. It is sometimes hypothesized to be some sort of minimum length scale for physics.

Table F.4 gives our preferred, more mnemonic notation for comparing the asymptotic order of growth of functions.

Name	Sym	Val	Name	Sym	Val
deka-	D	10^1	deci-	d	10^{-1}
hecto-	h	10^2	centi-	c	10^{-2}
kilo-	k	10^3	milli-	m	10^{-3}
mega-	M	10^6	micro-	μ	10^{-6}
giga-	G	10^9	nano-	n	10^{-9}
tera-	T	10^{12}	pico-	p	10^{-12}
peta-	P	10^{15}	femto-	f	10^{-15}
exa-	E	10^{18}	atto-	a	10^{-18}

Table F.1: Unit magnitude prefixes.

Symbol	Name	Measures	Some equivalences
Å	Ångstrom	length	10^{-10} m
B	byte	information	8 b
b	bit	information, entropy	$(\ln 2)$ nat
C	Coulomb	electric charge	
eV	electron Volt	energy	1.60217733×10^{-19} J
g	gram	mass	
Hz	Hertz	frequency	1/s
J	Joule	energy	1 N m
K	Kelvin	temperature	$\sim 1.38\times10^{-23}$ J/nat
kg	kilogram	mass	1000 g
m	meter	length	
N	Newton	force	1 kg m/s^2
nat	nat	entropy	k_{B}, 1 b/ln 2
L_{P}	Planck length	length	$\sqrt{G\hbar/c^3} \approx 1.616\times10^{-35}$ m
s	second	time	
V	Volt	electric potential	1 J/C

Table F.2: Some basic units of measurement used in this document.

Symbol	Meaning	Some equivalences	Approximate value
c	Speed of light		299792458 m/s
ϵ_0	Permittivity of free space	$1/\mu_0 c^2$	8.85418782×10^{-12} F/m
h	Planck's constant		6.6260755×10^{-34} J $\cdot$ s
$\hbar$	Reduced Planck's constant	$h/2\pi$	1.05457267×10^{-34} J $\cdot$ s
G	Gravitational constant		6.67259×10^{-11} N m^2/kg
k_{B}	Boltzmann's constant	1 nat	1.3806513×10^{-23} J/K
σ_{SB}	Stefan-Boltzmann constant	$\pi^2 k_{\mathrm{B}}^4/60c^2\hbar^3$	5.6704×10^{-8} J/s-K^4-m^2
q_{e}	Electron charge magnitude		1.60217733×10^{-19} C

Table F.3: Fundamental physical constants used in this document.

Cryptic standard notation	Our own mnemonic notation	Mathematical definition; English explanation
$f = \boldsymbol{\Theta}(g)$	$f \sim g$	$\exists c_1, c_2, n_0 > 0\colon\ \forall n \geq n_0\colon\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$; f is asymptotically proportional to g
$f = \mathcal{O}(g)$	$f \precsim g$	$\exists c, n_0 > 0\colon\ \forall n \geq n_0\colon\ 0 \leq f(n) \leq c g(n)$; f is asymptotically no more than proportional to g
$f = \boldsymbol{\Omega}(g)$	$f \succsim g$	$\exists c, n_0 > 0\colon\ \forall n \geq n_0\colon\ 0 \leq c g(n) \leq f(n)$; f is asymptotically no less than proportional to g
$f = \mathbf{o}(g)$	$f \prec g$	$\forall c > 0\colon\ \exists n_0 > 0\colon\ \forall n \geq n_0\colon\ 0 \leq f(n) < c g(n)$; f is asymptotically strictly less than proportional to g
$f = \boldsymbol{\omega}(g)$	$f \succ g$	$\forall c > 0\colon\ \exists n_0 > 0\colon\ \forall n \geq n_0\colon\ 0 \leq c g(n) < f(n)$; f is asymptotically strictly more than proportional to g

Table F.4: Asymptotic order-of-growth notation. In addition to reviewing the standard notation, we introduce a simplified notation that will be useful in some contexts.

Bibliography

[1] W. Athas. Low-power VLSI techniques for applications in embedded computing. In *IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, pages 14–22. IEEE Computer Society Press, March 1999. Abstract at `http://www.-isi.edu/acmos/abstracts/99-04.VOLTA.html`.

[2] W. Athas, N. Tzartzanis, L. Svensson, L. Peterson, H. Li, X. Jiang, P. Wang, and W-C. Liu. AC-1: A clock-powered microprocessor. In *Proc. of the International Symposium on Low-Power Electronics and Design*, Monterey, CA, 18–20 August 1997. `http://www.isi.edu/acmos/papers/97-08.MontereyAC1.-ps`.

[3] W. C. Athas, L. "J." Svensson, and N. Tzartzanis. A resonant signal driver for two-phase, almost-non-overlapping clocks. In *ISCAS-96*, 1996.

[4] W. C. Athas, N. Tzartzanis, L. "J." Svensson, and L. Peterson. A low-power microprocessor based on resonant energy. *IEEE Journal of Solid-State Circuits*, 32(11):1693–1701, nov 1997.

