Self-Assembled Computer Architecture: Design and Fabrication Theory

by

Christopher L. Dwyer

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science

Chapel Hill

2003

Approved by:

Advisor: Russell Taylor

Reader: Dorothy Erie

Reader: John Poulton

Reader: Leandra Vicci

Reader: Sean Washburn

ABSTRACT

CHRISTOPHER L. DWYER: Self-Assembled Computer Architecture: Design and Fabrication Theory

(under the direction of Russell M. Taylor)

This dissertation explores the design and fabrication of massively-parallel computers using self-assembling electronic circuitry. A DNA-guided self-assembly method, inspired by discoveries in chemistry, materials science, and physics, is used to develop an argument for the feasibility of constructing complex circuitry. The fabrication yield of such a process is calculated. Together, these form the foundation for a discussion of the computer architectures and implementations that this self-assembling process enables.

Simulations estimate the electrical performance of the components used in the self-assembly process. Traditional drift-diffusion simulations were used to evaluate a ring-gated field effect transistor and the results were applied to a circuit level simulation to evaluate specific circuit topologies. These circuits were then grouped into implementation level components (logic gates, memory elements, etc.) and used in an architecture-level behavior simulator.

The electrical performance results enable an informed evaluation of higher-level computer designs that could be built using self-assembly. Estimates of the performance, including power consumption, of two architectures are presented. These architectures appear to be impractical without a self-assembling fabrication process and are shown to have remarkable advantages over conventional computing machines. These machines are estimated to be nearly three orders of magnitude faster than the fastest next-generation supercomputer (IBM BlueGene /L) on certain classes of problems.

ACKNOWLEDGEMENTS

Thanks to

My committee members: Dorothy Erie, John Poulton, Richard Superfine, Russell Taylor, Leandra Vicci, and Sean Washburn for their guidance and support.

Mike Falvo, Martin Guthold, Garrett Matthews, Neal Snider, Phillip Williams, and all the nano-scientists for their help, experience, and charity.

The Nanoscale Science Research group.

Mark Kellam for his guidance with the PISCES-IIb simulations.

Catherine Perry, Andrea Bunn, Donna Boggs, Sandra Neely, Tammy Pike, Janet Jones, and all the other *great* staff members in the Computer Science department for making the wheels turn.

My cohort (and others) for many truly entertaining moments.

(in alphabetical order) Alexandra, Angela, Bamboo Stick, BrendOn, Caroline, Chris, Dave, Hedge & Ivy Patch, Kelly, Ketan, Mark, Matt, Michele, Mike, Montek, Nashel, Nick, Paul, Sharif, Shelby, Stefan, Steve, Tracy, Yoni, and Zac.

and most importantly,

My mother and father for their caring and enthusiastic support, and

Owen, Susan, Katie, Manuel, Alison, and Andrea for helping to remind me how precious it all can be.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

3D	Three dimensional
3-SAT	Three literals per clause satisfiability problem
AC	Accumulator control multiplexer
ACC	Accumulator
ALU	Arithmetic & logic unit
ASIC	Application specific integrated circuit
ATP	Adenosine triphosphate
AuPd	Gold-palladium
CAM	Content addressable memory
CMOS	Complementary metal oxide semiconductor
CREG	Control register
CSM	Control state machine
DAMP	Decoupled array multi-processor
DC	Direct current
DES	Data encryption standard
DNA	Deoxyribonucleic acid
DMF	Dimethyl-formamide
DTT	Dithiothreitol
EDC	1-Ethyl-3-(3-dimethylaminopropyl)carbodiimide Hydrochloride
EEC	Enable enable-carry-bit
EESD	Enable enable-set-D-bit
EL	Enable random constant load circuit
ERC	Enable reset carry-bit
ES	NEC Earth Simulator
ESA	Enable shift accumulator
ESC	Enable set carry-bit
ESR	Enable shift register
FEM	Finite element method
FET	Field effect transistor
F/F	Flip flop

fF	Femtofarad
FLOP	Floating-point & logic operation
FO	Fanout
FPGA	Field programmable gate array
FSA	Fluidic self-assembly
GB	Gigabyte
Gbps	Gigabits per second
GND	Electrical ground
HAM-PATH	Hamiltonian path problem
HP	Hewlett-Packard
IBM	International Business Machines Corporation
IMM	Immediate value (external)
Ids	Current drain-to-source
I/O	Input / Output
IV	Current / voltage
LC1	Logic control multiplexer one
LC2	Logic control multiplexer two
LC2b	Least significant control bit of logic control multiplexer two (LC2)
LSB	Least significant bit
MB	Megabyte
Mbps	Megabits per second
MFC	Microsoft Foundation Classes
MHz	Megahertz
MIMD	Multiple instruction multiple data
MIN-QUERY	Minimization query
MSB	Most significant bit
MUX	Multiplexer
MUX2	Two input multiplexer
MW	Megawatt
nF	Nanofarad
NFET	N-type field effect transistor
NMR	Nuclear magnetic resonance
NP	Non-deterministic polynomial time

oligo	Oligomer
P4	Intel Pentium 4
PAGE	Polyacrylamide gel electrophoresis
PBS	Phosphate buffer solution
PE	Processing element
PFET	P-type field effect transistor
PMMA	Poly-methyl-methacrylate (a photoresist)
PPS	Parallel pattern search
QCA	Quantum-dot cellular automata
R0	Register zero
R1	Register one
R2	Register two
R3	Register three
R4	Register four
RC0	Register control multiplexer zero
RC1	Register control multiplexer one
RC2	Register control multiplexer two
RC3	Register control multiplexer three
RC4	Register control multiplexer four
RCU	Register control unit
REGFILE	Register file
RG-FET	Ring gated field effect transistor
RX	Unspecified (any) register
SEM	Scanning electron microscope
SGT	Surrounded-gate transistor
SIMD	Single instruction multiple data
SPICE	Simulation Program with Integrated Circuit Emphasis
SREG	Setup register
SRH	Shockley-Read-Hall recombination
SWNT	Single-walled carbon nanotubes
TB	Terabyte
UV	Ultraviolet
Vdd	Voltage for common drains

Vds	Voltage drop from drain-to-source
Vgs	Voltage drop from gate-to-source
VLSI	Very large scale integration
WC1	Wait control multiplexer one
WC2	Wait control multiplexer two
Win32	Microsoft's 32-bit Windows Operating System Platform
WTC	Wait & trigger controller

Chapter 1. Introduction

1.1 Overview

Computer system design will change dramatically as nanoscale science and technology develop to the point where practical assembly mechanisms exist for building systems with 10^{19} components. These changes will be motivated by an interest in developing computing devices that exploit the new technology's features and avoid its pitfalls. The advent of massively parallel near molecular-scale electronic systems will enable solutions to problem spaces yet untouched by modern computing. This dissertation evaluates two such computer architectures and their feasible fabrication by DNA-guided self-assembly.

1.2 Thesis statement and contributions

> "DNA-guided self-assembled 3D silicon nanorod assemblies with 10^{19} components can be used to construct novel computers of theoretical and practical use. This method overcomes hurdles in attachment chemistry, assembly complexity, electronic behavior, fabrication yield, power consumption, thermal effects, system design, and performance."

This dissertation provides a proof-of-concept argument that shows there are no fundamental barriers to undertaking the construction of such computers. It uses published science, new simulations, and preliminary chemistry laboratory work to address all levels of design and fabrication. Actual construction of this system would require economic resources beyond those available here.

The work outlined in the following section forms the plausibility argument that massively parallel computing machines can be constructed in the foreseeable future, perhaps

even within the next decade, using self-assembly. The characteristics of nanoscale assembly require a fundamental shift in how computers are designed to harness this power.

This dissertation contributes to the field of computer engineering by establishing a first point of comparison showing how the design of traditional computing machines can change to produce a new class of computing machine enabled by near molecular-scale self-assembly. The following have been done to show the feasibility and usefulness of constructing such a machine:

- Published new work on DNA - nanotube attachment.
- Published simulation results of a new type of semiconductor transistor, the ring-gated field effect transistor, or RG-FET.
- Published simulation results on the use of RG-FETs in logic circuitry.
- Developed a theory of fabrication using self-assembly that takes into consideration known chemistry, thermodynamics, DNA reactions, and yield estimates.
- Developed a face-serial assembly ordering algorithm that can order the allocation of 15 unique DNA sequences for the logic structures used here.
- Developed the *Oracle* and the decoupled array multi-processor (DAMP), architectures that brackets a continuum of computer architectures that span from theoretically interesting to practical and are designed to take advantage of the strengths of self-assembly.
- Simulated the large-scale behavior of RG-FET circuitry in implementations of the oracle and DAMP architectures.
- Evaluated the performance of the DAMP and Oracle versus conventional supercomputers.

Each of the above is a piece of the larger contribution that is the end-to-end design of a new useful kind of computing machine, showing that at all levels it can be feasibly fabricated. This machine is inspired by the massive parallelism found in nature and

simulations show that it favorably compares with the fastest and most parallel machines available today.

1.3 Introduction

Methods using DNA-guided self-assembly are in their infancy and have to date only produced simple geometric structures. The complexity of these structures is far from approaching the complexity found in the biological organisms that have inspired such work. However, there are considerable quantities of human effort being put into developing this technology to the point where it can be usefully employed to build complete nanoscale structures.

Figure 1.1 illustrates the problem faced by self-assembled computer designers. The fundamental problem is how to make a design that starts from feasible physical processes and ends up being useful as a computing device. The problem can also be viewed from the other side - how to make an architecture that provides a useful computing model and is feasible to build with self-assembling techniques. Designing a useful architecture that can actually be built requires finding and traversing the complete path.

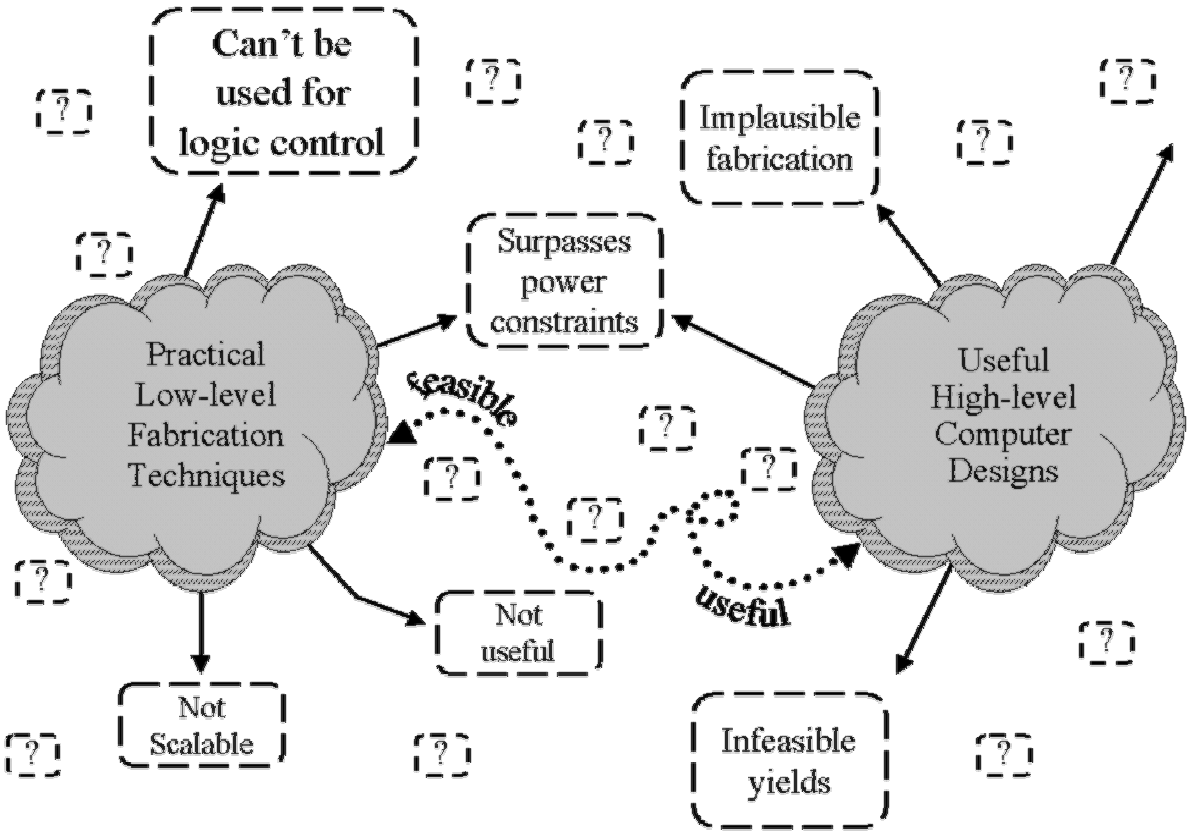

Figure 1.1. Technological design space. Illustration of the many challenges facing a self-assembled machine designer. In particular, the strong coupling between the low-level fabrication and high-level design makes the space treacherous.

1.4 Method

Some researchers have provided tantalizing glimpses of how DNA can self-assemble interesting shapes but leave open how to use these to build actual computers with large numbers of processors. This dissertation describes two computer architectures, the Oracle and DAMP, and shows that both can be plausibly implemented and realized and that both are useful, addressing problems of either theoretical or practical usefulness.

The question answered by the thesis above is simply how can self-assembly be used, in its current stage, to build computing machinery that is better than what we have today? Today's processors are designed according to a rule that says computation must largely occur

at runtime. Wide data paths, caches, branch prediction, fast multipliers all come from a computing heritage that emphasizes runtime execution. The inherent circuit complexity in this design philosophy is obvious from the intricate and idiosyncratic mask-work for contemporary processors, where large potions of the mask are custom-designed for specific functions.

Toward the end of the 20th century, multiple-instruction-multiple-data (MIMD) shared memory parallel computers became the dominant tools for solving some of the most complex problems faced by society. These machines were, and still are, massive constructions that take thousands of design hours and hundreds of millions of dollars to build. They are typically made of arrays of tightly coupled processors with memory that is either shared or distributed, and require very elaborate and clever schemes for balancing the problem load among processors and maintaining efficient communication between processors. In contrast, regularity and loosely coupled parallelism appear to be the only feasible architecture in self-assembled machines at scales that rival the largest distributed computer networks. This leads to a shift in machine design to machines with 10^{12} computing elements, each with a small amount of local memory, with no communication between processors.

Design shifts

Feature	Self-assembly	VLSI
Fabrication scale	**10^{19} components**	~10^{9} components
Required fault tolerance	high	**low**
Max. individual circuit size	small	**very large**
High bandwidth interconnections	impractical	**feasible**
Tightly coupled processors	impractical	**feasible**

Table 1.1. Design shifts between VLSI and self-assembly.

Table 1.1 indicates that the only advantage self-assembly has over conventional VLSI is the fabrication scale. Every other design feature has changed dramatically for the worse. This is the primary motivation behind the discussion in following chapters. Since the set of design features that are routinely used in conventional technologies has changed in the switch

to a self-assembling technology, how can the design of computing machines change to enable new useful designs?

Outline

Since the fabrication techniques used by chapters 3 and 4 may be foreign to the typical computer designer, these readers may find it useful to refer to section 3.2 and appendix A for a brief introduction to the means and methods available to self-assembling technologies.

Chapter 2 describes the state of the art in massively-parallel computer design, DNA computation, self-assembled circuitry and molecular electronics, fault-tolerant computing, quantum-dot cellular automata, and quantum computing and describes how the work presented in this dissertation differs from the prior art.

Chapter 3 discusses the circuit fabrication method with an emphasis on the physical details encountered in the implementation.

Chapter 4 explores the tradeoffs between yield and the size of a self-assembled computer and how structural rigidity plays a role in this relationship.

Chapter 5 describes the design and assembly-ordering tools developed to create designs of 3D nanorod structures for simulation.

Chapter 6 describes methods for supplying power to self-assembled computers and communicating between processors and the outside world.

Chapter 7 investigates the *Oracle* - a self-assembled computer architecture inspired by work in DNA computing that is enabled by the self-assembling process described in chapter 2.

Chapter 8 investigates the decoupled array multi-processor (DAMP) - a self-assembled computer architecture that is similar to conventional computer designs but incorporates 10^{12} processing elements.

Chapter 9 describes the electrostatic, semi-conductor, thermal, circuit-level, and behavioral simulation of circuitry that can be used to implement the oracle and DAMP architectures.

Chapter 10 investigates the thermal properties of the DAMP and cooling techniques.

Chapter 11 presents performance estimates for the DAMP on the blind data encryption standard (DES) decryption problem and a typical global optimization problem. The performance of the DAMP is compared against the fastest known computing machines.

Appendix A describes a method for chemically attaching DNA fragments to carbon nanotubes and an introduction to the terms and methods used in this field.

Appendix B includes a compact disc that contains the source code and text used within this dissertation.

Appendix C contains a detailed description of the instruction set implemented by the DAMP assembler described in chapter 8.

Chapter 2. Related work

2.1 Massively-parallel computer design

State-of-the-art supercomputer design has converged over the last decade to multiple-instruction-multiple-data (MIMD) architectures. This type of machine is flexible, has a high-bandwidth inter-processor connection network, and can solve many kinds of problems. The top 500 list of the fastest supercomputers on the planet is currently topped by the NEC EarthSimulator [Top500, 2003]. This machine is an array of 640 processor nodes interconnected as a hypercube, with 8 processors per node. Each processor has 16 GB of local memory (shared with the others) and can execute 8 GFlop/sec. The aggregate machine has a peak performance of ~40 TFlop/sec with 10 TB of memory. The entire machine occupies 13,000 m^3 without including the cooling and power supply systems. The processors in the Earth Simulator were fabricated using a 0.15 µm CMOS technology [Dongarra, 2002]. The next generation MIMD supercomputer, the IBM BlueGene /L, is planned to boast a peak performance of 360 TFlop/sec and occupy 533 m^3 [BlueGene, 2002]. Each consumes ~3 MW of power without considering the power consumed by cooling.

These machines use large numbers of tightly-coupled processors to solve large problems more quickly than any machine ever has. The natural design progression is to continue increasing the number of processors and inter-processor bandwidth to solve larger and larger problems. However, the size and power consumption of present-day supercomputers is already becoming a constraint on the design. The work presented in subsequent chapters will show how a smaller and less power-hungry self-assembled computer can out-perform present-day supercomputers for some classes of problems.

2.2 DNA computation

The pioneering work by Adleman, et al. in establishing a role for DNA (deoxyribonucleic acid) in the computation of large problems made the prospect of

molecular-scale computing systems appear to be within reach [Adleman, 1994]. The solution to a seven-node Hamiltonian path problem using DNA took several weeks of chemical laboratory processing. The 3-SAT problem[1] has also been solved for problems with up to 20 variables using DNA [Braich, 2002]. Unfortunately, these chemical methods must be repeated for each new instance of the problem, so it takes about a week to solve each.

This work has been followed, and expanded, by many others to address other problems with new methods of computation. The application of these methods to general-purpose "wet" computer architecture marks a new turn in how DNA has been used in computer design [Head, 2001]. This method can manipulate strings of symbols and is Turing complete. However, this method still relies on enzymatic processes and requires chemical laboratory work to extract results, like all DNA computing methods, which can take several weeks.

Similar investigations have developed physical representations of computation that use DNA to construct solutions to a problem [Seeman, 2001; Winfree, 2000]. These projects use the self-assembling properties of DNA to solve a problem by forming thin membranes of DNA tiles that follow binding rules. The binding rules define an automaton and have been shown to be Turing complete. The self-assembling fabrication method described in chapter 3 uses DNA to form electrical structures rather than to solve a computational problem - the resulting computer can then solve instances of the problem at a much faster rate. The idea is to solve all instances of the program at once, then select the appropriate solution electronically at runtime.

2.3 Self-assembled circuitry & molecular electronics

Molecular electronics has been the focus of many researchers looking at how to apply advances in self-assembly. There is interesting work that explores the use of nanowires and molecular systems to build self-assembled computing logic [Collier, 1999; Heath, 2001; Kovtyukhova, 2002; Melosh, 2003]. These studies have focused on simple molecular-scale

[1] The 3-SAT problem involves determining if there is some assignment of variable values that make a given sum-of-products Boolean equation, with three literals per clause, true. This problem is provably NP-complete.

memory elements in crossbar junctions and have not investigated the difficult problem of how to scale to more complex devices.

There is work that has simulated specific molecular electronic components in computing circuitry that elaborate on the feasibility of certain design spaces [Goldstein, 2002]. Most of this work uses wired-OR logic and requires pull-up resistors that would consume enormous amounts of power if scaled up to 10^{12} components. To date this work has not addressed the complexity scaling issues.

2.4 Fault-tolerant computing

Other work has focused on fault-tolerant circuit designs that are amenable to the imperfect circuit yields predicted in nanotechnology [Heath, 1998]. The Teramac is one such fault-tolerant machine made from faulty FPGAs that uses mapping routines to determine fault locations. Once a fault is detected, the silicon compiler is instructed to avoid that particular fault when allocating gates. These and similar techniques will be critical to building large, connected computers. This dissertation skirts the whole issue by decoupling the processors and using content-addressed random indexing[2] described in chapters 3 and 6.

2.5 Quantum-dot cellular automata (QCA)

The application of quantum dot structures to computing has become feasible since the advent of high-density quantum-dot fabrication methods. Through electrostatic interactions quantum-dot cells (QCs) change state according to well defined rules. Cells can be placed next to each other to create a network that computes a logic function. The limitations of QCAs comes from their sensitivity to small charge fluctuations which requires that they be operated at very low temperatures (< 80K.) A thorough overview of QCA methods can be found in [Lieberman, 2002]. The self-assembled nanorod circuitry described in chapters 3 and 9 can be operated at room temperature (and above) and is less sensitive to electric charges than QCA circuitry. Again, scaling QCAs up to larger problem sizes has not been

[2] Content-addressed random indexing is a method that uses random constants to index uncoupled arrays of processors and uses the constants to calculate a value (i.e. content) for the processor to store. The processor can then use the calculated value to index the original constant like a content-addressable memory.

investigated - doing this in 3D might be competitive to the DAMP, but the issue of how to build it remains an open problem.

2.6 Quantum computing

The result of a quantum computation is very similar to an exhaustive classical search of the same problem space, with the exception that quantum computing has a limited set of operators compared to classical computing (e.g. controlled-NOT, but no strict copy, etc.) A quantum bit (qubit) is represented by the physical state of a particle. The particle could be an electron, or hydrogen atom, or even molecular substituents of a larger molecule. Basic properties such as spin or magnetic moments are used to represent the discrete values of the qubit because they have only two possible values (spin up and spin down, etc.) Since the qubits are represented by quantum phenomena, a set of qubits that are entangled (i.e. were once in close contact) form a superposition of all possible qubit values. That is, the quantum nature of the data representation allows an entangled set of qubits to represent all possible binary combinations, so long as there is coherence between the states. The act of measurement typically breaks the coherence in the system and collapses the set to a particular state from the coherent superposition of all possible states. There are operations that can be performed on the set of qubits that do not disrupt the coherence of the system. Thus, the system remains entangled and the result is a new superpositioned state of all possible outcomes of the operation. A sequence of operations is used to perform useful computations on the system, such as the Shor factorization algorithm. The grand promise of quantum computing is that a 500-qubit system can enable an operation to work with all 2^{500} binary states of that system in parallel, at once! This is equivalent to performing a classical search for a solution from 2^{500} possible input combinations.

Most of these quantum computing methods use nuclear magnetic resonance (NMR) to operate on the entangled quantum states of molecules and have only been used to manipulate ~7 qubit systems to date. Advances in non-NMR based quantum computing may make it possible to manipulate larger systems of qubits with very low power consumption. Experiments to manipulate 5-10 qubits have been suggested [Brown, 2002].

The coherence problem of quantum computing (i.e. the inability to make entangled systems with more than 10 qubits) is currently unsolved. The oracle architecture described in

chapter 7 can be used to assemble the solution to many problems with a 40-bit input space. This is equivalent to a 40-qubit quantum computer. The self-assembled circuitry described in this dissertation also enable classical architectures like the decoupled array multiprocessor (DAMP.)

Chapter 3. 3D Nanoscale circuit fabrication

3.1 Introduction

We focus on the realization of a new computer architecture that is enabled by the development of DNA-guided self-assembled systems. The enormous parallelism and scale of this kind of self-assembling process has motivated research into novel forms of computation that use the intrinsic properties of DNA hybridization to form solutions to a problem [Roweis, 1998; Winfree, 2000]. We take a slightly different approach to developing computing devices using DNA: instead of depending on the computability of DNA hybridization events to do the computation, we investigate the structural use of DNA to create electrically active nanoscale rod-lattice structures. That is, rather than using DNA to assemble and to form the solution to one instance of a problem, we use it to assemble a computer that contains 10^{12} such solutions. The computer can then produce results every few microseconds vs. the weeks per solution taken by DNA computing techniques.

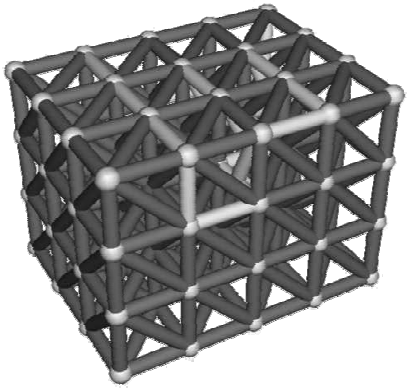

Figure 3.1. A DNA-guided assembly of rods that form a NAND gate.

The basic approach is to design 3D computing circuitry, as illustrated in figure 3.1, that is constructed using a series of DNA binding events to assemble nanorods. Each rod

may be an insulator, conductor, or a novel type of transistor described in section 3.2 and chapter 9. These basic 3D blocks are themselves assembled into larger structures to form computing elements. These elements can be connected to power and signal leads using massively-parallel self-assembly to produce a computer consisting of 6×10^{12} processors, of which an estimated 10^{12} will be functional.

Our method of constructing computing circuitry from nanoscale self-assembled components requires several capabilities described here. These capabilities include the functionalization of rod-like nanowires with DNA, DNA metallization, DNA-guided self-assembly, and a novel nanoscale transistor. We provide an overview of the capabilities here and point to more complete discussions elsewhere in the document.

Section 3.2 outlines the fabrication and methods of low-level assembly. Section 3.3 describes the nanorod synthesis and section 3.4 describes the structures that will be formed. Section 3.5 and 3.6 develop the assembly method further. Section 3.7 discusses the design of DNA strands and the assembly tolerances, and section 3.8 outlines the yield estimates for each DNA junction.

The components fabricated from the low-level assembly described here are used in a fluidic self-assembly method to create larger circuits as described in chapter 6. This method overcomes the low yield predicted in the DNA-guided self-assembly process. Since each estimate made here, and throughout this work, chooses the worst-case assumptions it is expected that these values (e.g. for yield, performance, power consumption, etc) are strictly lower bounds and that optimization in the manufacturing process can dramatically improve the overall performance of the technology.

3.2 Background

The following section is a brief primer on the use of nanotechnology in building electronic circuitry. The topics discussed here relate the basic properties of nanorods, DNA, attachment chemistry, conductivity, and transistors to self-assembly. Readers already well versed in such topics will likely find this section to be tedious and may wish to skip it entirely. A basic text on the biochemistry of DNA may be useful for other readers [Lewin, 1997].

Nanorods

Controlled self-assembly of nanoscale circuitry requires the ability to control the properties of individual components of the structure. Recent advances in silicon nanowire doping have proven that small nanoscale rods (> 50 nm diameters) can be doped controllably and can be made to behave like bulk semiconducting materials [Cui, 2000].

DNA

As mentioned in chapter 2, DNA can be used for more than just storing genetic information within biological organisms. The two forms of DNA that are important to this discussion of self-assembly are single-stranded and double-stranded DNA. Watson & Crick proposed the structure and means by which single-stranded DNA forms double-stranded DNA on April 2, 1953. Single-stranded DNA is a string of *nucleotides* (or chemical bases) that are attached to a sugar-phosphate backbone. Figure 3.2 illustrates the major nucleotides found in DNA.

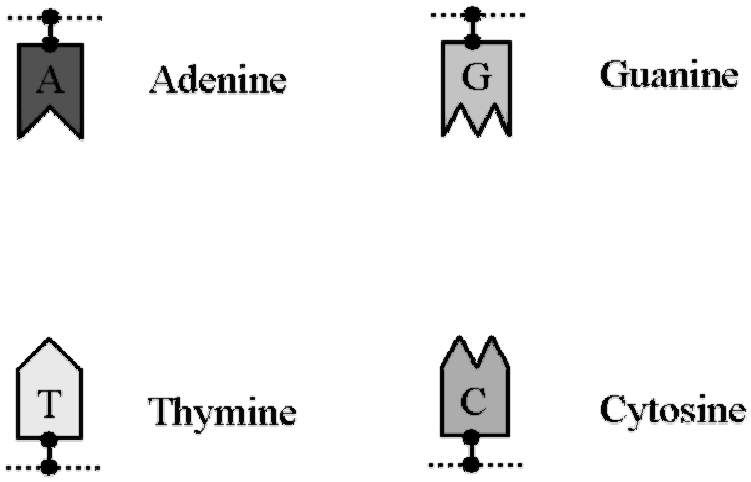

Figure 3.2. Schematic representation of the major DNA nucleotides. The dashed lines represent a backbone used to string nucleotides together.

Any of the nucleotides in figure 3.2 can be used in the sequence of a single-stranded piece of DNA. The mechanism that single-stranded DNA uses to form double-stranded DNA (in the shape of a double helix) is called *hybridization*. DNA hybridization is the

"coming together" of two single-stranded DNA fragments. However, there are rules about which single-stranded DNA fragments hybridize with each other.

The adenine and thymine nucleotides happen to prefer sticking to each other rather than to either guanine or cytosine. This is called *complementarity*. That is, adenine (A) and thymine (T) are complements. The same applies to guanine (G) and cytosine (C). Single-stranded DNA sequences can therefore be complements if for every occurrence of *A* or *G* in one strand there is a *T* or *C*, respectively, in the complementary strand, and vice-versa. Only complementary single-stranded DNA fragments hybridize to form double-stranded DNA. Complementarity is shown here through the use of single and double notches.

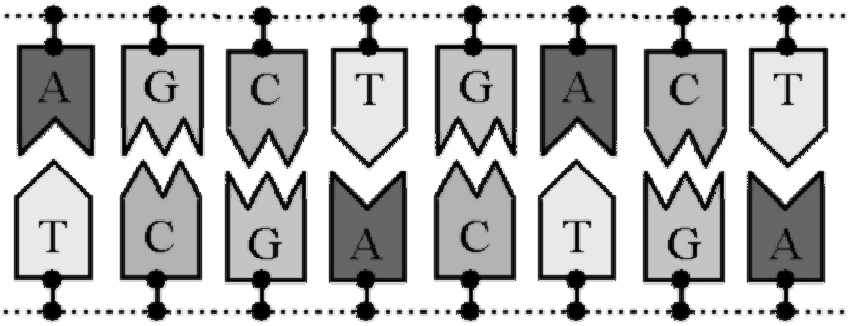

Figure 3.3. Schematic representation of two complementary single-stranded DNA fragments forming a double-stranded fragment.

The ability of complementary single-stranded DNA to form only the correct double-stranded DNA is also known as *specificity*. That is, DNA fragments that are not complements will not hybridize. The degree of specificity is dependent on the length of the DNA fragment, with 8 - 12 base fragments being optimal. Very long fragments of DNA hybridize with less specificity than shorter fragments.

The hybridization of single-stranded DNA into double-stranded DNA is temperature dependent. A mixture of double-stranded DNA can undergo a sharp transition to a mixture of single-stranded DNA (i.e. separate complementary strands) by raising the temperature of the mixture above the *melting* temperature of the DNA sequence. That is, above the melting temperature of a given fragment of DNA the fragment will not hybridize with its complement. The transition from single-stranded to double-stranded DNA is reversible by

lowering the mixture's temperature below the melting temperature. Cycling the mixture's temperature above and below the melting temperature is commonly used to improve hybridization specificity because only the most stable (i.e. complementary) interactions between strands are likely at these temperatures. Again, please refer to an introductory biochemistry text for a complete discussion of this topic.

DNA attachment

Nanowire-DNA *functionalization*, or chemical attachment, is the first step in implementing a DNA-guided self-assembly process. Our method requires the rod-like nanowires to have unique DNA sequences attached to each end. Section 3.5 discusses how to design the DNA and nanowire properties to assemble computing circuitry. The DNA-directed formation of nanowire-patterned surfaces has been reported and provides insight into how such nanowires can be functionalized [Mbindyo, 2001].

Our own work functionalizing carbon nanotubes with DNA [Dwyer, 2002], described in appendix A, also provides insight into the available attachment mechanisms. The most promising schemes employ nanorods formed in membranes or structures that can protect one end of the rods from reactions occurring at the other end. Such asymmetry in the reaction of the rods is important in controlling the sequence of the DNA strand on each end.

An important quality of DNA that makes it most suitable for self-assembly is its ability to hybridize with its complement with very high specificity. Consider the 4^8 different 8-base DNA sequences, for which there are 65,536 nearly orthogonal reactivities. This is a vast improvement over the handful of specific covalent chemical reaction schemes that are readily accessible using present-day organic chemistry.

Remarkable work has been undertaken in the effort to produce DNA assembled structures. Many of these efforts have focused on the structures created by clever designs of DNA sequences undergoing interesting thermodynamic transitions [Yan, 2002; Seeman, 2001]. Still others have focused on the formation of ordered superlattices made from nanorods [Mbindyo, 2001; Dujardin, 2001]. The experimental demonstration of mesoscopic DNA-guided assemblies is also of interest [Soto, 2002]. These results imply that there is considerable promise in the DNA-guided self-assembly of large-scale molecular structures.

Conductivity

The ability to convert double-stranded DNA into a highly conductive ohmic contacts (by a process known as *metallization*) makes the use of DNA in nanoscale circuitry extremely attractive. This work has shown that DNA can be used as a backbone for the formation of highly conductive nanowires with conductances greater than 1.4×10^{-3} S for micron long nanowires [Braun, 1998; Richter, 2001]. These techniques form a coating of metal by allowing positively charged metallic ions to coalesce around the negatively charged double stranded DNA. Such metallization techniques are suitable for either surface bound or suspended DNA strands. We anticipate that the DNA used to form our 3D self-assembled structures will exist in a suspended form similar to what is reported in [Braun, 1998].

Transistors

We have invented and evaluated a new kind of transistor that we call a ring-gated field effect transistor (RG-FET) for use in the context of self-assembled structures. Figure 3.4 illustrates the basic structure of the RG-FET.

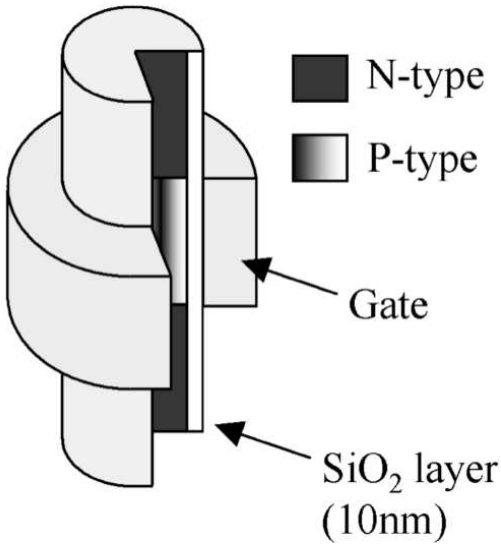

Figure 3.4. The basic structure of the ring-gated field-effect-transistor.

We have simulated the behavior of this kind of transistor in complementary metal-oxide-semiconductor (CMOS) logic circuits [Dwyer, 2002]. The details of these simulations are found in chapter 9.

We have also briefly explored the plausibility of fabricating such a transistor by using an electron beam lithography technique to form a nanoporous polymer surface. Figure 3.5 is a scanning electron microscope image of AuPd rods that we formed projecting out of a poly-methyl-methacrylate (PMMA) surface. Similar work in vapor-liquid-solid phase nanowire growth has uncovered promising synthesis methods [Lew, 2002]. The route that we expect will most likely lead to successful patterning of the rods is illustrated in figure 3.6.

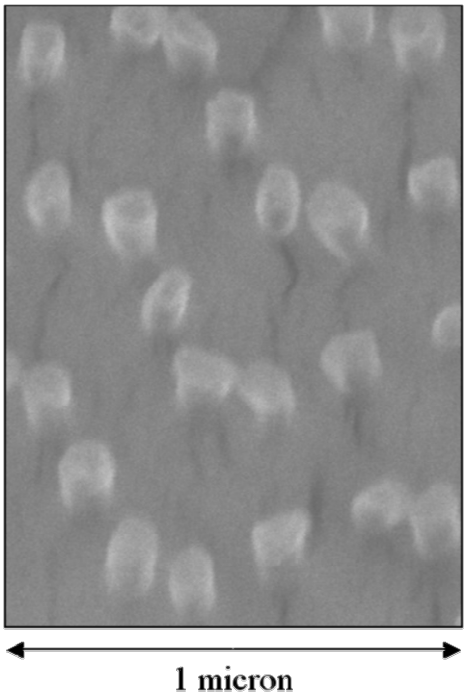

Figure 3.5. A scanning electron microscope image of AuPd rods protruding out of a poly-methyl-methacrylate (PMMA) surface.

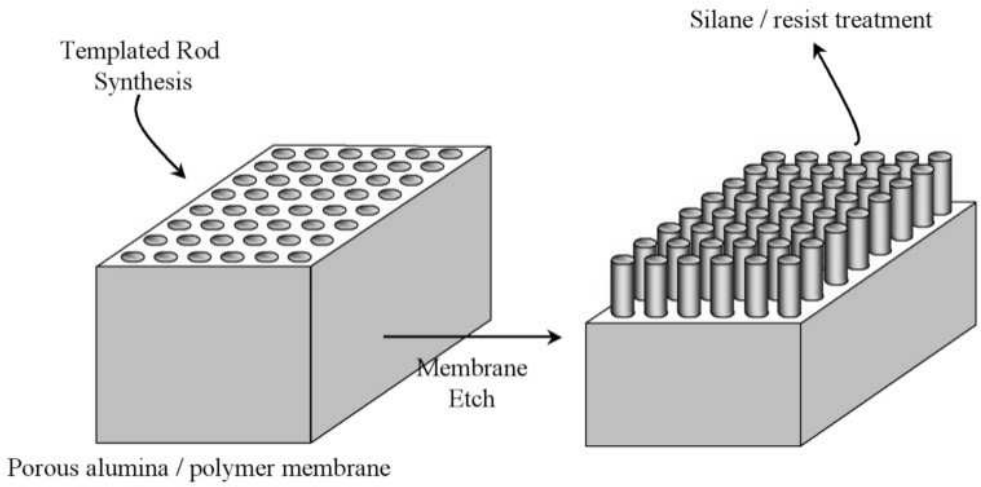

Figure 3.6. RG-FET synthesis scheme. Repeated membrane etching and rod surface treatments can be used to form banded structures along the rod.

The process begins by forming rods in a membrane (either ceramic or polymer) and using a selective etch to expose a portion of the rods. Using silane, polymer, or other resists, the top portion of the rods would be modified and protected from subsequent etching steps. A negatively charged silane monolayer could then be used to form a band around the rod that could be processed to create a metallic ring as reported by Richter [Richter, 2001]. Ring-gated structures similar to these have been formed on surfaces and their electrical properties have been measured [Lauhon, 2002; Gudiksen, 2002].

3.3 Nanowire synthesis & self-assembly

The synthesis of nanorods and nanowires is the first step in our process of fabrication for complex computing circuitry. The template directed synthesis of nanowires is of particular interest because such templates permit the asymmetric functionalization of the nanowire [Lew, 2002; Martin, 1996]. Such asymmetry is important in the control of how the nanowires attach to other objects, including surfaces and other nanowires. Our own work in the functionalization of carbon nanotubes underscores the difficulty in asymmetrically functionalizing rod-like nanoparticles without templates [Dwyer, 2002b]. Research in the

area of nanowire / surface interactions has produced valuable mobility information that makes it possible to estimate the yield of more complex structures [Martin, 2002].

The DNA-guided self-assembly of nanoparticles into complex structures began with pioneering studies into the formation of artificial geometric structures made from DNA [Yan, 2002]. The more interesting applications of DNA-guided self-assembly to computer design came in the form of the DNA-guided assembly of nanowires. Several landmark studies have shown that it is possible to assemble nanowires (and other nanoparticles) using DNA [Dujardin, 2001; Mbindyo, 2001; Soto, 2002]. These structures are rudimentary compared to what is needed by the designs presented in chapters 3 - 7, but demonstrate promising success in controlling self-assembly.

3.4 Cubic unit cell assembly

Figure 3.7 illustrates a simple cubic unit cell with diagonal supports. The particular function of any unit cell is determined by the electrical properties of each rod. By using the RG-FETs and rods described earlier, a cell can be combined with others to form logic circuitry.

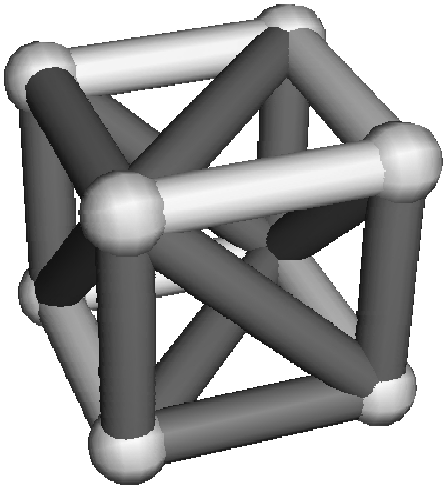

Figure 3.7. A cubic unit cell with diagonal supports (crossbars). The golden (light gray) rods are conducting and the dark gray rods are insulating.

The logic circuitry is first specified using a standard complementary metal-oxide-semiconductor (CMOS) logic style, as in figure 3.8. The NAND gate shown in figure 3.8 takes its two inputs, A and B, and produces an output of zero if and only if the two are both one (V_{dd}). This gate represents one of many complete logic sets because sets of NAND gates can be used to implement any Boolean logic function.

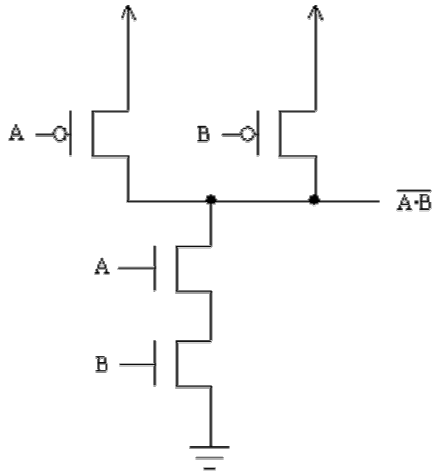

Figure 3.8. A CMOS implementation of a NAND gate

The circuitry of the NAND gate can be converted into a 3D structure suitable for self-assembly, illustrated in figures 3.9 and 3.10. The procedure described in section 3.5 for the formation of a triangular structure can be extended to form such a rectangular solid as this logic gate. One challenge is that the number of unique DNA sequences that a fully parallel self-assembly method requires scales with the surface area of the structure. A fully-parallel method would require a set of unique complementary strands for each junction; the entire structure could then be formed at once by mixing all of the functionalized rods.

This will only work if there are few internal rods that can be shielded from their assembly points by external rods that assemble prematurely. That is, if the outside of a 3D structure assembles before its inside, the rods will be unable to reach their intended positions and the core will not assemble correctly. This leaves the structure empty inside. To avoid this problem, rods must be assembled (inserted into the mixture) from the inside to the outside by sequential ordering.

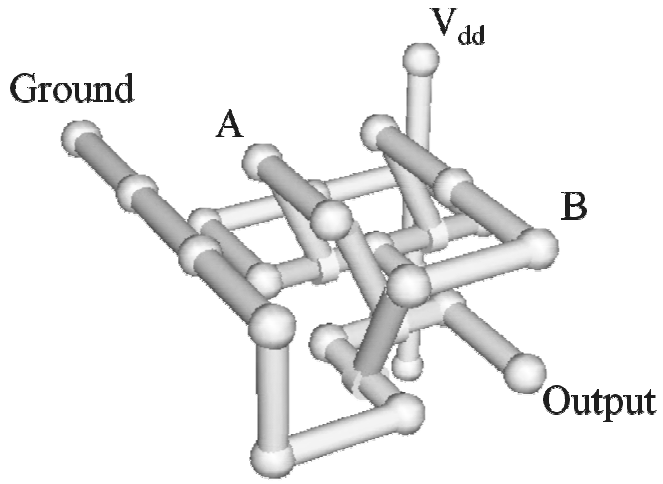

Figure 3.9. The conducting portions of the 3D structure for a NAND gate.

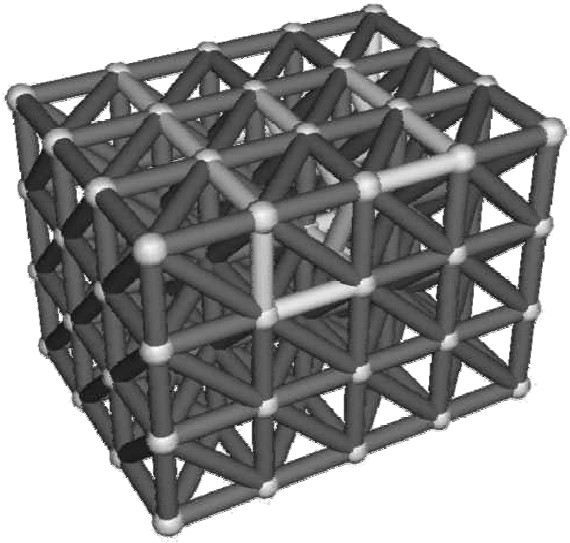

Figure 3.10. The physical 3D structure of a NAND gate embedded in insulating unit cells for structural support.

With a fully parallel approach, the number of unique DNA sequences needed by even a simple memory element (256 bits) could easily reach tens of thousands. Fortunately, it is possible to reduce the number of unique DNA sequences required to assemble a structure by using a face-serial approach. In this approach, each face of the structure is assembled in a

serial fashion. Since each face is assembled independently, different faces can share a common set of "active" DNA sequences. Within a face, the assembly moves from left-to-right, top-to-bottom. Figure 3.11 illustrates the assembly sequence.

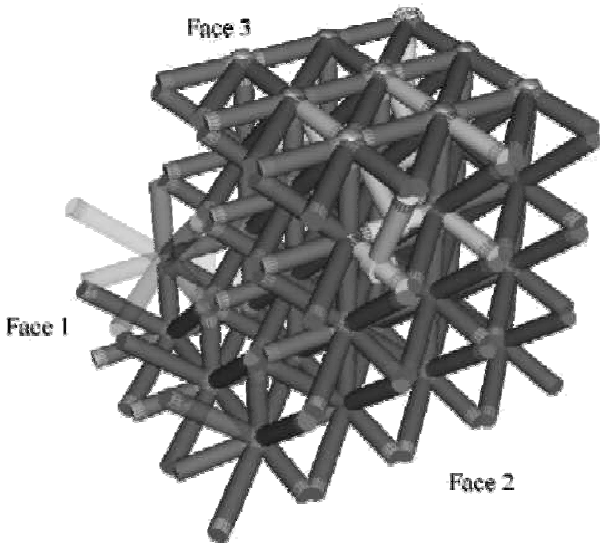

Figure 3.11. Face-serial assembly of a 3D structure. The assembly begins first on face 1 (partially completed in this figure), then face 2, and then face 3. The process repeats until the structure is complete.

An implementation of the assembly algorithm can be found in appendix B in the file AssemblyProgDlg.cpp in the function `void CAssemblyProcDlg::AssemblyPattern2()`. Each face is assembled using an alternating set of DNA sequences. A rod will only assemble between two other rods if it has DNA sequences on its ends that are complementary to both. The choice and size of the number of unique DNA sequences is related to graph coloring because adjacent rods must have different DNA sequences (or colors in this case.)

Since a common set of DNA sequences is shared among faces as well as among sites within a face, the total number of unique DNA sequences is fixed and independent of the surface area or volume of the structure. Our designs use 15 unique DNA sequences for this face-serial method. Table 3.1 contains the counts of our assembly method for several logic circuits. Figure 3.12 illustrates the NAND structure as viewed when each unique DNA strand is given its own index. The repetition among rows on each face is apparent and indicates that the total number of unique DNA strands is fixed. The trade-off for the face-

serial method versus full self-assembly is the increased number of steps and time required by the processing steps. Many structures can be constructed in parallel, as described in section 3.4, so this method still enables massively parallel self-assembly.

Since this method assembles the structure serially, the time required to build a structure is linearly proportional to the number of rods in the structure. It is difficult to predict the amount of time that each rod will require to assemble properly. The process can likely be automated to a high degree and optimized to improve the ~10 minute assembly times from [Dujardin, 2001]. If the per-rod assembly time can be optimized to less than 1 minute, the modular designs presented in section 3.6 and chapter 6 with ~800 rods per module would take ~13 hours to assemble.

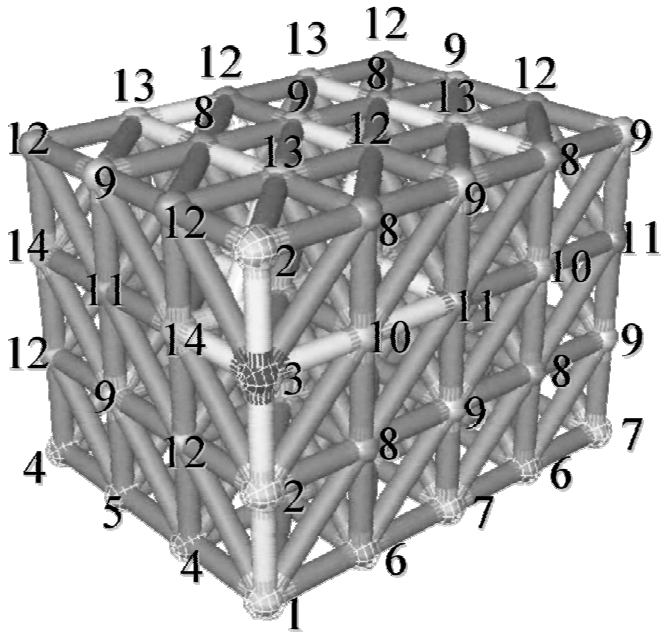

Figure 3.12. The 3D structure of a NAND gate with each unique DNA sequence represented by a different color (and number). Sequence 15 is not visible here because it happens to appear on the other side only.

Logic Gate	Total Rods	Metallic	Insulating	Diagonal Struts	RG-FETs	DNA Sequences*
NOT	90	5	47	34	2	26
NOR	252	18	122	104	4	55
NAND	328	18	164	138	4	63
XOR	522	86	190	214	16	77
Full Adder	1722	273	641	732	38	158

Table 3.1. Assembly statistics for several logic circuits. * The number of DNA sequences needed if the conserving allocation method is not used. The conserving allocation method needs 15 unique sequences.

The values for table 3.1 came from a custom assembly tool (described in chapter 5) we developed for converting 3D circuit specifications into rod-DNA allocations. As the logic circuitry becomes more complex, the number of required unique DNA sequences increases. This underscores the importance of the DNA conserving, face-serial assembly method described above.

3.5 Assembly method

Our proposed method for constructing computing devices employs the assembly of simple cubic unit cells (with diagonal supports) using DNA-guided self-assembly. Control over the electrical properties of the assembled structure comes from the choice of the electrical properties of the individual rods in the structure. For this purpose we have developed custom software for designing the 3D circuit layout of logic gates. The software automatically generates a list of rod types and the DNA sequences required on each end to form the 3D structure. First, it is important to understand our proposed assembly process before examining the algorithms used in the design software. As an example of the assembly process, let us consider the assembly of a simple three-rod, triangular structure.

Figure 3.13 illustrates the steps involved in the process. The process begins by hybridizing a rod with the solid support (or anchor). The solid support is used to anchor the intermediate structures during the cycling of reactants and rods. The first rod has DNA on one end that is complementary to a region of DNA on the solid support. The DNA sequences

attached to the solid support are extended away from the surface by a polymer arm that has a sufficiently negative linear charge to metallize DNA [Richter, 2001].

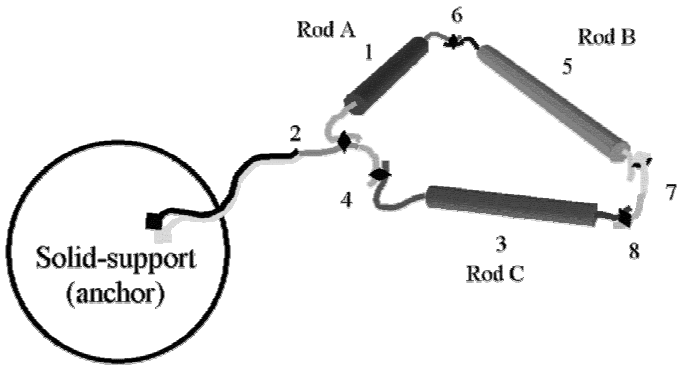

Figure 3.13. The assembly of a triangular rod structure. Each number indicates the order of operations. 1: Hybridize one end of rod A's DNA with the solid support's DNA. 2: Cross-link the duplex DNA. 3: Hybridize one end of the rod C's DNA with the solid support's DNA 4: Again, cross-link the duplex DNA. 5: Hybridize one end of rod B's DNA with rod A's free end. 6: Again, cross-link the duplex DNA. 7: Hybridize the "coupling" DNA strand with the DNA on rod B's free end. The coupling strand is made to site specifically cross-link to rod B's DNA. 8: Hybridize the other side of the coupling strand with the DNA on rod C's free end and cross-link the duplex DNA. DNA metallization can occur after the structure is complete to form conducting paths between the rods.

The hybridization event between the first rod and the solid support is carefully controlled to maximize the specificity of the interaction. By raising the temperature of the system above the temperature at which DNA strands separate from their complements (i.e. the melting temperature) of the DNA and then slowly cooling it back to room temperature, we can ensure a high degree of specificity between the DNA strands (i.e. complementary strands hybridizing with each other, and each other only).

Non-specific rod-rod interactions (rods sticking to each other when they should not) may interfere with this intended interaction but similar silicon particle systems have been developed that minimize this interaction [Martin, 2002]. Section 3.8 discusses this further.

After the hybridization event, the duplex DNA is cross-linked using cisplatin or some other cross-linking agent to "cement" the connections (forming covalent bonds) so that they stay connected during later processing steps. As the process proceeds, each hybridization

event is carried out under these same conditions to maximize specificity. This is important for correctly assembling structures with high yield. Assembly yield is discussed in section 3.8 and chapter 4.

A fluid containing a concentration of the second rod, which has DNA on one end that is complementary to a second region of DNA on the solid support, is flushed past the solid support. Under the same stringent conditions, it is allowed to hybridize with the solid support. Again, the duplex DNA is cross-linked to form a stable and covalently bound intermediate structure.

Unfortunately, the addition of the third rod is not as simple as the previous two. If we were to add a solution containing the third rod type with DNA on each end that was complementary to the first and second rod respectively, a triangular structure could form. But an open four-rod structure would also form with relatively high probability. This is because two rods of type 3 could hybridize independently with both the first and second rods, one on each. To avoid this ambiguity we need to introduce a "coupling" DNA strand.

The third rod is made so that it is complementary on one end to the first rod. The other end of the third rod is made to complement one side of a coupling strand. The third rod is hybridized with the first rod as described earlier. The coupling strand is made to complement the free end of the third rod and the free end of the second rod, with one modification: the portion of the coupling strand that hybridizes with the third rod has a psoralen-modified nucleotide, or some other site-specific mutagen. This modification ensures that the coupling strand irreversibly binds *only* to the free end of the third rod [Qiagen, 2003].

The coupling strand is hybridized with the third rod, as before. After the coupling strand has been cross-linked to the third rod irreversibly, the system's temperature is raised above the melting temperature of the coupling strand and the site is rinsed with buffer[3]. Upon cooling, the coupling strand that was bound to the third rod will hybridize with the second rod. This unambiguously closes the gap and forms the triangular structure. Cross-

[3] A buffer is an aqueous solution of salts that maintain a constant pH (acid or base) in the solution as other chemical reactions take place (that might otherwise change the pH.)

linking the duplex DNA again forms a covalently bound and stable structure. Metallization of the DNA can occur anytime after the structure has been formed.

3.6 Fluidic self-assembly

For reasons explained in chapter 4 and 6, simple DNA-guided self-assembly is not likely to have sufficiently high yield (in the near term) to form structures large enough to perform useful computations. As the processing techniques mature this yield may increase, making it possible to assemble larger structures. In the meantime, it is necessary to also use an intermediate form of self-assembly known as fluidic self-assembly.

Thin structures made using the DNA-guided self-assembly method described earlier can be used to form larger composite structures by stacking. Figure 3.14 illustrates the stacking of thin structures. Any circuit design that is too large to assemble with DNA-guided self-assembly can be divided into portions, or slices. DNA-guided self-assembly is used to fabricate each unique slice and the slices are then stacked on top of each other using fluidic self-assembly to form the circuit.

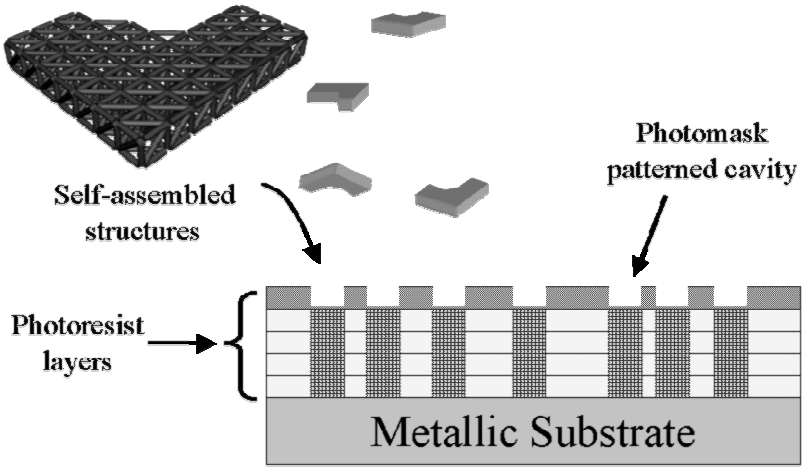

Figure 3.14. Fluidic self-assembly is used to stack thin structures to form a larger composite structure.

Fluidic self-assembly is simpler to understand than DNA-guided self-assembly because it relies on physical phenomena that are observable at the macroscopic (human)

scale. The application of a shear force in a fluid flow is what drives fluidic self-assembly. Each component is shaped so that it can only minimize the force acting on it from the fluid when it is tucked snugly into the proper hole. The flow must be weak enough so that it does not pull components from their holes, yet strong enough to drive each one into a hole. An example of this can be found in an industrial process where pyramidal solids are assembled into pyramidal "divots" in a silicon surface [Alien, 2003]. This scheme works in the same way that a traveler might use a doorway to take cover from the wind of a storm. Our use of fluidic self-assembly is described in section 6.1.

Similar work studying the effects of capillary forces[4] on the self-assembly of particles has illuminated many of the interesting properties of self-assembly [Srinivasan, 2001]. Among these properties is the influence rotational asymmetry has on the ability of a particle to assemble. While any child with a set of blocks and a peg board can tell you it is more difficult to place a square into the board than a circle, these studies have provided the first glimpse of why this is so at the mesoscopic-scale (between the macroscopic and nanoscopic worlds.) Further, rotational asymmetry can be used to control the placement of nanoparticles using a key and hole approach.

Other work has shown how capillary forces between metallic alloys can be used to assemble millimeter-scale particles [Clark, 2001; Clark, 2002; Gracias, 2000]. The particles are driven to assemble due to the same surface free energy minimization[5] phenomenon that creates a water meniscus in glass. The particles can reduce the strain on their surfaces by maximizing their interfacial contact areas, thus aligning themselves.

3.7 DNA strand design & assembly tolerances

The geometry of the assembled structure, and the tolerances allowed during that assembly, affect the design of the individual DNA strands attached to each rod end. The

[4] Capillary forces are what create menisci and curved droplets on flat surfaces. Surface free energy minimization drives these forces to maximize the contact area between the two materials.

[5] Surface free energy minimization is a phenomenon that helps to describe the behavior of surfaces in contact. Dangling bonds at the surface of a material create strain that can be reduced when they are brought into contact with a lower energy surface.

shape of the rods used also affects the DNA strand design. These issues are not difficult to analyze but are important to consider.

The basic rod used in the assembly process is a 50 nm diameter rod with a length of approximately 500 nm. (Typically, rod diameters deviate by only 1% to 10%, depending on template material, which implies that for a 50 nm diameter rod the deviation is negligible and can be ignored.) Tolerances in the synthesis of each rod will cause a non-negligible deviation from the average length. If we consider the distribution of lengths for any particular rod to be Gaussian, then to use 99% of the starting material, rods within $3 \cdot \sigma$ on either side of the mean must be accommodated by taking slack from the DNA strand. This is the only way a cubic cell can be formed from rods of uneven length. That is, the joints of a cube (with diagonals) made from rods with non-uniform lengths must somehow accommodate shorter or longer than average rods and remain closed. The elasticity of the joints determines how far from the average lengths an edge or diagonal rod can be and still form a closed cube. Therefore, the length of the DNA strands must accommodate the distribution of rod lengths and still form highly specific hybridization bonds with complementary strands.

If the DNA strand is simply made longer, these two requirements become mutually exclusive because longer DNA strands hybridize less specifically. It is necessary to reduce the length of the hybridizing portion of the DNA strand to about 12 bases to maintain high specificity [Mbindyo, 2001]. Unfortunately, 12 base pairs of DNA only stretch to 4 - 8 nm — far shorter than the ±10% (100 nm for a 500 nm long rod) seen in typical distributions of rod lengths [Mbindyo, 2002; Martin, 1996].

The use of a phosphorothioate oligomer (a so-called s-oligo) of universal bases (3-nitropyrrole 2'-deoxynucleoside) as a spacer is attractive. There are also other polymers that can be used to extend the DNA oligo without reducing its specificity, e.g. carbon polymers. The s-oligo will not hybridize with any other DNA strand because of the properties of universal bases but will retain the electrostatic and solution-phase properties of natural DNA [Loakes, 2001]. This property is important during the post-processing metallization step to fully metallize the DNA junctions.

The s-oligo is attached to the rod end and the 12 base pair DNA strand attached to the free end of the s-oligo. This synthesis can take place during the manufacturing of the DNA strand and is likely to be an easily-purified product [Loakes, 2001].

The extension, either an s-oligo or another polymer, accommodates uneven rod-length distributions by allowing the DNA junction to stretch. Single-stranded DNA is flexible and will allow rods to rotate relatively freely about a junction. In the following section, the specific geometry of a rod junction will be discussed in detail. We must consider some basic geometric properties to design the DNA and s-oligo strands.

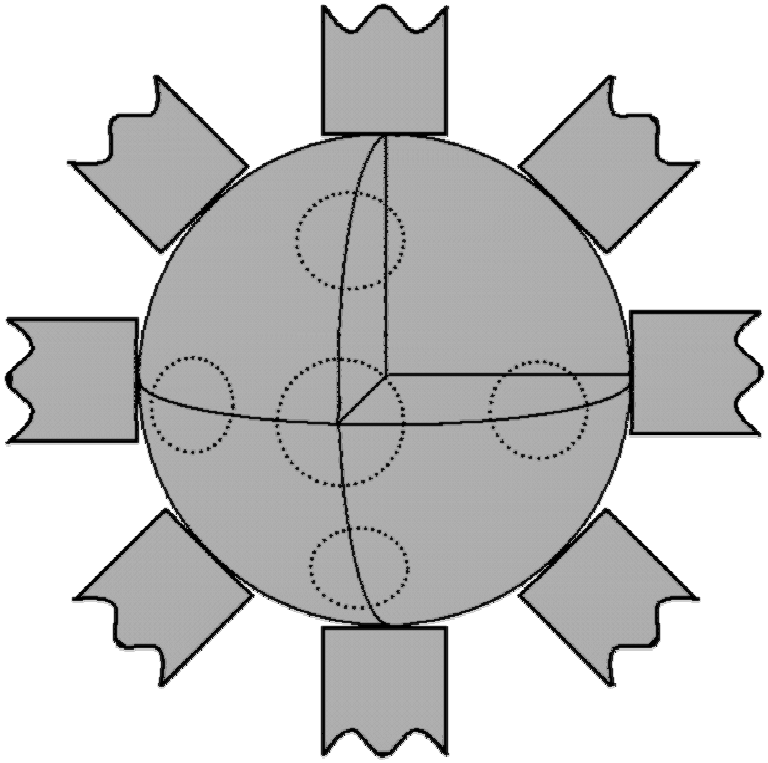

Figure 3.15. A junction sphere with eight perpendicular rods along each orthogonal equator. A total of 18 rods can fit around a junction in this manner.

The number of rods that can join at a junction is illustrated in figure 3.15. Each of the 18 rods that could join at a junction must be able to physically fit around the junction. It is helpful then to think of the junction as a sphere having some radius that defines the closest

approach of any surrounding rod. Figure 3.16 illustrates the constraint on the junction sphere's radius along a quarter of one of the orthogonal equators.

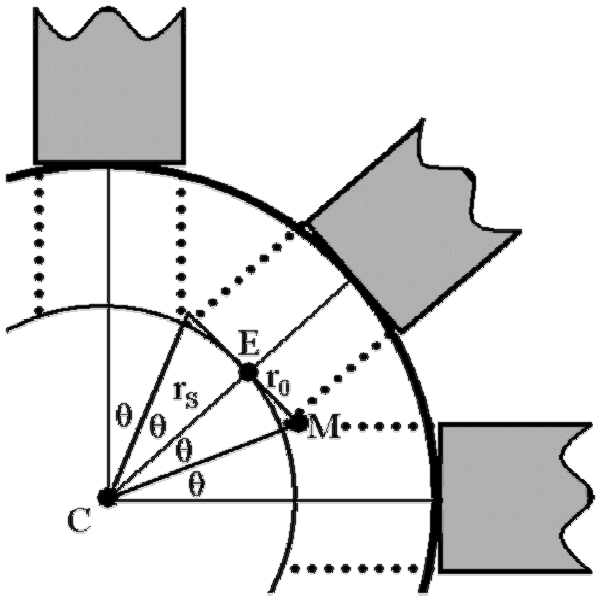

Figure 3.16. Geometry of the rod junction along a quarter of an equator. r_0 is the radius of each rod and r_S is the radius of the junction sphere. Only three rods are shown.

We need to know the minimum junction sphere radius to estimate how long the s-oligo extension must be to make a junction feasible. A simple way to find the minimum junction sphere radius, r_S, is to consider a junction sphere larger than it needs to be to fit all the rods, as represented by the outer quarter-circle in figure 3.16. While keeping the rods fixed to the circle, collapse the outer circle until the rods begin to bump each other. The radius of the inner circle is the smallest that still allows the three rods to remain fixed to the circle from the center of each rod end and perpendicular to the arc.

The angle θ in figure 3.16 is the angle between the colliding edges of two neighboring rods. Symmetry requires these angles to be the same. When the junction sphere is smallest the sum of these angles for one quadrant is 90˚, so:

$$4 \cdot \theta \leq \frac{\pi}{2} \qquad (3.1)$$

That is, θ can be no greater than π / 8 when all the rods fit around the junction. Using this we can calculate the relationship between θ, r_0, and r_S. The right triangle CME in figure 3.16 gives us the relationship:

$$\tan\theta = \frac{r_0}{r_S} \tag{3.2}$$

Substituting (3.1) into (3.2) and rearranging the result we have:

$$r_S \geq \frac{r_0}{\tan \pi/8} \tag{3.3}$$

That is, equation (3.3) states that the junction sphere radius must be approximately 2.4 times the common radius of the rods that join at that junction. For the rod dimensions considered here, 25 nm radius by 500 nm length, the junction sphere must have a radius of no less than 60 nm. Since s-oligo lengths are 0.34 - 0.7 nm per base (each base can stretch), we need a s-oligo with no fewer than 85 - 176 bases (poly-universal.)

This puts a lower limit on the length of the s-oligo extension. It must be at least 85 - 176 bases just for the junctions to be feasible for 25 nm radius rods. However, for any larger structure to be feasible, the extension must be able to give slack to rods of uneven lengths. That means the s-oligo must be long enough to connect the shortest rods to a junction. Figure 3.17 illustrates the worst-case situation.