[5] William C. Athas, Lars "J." Svensson, Jeffrey G. Koller, Nestoras Tzartzanis, and Eric Ying-Chin Chou. Low-power digital systems based on adiabatic-switching principles. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):398–407, December 1994.

[6] Stephen Avery and Marwan Jabri. Design of a register file using adiabatic logic. Technical Report SCA-06/06/97, University of Sydney SEDAL, June 1997.

[7] Henry G. Baker. Lively linear lisp—'Look ma, no garbage!'. *ACM SigPlan Notices*, 27(8):89–98, August 1992.

[8] Henry G. Baker. NREVERSAL of fortune — the thermodynamics of garbage collection. In Y. Bekkers, editor, *International Workshop on Memory Management*, pages 507–524. Springer-Verlag, 1992.

[9] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $\mathcal{P} =? \mathcal{NP}$ question. *SIAM J. Computing*, 4(4):431–442, December 1975.

[10] Adriano Barenco, David Deutsch, Artur K. Ekert, and Richard Jozsa. Conditional quantum dynamics and logic gates. *Physical Review Letters*, 74(20):4083–4086, 15 May 1995. Preprint at Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9503017`.

[11] Edward Barton. A reversible computer using conservative logic. Term paper for 6.895 at MIT, 1978.

[12] Matthew E. Becker and Thomas F. Knight, Jr. Transmission line clock driver. In *Power Driven Microarchitecture Workshop*, pages 80–85, Barcelona, Spain, 28 June 1998.

[13] Matthew E. Becker and Thomas F. Knight, Jr. Transmission line clock driver. In *IMAPS Int'l Advanced Technology Wkshp. on Flip Chip Technology*, Braselton, GA, 12–14 March 1999.

[14] Jacob D. Bekenstein. Universal upper bound on entropy-to-energy ratio for bounded systems. *Phys. Rev. D*, 23(2):287–298, January 1981.

[15] Jacob D. Bekenstein. Entropy content and information flow in systems with limited energy. *Phyical Review D*, 30(8):1669–1679, 15 October 1984.

[16] C. H. Bennett. Logical reversibility of computation. *IBM J. Research and Development*, 17(6):525–532, 1973.

[17] C. H. Bennett. The thermodynamics of computation, a review. *International Journal of Theoretical Physics*, 21(12):905–940, 1982.

[18] C. H. Bennett. Notes on the history of reversible computation. *IBM J. Research and Development*, 32(1):16–23, January 1988. Reprinted in [100], ch. 4, pp. 281–288.

[19] C. H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Computing*, 18(4):766–776, 1989.

[20] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh V. Vazirani. Strengths and weaknesses of quantum computing. Preprint at `http://vesta.physics.ucla.edu/cgi-bin/uncompress_ps_cgi?bbbv94h.ps`, 1994.

[21] Ethan Bernstein and Umesh V. Vazirani. Quantum complexity theory. In *25th Association for Computing Machinery Symposium on the Theory of Computing*, pages 11–20, 1993.

[22] A. Berthiaume and Gilles Brassard. Oracle quantum computing. In *Proceedings of the Workshop on Physics of Computation: PhysComp '92*, pages 195–199, Los Alamitos, CA, 1992. Institute of Electrical and Electronic Engineers Computer Society Press. Also to appear in *Journal of Modern Optics*.

[23] A. Berthiaume and Gilles Brassard. The quantum challenge to structural complexity theory. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pages 132–137, Los Alamitos, CA, 1992. Institute of Electrical and Electronic Engineers Computer Society Press.

[24] Gianfranco Bilardi and Franco Preparata. Horizons of parallel computation. Technical Report CS-93-20, Brown University, May 1993. Also available on the web at `http://www.cs.brown.edu/publications/techreports/reports/CS-93-20.html`.

[25] Bruce M. Boghosian and Washington Taylor IV. Simulating quantum mechanics on a quantum computer. In Toffoli et al. [163], pages 32–35. `http://xxx.lanl.gov/abs/quant-ph/9701019`.

[26] Dan Boneh and Richard Lipton. Quantum cryptoanalysis of hidden linear functions. In *Advances in Cryptology — CRYPTO '95*. Springer-Verlag, 1995. Proceedings of the Crypto '95 Conference, Santa Barbara, California, Aug. 27–31. To appear. `http://www.cs.princeton.edu/~dabo/papers/quantum.ps`.

[27] Bob Boothe. Algorithms for bidirectional debugging. Technical Report USM/CS-98-2-23, Computer Science Department, University of Southern Maine, 96 Falmouth St, Portland ME 04104-9300, February 1998. Author's email: `boothe@cs.usm.maine.edu`.

[28] A. R. Calderbank and Peter W. Shor. Good quantum error-correcting codes exist. Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9512032`, December 1995.

[29] C. S. Calude, J. Casti, and M. J. Dinneen, editors. *Unconventional Models of Computation*. Springer, 1998. (Proc. of the First International Conference on Unconventional Models of Computation (UMC'98), held at the University of Auckland, January 5–9, 1998).