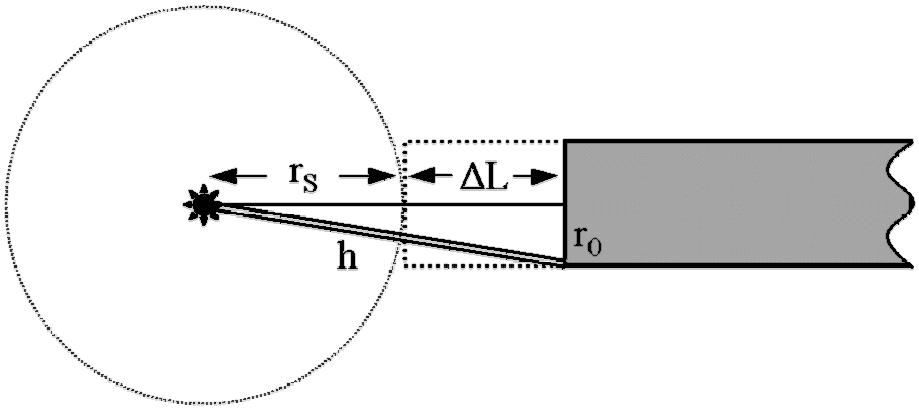

Figure 3.17. The worst case rod-length and DNA strand placement scenario. The circle represents the junction sphere and ΔL is the length discrepancy. The variable h is the length of the s-oligo and DNA strand that can accommodate the length discrepancy and still join at the center of the junction sphere (marked with a star.)

The triangle made from the line segments $r_S + \Delta L$, r_0, and h leads to the following:

$$h = \sqrt{r_0^2 + r_S^2 + 2 \cdot \Delta L \cdot r_S + \Delta L^2} \qquad (3.4)$$

The earlier discussion has fixed the value of h (from figure 3.17) to be larger than 60 nm since it must be at least that large to make the junction feasible if all the rods touch the junction. The additional length (beyond 60 nm) will depend on the length distribution of the rods. For example, if we assume that a 10% (or ±5%) variation in length covers 99% of all the rods in a particular distribution, then ΔL = 50 nm for a 500 nm rod. This length distribution is common for template-directed nanorod growth [Dujardin, 2001]. Using the previously established r_S = 60 nm and r_0 = 25 nm, the total s-oligo and DNA strand length, h, must be about 113 nm, or 161 - 332 bases in this case.

The compounding of this error, junction after junction can eventually lead to an infeasible geometry. In fact, two worst-case junctions next to each other will create an infeasible third junction because the adjacent rod would need to be 20% longer than average, which is outside of the range of rod-lengths we have assumed. Since we defined ΔL to be the length discrepancy that covers 99% of all rods, we have approximately a 1 in $(1 / 1\%)^2$

chance that such a situation will arise per junction. This means that no more than 99.99% of all junctions will be feasible.

The junction yield may be improved by grouping rods of similar length together before assembly. Such "binning" of rods based on length will reduce the ΔL that covers 99% of the rods in a bin and therefore increase the per junction feasibility yield.

3.8 Assembly yield

The hybridization efficiency of the DNA strands and the effect of non-specific rod-rod interactions will attenuate the actual junction yield. One particular study places the DNA hybridization efficiency of a 12-mer at around 98% [Pena, 2002]. Since the number of DNA hybridization events per junction is anywhere from 7 to 12 (discussed in detail later), we can estimate the DNA hybridization yield per junction to be in the range of $(98\%)^{12}$ to $(98\%)^{7}$, or [78.5%, 86.8%].

One way to keep non-specific rod-rod interactions to a minimum is by using a hydrophilic silane monolayer to coat the outside of the silicon rods. A related system of gold nanorods interacting with derivatized surfaces has shown that hydrophilic surface treatments greatly reduce the non-specific interaction [Martin, 2002]. This study showed that greater than 99% of all nanorod material remained free from non-specific interactions. The application of this method to a silicon rod system appears straightforward. If so, we can ignore the non-specific interactions and say that the total yield of structures from raw rods is as follows:

$$Y_T = (Y_{DNA} \cdot Y_{JUNCTION})^N \qquad (3.5)$$

where Y_{DNA} is the DNA hybridization efficiency, $Y_{JUNCTION}$ is the junction yield, and N is the number of junctions per structure.

Each of the structures described later can be divided into a number of "slices" with approximately 108 junctions per slice. We can expect no more than $(86.8\% * 99.99\%)^{108}$, or $2.267 \times 10^{-5}\%$, of the total number of possible structures (slices in this case) to have formed properly. This means that about 2×10^{19} raw rods will be required to produce 7×10^{12} final structures (or $\sim 6 \times 2^{40}$). Slicing the each structure into smaller modules, as described in

chapter 6, is important because many of the circuits described in chapters 7, 8, and 9 need far more than 108 junctions. A simple NAND gate from chapter 9 would require ~540 junctions for a final yield of $(86.8\% * 99.99\%)^{540}$, or 6×10^{-32} % if it were not sliced. Chapter 6 describes another form of self-assembly that can overcome this low yield.

Materials

Commercially available material used in fabricating rods by template-directed methods have produced as many as 10^{11} rods per cm^2 of membrane [Martin, 1996]. Each membrane can be re-used so that multiple rods can be produced from a single pore. If we assume that 25% of the space above and below each rod is dead space, then a typical membrane with a 60 µm thickness could support 80 x 0.5 µm long rods per pore. Using this technique, the 2×10^{19} raw rod material would require 2.5×10^6 cm^2, or the area of about 3,500 x 12" circular wafers. This is a large amount of membrane material but when compared to the wafer output of a silicon foundry (~12,000 12" wafers per month) the number is comparable. It is likely that as the demand for nanoporous membranes increases, large volume fabrication plants will become practical.

Chapter 4. Structural stability and yield

The discussion in section 3.5 underscores the importance of reducing the number of rods that join at each junction in a structure. Equation (3.5) shows that the final structural yield decreases exponentially in the number of junctions, and depends on the per-rod DNA hybridization efficiency.

It is clear from looking at illustrations of the unoptimized assembled structures seen so far that they are dense. Many of the rods inside these dense structures can be removed without consequence to the overall rigidity of the final structure. Removing non-essential crossbar and insulating supports dramatically affect the final yield by reducing the average number of rods participating in each junction.

The yield estimates from section 3.5 were obtained by taking the number of rods participating in a fully dense structure, from chapter 7, and reducing it by 50%. This is an upper bound on the number of required rods because simply removing every other crossbar reduces the average number of rods participating in a junction from 12 to 9. Further, alternating levels that have crossbars reduces the average to 7 rods per junction. Section 4.1 describes how the stability of this method was tested.

4.1 Structural stability

The structure becomes unstable when a grouping of junctions no longer has complete rigidity because the rods under constrain it and substructures can "slosh" around within the structure. This increases the chances of signal lines shorting together and should be avoided. Figure 4.1 illustrates one way to alternate the diagonal support rods so that no more than seven rods meet at any junction. Each unit cube in this figure has at least three faces whose normals are orthogonal with diagonal supports.

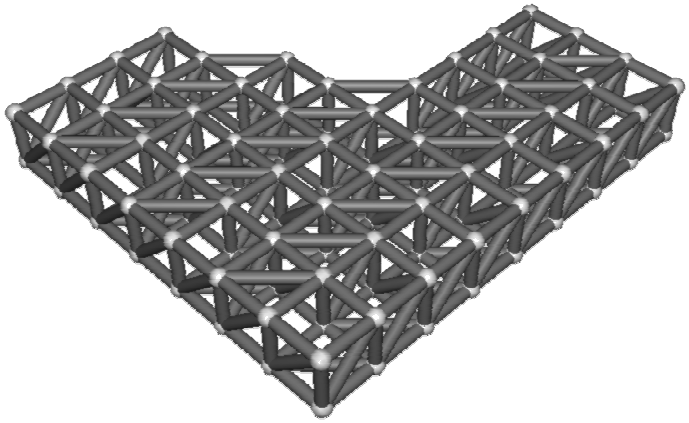

Figure 4.1. Structure with no more than seven rods meeting at any junction. Each unit cube has at least three perpendicular faces with diagonal supports.

Since the DNA junctions that bring the rods together are not rigid (i.e. rods are relatively free to rotate) but can act as harmonic oscillators (the DNA can stretch) the evaluation of the overall rigidity of the structure involves many coupled harmonic oscillators. This type of problem is difficult to solve exactly because of the number of oscillators. As a guide to determining the rigidity of the structure a spring-mass simulation was used.

Figure 4.2 illustrates the model used with a custom spring-mass simulator to test the structural stability of the module [Taylor, 2002]. Each junction in the module is replaced by a small mass connected by stiff springs to the other junctions (the rods and DNA are represented by the stiff spring.) Applying a force to one corner of the structure and then allowing the springs to come to equilibrium tests the structure's stability. If the geometry of the module remains intact after the perturbation, then it is likely to be structurally stable. The structure in figure 4.2 remains stable, suggesting that 7 rods per junction are sufficient.

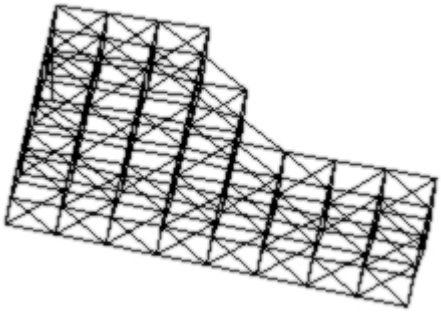

Figure 4.2. Spring-mass model used to test the structural stability of the module after a perturbing force has been applied. Only the springs are visible here.

4.2 DNA coupling yield

The face-serial assembly method described in chapter 3 reduces the total number of unique DNA sequences to 15. This section discusses the amount of raw DNA that is required to assemble the structures described so far. Whenever an estimate is used, the worst-case is used, making all yield estimates pessimistic. If after this evaluation the cost appears manageable, the actual cost is likely to be lower.

It is straightforward to calculate the total amount of DNA (with s-oligo extension) required to assemble a typical structure. There are seven types of rods: metallic, insulating, strut, metallic-strut, p-type RG-FET, n-type RG-FET, and gate. An upper bound on the quantity of DNA needed can be obtained by assuming that all rods in the structure need the same type of DNA strand and that there are 15 such cases, one for each unique DNA strand.

This estimate must be inflated because not all raw DNA-s-oligo strands will react with a rod end. Similar DNA-surface reactions have seen reactions progress to 10% of the saturated monolayer in 4 hours [Mbindyo, 2001]. The saturated DNA monolayer density was taken as 1×10^{14} strands / cm^2 and the starting stock concentration was 10μM. That is, about 1.6×10^{-4} % of the stock material reacted with the surface in 4 hours. Therefore, to

correct the total amount for reaction efficiency, the original figure must be multiplied by 1 / 1.6×10^{-4} %, or 6.25×10^{5}.

The earlier discussion of DNA strand design used 1×10^{14} rods as the starting point for the assembly. Using this same number of rods and accounting for the reaction efficiency of the DNA with the surface (of each rod end) yields a total amount of DNA (including the 15 types) of ~2×10^{21} strands. This number of strands at a typical concentration for DNA, ~10 µM (or 10×10^{-6} moles / liter), would occupy ~330 liters.

The cost of this quantity of DNA can be estimated using current prices available from vendors. One vendor, Qiagen, sells highly purified custom DNA strands in bulk for ~$25 USD that can be re-suspended into 1 mL of buffer (water plus some salts) to make a 10 µM DNA solution. The 330 liters of DNA solution required for the assembly process would cost ~$8M USD. Since only 1.6×10^{-4} % of this material is used during construction due to reaction efficiencies, recycling the waste material may dramatically reduce the cost for a second construction.

4.3 Purification

Non-functional instances of modules can probably be separated from correctly assembled ones by centrifugation[6]. Therefore, calculations of the modular assembly yield of a machine in chapter 8 will assume 100% purified modules. We can trade bulk material for a higher purity product. That is, if the final yield for a module is lower than we need we can purify the module and repeat the assembly method. The individual pieces, or modules, of a circuit can likely be purified because properly formed structures will have a different density and drag profile than structures which did not form properly. The combination of a unique mass and drag profile makes a centrifugation purification method an attractive way to separate good structures from bad ones as long as it does not damage the modules. Since the DNA-conserving assembly method assembles structures in a face serial fashion, if at any point a structure's face is not properly assembled, that structure will fall behind in the

[6] Centrifugation is a process that can separate particles (and molecules) based primarily on their density. A *centrifuge* applies a high acceleration (~10,000 x G) to a sample and over time low-density particles migrate toward the top and high-density particles migrate toward the bottom. The two can be separated by carefully decanting the top-most layers.

assembly. Even if the structure loses only one face due to an aborted assembly (and caught up on the next face) it will have a noticeably different mass. For example, a fault in the module assembly will on average reduce the mass of the module by ~20% (one face with 10 junctions) if a single face is missing. Ultracentrifugation is routinely used to isolate biomolecules with mass differences of ~5% [Lebowitz, 2002].

Chapter 5. Custom design tools

5.1 3D rod design tool

The 3D rod design tool is a program that facilitates the creation of 3D rod structures for use in self-assembled circuitry. The user has interactive control of the view of the structure and distracting rods can be temporarily hidden from view. Figure 5.1 is a screen capture from the design tool.

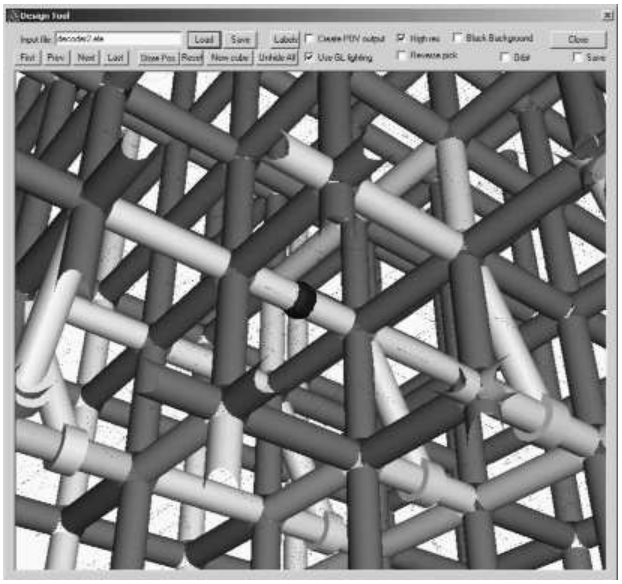

Figure 5.1. Screen capture from the custom design tool.

A user can use the design tool to load existing structures or create new rod structures. There are no design rules enforced other than the fact that rods must be aligned along one of the 12 allowed orientations, or be used as a gate electrode. The design tool also allows the user to label metallic rods that will be attached to that node in an extracted SPICE deck. This facilitates naming signal lines for electrical simulations later.

The following is an example design process:

1. Create a large cube of insulating rods with the "New cube" button. If the structure is not cubic, use the '1', '2', or '3' keys to add additional layers along each axis.
2. Select a rod by clicking on it with the mouse and change it to either a conductor ('m'), NFET ('n'), PFET ('p'), gate ('g'), conducting cross-bar ('c'), or insulating cross-bar ('s'). The rod type and orientation can alternatively be changed by double-clicking with the right mouse button on the rod. This constrains the properties of the rods to those properties supported by the fabrication technology.
3. Double click with the left mouse button on a rod to center the current view on that rod. This feature is used to navigate large and complex structures.
4. Pan with the left- or right- mouse button depressed to rotate the view. As with the centering command, this feature is used for navigation.
5. Control+S toggles between fully visible insulating crossbar (struts) and hidden struts. Control+T hides/unhides all insulators. Large and complex structures are dense and can become difficult to understand. Hiding insulating rods, which are only for support, helps reduce the visual complexity of the scene and emphasizes the structure of the electrical circuitry.
6. Repeat from step 2 until the circuit is complete.

The circuit can be saved and then opened with the assembly tool to create a SPICE deck for simulation and to gather assembly statistics.

Software platform

The custom design tools were created using the Microsoft Visual Studio in MFC/C++. The rods are rendered using a Win32 version of the OpenGL standard. The source code for this tool can be found in appendix B.

5.2 Assembly-order tool

The assembly-order tool takes an element structure from the design tool and creates a DNA conserving assembly order. The characteristics of this algorithm have already been described in chapter 3. The implementation of the algorithm can be found in appendix B.

Figure 5.2 is a screen capture from the assembly tool. The end of the each rod is colored to indicate the sequence of DNA that has been allocated for that rod-end.

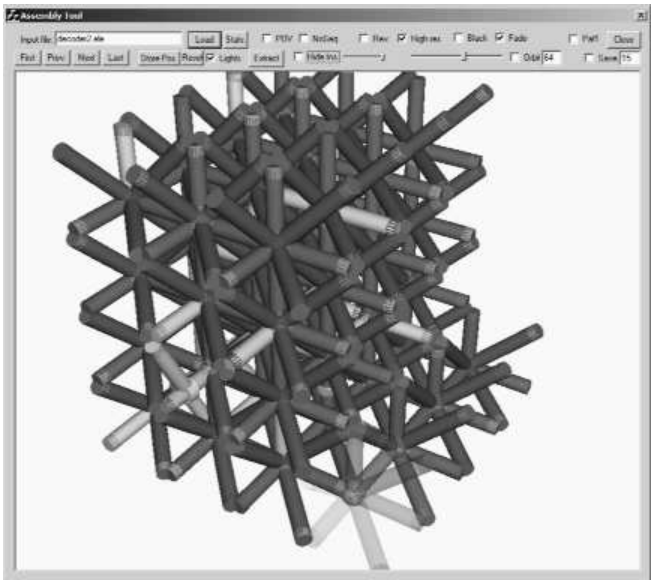

Figure 5.2. Screen capture from the custom assembly-order tool.

The assembly tool also extracts a SPICE deck, or input file, from the element structure. The final deck can be included in a test circuit to verify that the structure works properly and to estimate power consumption.

The assembly tool can also be used to create animations of the circuitry that rotate and step through the assembly process. This can display the structure of the circuit to help understand how the circuit must be connected to other modules.

Software platform

The assembly-order tool was created using the Microsoft Visual Studio in MFC/C++. The rods are rendered using a Win32 version of the OpenGL standard. The source code for this tool can be found in appendix B.

Chapter 6. Power and interconnect

Connecting self-assembled structures to power and I/O electrodes could easily be a manufacturing bottleneck given the vast number of structures that can be assembled at one time. This chapter describes two interconnection methods for 10^{12} devices that can be implemented in parallel.

6.1 Modular assembly

Section 8.4 analyzes the tradeoff that helps determine the optimal fabrication yield of a machine given various design parameters. Section 3.8 shows how the low yield of a small logic circuit is impractical and therefore why a monolithic, fully assembled computing machine is not feasible without improvements in current fabrication yields. This does not mean that modular self-assembly is infeasible. The design-yield tradeoffs help determine the level of modularity needed to achieve a given final yield by trading larger numbers of unique modules for higher raw (module) fabrication yields.

The placement of each self-assembled module must proceed unambiguously as did the rod assembly described in the earlier portions of this chapter. One simple way to assemble mesoscopic-scale objects unambiguously is to rely on geometric features and surface free-energy minimization [Clark, 2001; Clark, 2002]. This method of self-assembly works by creating modules with a particular geometric shape that can fit into a hole on the surface like a key. Materials coated on the sides of the module act like glue to keep the module in place. The module shape, or footprint, should be rotationally and reflectionally asymmetric so that a well-defined orientation for the modules on the surface can be maintained.

Adopting a standard footprint among modules facilitates the mesoscopic-scale assembly if that footprint allows only a single final resting-orientation. This technique is

used in fluidic self-assembly where an object minimizes the shear force exerted on it by the fluid when it lands on the substrate in a strictly unambiguous fashion [Alien, 2003].

Lithographically patterned planar substrates, as illustrated in figure 6.1, could be used as landing areas for the modules if the exposed and developed portions of the resist layer create recesses that fit the outlined shape of the self-assembled modules. Maintaining a registered substrate-to-mask alignment will allow multiple self-assembled modules to be stacked on top of each other [Srinivasan, 2001]. The tolerances for the lithography process need to be in the range of half a rod length, or 0.25 µm in this case. This is well within the capabilities of present day photolithography.

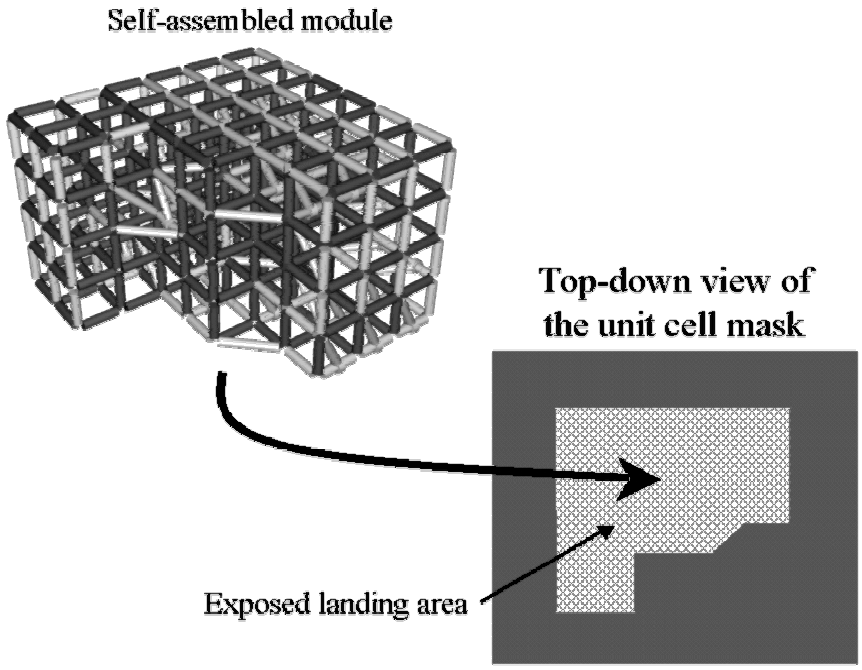

Figure 6.1. Modular-assembly unit cell mask and module. The module lands in the exposed landing area. A new photoresist mask is constructed and the process repeats.

Modular-assembly simplifies the interconnection problem by disambiguating the orientation of structures as they land on the substrate. Since the lithography pattern prevents improperly oriented modules from landing, the final structure will have a well-defined shape and orientation, as illustrated in figure 6.2. The particular photoresist chosen to form the cavities must support high aspect ratio features and have a low surface free energy. The

commonly used SU-8 photoresist has high aspect ratio features but needs to be doped to reduce its surface free energy. Research in this area has shown that it is possible to formulate epoxy-based photoresists and that the siloxane nature of the polymers makes self-assembled monolayer treatments feasible [Schmid, 1996; Martin, 2002].

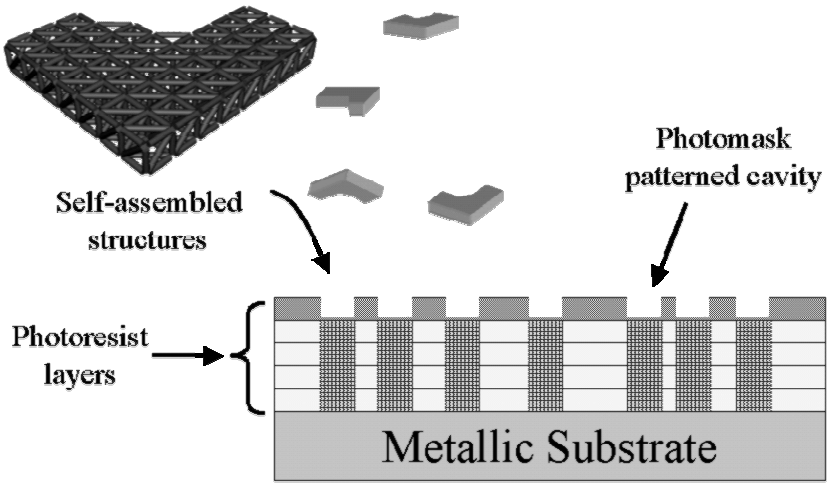

Figure 6.2. A modular assembly of stacked structures. Photolithographic steps create the sidewalls that corral the asymmetric structures.

The asymmetry in the footprint of each structure makes it possible to enforce a role for the substrate electrode and the top electrode (deposited during post-processing.) We know exactly how the final circuit will be connected if the method described in section 6.2 is used for power and I/O.

Using a lithographically prepared silicon substrate instead of a metallic substrate we can employ conventional VLSI techniques in making a footprint that can communicate with the processing elements. The circuitry that controls the communication to the processors can lie beneath the footprints and be shared by several "stacks". This is described further in section 6.3.

6.2 Monolithic assembly

Another solution to the I/O problem is to use fully self-assembled structures sandwiched between two power electrodes. This method is applicable to structures that are

fully assembled before finishing the interconnection method. Such monolithic assembly methods require higher yields as discussed in chapter 8. The most serious drawback of this method is the large capacitance of the sandwich. The DAMP design in chapter 8 would require the control circuitry to oscillate a 9 nF load at greater than 400 MHz, which is energy intensive. The alternatives to this method are presented in section 6.3.

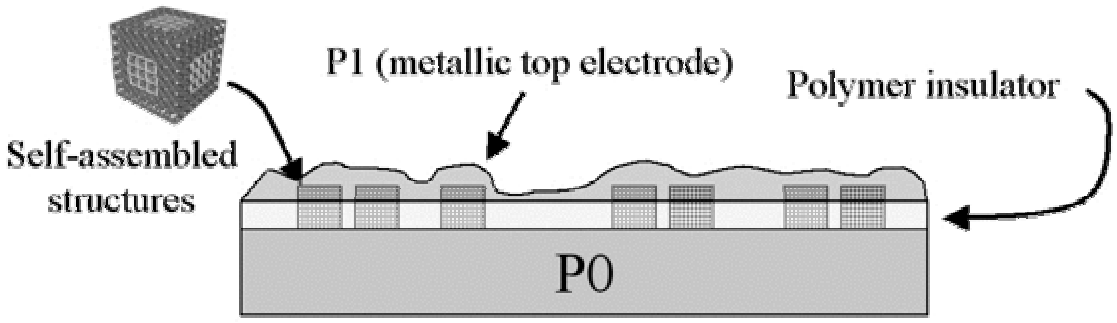

Figure 6.3. Layered interconnect method. The bottom electrode serves as ground while the top electrode serves as Vdd.

Figure 6.3 illustrates the layered interconnect method. Each electrode serves a dual purpose. The bottom electrode (P0) is used to electrically ground the circuitry and is also used as a clock signal. The top electrode (P1) is used to supply a positive voltage to the circuitry and is also used as a data signal.

This arrangement requires special "power-up" circuitry to be embedded within the structures. This circuitry serves to orient the structure as to which direction is "up" (the positive voltage electrode.) This same circuitry, through the use of a bridge rectifier, can supply power and provide a reference for how to use each electrode. By alternating between power and signaling phases, the electrodes can be used for both purposes. The initialization routine for this system is as follows.

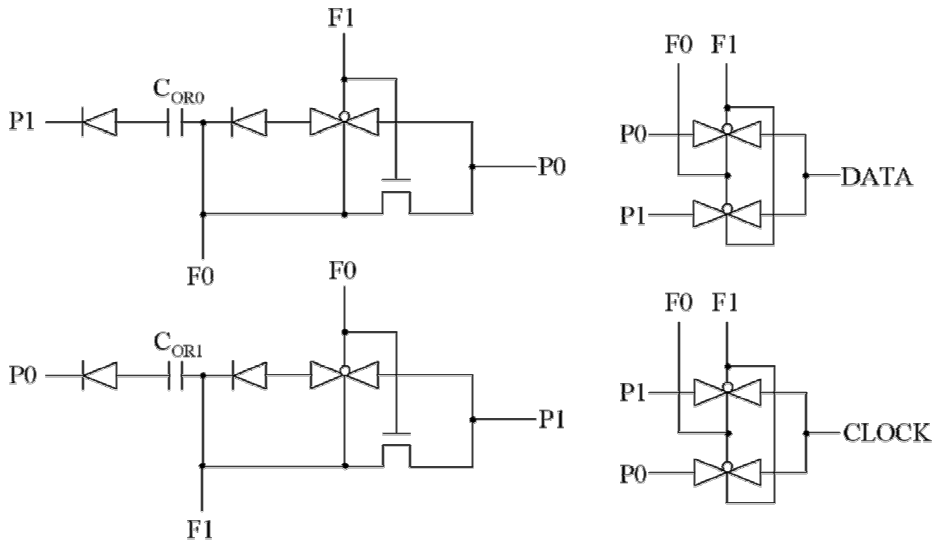

Figure 6.4. The power-up circuitry used to orient a structure after it has been sandwiched between two electrodes. The circuitry tells the structure which electrode was powered up first and therefore which electrode will serve as a data signal. The other electrode is the implied ground and clock signal.

The positive electrode (P1 in figure 6.4) is slowly ramped to V_{dd} (1V in the circuits we have considered) while the ground electrode (P0 in figure 6.4) is connected to the system ground (0V).

After some time, the orientation capacitors (C_{or0} and C_{or1}) will have fully charged or failed to charge depending on which electrode was powered-up and which was grounded. At this point the power-up circuit knows which electrode (P0 or P1) is the positive (and data) electrode and which is the ground (and clock) electrode. The signals F0 and F1 will reflect this orientation and select the proper electrode to be connected to the internal DATA and CLOCK wiring.

To signal a “1”, the positive electrode and ground electrodes are temporarily held high. The ground electrode is returned to ground potential after the circuitry has stored the input bit. To signal a “0”, first the positive electrode is grounded and then the ground electrode is raised to V_{dd}. Again, this condition is maintained for a sufficient time to allow the circuitry to latch the input bit before returning to the power phase (P1 high and P0 grounded).

The bridge rectifier in the power circuitry charges a capacitor (and the rest of the power-up circuitry) regardless of which electrode is positive. The circuitry will function properly as long as the data and clock steps (step 3 above) are short compared to the power-up time constant (i.e. the time required to charge the power circuitry).

This circuitry is useful because it works without regard to which electrode is positive and which is grounded. If the assembled structures are to be deposited from suspension, they should be encased in a way similar to that illustrated in figure 6.5.

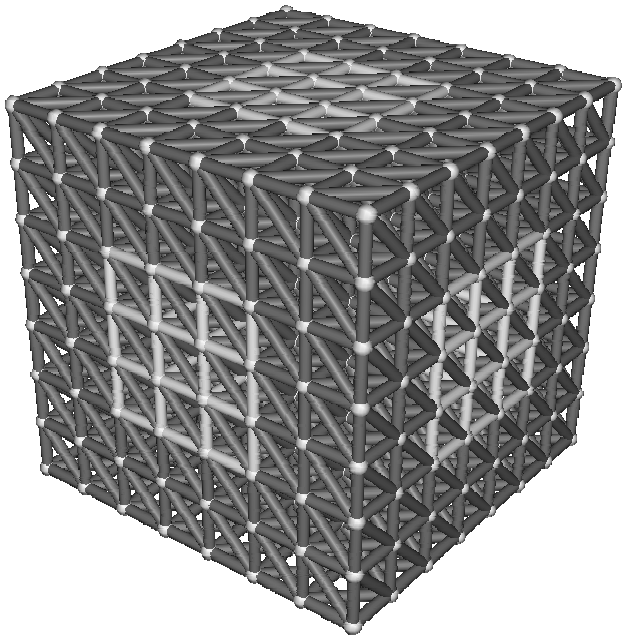

Figure 6.5. This structure connects opposing sides of a cube (or rectangular solid) to the P0 and P1 wiring inside the structure so that the structure can "land" with any side down and still receive electrical power and be able to communicate.

This structure connects opposing sides of a cube (or rectangular solid) to the P0 and P1 wiring inside the structure. Since opposite sides of the cube are connected either to P0 or P1, the structure can land on a metallic surface (the bottom electrode) with any of its sides. A layer of insulating material (e.g. a polymer) could be deposited onto the surface and etched back to expose the top side of each cube. Another layer of metal could then be deposited on top and used as the positive electrode (see figure 6.2.) This method can be used regardless of

how the cube lands as long as it makes (or can be made to make) good ohmic contact with the bottom and top electrodes.

6.3 Output methods

Regardless of which interconnection method is used there must be a way for each processing element to communicate with the outside world or each other. It may be possible using future versions of self-assembling technology that have higher yields to make a single monolithic device with many connections between elements. In the near-term where a single large monolithic device is not feasible and connections are restricted, we must adopt a modular approach like the one described above.

With no connection other than a common power supply, each processing element appears to be without any way to communicate. Aside from receiving commands, there must be a method for communicating calculation results to the outside. One potential solution to this problem is to use a switching noise detection circuit in the power supply. If tuned to detect a unique electrical oscillation made by the processing elements, it can communicate at least a single bit of information. The nature of this method makes the resultant bit take on the superposition of all bits being transmitted, which complicates the communication - only one processing element can communicate at a time. This can be used in a way similar to how auctioneers talk to an audience. A question is posed ("will anyone buy this vase for $10?") and any eligible members of the audience respond. The circuit that can implement this output method is described in the architectural description of the decoupled array multi-processor in chapter 8, and simulated in chapter 9.

The drawback with this method is the large capacitive load between the power electrodes and the frequency with which the voltage must change. That is, a single processor will have trouble overcoming the large capacitance of the power electrodes to produce a signal large enough to detect. A simple way around this problem is to use conventional I/O methods to clock data into the processors.

Figure 6.6 illustrates a modified footprint that has ports for clock and data signals that are shared in parallel by several processors. The ports are signal lines that protrude upward from the silicon substrate and through the passivation layer (e.g. glass.) This requires the

substrate to be more complex than a simple metallic substrate because it needs to have multiple routing layers and some control logic. This is possible when the size of the self-assembled circuits is comparable to the minimum feature size of the photolithography used to make the substrate. However, as the components used in self-assembling the circuitry shrink in size and it becomes impossible to interconnect multiple signals per processor, the two electrode interconnection method described in section 6.2 will become an attractive solution.