[30] Ruknet Cezzar. Reversible computer apparatus and methods of constructing and utilizing same. U.S. Patent #5,469,550, 21 November 1995.

[31] Anantha P. Chandrakasan and Robert W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[32] Isaac L. Chuang and Raymond Laflamme. Quantum error correction by coding. Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9511003`, November 1995.

[33] Isaac L. Chuang, Raymond Laflamme, Peter W. Shor, and W. H. Zurek. Quantum computers, factoring and decoherence. *Science*, 270:1635–1637, 1995. Preprint at Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9503007`.

[34] Isaac L. Chuang and Yoshihisa Yamamoto. A simple quantum computer. *Physical Review A*, 52:3489–3496, 1995. Preprint at Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9505011`.

[35] J. I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Physical Review Letters*, 74:4091–4094, 1995.

[36] D. Coppersmith. An approximate Fourier transform useful in quantum factoring. Research Report RC 19642, IBM, 1994.

[37] Don Coppersmith and Edna Grossman. Generators for certain alternating groups with applications to cryptography. *Society for Industrial and Applied Mathematics J. Appl. Math.*, 29(4):624–627, December 1975.

[38] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill, 1990.

[39] David G. Cory, Amr F. Fahmy, and Timothy F. Havel. Ensemble quantum computing by nuclear magnetic resonance spectroscopy. *Proceedings of the National Academy of Sciences of the United States of America*, 94(5):1634–1639, 4 March 1997. See `ftp://deas.ftp.harvard.edu/techreports/tr-10-96.ps.gz`.

[40] David G. Cory, M. D. Price, and T. F. Havel. Nuclear magnetic resonance spectroscopy: An experimentally accessible paradigm for quantum computing. *Physica D*, 120(1–2):82–101, 1 September 1998.

[41] B.C. Crandall and James Lewis, editors. *Nanotechnology: Research and Perspectives*. MIT Press, 1992.

[42] Pierluigi Crescenzi and Christos H. Papadimitriou. Reversible simulation of space-bounded computation. *Theoretical Computer Science*, 143:159–165, 1995.

[43] Defense Advanced Research Projects Agency (DARPA). Scalable computing systems. `http://www.darpa.mil/ito/research/scalable`.

[44] J. S. Denker, S. C. Avery, A. G. Dickinson, A. Kramer, and T. R. Wik. Adiabatic computing with the 2N-2N2D logic family. In *International Workshop on Low Power Design*, pages 183–187, 1994.

[45] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Ser. A*, A400:97–117, 1985.

[46] David Deutsch. Three connections between everett's interpretation and experiment. In R. Penrose and C. Isham, editors, *Quantum Concepts in Space and Time*, pages 215–225. Clarendon Press, Oxford, 1986.

[47] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London Ser. A*, A439:553–558, 1992.

[48] Alexander G. Dickinson. Adiabatic logic. U.S. Patent #5,521,538, 28 May 1996. Assigned to AT&T.

[49] David P. DiVincenzo. Two-bit gates are universal for quantum computation. *Physical Review A*, 51(2):1015–1022, February 1995. Also at Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/cond-mat/9407022`.

[50] M. Dorojevets, P. Bunyk, D. Zinoviev, and K. Likharev. COOL-0: an RSFQ subsystem design for petaflops computing. *IEEE Trans. on Appl. Supercond.*, jun 1999.

[51] K. Eric Drexler. *Nanosystems: Molecular Machinery, Manufacturing, and Computation*. John Wiley & Sons, Inc., 1992. `http://nano.xerox.com/nanotech/nanosystems.html`.

[52] Albert Einstein. *Relativity*. Crown Publishers, Inc., New York, 1961.

[53] Artur K. Ekert and Richard Jozsa. Shor's quantum algorithm for factorising numbers. To appear in Rev. Mod. Phys. Preprint at `ftp://eve.physics.ox.ac.uk/Archive/Numbered/EJ95/EJ95.ps`.

[54] Artur K. Ekert and Richard Jozsa. Notes on Shor's efficient algorithm for factoring on a quantum computer. Preprint at `http://feynman.stanford.edu/qcomp/ekert-jozsa/index.html`, 1994.

[55] François Englert. On the black hole unitarity issue. Los Alamos e-print `http://xxx.lanl.gov/abs/hep-th/9705115`, 15 May 1997.

[56] T. Pellizzari *et al.* Decoherence, continuous observation and quantum computing: a cavity QED model. Preprint, June 1995. University of Innsbruck.

[57] Hugh Everett, III. The theory of the universal wave function. In Bryce S. DeWitt and Neill Graham, editors, *The Many-Worlds Interpretation of Quantum Mechanics*, pages 3–139. Princeton University Press, 1973.

[58] William Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley & Sons, Inc., 1950.

[59] Richard Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6&7):467–488, 1982.

[60] Richard Feynman. Quantum mechanical computers. *Optics News*, 11, 1985. Also in *Foundations of Physics*, 16(6):507–531, 1986.

[61] David J. Frank. CMOS toggle flip-flop using adiabatic switching. U.S. Patent #5,517,145, 4 May 1996. Assigned to IBM.

[62] David J. Frank. Static combinatortial logic circuits for reversible computation. U.S. Patent #5,493,240, 20 February 1996. Assigned to IBM.

[63] David J. Frank and Paul M. Solomon. Energy conserving clock pulse generating circuits. U.S. Patent #5,506,520, 9 April 1996. Assigned to IBM.

[64] Michael Frank. Time-symmetric control-flow instructions for less garbage in reversible programs. MIT Reversible Computing Project Internal Memo #M1, February 1996. `http://www.ai.mit.edu/~mpf/rc/memos/M01_symmarch.html`.

[65] Michael Frank and Scott Rixner. Tick: A simple reversible processor (6.371 project report). Online term paper, May 1996. `http://www.ai.mit.edu/~mpf/rc/tick-report.ps`.

[66] Michael P. Frank. Low-energy computing for implantable medical devices. Talk presented to the MIT Clinical Decision Making Group, `http://www.ai.mit.edu/~mpf/rc/MedG_talk/webpage.html`, 21 February 1996.

[67] Michael P. Frank. Modifications to PISA architecture to support guaranteed reveribility and other features. MIT Reversible Computing Project Internal Memo #M7, February 1997. `http://www.ai.mit.edu/~mpf/rc/memos/M07/M07_revarch.html`.

[68] Michael P. Frank. Voltage scaling and limits to energy efficiency for CMOS-based SCRL. MIT Reversible Computing Project Internal Memo #M3, April 1997. Unfinished draft at `http://www.ai.mit.edu/~mpf/rc/memos/M03_scrllimits.html`.

[69] Michael P. Frank and M. Josephine Ammer. Separations of reversible and irreversible space-time complexity classes. Extended abstract submitted to FOCS-97. `http://www.ai.mit.edu/~mpf/rc/memos/M06_oracle.html`, 1997.

[70] Michael P. Frank and Thomas F. Knight, Jr. Ultimate theoretical models of nanocomputers. *Nanotechnology*, 9(3):162–176, 1998. Presented at the Fifth Foresight Conference on Molecular Nanotechnology, Palo Alto, CA, November 1997. `http://www.ai.mit.edu/~mpf/Nano97/paper.html`.

[71] Michael P. Frank, Thomas F. Knight, Jr., and Norman H. Margolus. Reversibility in optimally scalable computer architectures. In Calude et al. [29], pages 165–182. `http://www.ai.mit.edu/~mpf/rc/scaling_paper/scaling.html`.

[72] Michael P. Frank, Carlin Vieri, M. Josephine Ammer, Nicole Love, Norman H. Margolus, and Thomas F. Knight, Jr. A scalable reversible computer in silicon. In Calude et al. [29], pages 183–200. `http://www.ai.mit.edu/~mpf/rc/flattop/ft.html`.

[73] E. F. Fredkin and T. Toffoli. Design principles for achieving high-performance submicron digital technologies. DARPA Proposal, November 1978.

[74] E. F. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3/4):219–253, 1982.

[75] Ed Fredkin. Feynman, Barton and the reversible Schrödinger difference equation. In Hey [82], chapter 20, pages 337–348.

[76] Neil A. Gershenfeld and Isaac L. Chuang. Bulk spin resonance quantum computation. *Science*, 275(350), jan 1997.

[77] Lov. K. Grover. A fast quantum mechanical algorithm for database search. In *Proc. , 28th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 212–219, May 1996. `http://xxx.lanl.gov/abs/quant-ph/9605043`.

[78] Lov K. Grover. Quantum computers can search rapidly by using almost any transformation. Los Alamos e-Print Archive, December 1997. `http://xxx.lanl.gov/abs/quant-ph/9712011`.

[79] J. Storrs Hall. An electroid switching model for reversible computer architectures. In PhysComp92 [134], pages 237–247.

[80] J. Storrs Hall. An electroid switching model for reversible computer architectures. In *Proc. ICCI '92, 4th Int'l Conf. on Computing and Information*, 1992.

[81] J. Storrs Hall. A reversible instruction set architecture and algorithms. In PhysComp94 [135], pages 128–134.

[82] Anthony J. G. Hey, editor. *Feynman and Computation: Exploring the Limits of Computers*. Perseus Books, Reading, MA, 1999.

[83] R. T. Hinman and M. F. Schlecht. Recovered energy logic: A single clock AC logic. In *International Workshop on Low Power Design*, pages 153–158, 1994.

[84] David A. Hodges and Horace G. Jackson. *Analysis and Design of Digital Integrated Circuits*. McGraw-Hill, Inc., second edition, 1988.

[85] Tad Hogg. Quantum computing and phase transitions in combinatorial search. Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9508012`, August 1995.

[86] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1979.

[87] K. Huang. *Statistical Mechanics*. Wiley, 1963.

[88] E. Joos and A. Qadir. A quantum statistical upper bound on entropy. *Il Nuovo Cimento*, 107B(5):563–572, 1992.