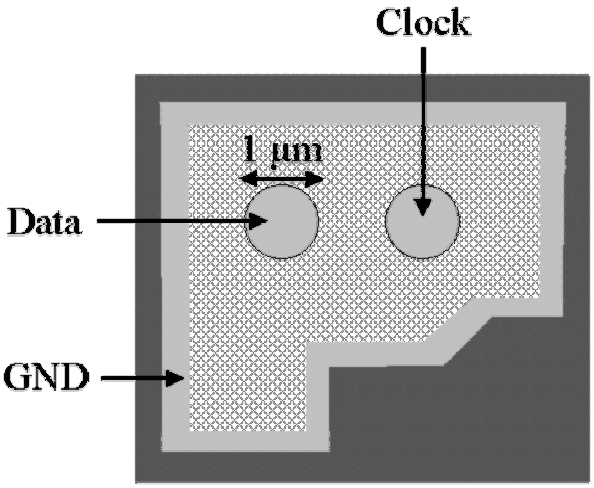

Figure 6.6. Modified footprint that provides a low capacitance clock and data port. The two circles inside the footprint are the data and clock electrodes, the ground electrode lines the rim of the footprint.

A processor can directly sense the data line (or port) by wiring it to a multiplexer input (see LC1 in section 8.2.) If the photolithographic process can place multiple viaducts to the substrate, the data channel can be expanded to multi-bit and/or full duplex. The external clock line can be driven by external amplifiers and connected directly to the internal processor clock line.

For output, external circuitry pre-charges the shared data line and any processor can pull the line to ground with pull-down logic. The extremely low drive current of the RG-FET (~1.6 µA) means that pulling the data line to ground could take a long time if the data line capacitance is large. The capacitance of the data line can be reduced by using an H-tree arrangement, illustrated in figure 6.7, with buffers placed at vertices. The nodes on the H-

tree sit below the modified footprints from figure 6.6 and contain pre-charge control circuitry. The buffers pass the result of the pull-down to higher levels in the H-tree, and ultimately to the top of the H-tree.

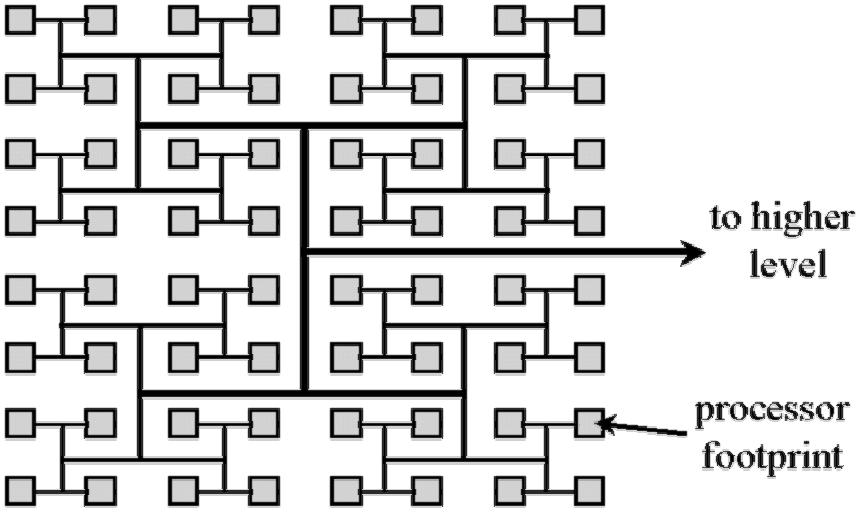

Figure 6.7. A portion of the processor H-tree. This circuitry lies beneath the processors (and the passivation layer) in the silicon substrate. Each processor footprint is ~4.5 µm on a side.

Since the data line may take a long time to pull to ground with the low drive current of the RG-FETs it is necessary to keep the data line capacitance below a critical value. Using the decay constant $\tau = R \cdot C$ and assuming the line to be can be sensed after $3 \cdot \tau$ (or after > 90% of the charge has dissipated), the maximum data line capacitance C can be calculated. The drive current of 1.6 µA means that the RG-FET (operating at 1V) has an on-state resistance of 625 kΩ, which we can use as R in the decay constant. Solving for the capacitance we get $C = \tau / R$. To operate at 400 MHz, the total decay time $3 \cdot \tau = 2.5$ ns, or $\tau = 0.83$ ns. Using the on-state RG-FET resistance we get a maximum data line capacitance of 1.3 fF which is practical for standard CMOS technology. (A 0.25 µm x 30 µm plate capacitor with a 0.25 µm separation and $\kappa = 5 \cdot \varepsilon_0$ also has a 1.3 fF capacitance.) As with the ring oscillator, the output is the logical "OR" of all element outputs.

Chapter 7. The oracle architecture

The oracle is a new architecture that uses brute force to solve instances of NP-complete problems. Inspired by the broad parallelism of DNA computing, the oracle stores all instances of a problem at the time of its assembly. Whereas the methods used in DNA computing that require weeks of laboratory work per problem instance, the oracle uses electrical circuitry in conjunction with self-assembly to enable it to solve instances of the problem rapidly.

7.1 Assembly-time computation

All computers require some degree of assembly-time complexity during their construction. Traditional computers require photolithographic masks to be fabricated and materials to be deposited and patterned in complex ways. This assembly-time complexity usually generates very little assembly-time computation. Incorporating portions of a computation into the design of a machine (e.g. ROMs, micro-code, etc.) has only limited application when compared against the vast set of problems traditional general-purpose machines are designed to solve.

The tradeoff between assembly-time and runtime complexity, however, can render gigantic improvements in performance. The advantage that application specific integrated circuits (ASICs) have over general-purpose circuitry is one example of this tradeoff. An ASIC design uses *a priori* understanding of a class of problems to build in assembly-time complexity that reduces the execution time of the circuit. For example, a carry-select adder uses redundant full adders to compute the sum of two operands speculatively. Only at runtime will the circuit choose the correct sum. ASICs are inherently faster than general purpose circuitry at solving problems because the problem has already been partially solved *by design*. That is, an ASIC solves a portion of the problem before runtime but still continues to employ a significant runtime component.

The type of machine described in this chapter is enabled by the extraordinarily parallel nature of self-assembly. The scale of this form of fabrication, discussed in chapter 3, makes it possible to cover an input space[7] that is much larger than what traditional ASIC designs can cover. Therefore, the advancement of near molecular-scale fabrication technology makes it possible to build ASIC-like computing machines that shift the majority of a computation so that it occurs during assembly of the circuit rather than during its runtime.

Oracles

An oracle is a class of machines that has within it a large number of question and answer pairs. Questions are posed to the machine. A response is generated if the question is contained in any of the oracle's question/answer pairs. In this fashion, the oracle is like a large content-addressable memory (CAM) that has been preloaded with the answers to a certain problem. An oracle differs from a CAM by the method the question and answer pairs are entered into the machine. The CAM requires $O(2^k)$ steps to load the answers (each of which have been computed) where k is the number of index bits that serve as an address. Each address is a question represented by up to k bits with its associated answer. The oracle requires O(k) steps to assemble and no runtime loading steps. The answers are determined by the manner in which the oracle is assembled. The self-assembly of each question and answer pair provides the oracle with the answers (with a high probability but not a certainty that a given question and answer pair will exist within the oracle.) If a particular question and answer pair did not form during the oracle's assembly then the oracle cannot solve that instance of the problem.

In the same manner that an ASIC is designed to solve a problem more quickly than a general-purpose machine by incorporating portions of the problem, the oracle is designed to solve huge portions of a problem space during its assembly.

[7] An input space is the set of all possible binary combinations (of some number of bits) that are within the range of meaningful inputs for a given problem.

7.2 Addition oracle

Architecture

A simple example of an oracle is the addition oracle (not useful in itself, but illustrative.) The addition oracle has a simply defined problem and a brief functional description, and performs all calculations at assembly time.

A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 7.1. Full-adder truth table

Table 7.1 lists the entries in the truth table for a full-adder. The addition oracle will be assembled so that it incorporates many of the possible combinations of the truth table entries. Each single line from the truth table represents an instance of an addition problem. A ripple-carry adder solves addition problems by chaining the carry-out from one full-adder to the carry-in of the next full-adder. In a similar way the addition oracle will chain carry-outs to carry-ins, but at assembly time rather than at runtime.

The oracle is queried for the answer to a problem (in this case an addition problem) by serially shifting in the operand bits. If the oracle contains the sum of the two operands, it responds with the answer by serially shifting out the sum.

Implementation

Each line in the truth table is converted to a "tile" that represents a particular input and output combination. This is similar to the way a carry-select adder speculatively pre-computes a carry pattern and at runtime selects the proper carry path, with the exception that the oracle pre-computes the entire carry path. The tiles that correspond to table 7.1 are shown in figure 7.1. Each tile has on its left side a carry input and an output. In this case, the

top portion of the tile is the carry-in and the bottom portion the carry-out. The carries are depicted in such a way that they fit together like the pieces of a jigsaw puzzle.

A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

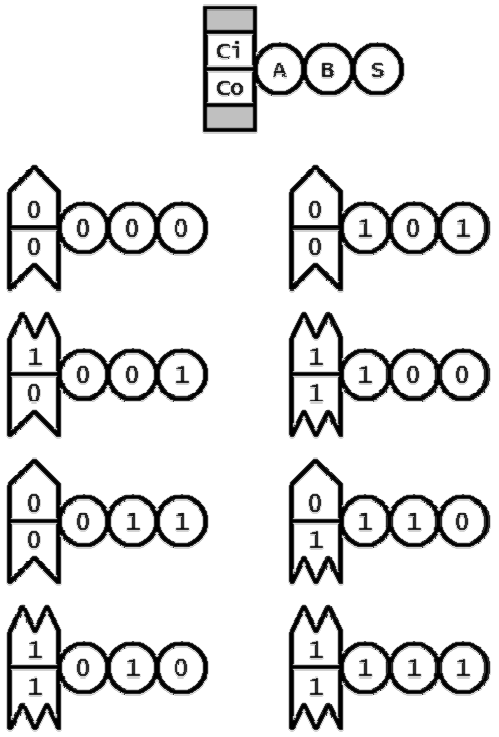

Figure 7.1. Assembly tiles for the addition oracle (with the full adder truth table.) A = a bit from the 1st operand, B = a bit from the 2nd operand, and S = the sum of the two bits.

The iterative nature of the function (e.g. output from step i produces input for step i+1) allows strings of these tiles to implement an instance of the function's evaluation. For example, figure 7.2 illustrates a simple 4-bit string made from the tiles in figure 7.1. This particular example is an instance of the addition function for "3 + 5 = 8". The shape of the carries on each tile dictates how the string is formed. Valid strings must match each carry-out with the corresponding carry-in. In this fashion, the tiles perform an assembly-time computation as they form valid strings. They assemble only into valid solutions for addition because the carries must match at each stage.

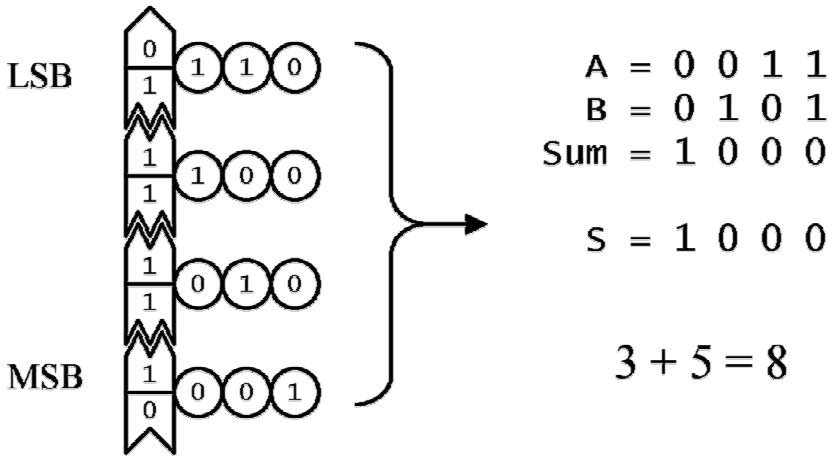

Figure 7.2. A 4-bit instance of the addition function. The carry-in and carry-out shapes determine valid strings. This string implements the "3 + 5 = 8" instance of the function.

A complete addition oracle is the collection of all possible N-bit strings of tiles. Each string represents one particular input and output combination, for example the "3 +5 = 8" string shown in figure 7.2. In this case, the string will respond with "8" to the question "what is 3 + 5". For all other questions the string will be silent.

The yield discussion in chapter 3, and later in chapter 8, indicates that it is feasible to have on the order of 1×10^{12} individual processing elements, or strings in this case. This means that it is feasible to implement all 40-bit input strings (N = 40), or in the case of an addition oracle all two 20-bit operand sums.

Again, the addition oracle is simply an illustration of an oracle rather than an exemplar of its usefulness. The circuit complexity of each string is determined by the circuitry needed to read the string and respond to queries. A possible circuit for the addition oracle is shown in figure 7.3.

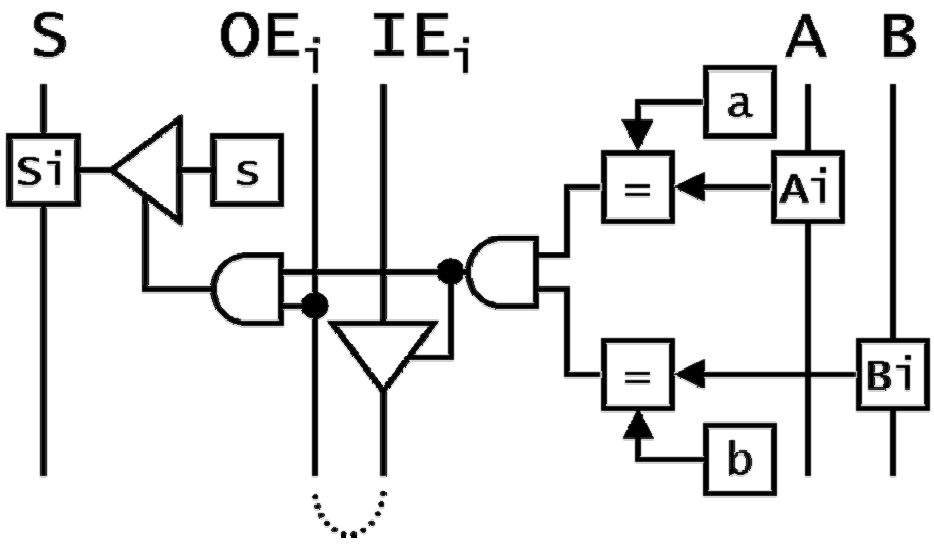

Figure 7.3. Circuit for an addition tile. The *a*, *b*, and *s* bits are consants assembled into a particular tile. The tile is assembled into strings that respect the carry-in and carry-out matching. The input query and output bits are serial shifted into and out of the circuit.

The *A*, *B*, and *S* signals illustrated in figure 7.3 carry the input query and response bits, respectively, for each tile. The OE_i and IE_i signals are the output and input enable signals, respectively, that coordinate the individual tiles in a string so that the string responds to a query if and only if *all* tiles in the string match the input query. The input enable signal is passed downward along the string and at the very last tile reflected upward as the output enable signal. Each tile can interrupt the input enable signal depending on the value of the current query, or the latched A_i and B_i input signals. Input queries are serially shifted into all strings (i.e. the circuits that implement each string) simultaneously. When the *A* and *B* values match the particular inputs of a string, all the tiles latch their sum values into the S_i latches. The only strings that respond to the query are those that have successfully reflected the input enable signal to their output enable line. The output enable signal could be used to trigger a ringer circuit, described in chapter 6 and 9, that creates an oscillating signal that can be detected by an external receiver. This method is useful for problems that require only a single bit of output (e.g. NP-complete problems). Alternatively, the output enable signal can be used, as shown in figure 7.3, to load the sum bit into a D-latch that can be shifted downward along the string to a ringer at the bottom that responds to the shift-out from the

string. An analysis of the power consumption and performance of this kind of circuitry is presented in chapters 8 and 9.

Generalization of the oracle

An oracle can solve any problem that is expressible as the form illustrated in figure 7.4. The functions $\mathbf{F_i}$ and $\mathbf{G_i}$ take the f_{i-1} and X_i inputs and generate the f_i and g_i outputs, respectively.

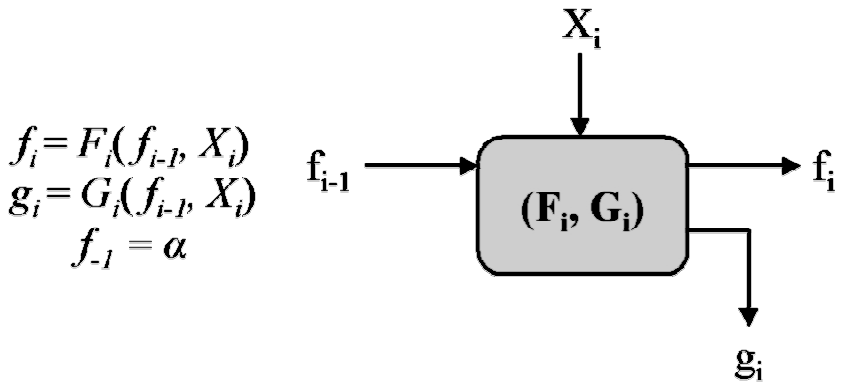

Figure 7.4. Problem expression solvable by an oracle.

Each input (f_{i-1} and X_i) and output (f_i and g_i) are bit vectors. To aid in initializing the system, f_{-1} is assumed to be α, which is a constant defined at assembly-time.

Equation (7.1) through (7.3) describe the addition oracle using the form illustrated in figure 7.4. These equations are derived from the truth table for addition, shown in table 7.1. The input vector **X** has two elements, **A** and **B**, that represent the input operands. Equation (7.1) is the carry-out bit, and equation (7.2) is the sum bit.

$$F_i(f_{i-1}, X_i) = f_{i-1} \cdot (X_i[A] + X_i[B]) + (X_i[A] \cdot X_i[B]) \tag{7.1}$$

$$G_i(f_{i-1}, X_i) = \overline{f_{i-1}} \cdot (\overline{X_i[A]} \cdot X_i[B] + X_i[A] \cdot \overline{X_i[B]}) + f_{i-1} \cdot (\overline{X_i[A]} \cdot \overline{X_i[B]} + X_i[A] \cdot X_i[B]) \tag{7.2}$$

$$\alpha = 0 \tag{7.3}$$

Realization

The monolithic fabrication techniques described in chapter 3 can be straightforwardly applied to realize each tile and string. The DNA strands used to form each junction can be selected so that when a new tile is being added to the string the assembly of a single anchor rod/DNA type determines the tile choice and also respects the carry-out / carry-in matching rule. Then, each tile type is assembled in parallel using the face serial method described in chapter 3. Since each tile will only assemble to the proper anchor, the new tile's carry-in will necessarily match the previous tile's carry-out.

However, the near-term self-assembly processes that appear to be most feasible will require each string of tiles to be divided into thin modular slices. The individual slices will not undergo DNA-guided self-assembly but rather fluidic self-assembly. This means that the DNA specificity that allowed each tile to respect the carry-out / carry-in matching rule no longer exists. Instead, thin modular slices must be stacked on top of each other as described in chapter 6. Each module has no preference for which stack it lands on but lands in the proper orientation on any stack due to its rotational asymmetry. This requires the tile set from above to be modified since the carry-out / carry-in matching rules must be enforced.

Rather than using a single tile for each entry in the full-adder's truth table as before, the entries are grouped according to a carry-in and carry-out pattern. Figure 7.5 illustrates the new tile design. The entire set of tiles will consist of the nine tile types like the one in the upper left-side of figure 7.5, one tile type like the one on the right side of the figure, and three tiles like the one in the lower left. Just as before, the tiles are stacked randomly (valid because the carries necessarily match between tiles) to form all possible strings of tiles. The "LSB" (least significant bit) tile from figure 7.5 is the last tile in each string (at the top) - since all sums start with a carry-in of zero.

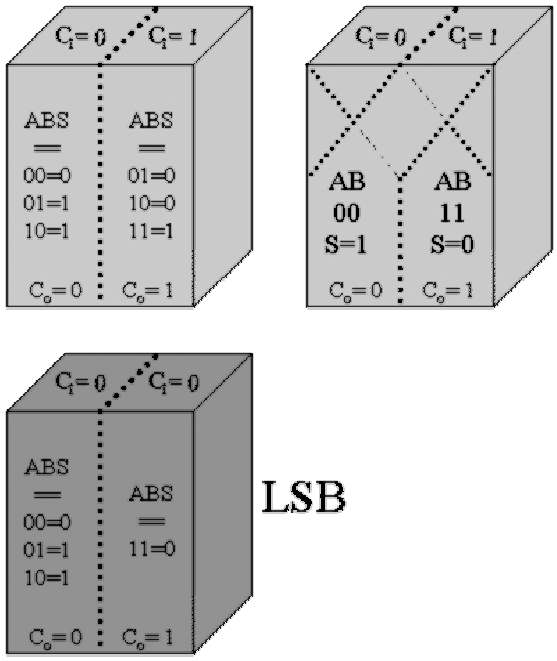

Figure 7.5. Addition oracle tiles for modular self-assembly. The constant 0-1 carry-in / carry-out pattern implicitly allows for the proper matching of stacked and oriented tiles. The circuitry in the tile on the right must swap sides to preserve the carry-in / carry-out pattern. The tile in the upper-left represents 9 different tiles, each with two of the AB pairs shown. The tile in the lower-left represents 3 different tiles and is the first (or least significant bit) in any string (both carry-in bits are zero.)

For example, the sum "5 + 6 = 11" in binary is "0101 + 0110 = 1011" with carry-in values of "1000", and the carry-out values of "0100" is illustrated in figure 7.6.

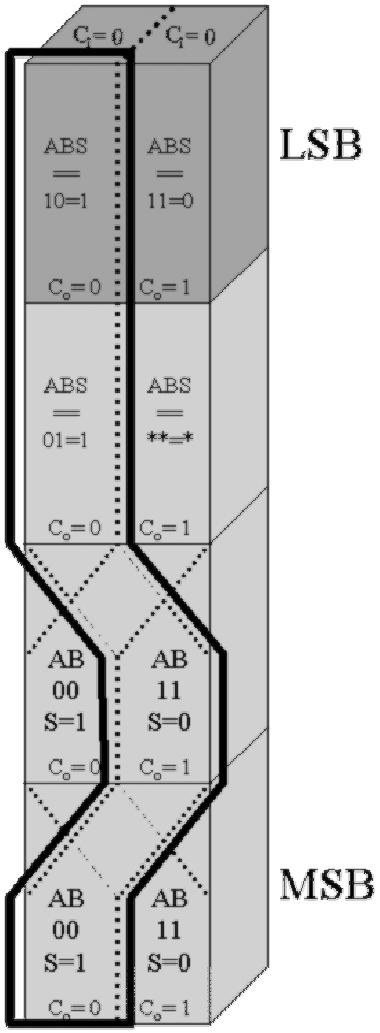

Figure 7.6. "5 + 6 = 11" example using the modular tile set. The circled portions represent the sum.

The methods used to orient modules in chapter 6 will need to be applied to the tiles described here. The rotational asymmetry of the tile shapes in figure 7.5 is required, but has been omitted from the diagram here for the sake of clarity. As with the earlier tile set, the new tile set stacks to form N-bit strings representing the sum of two N-bit numbers. The circuitry of both tile sets (from figures 7.1 and 7.5) is identical with the exception that each tile in figure 7.5 has two copies of the circuit. The tile on the right-side of figure 7.5 must swap sides to maintain the carry pattern from the bottom of the tile since both the AB=11, C_i=0 and AB=00, C_i=1 truth table entries produce C_o = !C_i.

7.3 Hamiltonian path oracle

The Hamiltonian path (HAM-PATH) problem is in a complexity class known as NP-complete and represents what is considered to be an intractable problem. The problem consists of finding a path through a given graph of nodes connected by arcs that visits each node exactly once.

The HAM-PATH oracle computes all paths through a fully-connected graph at assembly-time in a manner very similar to the way Adleman solved the HAM-PATH problem using DNA [Adleman, 1994]. The difference is that Adleman's approach solves the problem for one particular graph while the HAM-PATH oracle can solves it for any instance of the problem with a fixed number of nodes.

Adleman's solution encodes each edge in a graph as a DNA fragment that has two "sticky" ends representing the starting and ending nodes of the edge. Each node in the graph is allocated a sequence of DNA and any edge that starts at that node will use this sequence on one end. The other sticky end of the DNA fragment uses the complement of the DNA sequence assigned to the ending node. All of the fragments are mixed together and form strings of edges (in the form of DNA fragments) that represent feasible paths through the graph. Since Hamiltonian paths visit each node once, only strings with as many edges as there are nodes in the graph are feasible Hamiltonian paths. All other strings are discarded. Cycles in the graph need special treatment [Adleman, 1994]. The entire process takes on the order of weeks.

The way the HAM-PATH oracle solves all instance of the problem is by solving the problem for a fully connected graph and then discarding solutions at runtime based upon a particular input graph. Paths from the fully connected graph that do not appear in the problem instance are deleted. This idea is illustrated in figure 7.7.

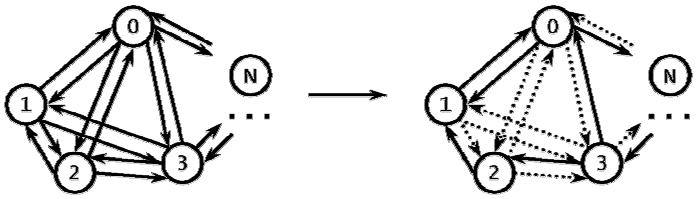

Figure 7.7. The fully connected graph on the left is collapsed to a particular graph on the right by deleting edges that do not appear in the problem instance. The dashed lines represent deleted edges. Conversely, the solid lines represent the remaining edges that have been selected for the current problem (specifying all edges in the problem graph.)

Like the addition oracle, the HAM-PATH oracle uses random strings of tiles to perform an assembly-time computation. The addition oracle formed all N-bit sums at assembly time. The HAM-PATH oracle forms all paths through the fully connected graph. At runtime the HAM-PATH oracle selects the edges that exist in the current problem instance. After selecting the edges in the problem instance one or more computing elements within the HAM-PATH oracle responds (electrically) to indicate that a Hamiltonian path exists through the graph if it has a solution.

The design of each HAM-PATH tile is somewhat more complicated than the tile designs for the addition oracle because the tiles need to support removing nodes from a set and responding to selected graph edges. Strings of tiles (computing elements) without the proper edges must disable themselves. For brevity, only the modular self-assembly technique is considered here.

Algorithm

Details of how the tiles solve Hamiltonian graph problems are given later in this section. The basic idea is that each tile randomly chooses a node from the fully connected graph and tests for the edge between that node and the node chosen by the next tile (above) in the string at runtime. If the edge is in the current problem instance (a particular graph less connected than the fully-connected graph), the SR-latch is set and the output enable (OE) signal is passed upward. If a string has edges that are in the problem instance, then that string represents a valid Hamiltonian path since each node occurs only once in the string of tiles. Each tile selects randomly from the remaining set of nodes after all the earlier tiles

have chosen their nodes. This means there cannot be any repeated nodes in the path chosen by a string.

The following example illustrates how a particular path is assembled using the HAM-PATH tiles for a 4 node graph.

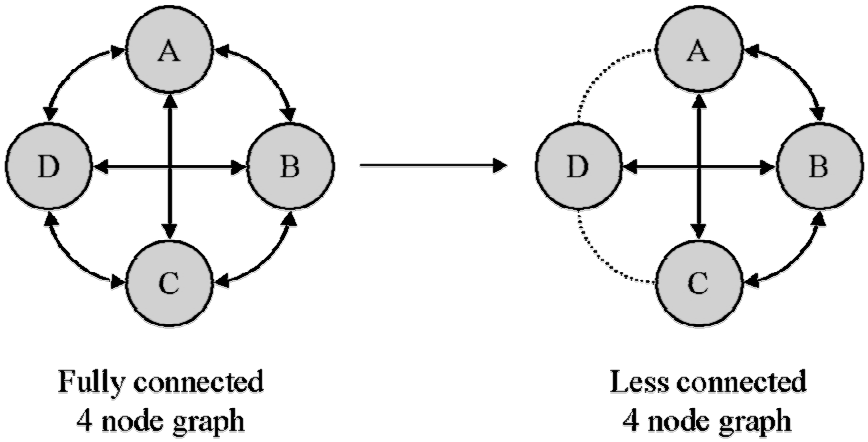

Figure 7.8. The fully connected graph on the left is reduced to the graph on the right by removing edges. The graph on the right represents an example graph.

Since a fully connected graph has all possible paths there is no question that it will have a Hamiltonian path. The real question is if a graph that is *not* fully connected has a Hamiltonian path. Therefore, the fully connected graph (represented by all possible strings of the HAM-PATH tiles) must be reduced to a graph of interest. Edges that do not appear in the instance graph (but that necessarily do appear in the fully connected graph) must be removed. This means that any path that uses an edge that has been removed from the fully connected graph cannot be a valid path through the instance graph.

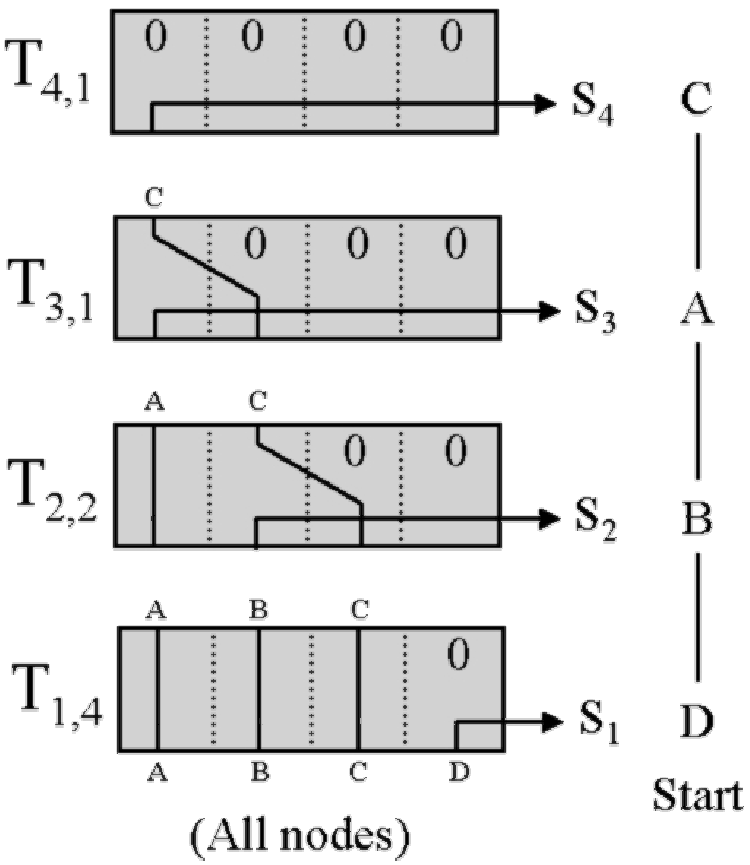

Figure 7.9. An example string that represents a Hamiltonian path (D-B-A-C) through the example graph. This string is only one from the 4! (24) randomly assembled strings that represent all paths through the example graph (some of them are Hamiltonian paths.)

Figure 7.9 illustrates a string pulled from the soup of all randomly assembled 4-tile strings in the oracle. Since all the edges represented by the string (D-B, B-A, A-C) are in the example graph this string represents a Hamiltonian path. Figure 7.10 illustrates the basic tile schematic.

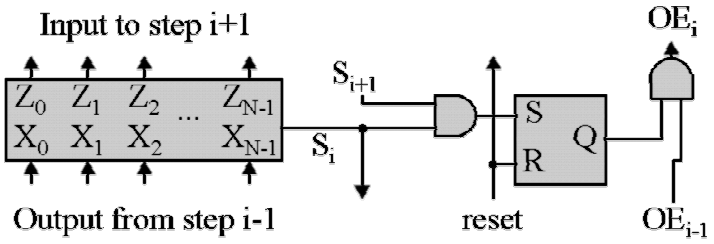

Figure 7.10. Basic HAM-PATH tile schematic.

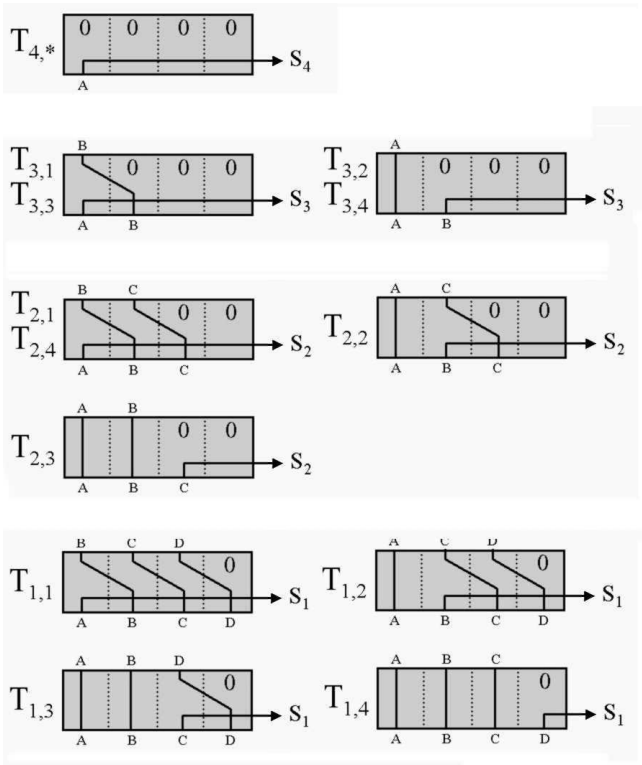

Figure 7.11. The four node tile set for the HAM-PATH oracle. A mixture of each tile, $T_{i,j}$, is used to assemble all possible 4-tile strings randomly in 4 steps. $T_{1,*}$ tiles are used during the first step, $T_{2,*}$ tiles during the next, and so forth. Inputs 1,2,3, and 4 represent node values that are propogated through the tiles as shown in figure 7.11.

Notice that the T1,4 tile at the bottom of the string in figure 7.9 could have been any of the $T_{1,*}$ tiles. The string would be valid (with other tiles) so long as the respective edges

exist in the example instance graph. The $T_{2,2}$ tile could have also been any of the $T_{2,*}$ tiles in the same way, and so forth.