[89] Thomas F. Knight, Jr. and Saed Younis. Charge recovery logic including split level logic. U.S. Patent #5,378,940, 3 January 1995. Assigned to MIT.

[90] Tom Knight. An architecture for mostly functional languages. In Patrick Henry Winston and Sarah Alexandra Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, volume 1, chapter 19, pages 500–519. The MIT Press, Cambridge, Massachusetts, 1990.

[91] Donald E. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley, second edition, 1981.

[92] J. G. Koller and W. C. Athas. Adiabatic switching, low energy computing, and the physics of storing and erasing information. In PhysComp92 [134], pages 267–270.

[93] A. Kramer, J. S. Denker, S. C. Avery, A. G. Dickinson, and T. R. Wik. Adiabatic computing with the 2N-2N2D logic family. In *1994 Symp. VLSI Circ.: Digest of Tech. Papers*. Institute of Electrical and Electronic Engineers Press, June 1994.

[94] A. Kramer, J. S. Denker, B. Flower, and J. Moroney. 2nd order adiabatic computation with 2N-2P and 2N-2N2P logic circuits. In *Intl. Symposium on Low Power Devices*, pages 191–196, Dana Point, CA, 1995. ACM.

[95] R. Landauer. Is quantum mechanically coherent computation useful? In D. H. Feng and B-L. Hu, editors, *Proceedings of the Drexel-4 Symposium on Quantum Nonintegrability—Quantum Classical Correspondence*. International Press, 1995.

[96] R. Landauer. Is quantum mechanics useful? *Philosophical Transactions of the Royal Society of London Ser. A*, 1995. In press.

[97] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM J. Research and Development*, 5:183–191, 1961. Reprinted in [100], ch. 4, pp. 188–196.

[98] Klaus-Jörn Lange, Pierre McKenzie, and Alain Tapp. Reversible space equals deterministic space. In *Proc. 12th Annual IEEE Conf. on Computational Complexity (CCC '97)*, pages 45–50, June 1997. `http://www.iro.umontreal.ca/~tappa/Publications/LMT'97_abstract.html`.

[99] Y. Lecerf. Machines de Turing réversibles. Insolubilité récursive en $n \in N$ de l'équation $u = \theta^n$, où θ est un $\ll$ isomorphisme de codes $\gg$ [Reversible Turing machines. Recursive insolubility in $n \in N$ of the equation $u = \theta^n$, where θ is an "isomorphism of codes"]. *Comptes Rendus Hebdomadaires des Séances de L'académie des Sciences [Weekly Proceedings of the Academy of Science]*, 257:2597–2600, October 28, 1963. Unauthorized English translation at `http://www.ai.mit.edu/~mpf/rc/Lecerf/lecerf.html`.

[100] Harvey S. Leff and Andrew F. Rex, editors. *Maxwell's demon: entropy, information, computing.* Princeton series in physics. Princeton University Press, Princeton, NJ, 1990. May be ordered through `http://www.ioppublishing.com/Books/Catalogue/020/__26/0750300566`.

[101] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes.* Morgan Kaufmann Publishers, San Mateo, California, 1992.

[102] A. K. Lenstra and H. W. Lenstra, editors. *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, 1993. .

[103] Robert Y. Levine and Alan T. Sherman. A note on Bennett's time-space tradeoff for reversible computation. *SIAM J. Computing*, 19(4):673–677, 1990.

[104] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Graduate Texts in Computer Science. Springer-Verlag, 2nd edition, 1997.

[105] Ming Li and Paul M. B. Vitányi. Reversibility and adiabatic computation: trading time and space for energy. *Proceedings of the Royal Society of London Ser. A*, 452:1–21, 1996.

[106] Ming Li and Paul M. B. Vitányi. Reversible simulation of irreversible computation. In *Proc. 11th IEEE Conference on Computational Complexity*, Philadelphia, Pennsylvania, May 24–27, 1996.

[107] David R. Lide, editor. *CRC Handbook of Chemistry and Physics: A Ready-Reference Book of Chemical and Physical Data*. CRC Press, Boca Raton, 79th edition, 1999.

[108] K. K. Likharev. Classical and quantum limitations on energy consumption in computation. *International Journal of Theoretical Physics*, 21(3/4):311–326, 1982.

[109] K. K. Likharev. Rapid single-flux-quantum logic. `http://pavel.physics.-sunysb.edu/RSFQ/Research/WhatIs/rsfqre2m.html`, 1992.

[110] Konstantin K. Likharev. Rapid single flux quantum (RSFQ) superconductor electronics. See `http://pavel.physics.sunysb.edu/RSFQ/Research/WhatIs/rsfqwte1.html`, may 1996.

[111] Konstantin K. Likharev and Alexander N. Korotkov. "Single-electron parametron": Reversible computation in a discrete-state system. *Science*, 273:763–765, 9 August 1996.