The relationship between the input vector X and output vector Z in figure 7.10 depends on the position of the tile in the path string because each tile must remove the node it chooses from the set of nodes that can be chosen by subsequent tiles. Equation (7.4) defines the relationship between a tile's input and its output and figure 7.11 illustrates an example 4 node tile set.

$$Z_j(i,n) = \left\{ \begin{array}{c} 0, j < N - i \\ \left\{ \begin{array}{l} X_j, j < \left\lfloor \dfrac{n-i}{i} \right\rfloor \\ X_{j+1}, j \geq \left\lfloor \dfrac{n-i}{i} \right\rfloor \end{array} \right\}, j \geq N - i \end{array} \right\} \tag{7.4}$$

where N is the number of nodes in the fully connected graph, the output and input ordinal (from figure 7.8) $j \in \{0 \ldots \text{N-1}\}$, the tile position (or step) $i \in \{1 \ldots \text{N}\}$, and the tile type $n \in \{1 \ldots \text{N}\}$.

The S_i signal from figure 7.10 is the node selection signal that tells a tile that its randomly chosen node is the end point of an edge in the problem instance. The logical AND of this signal and the one from the tile above (S_{i+1}) is used to store a bit that reflects the existence of the edge in the graph. This bit allows the output enable signal to propagate upward through the string so that if all the tiles have valid edges, an output circuit at the top of the string (of circuitry) may indicate to the controller that a Hamiltonian path exists in the graph.

Selecting edges in the instance graph

A circuit at the bottom of the stack of tiles provides graph edges to the string of tiles above. This circuit, illustrated in figure 7.12, uses a serial input, *X*, and a clock signal to shift bits into an N-bit shift register, where N is the number of nodes in the fully connected graph. The N-bit word shifted into the register must have a one as its left-most bit to trigger the

input to be connected (through pass-gates) to the bottom tile and later a clear signal to reset the register for the next edge.

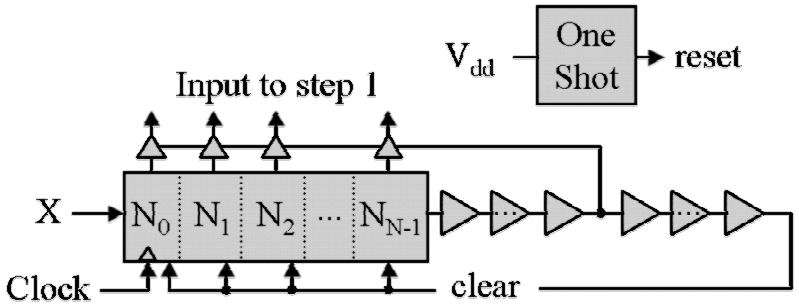

Figure 7.12. Serial control circuit that provides edge information to a string of tiles stacked above.

The *reset* signal is asserted at power-up long enough to reset all the RS-latches in the string of tiles. This allows a sequential list of edges to be clocked into the serial control circuit that selects each pair of tiles.

The circuits described above can be run at a clock rate of at least 400 MHz, as discussed in chapter 9, which means that a single edge can be selected from the fully connected graph in less than 30 ns. (10-bit edge identifiers clocked in serially (25 ns / edge) + 5 ns of propagation time per edge = 30 ns / edge.) Therefore, the time required by the HAM-PATH oracle to solve any 15 node graph problem (with $< 15^2$ edges) is less than 6.75 μs. The material limitations described in chapter 3, and later in chapter 9, prevent the HAM-PATH oracle from being able to handle graphs with more than 15 nodes. An Intel Pentium 4, with an estimated sustained path evaluation rate of 5.375×10^8 paths / second, can solve the worst-case 15 node Hamiltonian path problem in about 40 minutes. That is, the HAM-PATH oracle is 3.5×10^8 (350,000,000) times faster than the Intel Pentium 4 with worst-case 15 node Hamiltonian path problems.

Comparing the performance of the HAM-PATH oracle to a comparable supercomputer, the NEC Earth Simulator (ES), we find that the ES can perform an estimated 6.56×10^{12} paths / second or a complete problem in ~200 ms. This means that the HAM-

PATH oracle is ~40,000 times faster than the NEC Earth Simulator with worst-case 15 node Hamiltonian path problems.

The HAM-PATH oracle performance can be degraded by a factor of 40,000 and still match the ES performance. Since the power consumed by CMOS circuitry scales linearly with clock frequency the slower DAMP clock rate of 10 kHz would consume only 88 W of power compared to 12.8 MW for the ES. (The HAM-PATH oracle consumes ~15 mW while idle.)

Chapter 8. The decoupled array multi-processor (DAMP)

Architecture of practical value

En masse, simple processing elements can perform incredible computational feats. Consider multi-bit distributed computing projects like "SETI @ home" and the "United Devices Cancer Project" that use massive parallelism to accomplish super-computer scale tasks with idle desktop computer processing cycles. The early limitations of a self-assembling realization technology will require small circuitry. Single bit, serial processing elements are well suited to such limitations. They require less circuitry and have simple interfacing requirements.

The tradeoff between assembly-time and runtime complexity has at one end the oracle, described in chapter 4, and at the other end traditional sequential and parallel machines. The oracle has nearly no runtime complexity because its computation is performed during the assembly process. This limits the oracle to solve one particular class of problem. Greater flexibility comes by introducing more runtime capabilities into the machine design. The topic of this chapter is a particular machine design at the other end of the spectrum from the oracle. The decoupled array multi-processor (DAMP) employs much greater runtime computation to achieve greater problem flexibility than the oracle. In this sense, the DAMP is more practical because it has broader applicability that can offset the cost of developing the self-assembly techniques required to build the machine.

The DAMP is similar to traditional single-instruction multiple-data (SIMD) machines with two important differences: no inter-processor communication, and many more processors. The most significant difference is the lack of any communication hardware between processors. The large machines found in supercomputing centers today have high-bandwidth interconnections between processors. Unlike these modern machines, the processors in the decoupled array multi-processor have no way to communicate with each

other except through a shared control unit. This limits the decoupled array multi-processor to "embarrassingly parallel" problems.

The magnitude of the number of processors in each machine type is also dramatically different. Whereas most machines typically have less than 100,000 processors, the decoupled array multi-processor has on the order of 10^{12} processors (at least seven orders of magnitude larger than in conventional SIMD designs.) The complexity of any individual processor is greatly diminished with respect to the processors used in SIMD machines.

For comparison, an Intel Pentium 4 (P4) has ~55 x 10^6 transistors while a single DAMP processor has ~1600 transistors. That is, the P4 has the equivalent of 34,375 DAMP processors worth of transistors. The entire DAMP has ~1.75 x 10^{15} transistors, or the equivalent of ~32,000,000 Intel Pentium 4 processors. If a single P4 (and its accompanying hardware) occupies a 0.08 m^3 volume, the number of P4s that are equivalent to the entire DAMP (12 m^3 volume) would completely occupy 2.5 x 10^6 m^3 or a square room ~3,300 feet on a side with a 6-foot high ceiling.

The following sections describe in detail the architecture, implementation, and the first-level realization of the decoupled array multi-processor. The last section of this chapter deals with design / yield tradeoffs that are important to consider in light of finite material budgets.

8.1 Architectural description of the machine

The basic structure of the DAMP is illustrated in figure 8.1. A node controller sends control signals to each processor node. The processor node transmits these signals to each processor under its control through two signaling electrodes.

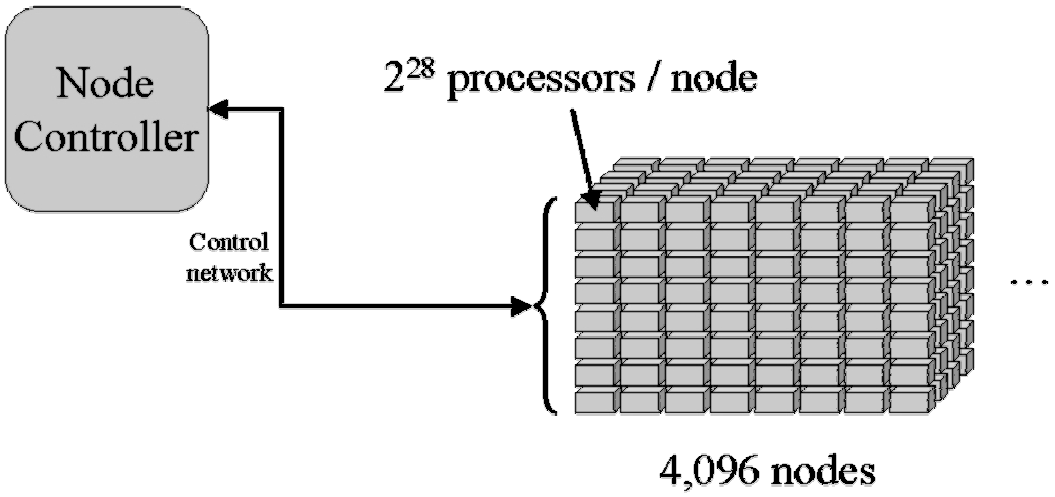

Figure 8.1. The node controller and processor node arrangement.

A single instruction stream is broadcast to all processors in a node and each processor conditionally executes the instructions depending on the value of the wait-status bits. Figure 8.2 illustrates the basic programming model for a DAMP processor. The processor is a bit serial machine with a 16-bit accumulator and five 16-bit general-purpose registers. The accumulator, R0, and R1 have the unique ability to load a random constant that is unique to each processor. The random constant is determined at assembly time as described in the realization. It is used as an index or a random seed for placement in the problem space. Otherwise, there is no way to differentiate between processors.

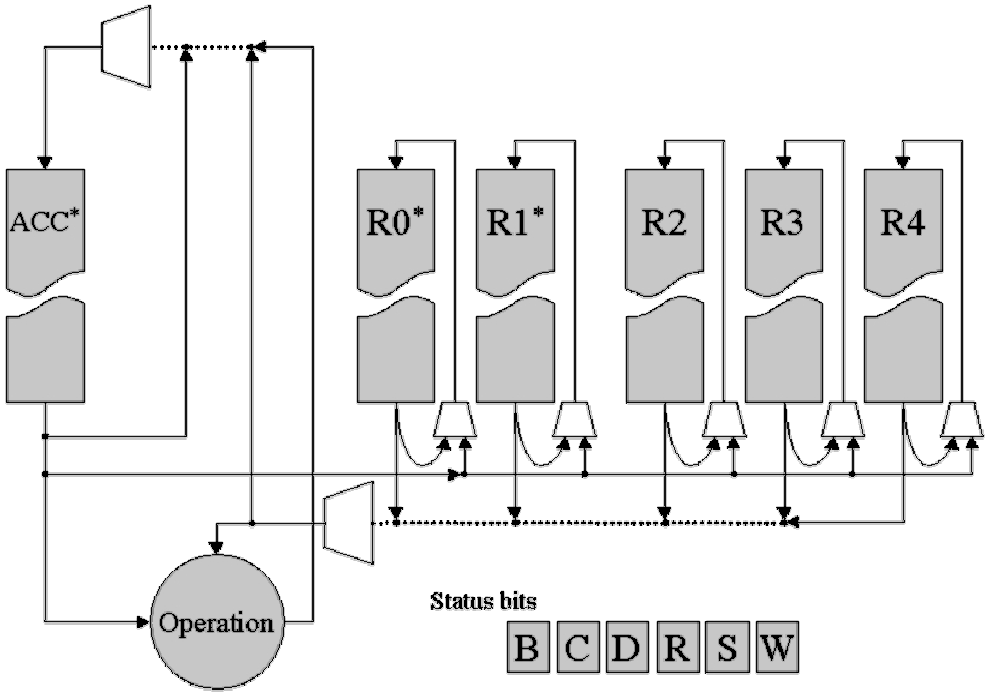

Figure 8.2. Processor diagram. ACC, R0, and R1 can be loaded with random bits.

In this bit-serial design, the least significant bit is the first bit to participate in each operation; the bit at the bottom of a register in figure 8.2. The accumulator can shift independently from the R0 to R4 registers because of separate shift controls, enabling relative data shifts. The operational unit is a full-adder that can provide either the carry out or sum signal to the accumulator input. Each register R0 through R4 can receive either their own lowest significant bit or the accumulator output as input during a shift.

The six status bits can be used to implement a wide range of operations as described in the later portion of this section. The definition and operation of each status bit is listed in table 8.1.

Status Bit	Description
B	Loads the current operand bit from R0 - R4
C	Loads the current carry out from the current operation
D	Detects a one on the output of the current operation
R	"Ringer" control set or cleared by special instruction. The control transceiver on the processor node can detect when any of the processors within the node have set R = 1
S	Loads with the output from the current operation
W	Set according to the value of another status bit (B, C, D, or S), or cleared. The machine ignores instructions when W = 1 with the exception that the RESUME instruction can clear W (W = 0) and start normal instruction execution.

Table 8.1. Description of the status bits.

The random constants mentioned earlier are a peculiar feature of this architecture that enables it to tackle large combinatorial problems. The constant replaces the processor index commonly used by SIMD machines. In a manner similar to DNA-computing [Adleman, 1994] the random constants can be cast as instances of problem variables. The input space for the DAMP considered here is illustrated in figure 8.3.

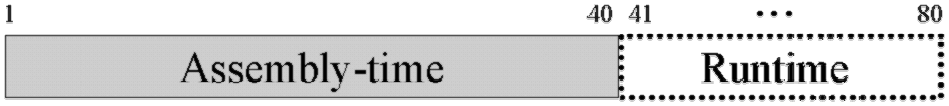

Figure 8.3. The input space of the DAMP.

The 2^{40} processors in the DAMP suffice to evaluate any 40 bit input space with only one run of a program. A program instructs each processor to manipulate their random inputs to produce an answer to a problem. If the answer is satisfactory, e.g. a minimum or below some threshold, the processor can alert the node controller and a binary search over that node's input space can begin. The search is complete when all bits of the random input used by the winning processor are determined by reading them one bit at a time as described in chapter 6 and section 11.1.

If the input space of a problem cannot be fit into 40 bits, additional bits computed at runtime may be used to augment them. By using a counter as the least significant bits of the input space, the 40 random bits can be treated as the most significant bits of the input space. Each time the program that calculates an answer from the input parameter is run, the counter is incremented. This will allow the DAMP to uniformly cover the larger input space, provided that the assembly-time random bits uniformly cover the 40-bit input space.

The low per-element yield of the fabrication technology used to build the DAMP demands that each processor be simple. This drives the use of bit serial processors and a simple controller. There is simply not enough room to devote circuitry to storing micro-code. As a consequence, all instructions are software encoded that is, synthesized by an assembler. This makes the DAMP rely heavily on assembler design.

Since there are only three defined low-level instructions (SETCREG, SETSREG, and CYCLE), the assembler can arbitrarily choose to implement high-level instructions that appear to be useful. The remainder of this section is devoted to the instruction set that is implemented by the assembler described in chapter 9. These instructions were chosen because they have been useful in diagnosing the logic implementation of the DAMP and have use in the programs described in chapter 11. In each of the instruction descriptions below, RX represents any register R0 to R4. In general, C/C++ style expressions are used to describe the operation of each instruction.

ADD(RX)	
Cycles	159
Operation	ACC = ACC + RX

Description: This instruction adds the 16-bit value in the accumulator to the 16-bit value stored in a register and replaces the accumulator value with the result. The carry status bit contains the value of the carry out from the operation. The carry status bit is reset before the operation begins.

ADDC(RX)	
Cycles	131
Operation	ACC = ACC + RX +C

Description: ADDC behaves similarly to ADD with the exception that the current carry bit is also included in the sum. This can be used to implement 2's complement subtraction as follows:

```
// Implement ACC = RX - ACC
NOT
SETC
ADDC(RX)
```

The ADDC instruction can also be used to implement multi-word arithmetic beyond 16 bits.

ADDI(constant)	
Cycles	170
Operation	ACC = ACC + 16-bit constant

Description: ADDI simply adds a 16-bit literal to the accumulator. The carry bit is cleared before the operation begins.

ANDI(constant)	
Cycles	136
Operation	ACC = ACC & 16-bit constant

Description: This instruction performs a bit-wise AND operation between the 16-bit literal and the contents of the accumulator. The result is stored back into the accumulator.

ASR(N)	
Cycles	116 + N
Operation	ACC = ACC ÷ 2^N

Description: ASR implements an arithmetic shift right. That is, the accumulator is shifted to the right (toward the least significant end) and the current sign bit is used to pad the upper portion.

CLEARB	
Cycles	77
Operation	B = 0

Description: Clears the 'B' status bit. That is, 'B' = 0.

CLEARC	
Cycles	55
Operation	C = 0

Description: Clears the 'C' status bit. That is, 'C' = 0.

CLEARD	
Cycles	77
Operation	D = 0

Description: Clears the 'D' status bit. That is, 'D' = 0.

CMP	
Cycles	131
Operation	D = !(ACC == 0)

Description: This instruction compares the accumulator with zero and if they are the same clears the 'D' status bit (D = 0). If the accumulator is not equal to zero, the 'D' status bit is set (D = 1).

CMPI(constant)	
Cycles	307
Operation	D = !(ACC == 16-bit constant)

Description: Similar to the CMP instruction, CMPI clears the 'D' status bit if the accumulator is equal to the 16-bit literal. If the accumulator is not equal to the 16-bit literal, the 'D' status bit is set (D = 1).

CMPI8(constant)	
Cycles	211
Operation	D = !(0xFF & ACC == 8-bit constant); CSR(8)

Description: This instruction compares the lower 8-bits of the accumulator with an 8-bit literal and clears the 'D' status bit if they are equal. If the 8-bit literal and the lower 8-bits of the accumulator are not equal, the 'D' status bit is set (D = 1). This instruction has the side-effect of circularly shifting the accumulator by 8-bits (see CSR(N)).

COPY(RX)	
Cycles	92
Operation	ACC = RX

Description: The COPY instruction is used to copy the contents of a register into the accumulator.

COPYH(RX)	
Cycles	215
Operation	ACC = (0xFF00 & RX) ÷ 256

Description: COPYH will copy the upper 8-bits of the specified register into the lower 8-bits of the accumulator padding the upper 8-bits of the accumulator with the register's sign bit.

COPYL(RX)	
Cycles	154
Operation	ACC = RX ÷ 256

Description: Similar to the COPYH instruction, COPYL copies the lower 8-bits from a register into the accumulator. The upper 8-bits of the accumulator receive the register's sign bit.

COST(RX1, RX2)	
Cycles	120
Operation	RX2 = ACC; ACC = RX1

Description: The COST ("Copy and Store") instruction merges two common instructions into one because each can be made more efficient by using the other's control

signals. First, COST stores (see STORE) the contents of the accumulator into RX2. Secondly, COST copies (see COPY) the contents of RX1 into the accumulator.

CSR(N)	
Cycles	88 + N
Operation	S = ACC[0] ACC = (ACC >> N) | (ACC << (16 - N))

Description: CSR implements a circular right-shift of the accumulator by N bits.

CYCLE(N)	
Cycles	N
Operation	-

Description: CYCLE is used to shift bits and execute operations setup by the SETCREG and SETSREG instructions. This instruction can be used to synthesize instructions not previously defined by the assembler. See the implementation details in section 8.2 for more information.

DEC	
Cycles	159
Operation	ACC = ACC - 1

Description: The DEC instruction decrements the accumulator by 1. If the contents of the accumulator decrement past zero, the carry bit will be set and the accumulator filled with 1s.

GRAB(N)	
Cycles	157
Operation	S = ACC[N]

Description: This instruction is used to copy the N^{th} bit from the accumulator into the 'S' status-bit. This is useful in interpreting a value as a bit mask.

INC	
Cycles	159
Operation	ACC = ACC + 1

Description: This instruction increments the accumulator. If the value of the accumulator increments to zero (overflow), the carry bit will be set.

LOAD(constant)	
Cycles	280
Operation	ACC = 16-bit constant

Description: The LOAD instruction is used to load a 16-bit constant into the accumulator.

LSR(N)	
Cycles	88 + N
Operation	S = ACC[0] ACC = ACC >> N

Description: LSR implements a logical right-shift of the accumulator by N bits. That is, the contents of the accumulator are shifted toward the least significant end by N bits. Zeros are shifted into the upper N bits of the accumulator.

LSRC(N)	
Cycles	66 + N
Operation	ACC = (ACC >> N) | ('C' & (2^{16-N} - 1))

Description: LSRC is similar to LSR with the exception that the logical right-shift pulls the current carry bit into the upper N bits of the accumulator.

MCOPY(constant, RX)	
Cycles	103
Operation	ACC = (ACC & !(16-bit constant)) | (16-bit constant & RX)

Description: The masked-copy instruction, MCOPY, uses a 16-bit literal to select bits from a register to be copied into the accumulator. For example, MCOPY(0x0FF0, R4) copies bits 4 - 11 from register R4 into bits 4 - 11 of the accumulator. The remaining bits in the accumulator are left unchanged.

NOT	
Cycles	171
Operation	ACC = !ACC

Description: This instruction stores the logical complement of the accumulator back into the accumulator.

ORI(constant)	
Cycles	202
Operation	ACC = ACC | 16-bit constant

Description: ORI stores the bit-wise logical OR of a 16-bit literal and the accumulator back into the accumulator.

RANDOM	
Cycles	55
Operation	ACC = rand; R0 = rand; R1 = rand;

Description: RANDOM loads three processor dependent 16-bit random values into the accumulator, R0, and R1 registers. The number for each processor is determined at assembly time as described in chapter 9.

RESUME	
Cycles	6
Operation	'W' = 0

Description: This instruction clears the 'W' status bit on all processors. RESUME instructs any previously waiting processor to begin executing instructions.

RINGOFF	
Cycles	11
Operation	'R' = 0

Description: RINGOFF is used to turn off the output circuitry of the processor. Each processor node can detect when the output circuitry of a processor is on, that is when 'R' = 1. This instruction is used to turn that circuitry off, or to clear the 'R' status bit (R = 0). See section 6.3 for further details.

RINGON	
Cycles	11
Operation	'R' = 1

Description: The RINGON instruction sets the 'R' status bit (R = 1). This creates an electrical condition that the node controller can detect, as described in section 6.3. See RINGOFF for more information.

SETGE, SETLT, SETNGE, SETNLT	
Cycles	55
Operation	Sets 'C' to the appropriate value so that the corresponding WAIT instruction will set 'W' = 1.

Description: These instructions are helpful in restoring the wait status bit just before a RESUME instruction. The suffix indicates which status bit is consulted before setting or not setting the 'W' status bit. The GE, LT, NGE, and NLT suffixes mean greater-than-or-equal, less-than, not greater-than-or-equal, and not less-than respectively.

A simple IF-THEN-ELSE clause can be implemented as follows:

```
// Implement IF-THEN-ELSE statement
CMPI(constant)    // if(ACC >= constant) {
WAITNGE     // All processors failing the condition WAIT...
...THEN-clause instructions...
SETGE       // restore the status bit
RESUME      // All processors RESUME execution...
WAITGE      // All processors that had previously executed
            //  the THEN-clause WAIT...
            // } else {
...ELSE-clause instructions...
RESUME      // }  // All processors RESUME execution...
```

Nested IF-THEN-ELSE statements can be implement in a similar manner as long as the corresponding SET* and WAIT* instructions are executed before and after each clause (or closing bracket.)

SETB	
Cycles	77
Operation	'B' = 1

Description: This instruction sets the 'B' status bit. That is, B = 1.

SETCREG(constant)	
Cycles	6
Operation	-

Description: This instruction is used to set the target control register that SETSREG will eventually be used to fill. SETCREG, in conjunction with SETSREG and CYCLE, can be used to implement instructions that have not been previously defined by the assembler.

SETC	
Cycles	55
Operation	'C' = 1

Description: This instruction sets the 'C' status bit. That is, C = 1.

SETSREG(constant)	
Cycles	5
Operation	-

Description: SETSREG loads a constant into the control register pointed to by a previous SETCREG instruction. See SETCREG and CYCLE for more information. See also, section 8.2 and chapter 3.

STORE(RX)	
Cycles	109
Operation	RX = ACC

Description: The STORE instruction stores the contents of the accumulator in the specified register.

STOREH(RX)	
Cycles	171
Operation	RX = (RX & 0xFF) | ((ACC & 0xFF) << 8)

Description: Similar to the COPYL and COPYH instructions, STOREH stores the lower 8-bits of the accumulator in the upper 8-bits of the specified register. The lower 8-bits of the register are preserved.

STOREL(RX)	
Cycles	109
Operation	RX = (RX & 0xFF00) | (ACC & 0xFF)

Description: The STOREL instruction stores the lower 8-bits of the accumulator in the lower 8-bits of the specified register. The upper 8-bits of the register are preserved.

WAITB, WAITNB, WAITC, WAITNC, WAITD, WAITND, WAITS, WAITNS, WAITGE, WAITNGE, WAITLT, WAITNLT	
Cycles	12
Operation	Set 'W' = 1 if the status bit condition is true.

Description: These instructions set the 'W' status bit (W = 1) if the corresponding status bit is set. The suffix indicates which status bit is consulted before setting or not setting the 'W' status bit. The GE, LT, NGE, and NLT suffixes mean greater-than-or-equal, less-than, not greater-than-or-equal, and not less-than respectively. These suffixes will cause the 'W' status bit to be set if a previous CMP or subtraction (see ADDC) generated the condition implied by the suffix.

XOR(RX)	
Cycles	148
Operation	ACC = ACC ^ RX

Description: This instruction stores the bit-wise exclusive-OR of the accumulator and specified register back into the accumulator.

8.2 One implementation of the DAMP

The 2^{40} processors in the DAMP are spread out evenly across 4,096 processor nodes, 2^{28} processors per node. As described in chapter 6, the processors at each node are contained within a processor substrate that is a thin wafer of processors either connected on both sides to metallic electrodes or through a patterned silicon substrate. The output circuit will be described in more detail and is simulated in chapter 9. As described below, each processor node is connected to a high-speed control network that is capable of distributing control

words at a rate high enough to keep up with the maximum processor clock rate. The processor node requires sufficient packet buffering capability to deal with control network traffic.

A central node controller distributes signaling commands to the array of processor nodes. The node controller is responsible for splitting up a problem space among the processor nodes and collecting the results. Chapter 9 discusses the simulation of the circuitry for this and estimates of the maximum clock rate. At a clock rate of 400 MHz and an average of 158 bits per integer instruction, the node controller must be able to distribute about 3 Mbps to each processor node. The specific topology of the control network will determine how fast the network interface on the node controller needs to be. Even if each processor node is connected to a single interface on the node controller the total throughput requirement will be less than 4,096 x 3 Mbps, or about 12 Gbps to the processor nodes.

The DAMP processors are implemented using complementary metal-oxide semiconductor (CMOS) logic circuitry as described in chapter 6. This circuitry behaves similarly to conventional CMOS circuitry and can be treated as such in this implementation. The performance of the DAMP will be evaluated using a power budget similar to that of the NEC Earth Simulator and IBM BlueGene /L, or ~3.5 MW. This makes the power allocation about 850 W per processor node.

Each DAMP processor element has a register file of five 16-bit registers, a register control unit (RCU), an arithmetic-logic unit (ALU), a control state machine (CSM), control registers, and a wait & trigger controller (WTC). The four-gate path from the control register to the reset signal in the WTC is the longest logic path through the processor, and is therefore the critical timing path. The following describes each of these units and the logic from which they are built.

8.2.1 Register file, register control unit (RCU), and arithmetic-logic unit (ALU)

Figure 8.4 illustrates the logic and arrangement of the register file, RCU, and ALU. The register file is composed of six 17-bit shift registers. Even though each register has 17 bits, the last bit is consumed during a shift operation leaving an effective 16 bits per register. One of the six shift registers is controlled independently from the other five. The five

dependent registers are numbered R0 through R4. The one independent register is called the accumulator (ACC). The accumulator, R0, and R1 can each be loaded with a random number that was encoded into their circuitry during assembly. This circuitry, described in chapter 9, requires the register values to be zero before the random bits can be loaded.

The number of bits per register can be easily increased as the processing yield is improved since the entire machine is serial. In particular, the fluidic self-assembly yield must be improved so that more modules, as described in chapter 3 and later in section 6.3, can be used per processor. The extra modules can then be devoted to register bits without redesigning the remainder of the processor. This will, however, increase the power consumed per instruction and the simulations in chapter 9 will need to be re-done.

The RCU is used to control the source and destination of bits for the R0-R4 registers and the accumulator. The multiplexers of the RCU coordinates swaps, copies, and bit interleaving between registers. Additionally, the RCU selects operands for the ALU and controls what signals the accumulator will use as input.

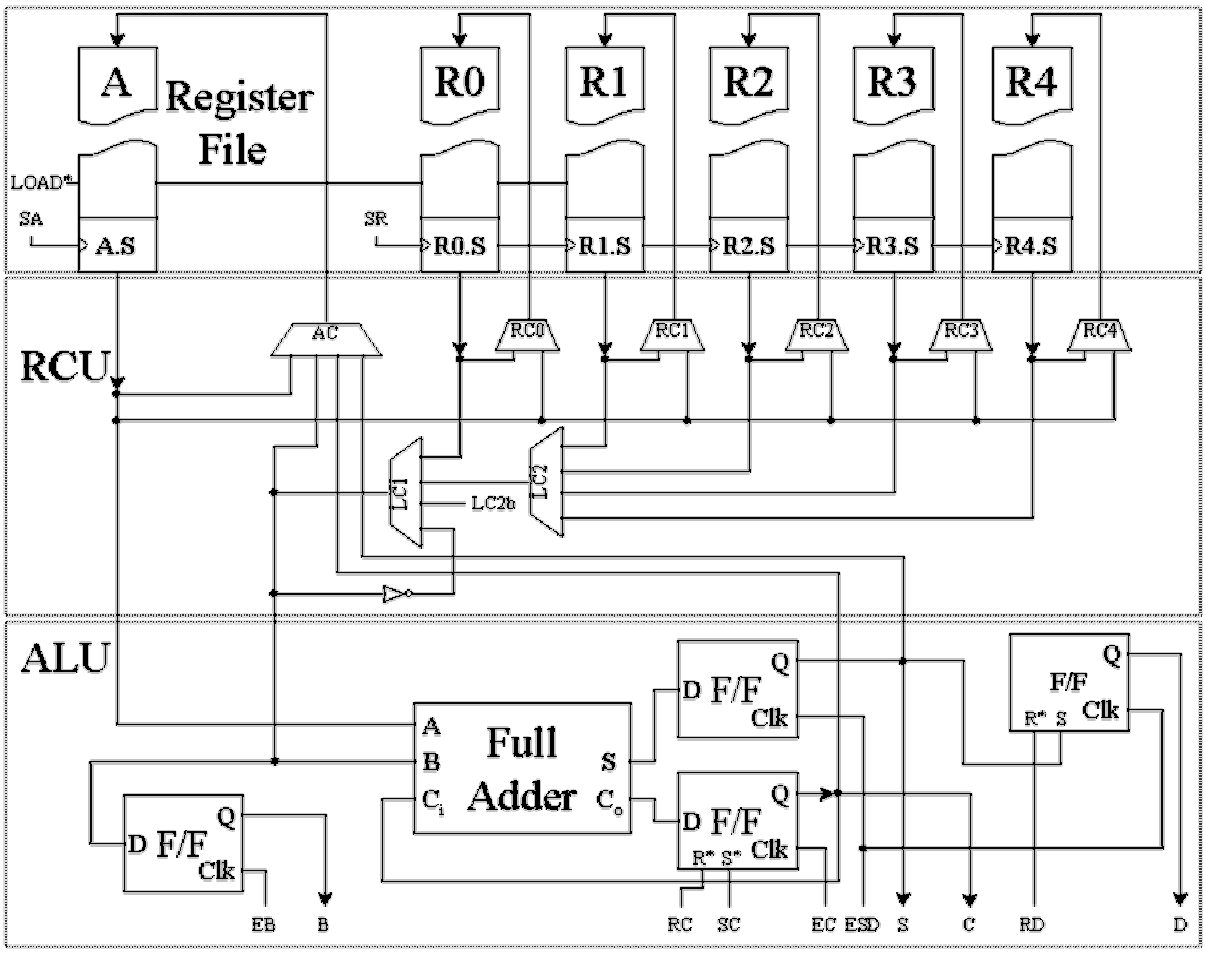

Figure 8.4. Implementation of the register file, register control unit (RCU), and arithmetic & logic unit (ALU). "R*" and "S*" are asynchronous reset and set control signals, respectively. The AC (2-bits), RC0 - RC4 (1-bit each), LC1 (2-bits), and LC2 (2-bits) multiplexers are each set by control bits from the control registers.

8.2.2 Control state machine (CSM)

Figures 8.5 and 8.6 illustrate the control flow and logic of the CSM. The inputs to the CSM are the immediate bit (X) and clock signal (CLK) taken from the control electrodes that sandwich the processor. The CSM is the starting point for all asynchronous and synchronous signals in the processor.

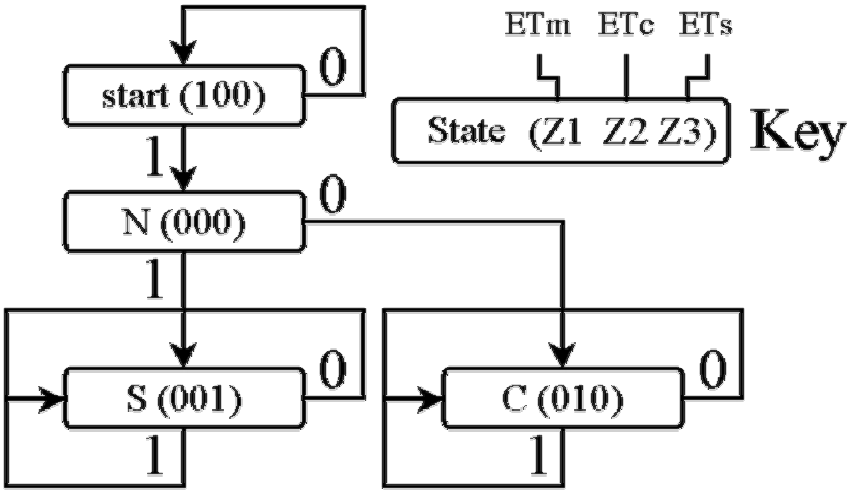

Figure 8.5. The serial control diagram for the control state machine. An input bit of 0 moves along the side-arrow while an input bit of 1 moves along the bottom-arrow. "N" is the neutral-state, "S" is the setup-state, and "C" is the control-state.

The state machine will be in the 'start' state upon reset. An input sequence of '11' puts the machine into the 'S', or setup state, and a sequence of '10' puts the machine into the 'C', or control state. The 'S' and 'C' states are sinks that can only be moved from with a reset (RESET_S0S1.) The four states of the CSM can be encoded with two bits (S0 and S1.) The reset signal is generated after a setup register or control register has been filled with data. This is described in more detail below.

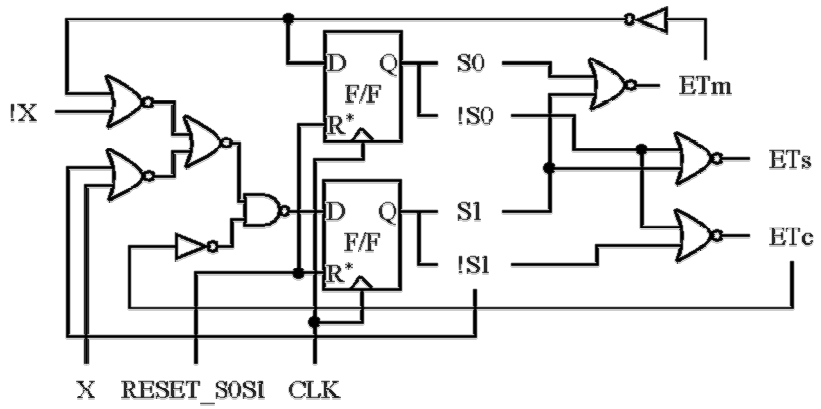

Figure 8.6. The control state machine (CSM) that implements the state diagram from figure 8.5. "R*" is an asynchronous reset control signal.

8.2.3 Control registers

Figure 8.7 illustrates the structure and logic used by the control registers. These registers control the multiplexers and decoders found throughout the processor. For example, LC1a, and LC1b are the two control bits that set the LC1 multiplexer. When the word (LC1a, LC1b) is 00, LC1 selects its top-most input, R0. Table 8.2 lists the various multiplexer control words and input selections.

Multiplexer	Control Signal Word	Inputs Selected by Control Word
AC	ACa, ACb	00: ACC, 01: LC1, 10: Co, 11: S
LC1	LC1a, LC1b	00: R0, 01: LC2, 10: LC2b, 11: !LC1*
LC2	LC2a, LC2b	00: R1, 01: R2, 10: R3, 11: R4
WC1	WC1a, WC1b	00: B, 01: C, 10: D, 11: S
WC2	WC2	0: WC1, 1: !WC1
RC0 - RC4	RCx	0: Rx, 1: ACC

Table 8.2. Multiplexer selections. * - the LC1 multiplexer is used to signal a bit to the controller. See chapter 6 for details.

Each control signal has been grouped into a control register by function. For example, control register S1 selects the input bit for the accumulator. This is a common operation and should be efficient, therefore no other signals are grouped into the S1 control

register. In order to minimize circuitry, each control register has a detection circuit attached to its right-most bit. This detection circuitry, found in the WTC, triggers a reset condition after the contents of the control register have been loaded. For example, before a control register is written to a clear signal is generated by the WTC. This clear signal asynchronously sets the contents of the target control register to zero. Next, the control word for the register is serially clocked into the register with a one as its prefix. The prefix bit signals to the detection circuit that the word has been clocked in successfully by shifting out at the right end of the register. This simple scheme replaces the need for a counter or other state machine to keep track of the progress of control word loading.

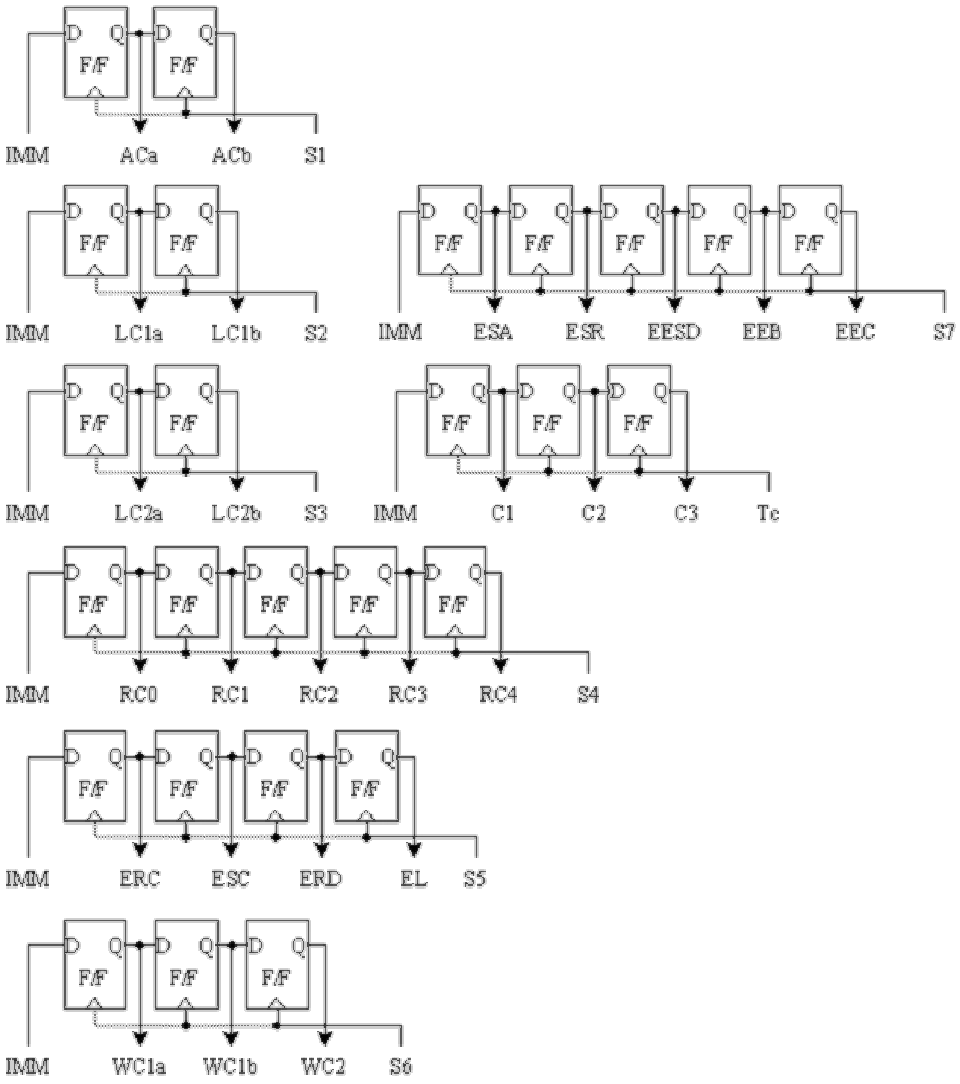

Figure 8.7. The control registers. Each shift register is triggered by a circuit from the wait & trigger control unit (WTC). The input to each register is taken from the current serial input bit (IMM).

8.2.4 Wait & trigger controller (WTC)

Figure 8.8 illustrates the logic and structure of the WTC. The WTC is primarily responsible for generating control register shift signals, register file shift signals, the reset signal, and the wait signal. The WTC also takes input from the CSM to indicate when the selected control register should be cleared.

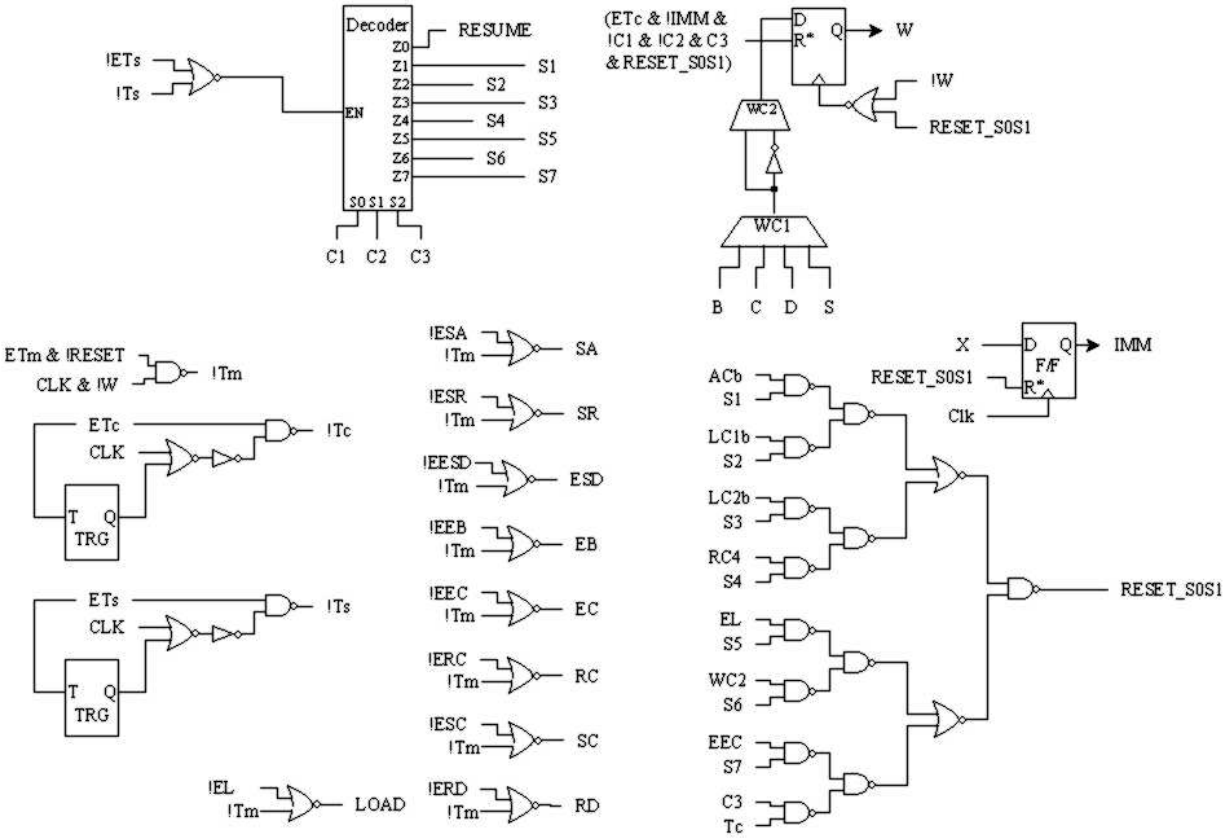

Figure 8.8. Implementation of the wait & trigger controller (WTC).

The instructions implemented by the assembler described in section 8.1 require specific bit-serial signals to instruct the processor to perform the given operation. The implementation details for these instructions can be found in appendix C.

Physical housing

The size of the physical enclosure for a processor node depends on the yield estimates made in chapter 3 and further expanded in section 8.4. These yield estimates indicate that we can expect 16% of the total number of processors in a node, or about 2^{28} processors, to work properly (7×10^{12} processors / 4096 nodes = 1.7×10^{9} processors per node.) Since the non-functional processors take up space on the substrate, the number of processors along the side

of a square substrate is the square root of the total number of processors per node, or about 41,231 processors.

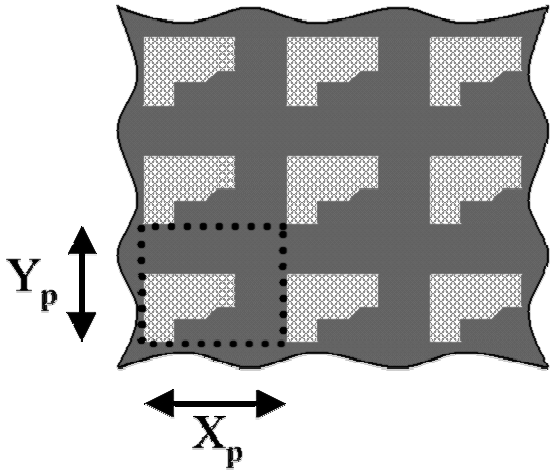

Figure 8.9. Processor spacing pitch on the substrate.

The processor pitch X_p and Y_p is equal to 5 µm, illustrated in figure 8.9, yielding a processor substrate width, W_N in figure 8.10, of approximately 205 mm square (7.4" x 7.4".) This pitch can be taken from the nanorod layout described in section 8.3. If a 0.5 µm thick metallic electrode is used as the top electrode and the substrate is 5 mm thick, the total processor substrate thickness, H_N in figure 8.10, is about 5.125 mm.

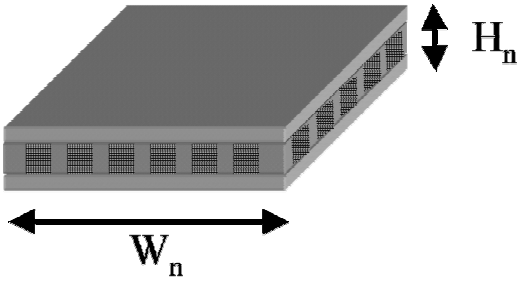

Figure 8.10. Processor substrate dimensions.

The dimensions of the processor substrate directly impact the size of the processor node housing, illustrated in figure 8.11. If we use a 20% margin on each side of the processor substrate W_N = 246 mm, or about 9". The space above the processor substrate will need to be large enough to accommodate the bubbles created by a boiling pool of coolant. Without experimental data on the size of pool boiling bubble sizes, it is difficult to know how large this space should be. However, this spacing must be determined to estimate the size of the DAMP. As a conservative estimate, the spacing will be taken as 5 times the thickness of the processor substrate, or about 25 mm (~1").

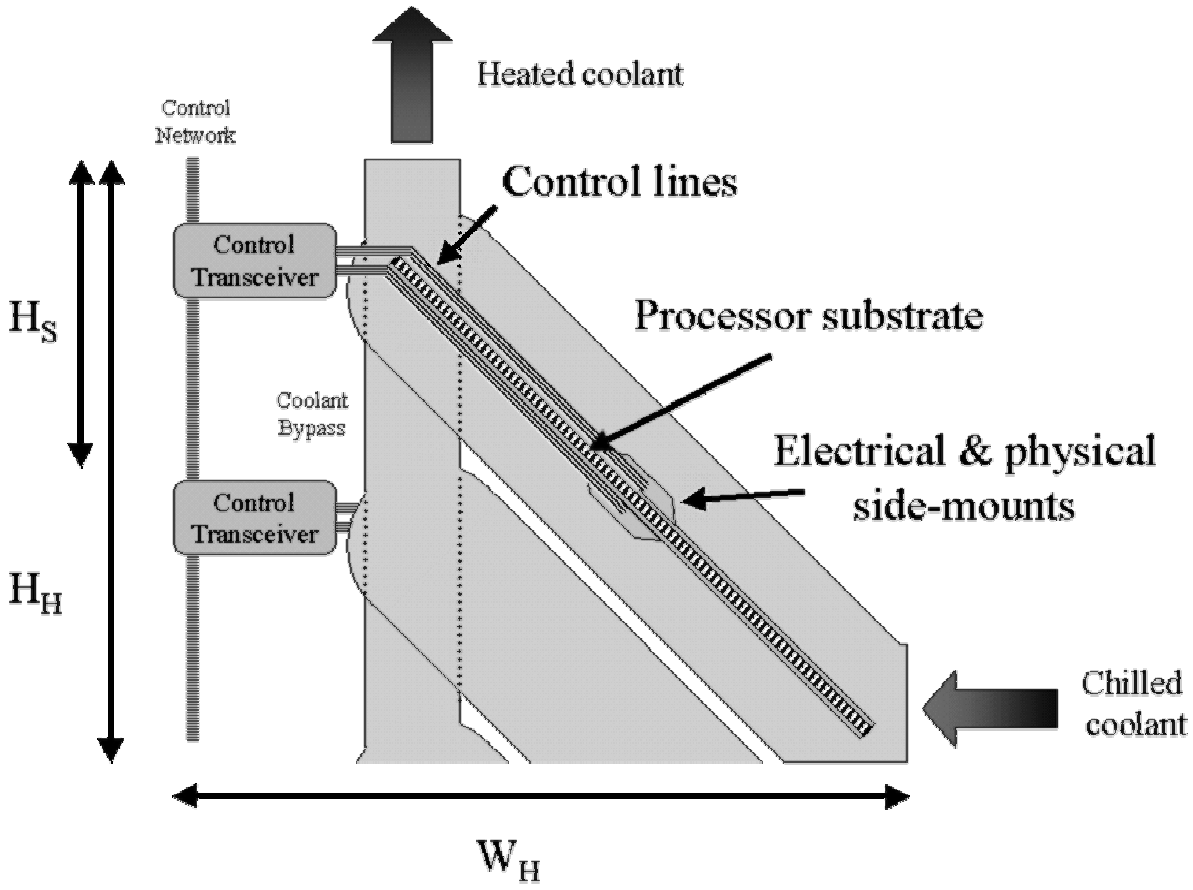

Figure 8.11. Processor node housing and cooling jacket. The housing has a stacking height, H_S, housing height, H_H, and a housing width, W_H.

The stacking height, H_S from figure 8.11, can be estimated by calculating the distance between processor node housings. Since the housing is 25 mm thick, the minimum pitch is also 25 mm, or about 1". Again, adding a 20% margin on the spacing makes for a pessimistic housing pitch of 30 mm, or about 1.1". The housing is held at a 45˚ angle to allow convection and pool boiling to remove heat. The housing height, H_H, is therefore $W_N \cdot \sin(45^\circ) = 174$ mm, or about 6.3".

The final dimension to be determined for the housing is W_H, or the housing width. This parameter is needed to determine how closely each node stack can be packed. To determine this parameter, the size of the control transceiver must be estimated. The control circuitry can extend along the 9" width of the node housing (into the paper in figure 8.11)

and be at least as deep as the node stacking height, or 1". A conservative estimate of 3" for the control circuitry extension makes the housing width parameter W_H about 3" longer than H_H, or 7.3", given the 45˚ angle of the housing.

The volume of a DAMP rack is the volume of a single stacked-processor node multiplied by the number of nodes per rack. If a rack is 6' tall, then it can house about 72 nodes. Each rack has a footprint of about 8" x 9" without considering the coolant delivery and return lines. Since each processor node has 2^{28} processors and each rack has 72 nodes, a full DAMP (2^{40} processors), requires about 57 racks. These racks can fit into a 6' x 6' footprint. Again, these estimates do not take into account the coolant delivery and return lines.

8.3 Nanorod layout

This section examines the 3D layout of several commonly occurring circuit structures in the DAMP. The complete layout of the DAMP is left as future work but would include each of these component circuit structures. The circuit schematics and simulation results for these structures can be found in chapter 9. The most important goal of producing these layouts was to determine a first draft footprint and size for each circuit so that later yield estimates could be based on actual designs rather than *estimates* of circuit designs. As with conventional VLSI, signal routing was the biggest challenge to fitting a circuit within the standard footprint. This footprint was chosen because each required circuit could fit and that it was small enough to be plausibly fabricated. The footprint shape was also chosen to have neither rotational nor reflectional symmetry, thus ensuring proper unambiguous lock-and-key self-assembly.

The illustration in figure 8.12 is the layout footprint. Each circuit was designed to have a projection that fits within the layout footprint. The two full sides of the footprint measure approximately 4 µm and 3.5 µm.

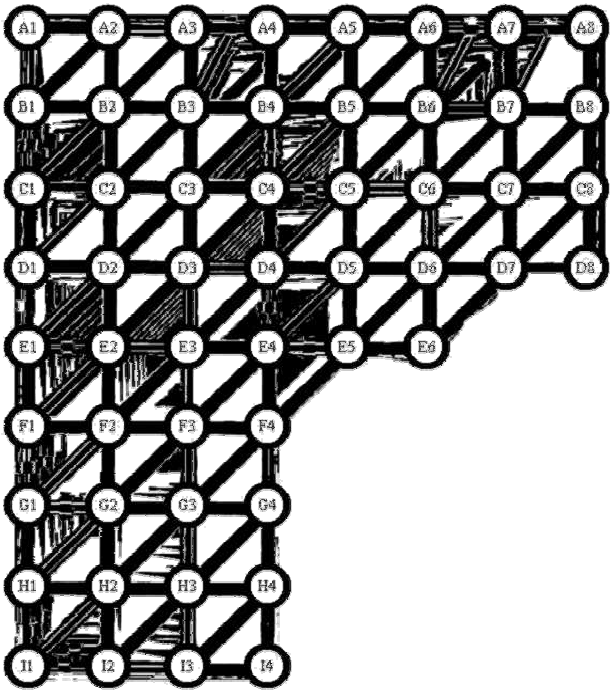

Figure 8.12. Layout footprint with row and column indicators.

Some of the layouts illustrated later in this section does not have the exact outline of the footprint. This is because many of the basic circuits do not occupy the entire footprint and can be put side-by-side with other circuitry. These primitive circuits can be pieced together to fit into the full DAMP layout footprint. All layout designs, except figure 8.13, do not show the insulating rods.

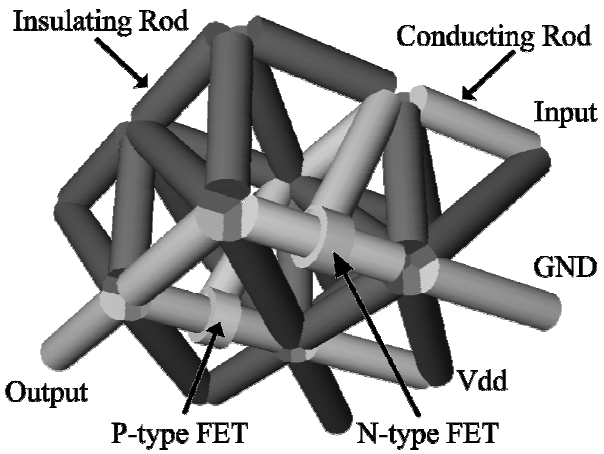

Figure 8.13. Nanorod layout of an inverter. Insulating and conducting rods are used for support. Circuit diagram in figure 9.11.

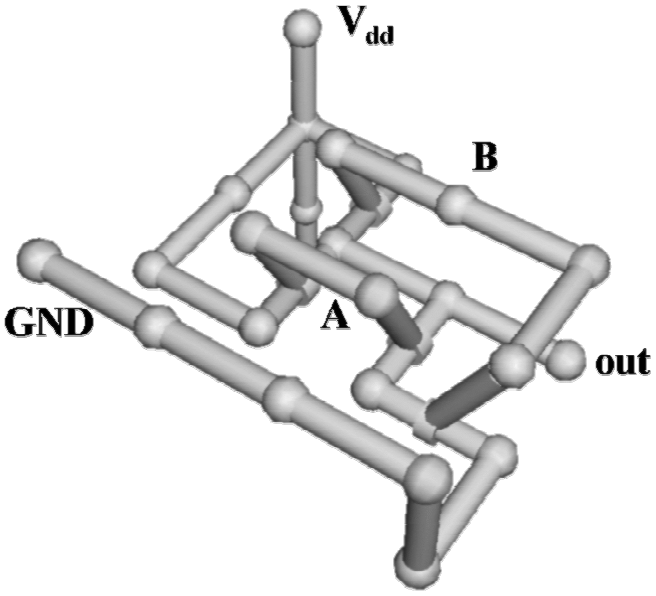

Figure 8.14. A view of the nanorod layout for a NAND gate. Circuit diagram in 9.12.

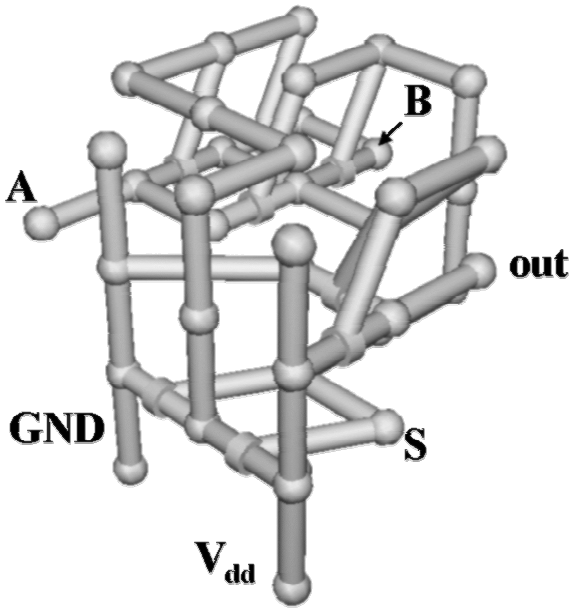

Figure 8.15. The 2-input multiplexer (MUX2). Circuit diagram in figure 9.13.

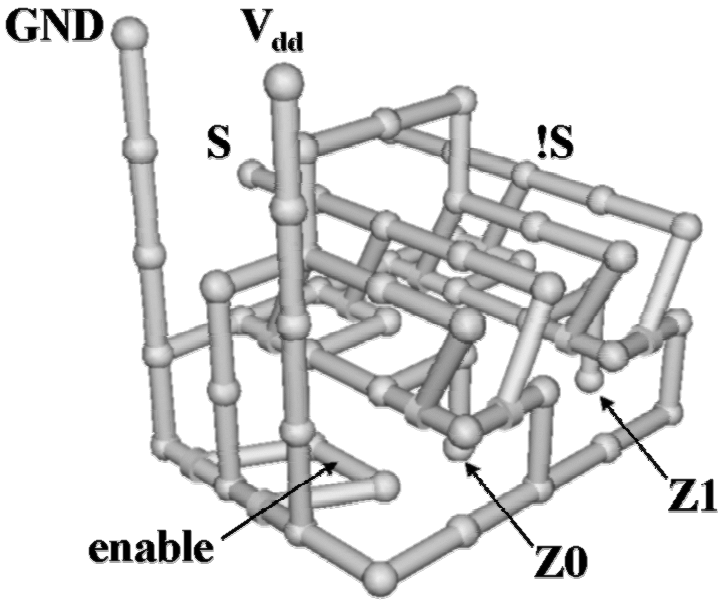

Figure 8.16. One-bit decoder. Circuit diagram in figure 9.14.

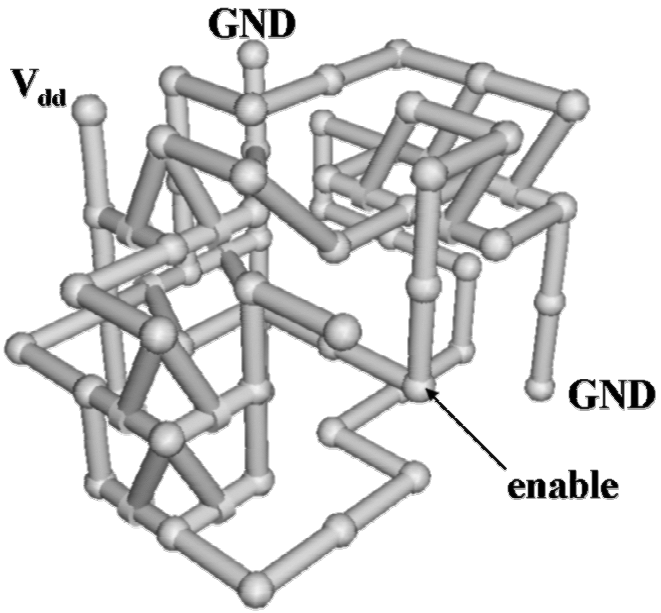

Figure 8.17. The ringer circuit. A two input multiplexer, with an inverter on its output, is tied back to an input. Selecting that input will create an externally detectable oscillating power signature. Circuit diagram in figure 9.18.

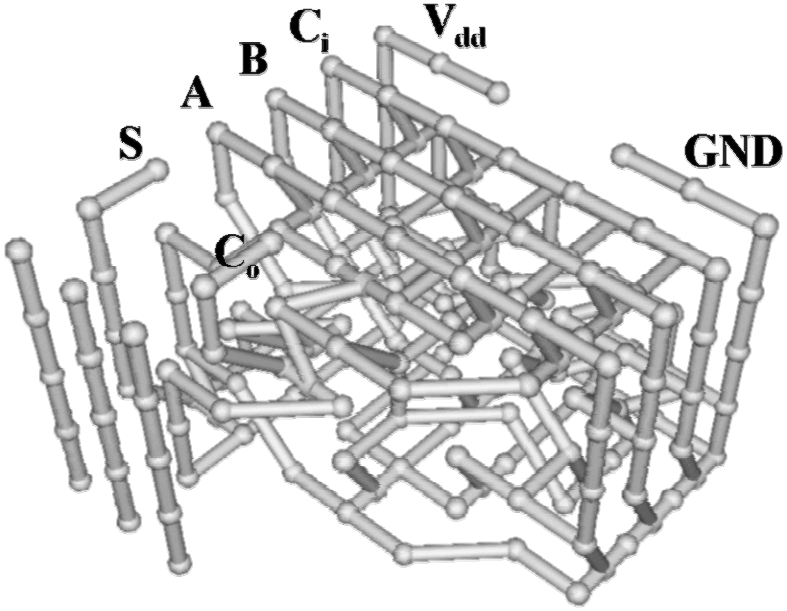

Figure 8.18. A view of the nanorod layout for a full adder. Circuit diagram in figure 9.15.

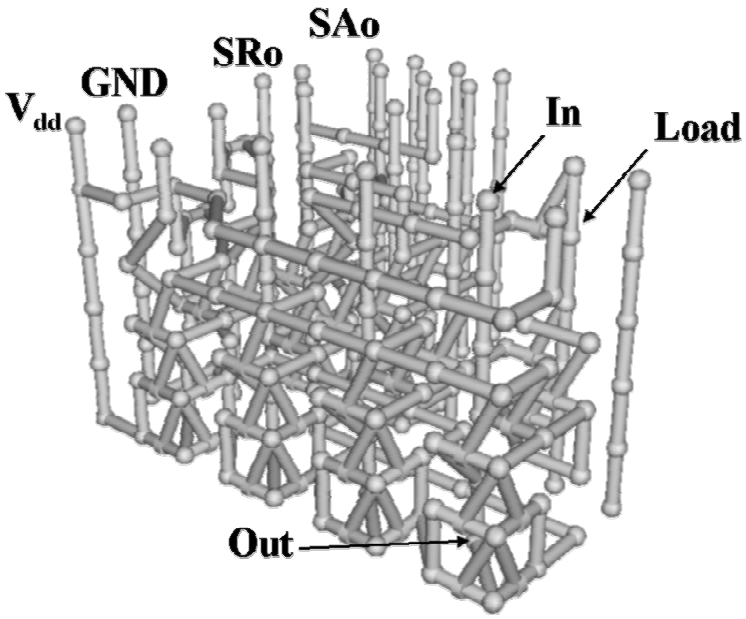

Figure 8.19. View of the nanorod layout for a single bit of the six-register cell. One of the six D-latches is clocked independently (for the accumulator) from the other five by using SAo/SAi instead of SRo/SRi. Only one of the six pairs of In and Out signals are labelled, all are present. Some GND signal labels have also been omitted. Circuit diagram in figure 9.17.

The register cell from figure 8.19 uses randomized rod assembly events to create a random constant that can be loaded into three of the six registers. The 'Load' signal is tied to a load circuit by a single rod for each of the three random-enabled registers. If the rod is conducting, the particular bit will be raised to a one. If the rod is insulating, the particular bit will remain unchanged. A mixture of insulating rods and conducting rods can be used during that step in the assembly to uniformly distribute random bits since each rod has a 50% chance of being either insulating or conducting. The load procedure requires the initial contents of the register to be zero since the load circuitry cannot set bits to zero.

This mechanism of introducing random bits into the register file should produce a uniform distribution of numbers since each bit has a 50% chance of being either a one or a zero. Since three of the six registers (each with 16 bits) can be loaded randomly, 2^{48} processors are required to cover the entire 48-bit input space. The DAMP has only 2^{40} processors, which means that the 48 random bits must be mapped to 40 bits for the input space to be fully covered.

If the distribution of bits within the registers is uniform, then the probability that a given $\log_2(M)$-bit value does not occur in any processor is (M-1 / M). For example, if M=4 and we pick a value randomly and compare it against any other value it is clear that there is a 3/4, or 75%, chance that the two will *not* be the same. The probability of *not* choosing a particular value after two trials is $(3/4)^2$ or 56%. This can be generalized as follows. The probability of not choosing a given value from M distinct values after N trials is $(M-1 / M)^N$. In the case of $M=2^{28}$ (28-bit values and $N=2.68 \times 10^8$, the processor count per node from chapter 8), the probability of not choosing a given 28-bit value is 36.7%. That is, roughly 1 in 3 processor nodes will have at least one missing 28-bit value. However, since the processors have 48-bit random values, the smaller 28-bit space *may* be covered if an appropriate hash function is used to convert the 48-bit number to a 28-bit number. This may be as simple as incrementing each random number and repeating the calculation for each known "gap" in the input space. The determination of the hash function and procedure to guarantee coverage is a topic for future work.

Each of the circuits illustrated in figures 8.13 to 8.19 can fit into the layout footprint illustrated in figure 8.12. The yield and tolerance analyses provided in chapter 3 and in the next section make it clear that feasible device structures must have fewer than about 108 junctions. This is possible if the full DAMP circuit layout is "sliced" into modules with the shape of the layout footprint and are only one rod "thick". Such modules will have 108 junctions arrayed in a fashion similar to the illustration in figure 8.20.

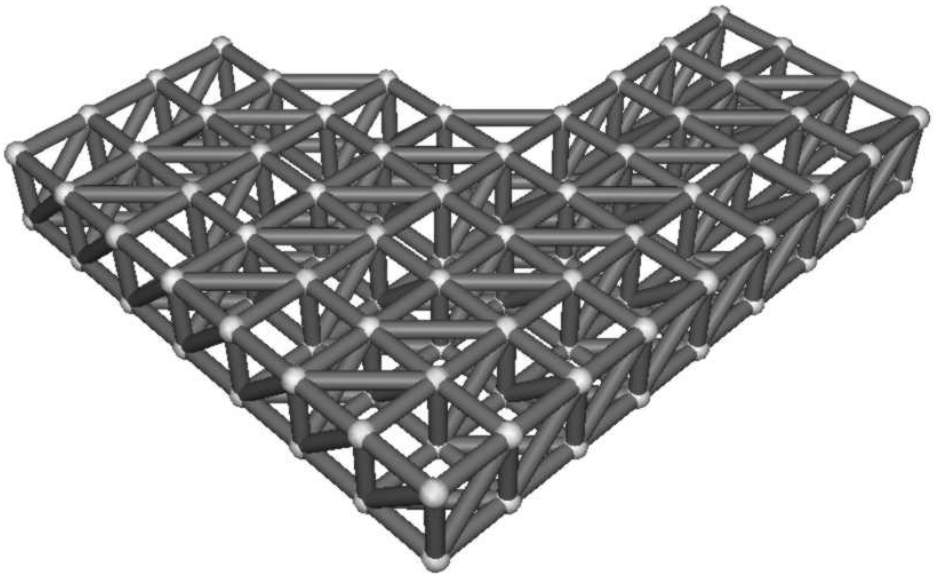

Figure 8.20. A thinly sliced module.