[112] Seth Lloyd. A potentially realizable quantum computer. *Science*, 261:1569–1571, 17 September 1993. Preprint at `http://www-im.lcs.mit.edu:80/poc/lloyd/lloyd.ps.Z`.

[113] Seth Lloyd. Envisioning a quantum supercomputer. *Science*, 263:695, 4 February 1994. Technical comment on [112].

[114] Chris Lutz. Janus: A time-reversible language. Letter from Chris Lutz to Rolf Landauer. Unauthorized reproduction at `http://www.ai.mit.edu/~mpf/rc/janus.html`, 1 April 1986.

[115] N. H. Margolus. *Physics and Computation*. PhD thesis, Massachusetts Institute of Technology, 1988.

[116] Norman Margolus. Physics-like models of computation. *Physica D*, 10:81–95, 1984.

[117] Norman Margolus. Crystalline computation. In Hey [82], chapter 18, pages 267–305.

[118] Norman Margolus and Lev B. Levitin. The maximum speed of dynamical evolution. In Toffoli et al. [163], pages 208–211. Available through `http://www.-interjournal.org`. Revised version available at `ftp://im.lcs.mit.edu/poc/margolus/speed.of.dynamics.ps.Z`.

[119] Norman Margolus, Tommaso Toffoli, and Gérard Vichniac. Cellular-automata supercomputers for fluid dynamics modeling. *Phys. Rev. Lett.*, 56(16):1694–1696, 21 April 1986.

[120] Vladimir S. Mashkevich. Conservative model of black hole (sic) and lifting of the information loss paradox. Los Alamos e-print `http://xxx.lanl.gov/abs/gr-qc/9707055`, 27 July 1997.

[121] Carver A. Mead. Scaling of MOS technology to submicrometer feature sizes. *Journal of VLSI Signal Processing*, 8:9–25, 1994. Reprinted as chapter 9 of [82].

[122] Ralph C. Merkle. Towards practical reversible logic. In PhysComp92 [134], pages 227–228.

[123] Ralph C. Merkle. Reversible electronic logic using switches. *Nanotechnology*, 4:21–40, 1993.

[124] Ralph C. Merkle. Two types of mechanical reversible logic. *Nanotechnology*, 4:114–131, 1993.

[125] Ralph C. Merkle. Reversible charge transfer and logic utilizing them. U.S. Patent #5,357,548, 18 October 1994. Assigned to Xerox.

[126] Ralph C. Merkle and K. Eric Drexler. Helical logic. *Nanotechnology*, 7(4):325–339, 1996. Available through `http://www.ioppublishing.com`.

[127] Cesar Miquel, Juan Pablo Paz, and Roberto Perazzo. Factoring in a dissipative quantum computer. Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9601021`, January 1996. Discusses a simulation of Shor's algorithm with error correction.

[128] J. E. Mooij, T. P. Orlando, L. Levitov, Lin Tian, Caspar H. van der Wal, and Seth Lloyd. Josephson persistent-current qubit. *Science*, 285:1036–1039, 13 August 1999 1999.

[129] C. D. Motchenbacher and J. A. Connelly. *Low-Noise Electronic System Design.* John Wiley & Sons, Inc., 1993.

[130] Robert C. Myers. Pure states don't wear black. Los Alamos e-print `http://xxx.lanl.gov/abs/gr-qc/9705065`, May 1997.

[131] William J. Ooms and Jerald A. Hallmark. Complementary logic recovered energy circuit. U.S. Patent #5,426,382, 20 June 1995. Assigned to Motorola.

[132] G. Massimo Palma, Kalle-Antti Suominen, and Artur K. Ekert. Quantum computers and dissipation. *Proceedings of the Royal Society of London Ser. A*, 1995. Submitted.

[133] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[134] *PhysComp '92: Proceedings of the Workshop on Physics and Computation, October 2–4, 1992, Dallas, Texas*, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[135] *PhysComp '94: Proceedings of the Workshop on Physics and Computation, November 17–20, 1994, Dallas, Texas*, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[136] J. E. Pin. On the languages accepted by finite reversible automata. In Thomas Ottman, editor, *Automata, Languages and Programming, Proc. 14th Int'l Colloq. (ICALP)*, volume 267 of *Lecture Notes in Computer Science*, pages 237–249. Springer-Verlag, 1987.

[137] M. Planck. *The Theory of Heat Radiation.* Dover, New York, 1991.

[138] John Preskill. Do black holes destroy information? Los Alamos e-print `http://xxx.lanl.gov/abs/hep-th/9209058`, September 1992.

[139] J.M. Rabaey. *Digital Integrated Circuits: A Design Perspective.* Prentice-Hall, 1996. `http://infopad.EECS.Berkeley.EDU/~icdesign`.

[140] A. L. Ressler. The design of a conservative logic computer and a graphical editor simulator. Master's thesis, MIT Artificial Intelligence Laboratory, 1981.

[141] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* The MIT Press, first mit press paperback edition edition, 1987. Original edition published by McGraw-Hill Book Company, 1967.

[142] Martin F. Schlecht and Roderick T. Hinman. Recovered energy logic circuits. U.S. Patent #5,396,527, 7 March 1995. Assigned to MIT.