The estimated number of sliced modules required to form a full DAMP processor is 250. This comes from the following calculations that are derived from the layout described earlier: 17 x 5 register cell slices, 8 x 6 D-latch slices, 6 x 4 MUX2 slices, 3 x 5 decoder slices, 1 x 5 full adder slices, 5 x 5 control state machine slices, and an estimated 20% for routing slices (48) = 250 slices.

8.4 Design & yield tradeoffs

Research has shown that an 89% to 100% yield can be achieved by the fluidic self-assembly of micron-scale plate structures onto patterned surfaces within minutes [Clark, 2002; Srinivasan, 2001]. The possibility of re-flow chambers capable of recycling material that is not properly assembled makes this method of surface assembly especially attractive.

A re-flow step is a fluidic "flushing" of a surface in the hopes of recovering material that did not properly assemble. Multiple re-flow steps can be used to increase the net surface assembly yield as long as material is successfully removed from the surface and placed back into suspension. The re-flow procedure is only productive if properly assembled structures remain untouched. The new re-cycled suspension is then either augmented with new

material or simply re-applied to the surface for self-assembly. Hydrophilic surface[8] treatments have been successful in making the surface mobility[9] of nanoscale particles very high [Martin, 2002]. This raises the prospect of the possible development of high efficiency re-flow methods.

The relationship between the net yield of the surface assembly and the number of re-flow steps required can be derived as follows:

$$B_0 = Q \cdot (1 - Y_{FSA}) \tag{8.1}$$

$$G_0 = Q \cdot Y_{FSA} \tag{8.2}$$

Where G_0 is the number of modules which successfully assemble onto the surface initially, B_0 is the number of modules which fail to assemble initially, Y_{FSA} is the fluidic self-assembly yield, or the fraction of modules that assemble onto the surface properly, and Q is the starting quantity of modules (an integer).

Let Y_{RF} be the re-flow yield, or the fraction of unsucessfully assembled modules that can be recovered. After the initial step, a re-flow step can be used to recover some of the modules counted in B_0 and re-apply it to the surface such that we have:

$$B_1 = B_0 \cdot Y_{RF} \cdot (1 - Y_{FSA}) \tag{8.3}$$

$$G_1 = Y_{RF} \cdot Y_{FSA} \cdot B_0 + G_0 \tag{8.4}$$

Equation (8.3) holds because we have defined the re-flow step as the step that takes a fraction, Y_{RF}, of all previously mis-assembled material and attempts to reassemble the parts in it onto the surface with yield Y_{FSA}. Equation (8.4) is the accumulation of the "good" modules from the initial step, G_0, and the newly assembled modules from the re-flow step.

---

[8] A hydrophilic surface is one that "likes" interactions with water. The contact angle between the edge of a water droplet and the surface is used to measure the degree of this interaction. Small contact angles represent a hydrophilic interaction, large contact angles represent hydro*phobic* interactions. For example, a water droplet on clean glass will make a small contact angle (hydrophilic), while a droplet on common plastics will make a large contact angle (hydrophobic.)

[9] Surface mobility is the ability of a particle to slide along a surface without sticking. That is, the particle does not bind easily to the surface even though the two are close together.

This can continue as long as Bi, where i indicates the current step, is greater than 1 module. Accordingly we have equations (8.5) and (8.6), and their generalizations in equations (8.7) and (8.8).

$$B_2 = B_1 \cdot Y_{RF} \cdot (1 - Y_{FSA}) \tag{8.5}$$

$$G_2 = Y_{RF} \cdot Y_{FSA} \cdot B_1 + G_1 \tag{8.6}$$

$$B_i = B_{i-1} \cdot Y_{RF} \cdot (1 - Y_{FSA}) \tag{8.7}$$

$$G_i = Y_{RF} \cdot Y_{FSA} \cdot B_{i-1} + G_0 \tag{8.8}$$

The generalized equations (8.7) and (8.8) are recurrence relations that can be reduced to equations (8.9) and (8.10), respectively.

$$B_i = Q \cdot Y_{RF}^{\,i} \cdot (1 - Y_{FSA})^{i+1} \tag{8.9}$$

$$G_i = Q \cdot Y_{FSA} \cdot \sum_{j=0}^{i} Y_{RF}^{\,j} \cdot (1 - Y_{FSA})^{j} \tag{8.10}$$

Noting that the summation,

$$S = \sum_{j=0}^{i} a^{j} \cdot b^{j}$$

can be written in closed-form as,

$$S = \frac{a^{i+1} \cdot b^{i+1} - 1}{a \cdot b - 1}$$

we can let a=Y_{RF} and b=(1-Y_{FSA}) and reduce equation (8.10) to the following:

$$G_i = \frac{Q \cdot Y_{FSA} \cdot \left(Y_{RF}^{\,i+1} \cdot (1 - Y_{FSA})^{i+1} - 1 \right)}{Y_{RF} \cdot (1 - Y_{FSA}) - 1} \tag{8.11}$$

If we let Q = 1.0, then G_i becomes the total surface assembled yield after i re-flow steps, or Y_{Ti}. Rearranging (8.11) to solve for i, or the number of re-flow steps needed to obtain a given Y_{Ti}, produces (8.12).

$$i = \frac{\log\left(\frac{Y_{Ti}}{Y_{FSA}} \cdot \left(Y_{RF} \cdot \left(1 - Y_{FSA}\right) - 1\right) + 1\right)}{\log\left(Y_{RF} \cdot \left(1 - Y_{FSA}\right)\right)} \tag{8.12}$$

Equation (8.12) is constrained by the following inequality:

$$\frac{Y_{Ti}}{Y_{FSA}} \cdot \left(Y_{RF} \cdot \left(1 - Y_{FSA}\right) - 1\right) + 1 > 0 \tag{8.13}$$

Rearranging (8.13) we find,

$$Y_{Ti} < \frac{Y_{FSA}}{1 - Y_{RF} \cdot \left(1 - Y_{FSA}\right)} \tag{8.14}$$

Equation (8.14) states that regardless of how many re-flow steps, there is a limit to the final surface assembly yield. This is, of course, a reasonable result given that Y_{RF} is less than 1.0, or that the re-flow efficiency is less than perfect, and that the fluidic self-assembly yield is also less than 1.0, also imperfect.

The hydrophilic surface treatments mentioned earlier have shown that greater than 99% of all deposited nanoparticle material can be kept from adsorbing to a treated silicon surface, with each particle free to move on the surface [Martin, 2002]. If the losses along a re-flow system are kept to less than 5%, then we can safely use an estimated $Y_{RF} = 0.94$. The previously mentioned fluidic self-assembly yield was between 89% and 100%, which makes for a conservative $Y_{FSA} = 0.89$. Using (8.14) and (8.12) we can calculate a maximum re-flow yield, Y_{Ti}, of 99.26% achievable in 5 re-flow steps.

The lithographic step involved in preparing the substrate, between module assembly steps, may introduce an imperfect processing step, but since conventional photolithography can routinely align features on the scale of the nanorods considered here, it is unlikely that

this will reduce the yield. The annealing of metallized DNA junctions may also reduce the final yield. However, the ability to continue the metallization process after the modular assembly is finished and electrochemically "welding" the junctions together makes this also appear to be a negligible reduction in yield [Richter, 2001].

The designs from section 8.3 and chapter 6 can be implemented by stacking the approximately 250 modules (each one is a slice of the total processor circuit.) The per step yield estimate derived earlier can be used to estimate the final assembly yield after 250 steps. That is, approximately $(0.9926)^{250}$, or 16% final yield. In order to maintain the number of functioning processors in the final machine (2^{40}) from the 7×10^{12} possible processors (see chapter 3), this yield requires about 0.99 times as much starting material and is within our estimates.

Chapter 9. Simulation methods and results

The ring-gated field effect transistor (RG-FET) simulated here is similar in structure to the surrounding-gate transistors (SGTs) that have been studied for more than a decade as high-density alternatives to planar transistors [Takato, 1988; Takato, 1992; Miyano, 1992; Jang, 1998]. The RG-FET is novel because of the nature of its fabrication and placement within a self-assembling device structure. The RG-FET can plausibly be incorporated into a DNA-guided self-assembly process by chemically attaching different DNA strands to each end of the rod and to the gate during the rod's formation.

The importance of low power digital circuitry to conventional devices is well known and will become even greater as both transistor density and clock rates increase. Molecular scale electronic devices have orders of magnitude more gates and will therefore require either ultra low power consumption gates or slow clock rates, and perhaps both. Thus, the need for low power logic circuitry becomes an important issue to molecular scale device design.

This chapter is devoted to the simulation and estimation of the power consumption of the components used by the decoupled array multi-processor.

9.1 PISCES-IIb simulation of nanorods

The size of the RG-FET silicon rod we have considered (50 nm diameter, 500 nm length) has been shown to be large enough to use classical drift-diffusion simulations[10] [Sano, 2002]. This makes the type of mixed-mode simulation (drift-diffusion with transistor-level simulation) less computationally intensive than more sophisticated methods developed to handle smaller sized junctions accurately. Drift-diffusion simulations were performed using a Win32 port of PISCES-IIb [Pinto, 1988].

[10] Drift-diffusion simulations model the motion of electrons and holes through a semiconductor in the presence of electric fields.

The geometry of the RG-FET is depicted in figure 9.1. Using cylindrical symmetry PISCES-IIb was able to simulate the structure in 3D.

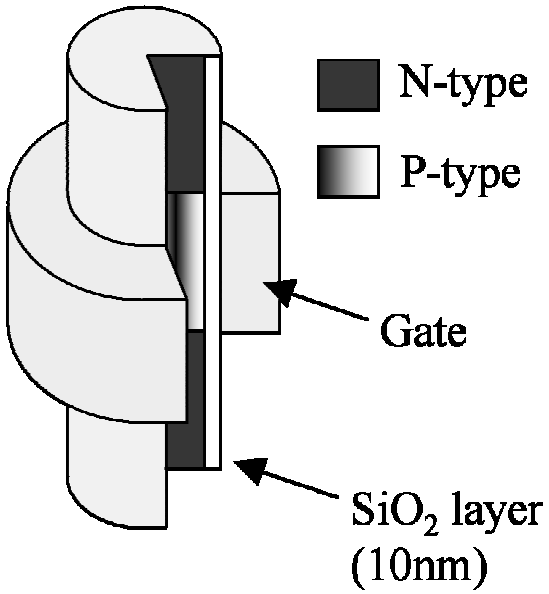

Figure 9.1. N-type RG-FET. The rod length is 500nm and its radius is 50nm. Channel length is 150nm. The source/drain contacts are at the top or bottom of the rod. The gate contact is a metal band around the rod. All contacts are palladium ($\phi_m \approx 5.0$ eV.)

The doping profile used by PISCES-IIb is shown in figure 9.2. The substrate (a silicon rod in this case) was doped to $1x10^{15}$ p-type (boron) atoms/cm^3 with the ends doped to $1x10^{21}$ n-type (phosphorous) atoms/cm^3 for the n-type RG-FET. The p-type RG-FET was doped to $1x10^{18}$ n-type atoms/cm^3 with the ends doped to $1x10^{22}$ p-type atoms/cm^3. Each FET was doped using a Gaussian profile with n-type and p-type characteristic lengths of 0.0475 μm and 0.061 μm respectively. Figure 9.2 is a plot of the dopant concentrations inside a radial slice of the rod. The plot is revolved around the Y-axis to form the 3D rod.

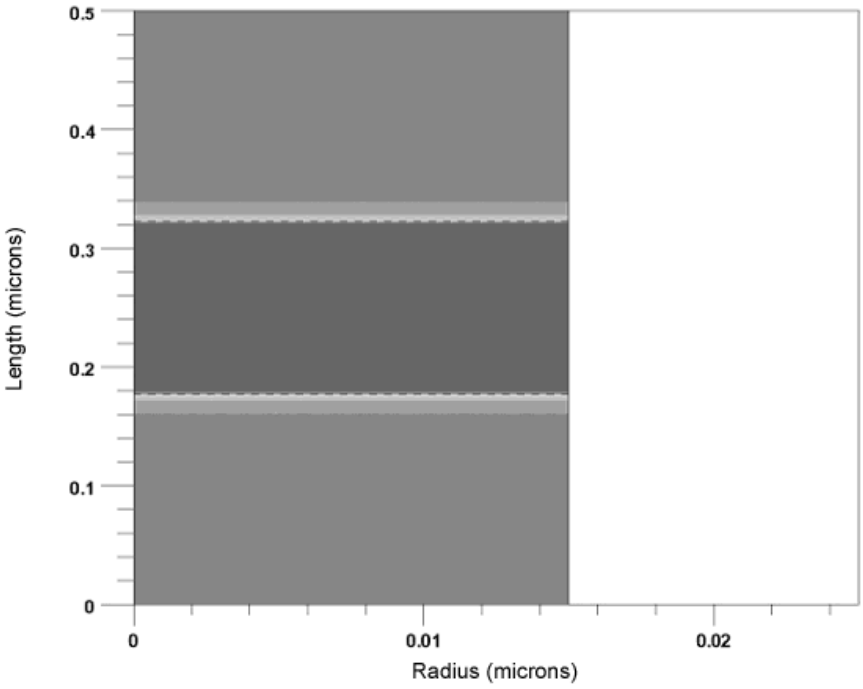

Figure 9.2. RG-FET doping profile along the 0.025 μm radius of rod (not to scale). The gate oxide layer (0.010 μm) is on the right, the PNP or NPN layers (0.015 μm) are on the left.

We performed a time independent simulation by applying a V_{ds} bias across the device (top to bottom) and a V_{gs} bias between the oxide side (right) and the bottom electrode. The simulation model included Shockley-Read-Hall (SRH) recombination with concentration-dependent lifetimes as well as concentration and lateral field-dependent mobility. Boltzmann statistics were used throughout with an operating temperature of 300K. To capture the time independent behavior of the RG-FET we swept V_{gs} and V_{ds} from 0.0v to ±1.0v (-1.0v for the PFET and 1.0v for the NFET). Each step was 0.5mV and 1mV steps along V_{gs} and V_{ds}, respectively, and I_{ds} recorded (current from top to bottom) to form the IV-curves in figures 9.3 and 9.4. The simulated transconductances of the n-type and p-type RG-FETs are plotted in figures 9.5 and 9.6 respectively.

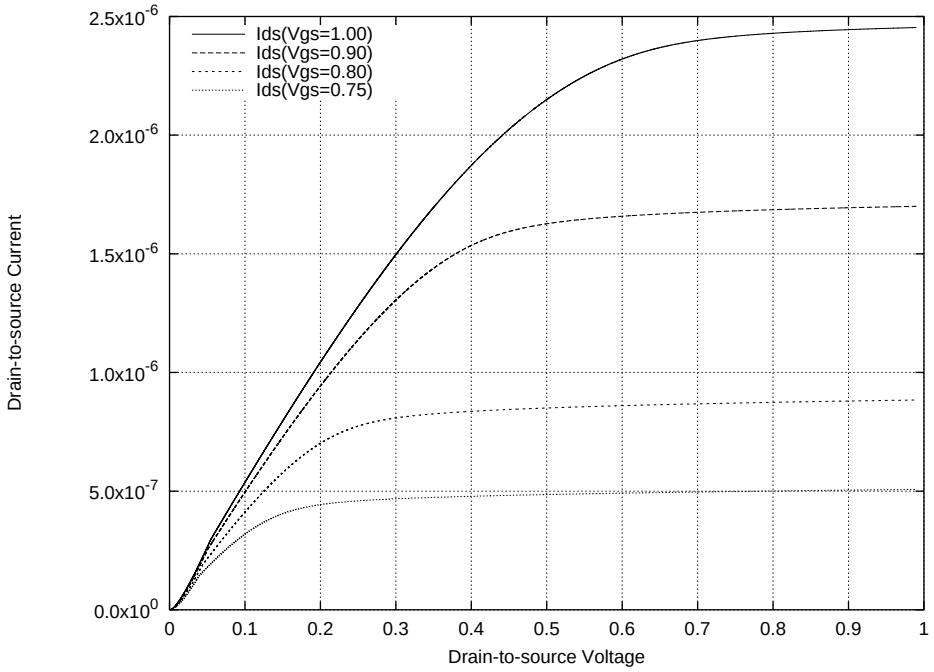

Figure 9.3. N-Type RG-FET IV curves.

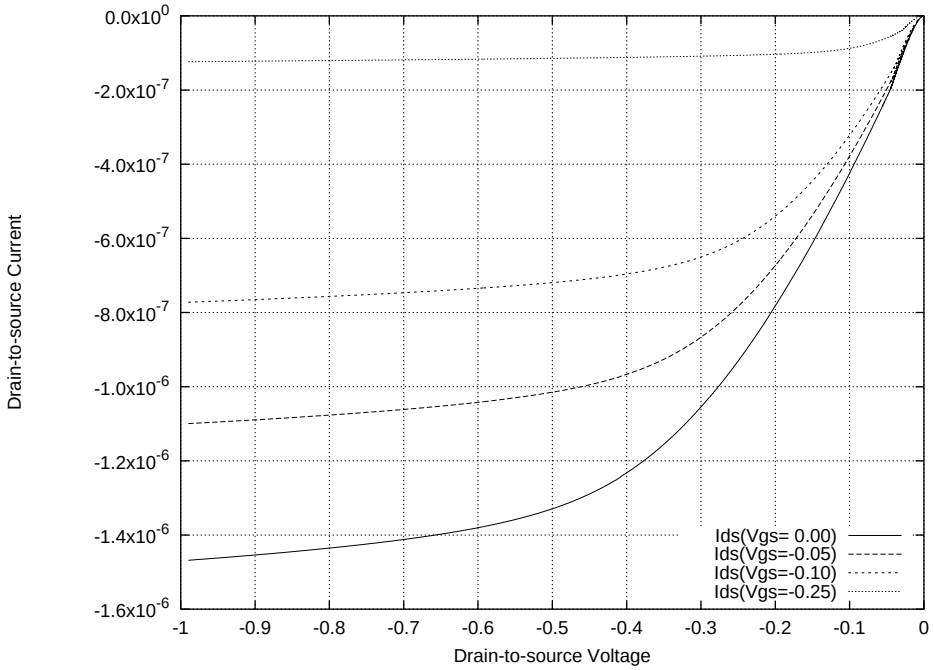

Figure 9.4. P-Type RG-FET IV curves.

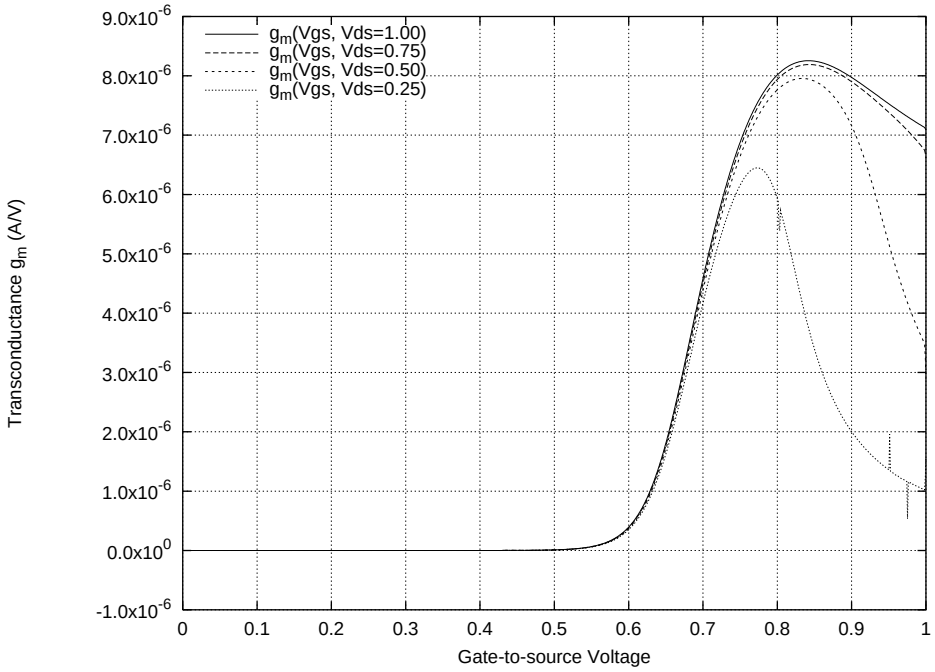

Figure 9.5. N-Type RG-FET transconductance at several source-drain voltages. The glitches in the Vds=0.25 transconductance trace are a computational artifact due to the low drain-to-source voltage.

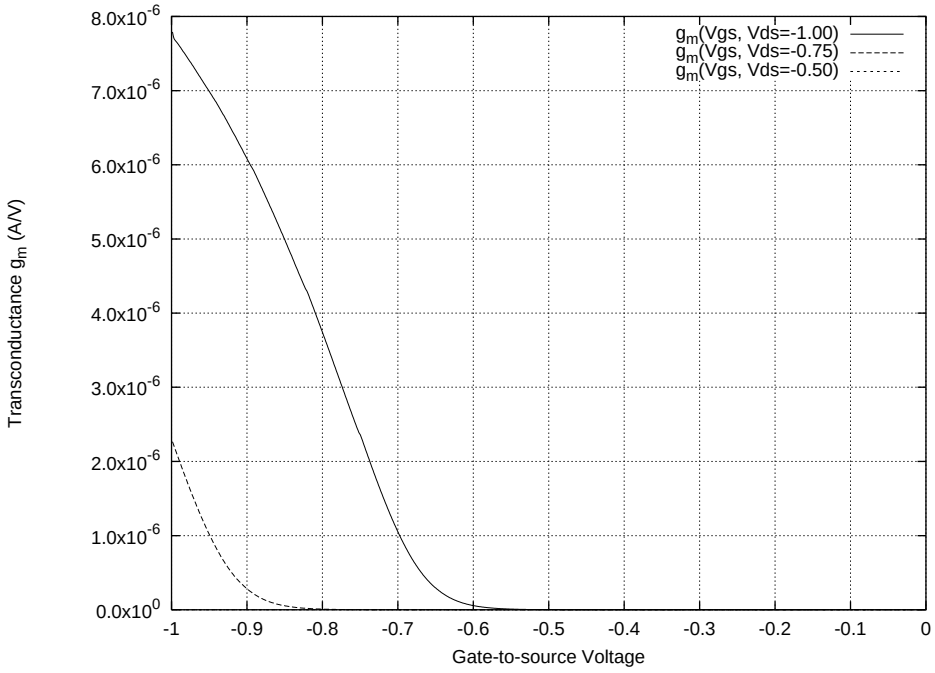

Figure 9.6. P-Type RG-FET transconductance at several source-drain voltages.

The data illustrated in figures 9.3 and 9.4 were stored on disk for later use by our modified SPICE 3f5 kernel. Our method of mixed-mode simulator coupling is similar to that used in [Rollins, 1988]. Instead of using an inner Newton iteration we simply preprocess the analog response of the RG-FET for later use by SPICE.

In addition to the RG-FET, a simple, heavily n-type doped nanorod of the same dimensions as the RG-FET (without a gate), was simulated to estimate the conductance properties of a conducting nanorod. The electrical properties, illustrated in figure 9.7, indicate that the rod has a bias-independent resistance of approximately 2.5 kΩ. This value is used in the SPICE models described in later sections.

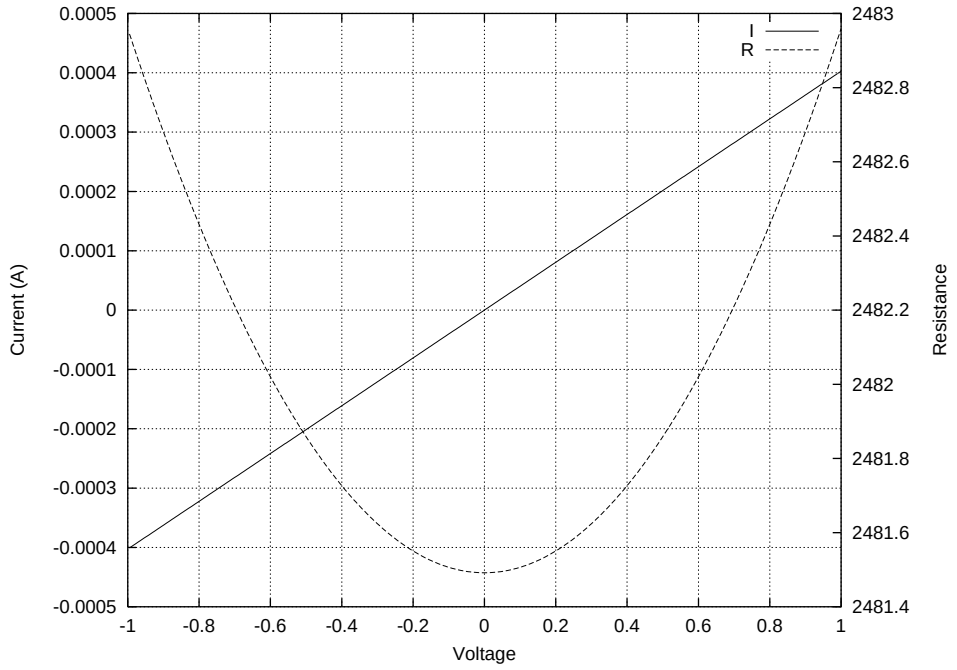

Figure 9.7. I-V plot of a heavily n-type doped silicon nanorod. The nearly linear current response *(I)* indicates that the rod behaves similar to a ~2.5 kΩ resistor *(R)* over the voltage range we use. The resistance varies by less than 2 Ω over the voltage range.

9.2 COULOMB simulation of capacitance

The values for the parasitic capacitances for the RG-FET were derived from a boundary element method solution to the electrostatic field problem (COULOMB) for a conducting rod surrounded by grounded rods as shown in figure 9.8 [IES, 2001].

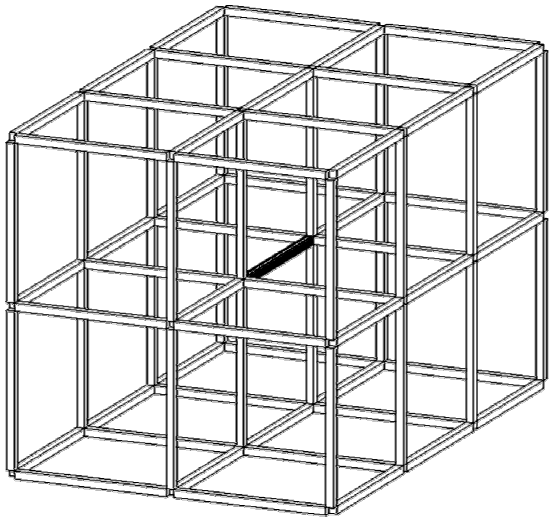

Figure 9.8. Rod geometry for parasitic capacitance calculation. The capacitance is measured between the center rod (shaded) and the outer shell of rods.

The COULOMB simulation results reported a capacitance of 1.71×10^{-17} F between the center rod in figure 9.8 and the surrounding shell of rods. This structure resembles the geometry of the self-assembled circuitry discussed in chapter 8. COULOMB also reported the capacitance between two parallel and adjacent rods to be 1×10^{-19} F. The values of R_{gs} and R_{gd} were estimated using the calculated resistance of a 10 nm thick silicon dioxide disk. The simulated "caged" capacitance value was used for C_{gb}, C_{sb}, and C_{db} and the adjacent rod capacitance for C_{gs}, C_{gd}, and C_{ds}. (C_{gb} is the gate-to-bulk capacitance, C_{sb} is the source-to-bulk capacitance, C_{db} is the drain-to-bulk capacitance, C_{gs} is the gate-to-source capacitance, C_{gd} is the gate-to-drain capacitance, and C_{ds} is the drain-to-source capacitance.)

The small inter-rod capacitance implies that cross-talk between signal lines will not be worse than in conventional technologies. The reactance between two signal lines with 1.71×10^{-17} F capacitive coupling at 400 MHz is greater than 23 MΩ ($1 / \omega C$.)

9.3 SPICE simulation of nanorod circuitry

A modified SPICE 3f5 [Quarles, 1991] circuit simulation kernel was used to simulate the behavior of several RG-FET based logic devices. The kernel was modified to include a simple file-based table lookup feature for the arbitrary current or voltage source device. The

file-based table lookup was used to read current data from the PISCES-IIb output files. The data is loaded into a memory table by the SPICE kernel and current values (I_{ds}) are linearly interpolated between V_{gs} and V_{ds} data points [Dwyer, 2002b; Dwyer, 2003]. The source for this modified kernel can be found on the compact disc included with appendix B, and on the dissemination page at www.cs.unc.edu/nano.

The behavioral simulations described in the next section estimate the total power budget for the machine given an instruction stream, and accurate power estimates of the component circuits. The parasitic capacitance values described in the previous section are used in the circuit simulations that model power consumption. Where appropriate, each gate or circuit was loaded with a $C_{gate} = 3 \cdot C_{gb}$ (FO-3) capacitance on its outputs to simulate a plausible fan-out. Each circuit was extracted from the nanorod layout described in section 8.3 by using the element models illustrated in figures 9.9 and 9.10.

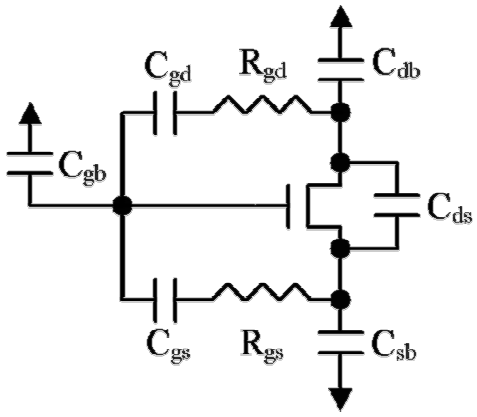

Figure 9.9. RG-FET capacitance circuit model.

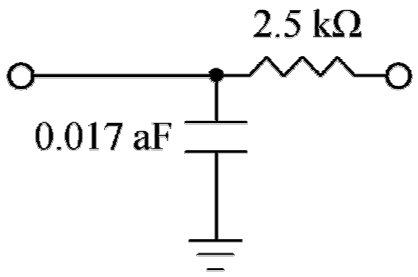

Figure 9.10. Conducting rod electrical model. Each conducting rod from a circuit layout is replaced by this model.

Despite the seemingly large resistance in figure 9.10, the small capacitance means that only a small amount of charge must move through the circuitry for it to work, resulting in lower power consumption.

Table 9.1 summarizes the gate delays, power consumption, and power-delay products (PT) for each circuit. The specific implementation details for each circuit are illustrated in figures 9.11 through 9.18. Figures 9.19 through 9.26 depict the results of the simulations.

Gate	Delay (ns)	Power (nW)	PT-product (J)
NOT	0.125	15	1.875×10^{-18}
NAND2	0.625	40	2.75×10^{-17}
2:1 MUX	0.3	300	9.0×10^{-17}
3:8 Decoder	1.0	10	1.0 x 10-17
Full adder	0.625	1600	1.0×10^{-15}
D-latch	0.3125	200	6.25×10^{-17}
Register cell (per bit)	1.25	245	3×10^{-16}
Ringer circuit (full on)	-	30	-

Table 9.1. Gate and circuit delays, power, and energy consumption.

The ringer circuit is formed by a feedback loop between the output of the LC1 multiplexer and one of its inputs, as illustrated in figure 8.4 of chapter 8. When the multiplexer selects this input an oscillation begins that produces a characteristic power signature. Figure 9.26 illustrates the oscillatory signature of the ringer circuit power consumption. Chapter 6 describes alternative output methods to the ringer circuit.

The average power consumption of this circuit while it is oscillating is approximately 30 nW. The DAMP with 6 x 10^{12} processors would consume 180 kW if all ringers were full on. Given a target power budget of 3.5 MW, the ringer circuit could consume as much as 580 nW, or almost 20 times as much power as it does now. Increasing the drive strength of the ringer would simplify the detection of the oscillatory power signal by making the signal larger.

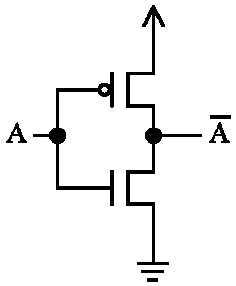

Figure 9.11. An inverter.

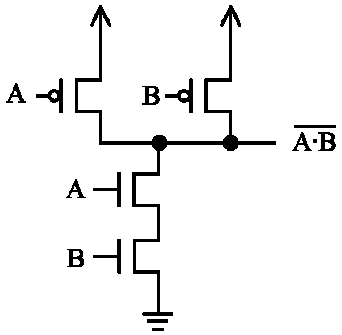

Figure 9.12. The NAND2 gate.

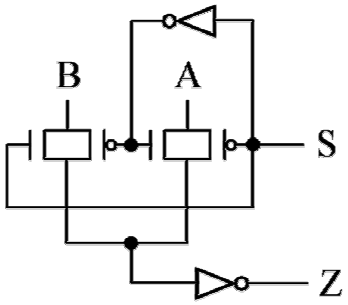

Figure 9.13. The 2:1 MUX circuit.

Three 2:1 MUX circuits are used to build the 4:1 MUX circuit needed by the DAMP.

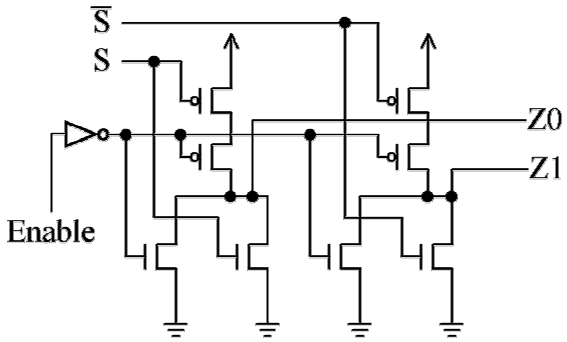

Figure 9.14. The 1:2 decoder circuit used to form the 3:8 decoder circuit.

Seven of the 1:2 decoder circuits were used to create the full 3:8 decoder circuit that was simulated. Parasitic capacitances of $3 \cdot C_{gb}$ were attached to each 2:1 decoder output to simulate a worst-case environment.