[143] Francis W. Sears, Mark W. Zemansky, and Hugh D. Young. *University Physics.* Addison-Wesley Series in Physics. Addison-Wesley Publishing Company, Reading, Massachusetts, sixth edition, February 1984.

[144] Charles L. Seitz, Alexander H. Frey, Sven Mattisson, Steve D. Rabin, Don A. Speck, and Jan L. A. van de Snepscheut. Hot-clock nMOS. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 1–17. Computer Science Press, 1985.

[145] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors: Technology Needs.* SEMATECH, Inc., 1997 edition, 1997. `http://notes.sematech.org/ntrs/Rdmpmem.nsf`.

[146] Peter W. Shor. Algorithms for quantum computation: Discrete log and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. Institute of Electrical and Electronic Engineers Computer Society Press, November 1994. `ftp://netlib.att.com/netlib/att/math/shor/quantum.algorithms.ps.Z`.

[147] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9508027`, August 1995. Expanded version of [146].

[148] Hava (Eve) Tova Siegelmann. *Foundations of Recurrent Neural Networks.* PhD thesis, Rutgers University, 1993.

[149] David R. Simon. On the power of quantum computation. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 116–123, Los Alamitos, CA, 1994. Institute of Electrical and Electronic Engineers Computer Society Press. Preprint at `http://vesta.physics.ucla.edu/cgi-bin/uncompress_ps_cgi?simon94.ps`. Improves on Bernstein & Vazirani '93 [21]. Shor '94 [146] was inspired by this.

[150] Tycho Sleator and Harald Weinfurter. Realizable universal quantum logic gates. *Physical Review Letters*, 74:4087–4090, 1995. Proposes a universal gate and a Cavity QED implementation.

[151] Warren D. Smith. Church's thesis meets the N-body problem. Technical Report TM 93-105-3-0058-6, NECI, September 1993. `http://www.neci.nj.nec.com/homepages/wds/church.ps`.

[152] Warren D. Smith. Fundamental physical limits on computation. Technical report, NECI, May 1995. `http://www.neci.nj.nec.com/homepages/wds/fundphys.ps`.

[153] Paul M. Solomon and David J. Frank. Power measurements of adiabatic circuits by thermoelectric technique. In *Symposium on Low Power Electronics*, pages 18–19, 1995.

[154] Dinesh Somasekhar, Yibin Ye, and Kaushik Roy. An energy recovery static RAM memory core. In *Symposium on Low Power Electronics*, pages 62–63, 1995.

[155] Andrew Steane. Multiple particle interference and quantum error correction. *Proceedings of the Royal Society of London Ser. A*, 1996. (Submitted.) Preprint available at Los Alamos Physics Preprint Archive, `http://xxx.lanl.gov/abs/quant-ph/9601029`. Introduces a parity encoding for quantum error correction.

[156] L. "J." Svensson and J. G. Koller. Adiabatic charging without inductors. In *Proc. Int'l Workshop on Low-Power Design*, pages 159–164, apr 1994.

[157] Lars Svensson. Adiabatic switching. In *Low Power Digital CMOS Design* [31], chapter 6, pages 181–218.

[158] W. G. Teich, K. Obermayer, and G. Mahler. Structural basis of multistationary quantum systems *ii*: Effective few-particle dynamics. *Physical Review B*, 37:8111–8121, 1988.

[159] Steven D. Thomas. Precharged adiabatic pipelined logic. U.S. Patent #5,602,497, 11 February 1997.

[160] Tommaso Toffoli. Computation and construction universality of reversible cellular automata. *J. Computer and System Sciences*, 15:213–231, 1977.

[161] Tommaso Toffoli. Reversible computing. Technical memo MIT/LCS/TM-151, MIT Lab for Computer Science, February 1980. Out of print; available from NTIS. Abridged version available as [162].

[162] Tommaso Toffoli. Reversible computing. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming (Seventh Colloquium, Noordwijkerhout, the Netherlands, July 14–18, 1980)*, volume 85 of *Lecture Notes in Computer Science*, pages 632–644. Springer-Verlag, 1980. Abridged version of [161].

[163] Tommaso Toffoli, Michael Biafore, and João Leão, editors. *PhysComp96 (Proceedings of the Fourth Workshop of Physics and Computation, Boston University, 22–24 November 1996)*. New England Complex Systems Institute, 1996. Copies may be ordered from *PhysComp96*, 44 Cummington St., Boston MA 02215 (`PhysComp96@pm.bu.edu`). Individual papers are available through `http://www.interjournal.org`.

[164] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, 1987.

[165] Kai-Yap Toh, Ping-Keung Ko, and Robert G. Meyer. An engineering model for short-channel MOS devices. *IEEE Journal of Solid-State Circuits*, 23(4):950–958, August 1988.

[166] D. B. Tuckerman and R. F. W. Pease. High-performance heat sinking for VLSI. *IEEE Electron Device Letters*, EDL-2(5):126–129, May 1981.

[167] David Bazeley Tuckerman. *Heat-Transfer Microstructures for Integrated Circuits*. PhD thesis, Stanford University, February 1984.

[168] Q. A. Turchette, C. J. Hood, W. Lange, H. Mabuchi, and H. Jeffrey Kimble. Measurement of conditional phase shifts for quantum logic. Submitted to Physical Review Letters. Abstract at `http://www.cco.caltech.edu/~hood/QO/Abstracts/Turc95b.html`.

[169] A. M. Turing. On computable numbers, with an application to the *entscheidungsproblem*. *Proc. London Math. Society*, 2(42):230–265, 1936. Also no. 43, pp. 544–546, 1937.

[170] N. Tzartzanis and W. Athas. Clock-powered CMOS: A hybrid adiabatic logic style for power-efficient computing. In *20th Anniversary Conference on Advanced Research in VLSI*, pages 137–151. IEEE Computer Society Press, March 1999. Slides at `http://www.isi.edu/acmos/presentations/99-03.ARVLSI.pdf`.

[171] Nestoras Tzartzanis and William C. Athas. Energy recovery for the design of high-speed, low power static RAMs. In *International Symposium on Low Power Electronics and Design*, pages 55–60, 1996.

[172] David Ungar, Henry Lieberman, and Christopher Fry. Debugging and the experience of immediacy. *Communications of the ACM*, 40(4):38–43, April 1997.

[173] W. G. Unruh. Decoherence and quantum computers: a problem. In *Proceedings of the Workshop on Physics of Computation: PhysComp '94*, Los Alamitos, CA, 1994. Institute of Electrical and Electronic Engineers Computer Society Press.

[174] W. G. Unruh. Mainraining coherence in quantum computers. *Physical Review A*, 51:992–997, 1995. Preprint at `http://feynman.stanford.edu/qcomp/unruh/index.html`.

[175] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 1–66. Elsevier, Amsterdam, 1990.

[176] A. Vergis, K. Steiglitz, and B. Dickinson. The complexity of analog computation. *Math. and Computers in Simulation*, 28:91–113, 1986. .

[177] Anastasios Vergis, Kenneth Steiglitz, and Bradley Dickinson. The complexity of analog computation. *Mathematics and Computers in Simulation*, 28:91–113, 1986.

[178] Carlin Vieri. *Reversible Computer Engineering and Architecture.* PhD thesis, Massachusetts Institute of Technology, 1999.

[179] Carlin J. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.

[180] Paul M. B. Vitányi. Locality, communication and interconnect length in multicomputers. *SIAM J. Computing*, 17:659–672, 1988.

[181] J. von Neumann. Non-linear capacitance or inductance switching, amplifying, and memory organs. U.S. Patent #2,815,488, December 3, 1957. Assigned to IBM.

[182] John von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, 1966.

[183] Robert M. Wald. *General Relativity.* University of Chicago Press, 1984.

[184] L. Wang. On the classical limit of phase-space formulation of quantum mechanics: entropy. *J. Math. Phys.*, 27:483–487, 1986.

[185] Steve Ward, John Nguyen, and John Pezaris. 3D-3N meshes. MIT LCS NuMesh project internal memo, sep 1991. `http://www.cag.lcs.mit.edu/numesh/papers/memos/3d3n.html`.

[186] Boyd G. Watkins. A low-power multiphase circuit technique. *IEEE Journal of Solid-State Circuits*, pages 213–220, December 1967.

[187] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley, second edition, 1993.

[188] R. L. Wigington. A new concept in computing. *Proceedings of the IRE*, 47:516–523, April 1961.

[189] Philip Yam. Rubbed out with the quantum eraser: Making quantum information reappear. *Scientific American*, January 1996.

[190] Yibin Ye and Kaushik Roy. Energy recovery circuits using reversible and partially reversible logic. *IEEE Transactions on Circuits and Systems—I: Fundamental Theory and Applications*, 43(9):769–778, sep 1996.

[191] S. G. Younis. *Asymptotically Zero Energy Computing Using Split-Level Charge Recovery Logic.* PhD thesis, MIT Artificial Intelligence Laboratory, 1994.

[192] S. G. Younis and T. F. Knight, Jr. Practical implementation of charge recovering asymptotically zero power CMOS. In *Proc. 1993 Symp. on Integrated Systems*, pages 234–250. MIT Press, 1993.

[193] S. G. Younis and T. F. Knight, Jr. Asymptotically zero energy split-level charge recovery logic. In *International Workshop on Low Power Design*, pages 177–182, 1994.

[194] S. G. Younis and T. F. Knight, Jr. Harmonic resonant rail drivers for adiabatic logic. In *Proceedings of the 1995 Symposium on Advanced Research in VLSI.* MIT Press, 1995.

[195] D. Zinoviev. Design issues in ultra-fast ultra-low-power superconductor batcher-banyan switching fabric based on RSFQ logic/memory family. *Applied Superconductivity*, 5(7–12):235–239, aug 1998.

Index

That’s all, folks!