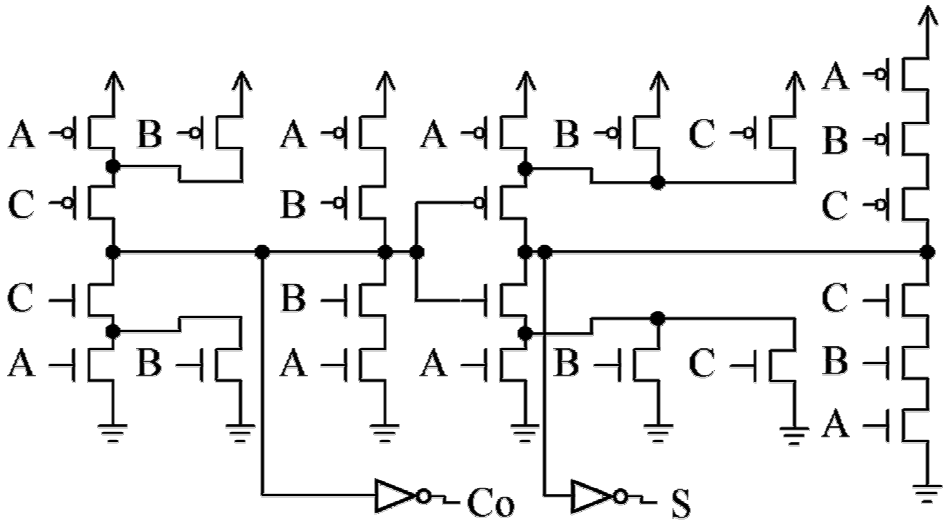

Figure 9.15. The full adder circuit.

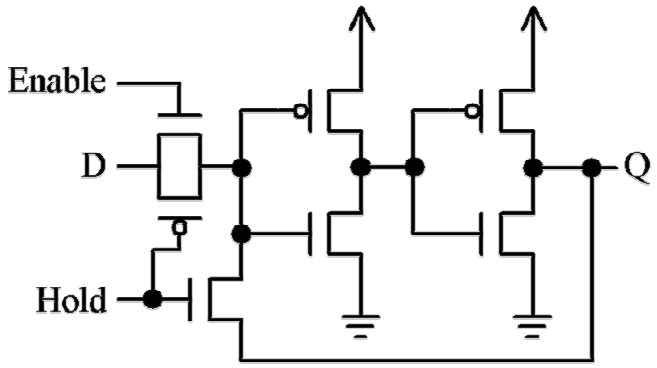

Figure 9.16. The D-latch.

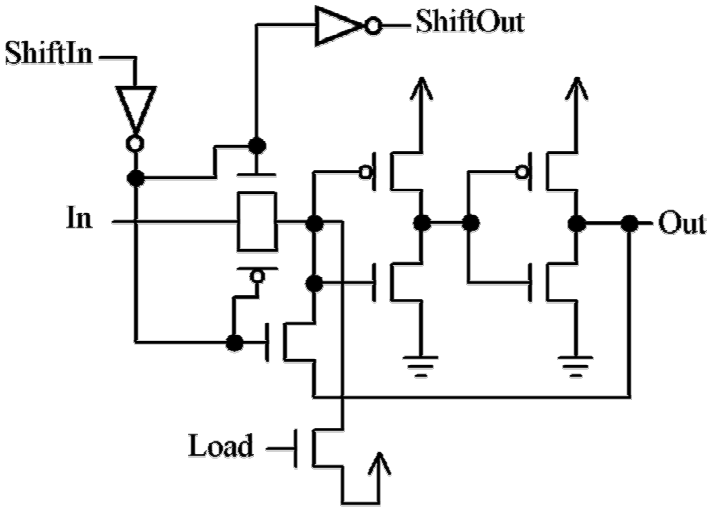

Figure 9.17. The register cell circuit. This circuit is arrayed 6 times to create the full register cell simulated here. The ShiftIn and ShiftOut signals are shared by 5 of the 6 registers (also called SRi and SRo.) The other register uses a dedicated ShiftIn and ShiftOut (also called SAi and SAo.)

The full register file is composed of 17 register cells connected in series. The "ShiftOut" signal from one cell is coupled to the "ShiftIn" of the next stage through a small delay line circuit of several inverters. Two such serial shift controls are used in the full register file, one controlling five of the six bits and the other controlling the remaining sixth bit. The simulation results that are plotted in figure 9.25 are for a register cell of six bits, each of them shifting the same data at the same time (both shift lines active).

The full register file also has a random number loading ability controlled by the "Load" signal. Only three of the six bits in each register cell can be randomly raised to a logical one, or true, value this way. The randomness of the loaded number is derived from the distribution of conducting and insulating rods used at a particular step in the assembly of the register cell as described in section 8.3.

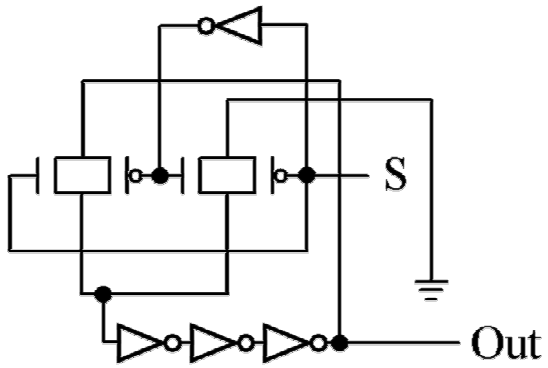

Figure 9.18. The ringer circuit.

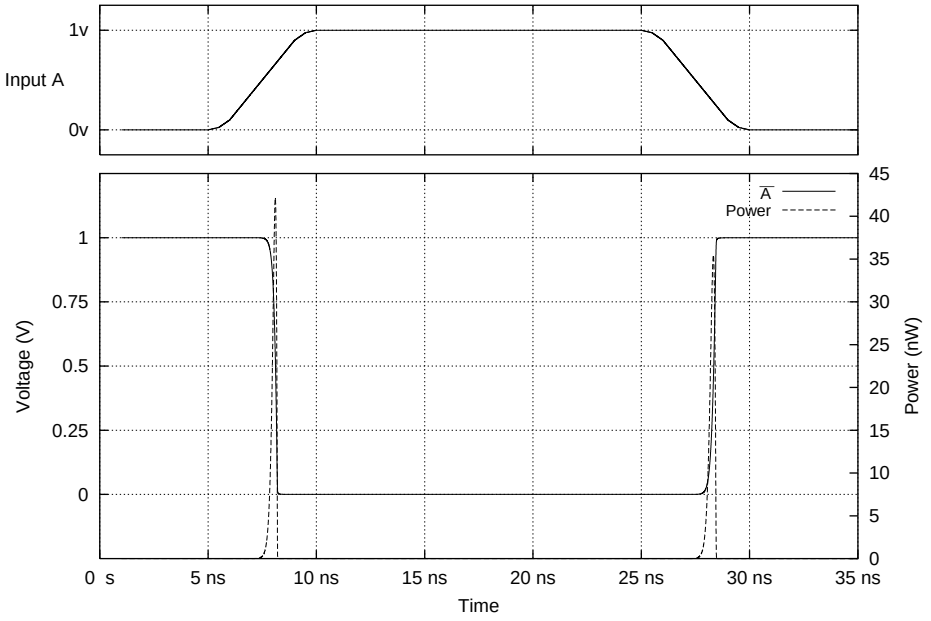

Figure 9.19. The inverter simulation results.

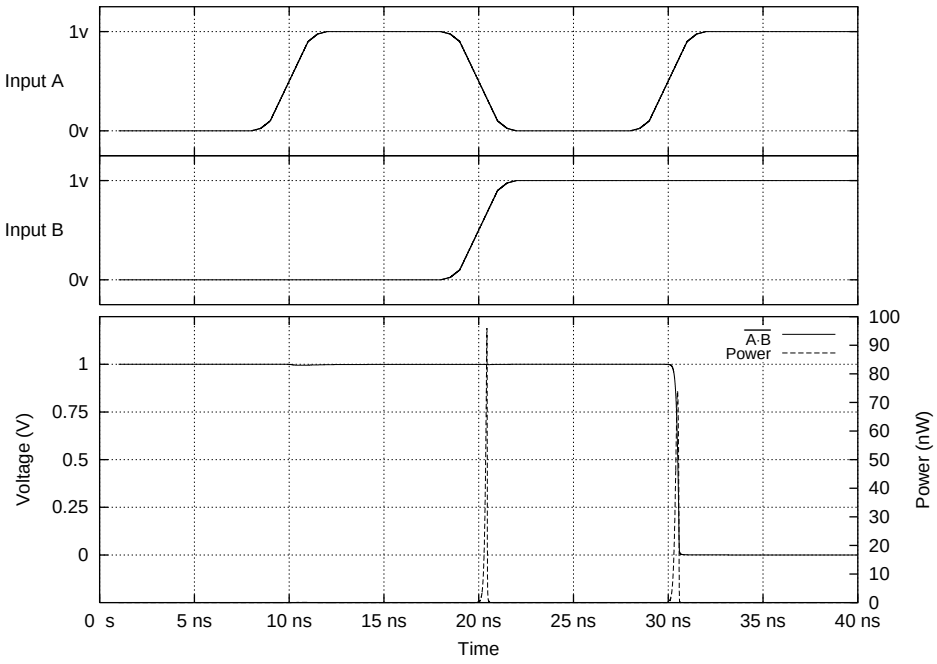

Figure 9.20. The NAND2 simulation results.

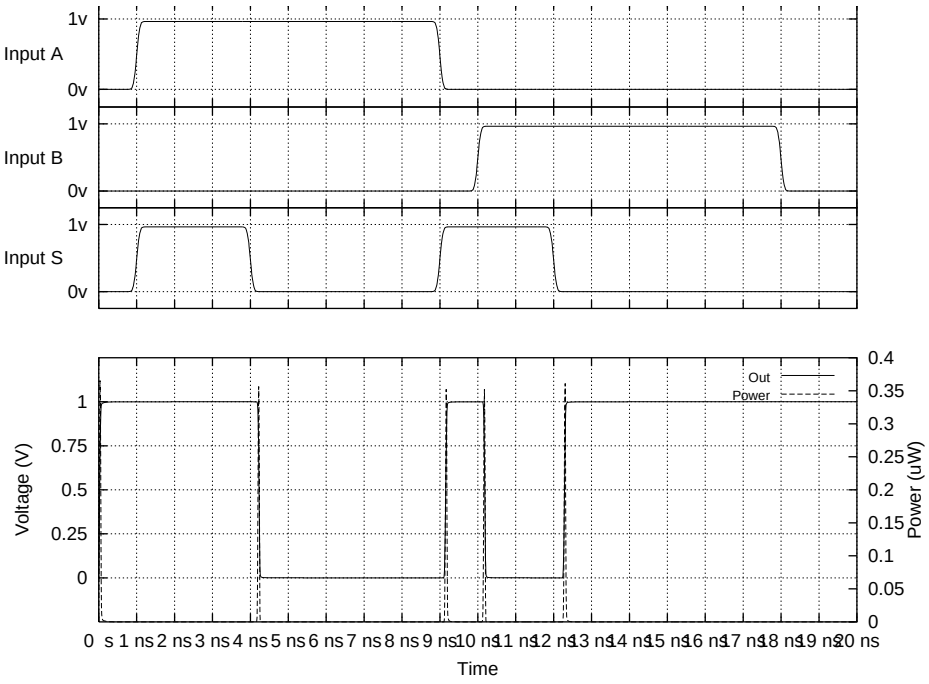

Figure 9.21. The 2:1 multiplexer simulation results. There is an inverter on the output of the multiplexer.

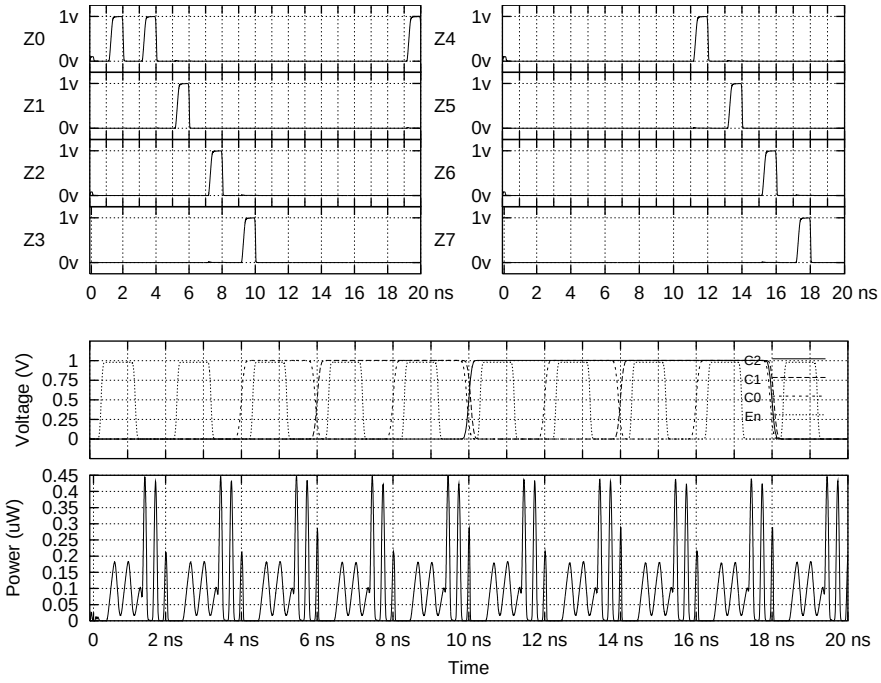

Figure 9.22. The 3:8 decoder simulation results.

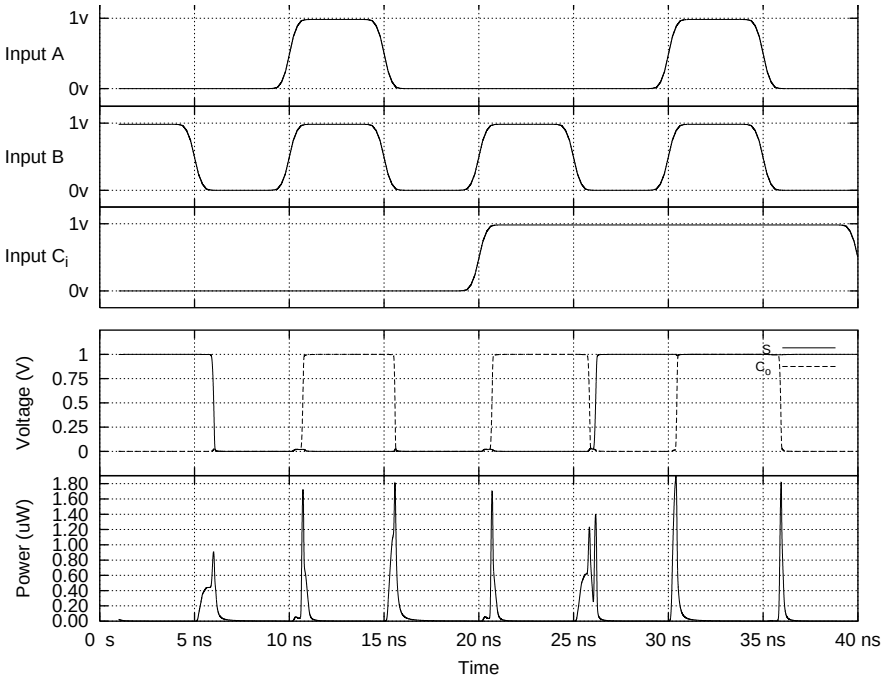

Figure 9.23. The full adder simulation results.

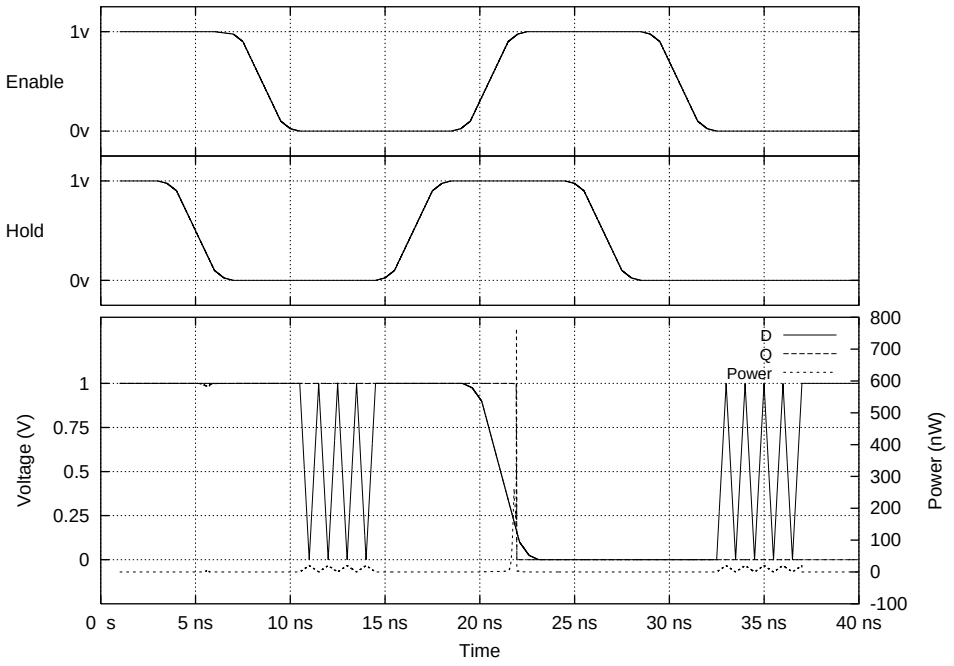

Figure 9.24. The D-latch simulation results.

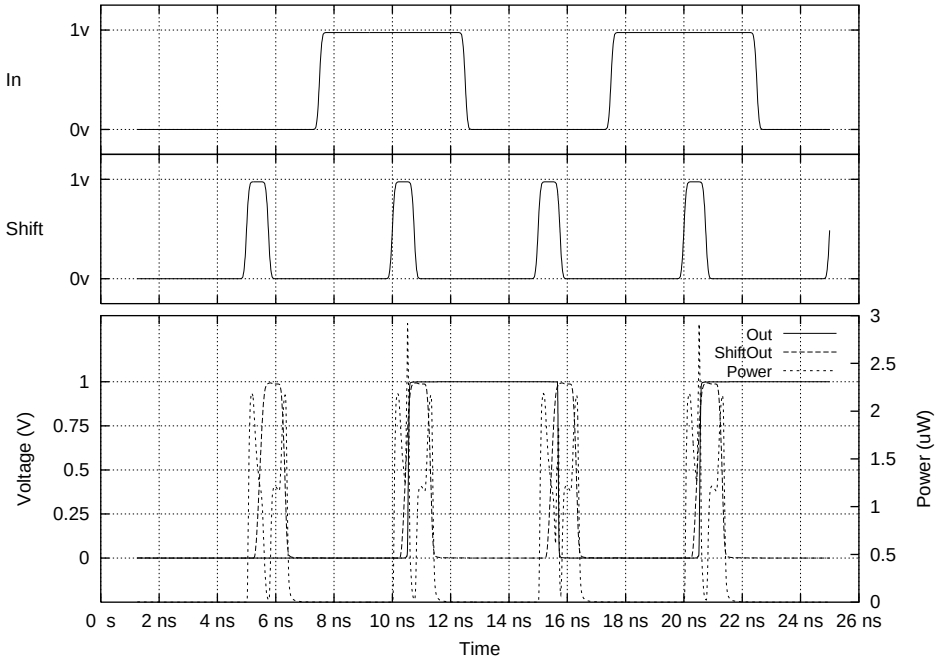

Figure 9.25. The register cell (6 bits) simulation results.

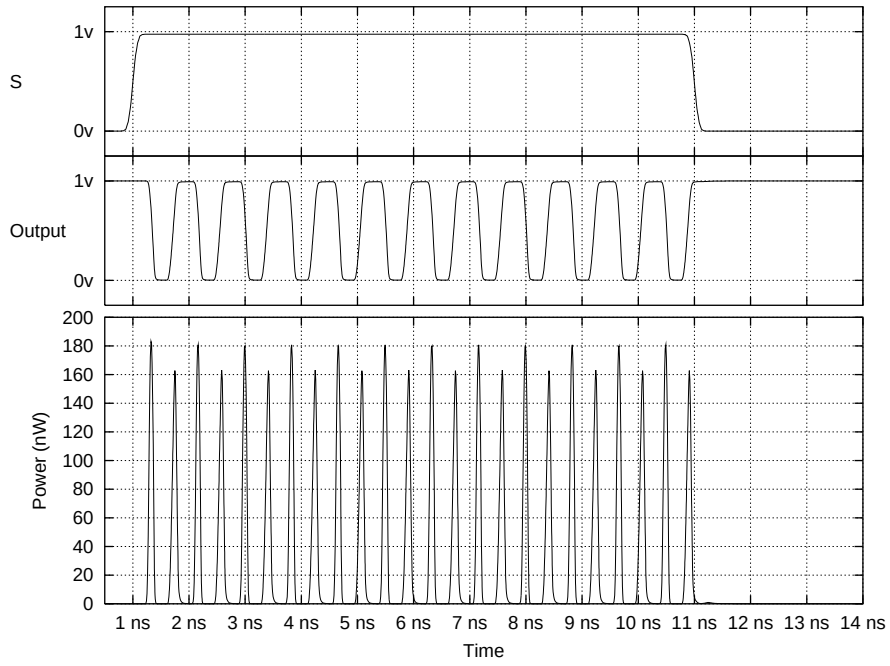

Figure 9.26. The ringer circuit simulation results.

9.3.1 Conclusions

The evaluation of the RG-FET, nanorods, and capacitance has led to the development of a set of power and performance estimates that will be used in chapter 11 to gauge the performance of the DAMP on various problems. The particular design decisions made before simulating the RG-FET logic gates were inspired from geometric and processing plausibility arguments starting from a 1V process with 500 nm long rods that are 50 nm in diameter. As figure 9.1 illustrates, the RG-FET cannot accommodate oxide thicknesses much greater than about 20 nm. Similarly, oxides thicker than 20 nm will reduce the channel diameter below the limit of continuum transport mechanisms [Sano, 2002]. Oxides less than 5 nm thick will require extremely precise control of the oxide growth to ensure a highly uniform and strong crystal. The quality of the oxide is important in preventing breakdown of the film at gate voltages of approximately 1V. Therefore, we chose an oxide thickness of 10 nm because it can withstand the electric fields developed at a gate voltage of 1V. Figure 9.27 illustrates the change in transconductance as oxide thickness is varied from 5 nm to 20 nm.

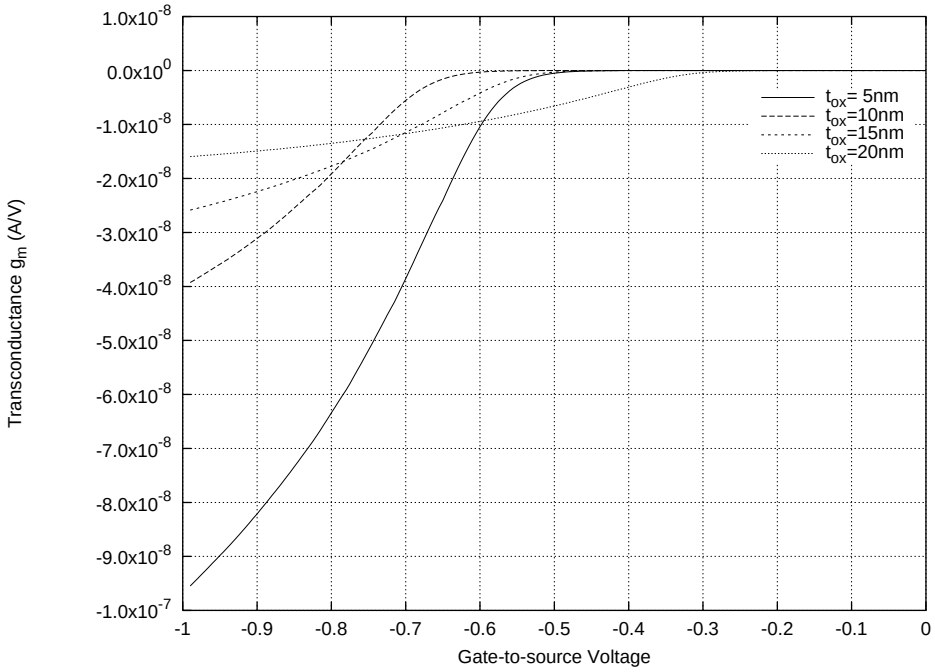

Figure 9.27. P-type RG-FET transconductance as a function of oxide thickness with a 150 nm channel. We used the 10 nm thick oxide RG-FET in our performance evaluations.

Expected processing limitations in the fabrication of an RG-FET motivated our choice of a 150 nm long channel. The lengthwise etch of the RG-FETs channel region may not be precisely controllable. Therefore, a sufficiently large margin (channel extension) must be left on either side of the channel. Channel lengths greater than 200 nm leave only 150 nm of rod on either side of a 500 nm long rod. Channel lengths below 75 nm will experience poor off-state leakage currents [Thompson, 1998] and may not be properly simulated by PISCES-IIb. Figure 9.28 illustrates the change in transconductance as the channel length is varied.

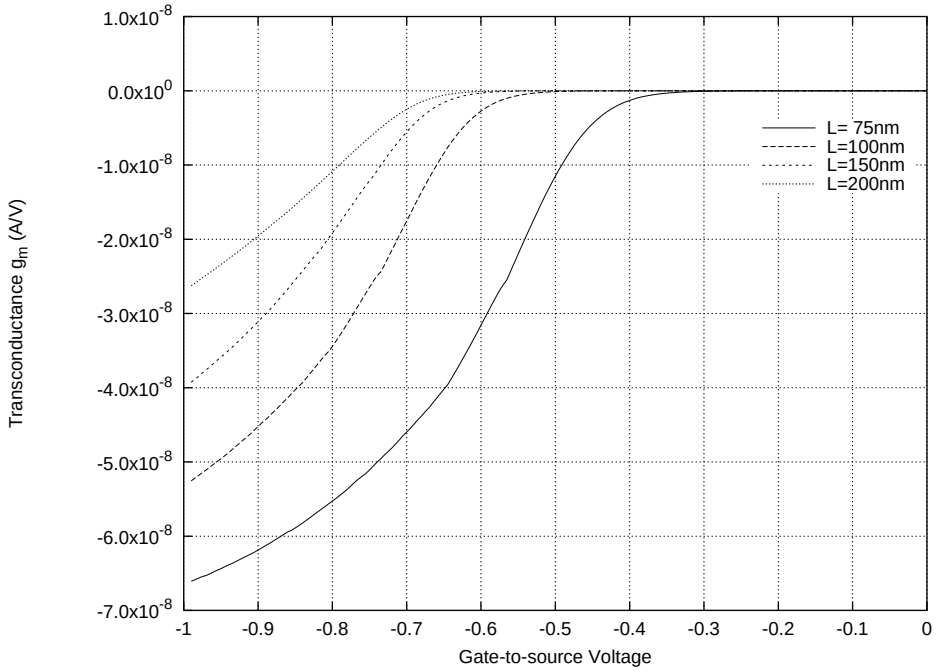

Figure 9.28. P-type RG-FET transconductance as a function of channel length using a 10 nm gate oxide. We used the 150 nm channel length RG-FET in our performance evaluations.

The input slew rates used in our simulations (0.2 V/ns) lead to conservative power estimates because they prolong the overlapping n-type and p-type RG-FET transition period with respect to the output transition, time thus increasing the switching energy estimate. The output transition times are consistent with the turn-on time observed in a time-dependent PISCES-IIb simulation of a p-type RG-FET. The time dependent simulations also show that the gate charging current due to the voltage-dependent channel capacitance is instantaneously never greater than 1 nA for 0.2 V/ns input slew rates (it drops to zero as the slew rate goes to zero).

We have performed additional transient-response simulations to clarify the error that we incur by using a DC approximation to the RG-FETs transfer function. We measured the time varying current response (I_{ds} vs. time) using several different slew rates (1 V/ms, 0.4 V/μs, 0.8 V/μs, 1.5 V/ns, 3 V/ns, 6 V/ns, 12.5 V/ns, 25 V/ns, and 50 V/ns). Figure 9.29 is a representative result from the transient simulations. The positive source and drain currents counter the displacement current seen in the gate. This is presumably due to the movement

of charges as the channel forms underneath the gate. To compare the DC response with the different transient responses we plot the log of the absolute difference between the currents versus time. Figure 9.30 illustrates several of these error curves. The errors we observe during the 3V/ns slew rate simulation can be as low as a few hundred electron/holes per second. This current is smaller than what can typically be simulated by the PISCES-IIb simulator. This implies that our DC method is as accurate as a transient simulation using PISCES-IIb for input slew rates less than 3 V/ns.

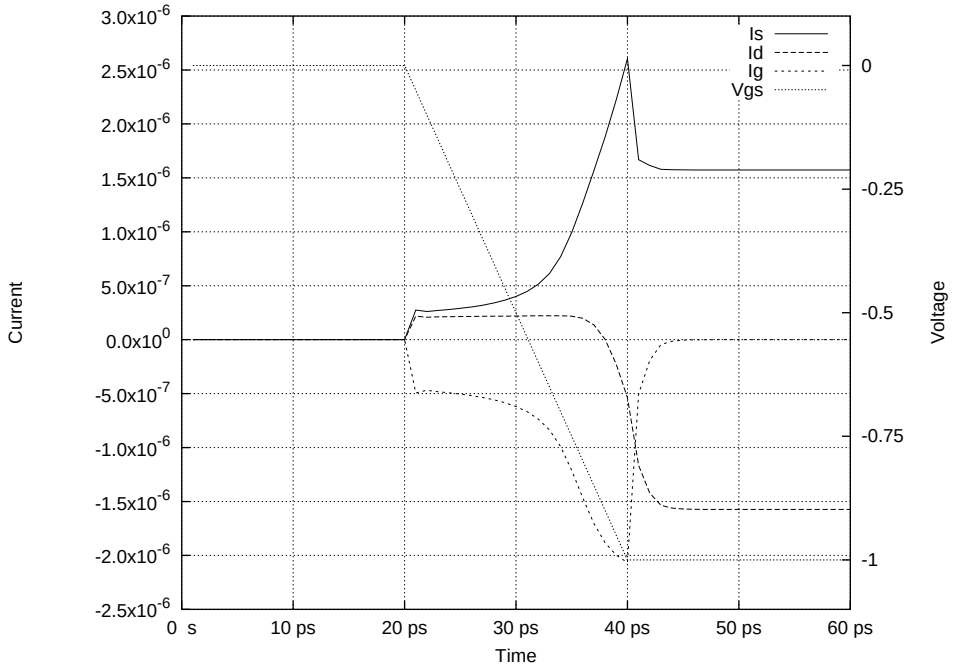

Figure 9.29. Transient response of a p-type RG-FET during a 20ps input voltage ramp.

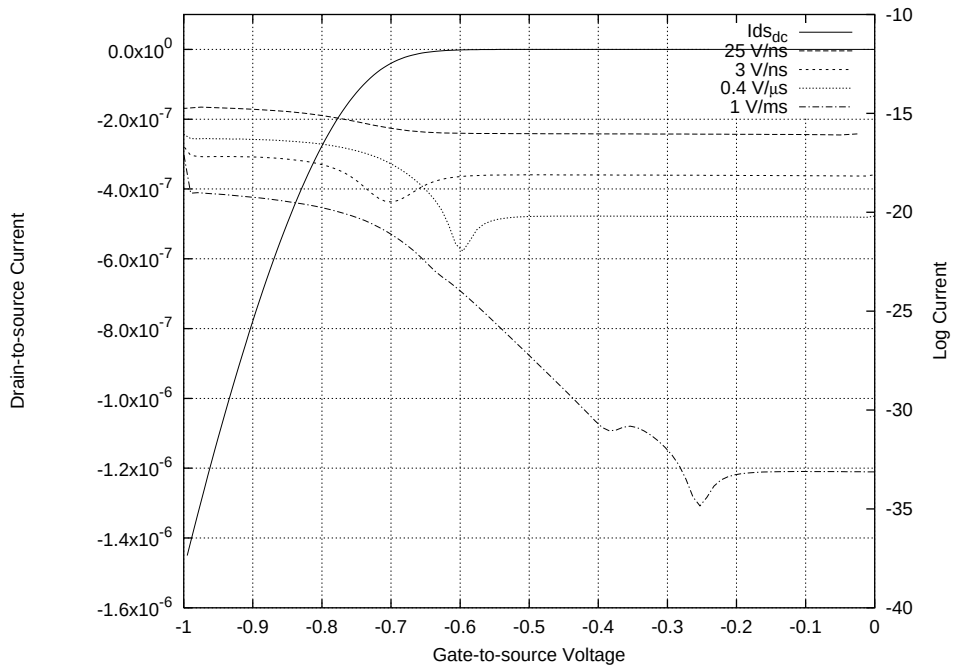

Figure 9.30. Absolute current error between DC and transient p-type RG-FET response.

9.4 Custom behavioral simulation

The SPICE simulation described earlier use a common technique to produce solutions to the various problems they are each suited to answer. That is, they each create massive systems of simultaneous equations that need to be solved for each time step of the simulation. This imposes a limit on the size of problems because of memory and patience constraints that make solving such large systems of equations difficult or impossible.

The common solution to this is to simulate larger components at a higher level of abstraction. Just as the PISCES simulation results were used in a more abstract (from PISCES's perspective) simulation in SPICE, the SPICE simulation results are used in a more abstract behavioral simulation.

Behavioral simulation is a way of evaluating complex logic circuitry without having to resort to the complex physically based simulations of electronic circuitry (PISCES and SPICE). Besides evaluating logic circuitry, behavioral simulation can also be used to

estimate timing requirements and power dissipation. These estimates require the detailed simulation results from the physical simulations (PISCES and SPICE).

The behavioral simulation described here was used predominantly to estimate the amount of power dissipated by the logic circuitry of a single processor in the DAMP. The timing issues are relatively less difficult to analyze because of the serial nature of the processor, and the limit the power budget places on the clock rate. Compared to how fast and how deep the typical logic circuits are in the DAMP (see sections 9.3, and 8.2) the reduced power-conscience clock rate will be far below the maximum operating frequency.

9.4.1 Behavioral Model

The behavioral simulator was programmed using the Microsoft Visual Studio in MFC/C++. The interface for the tool is shown in figure 9.31.

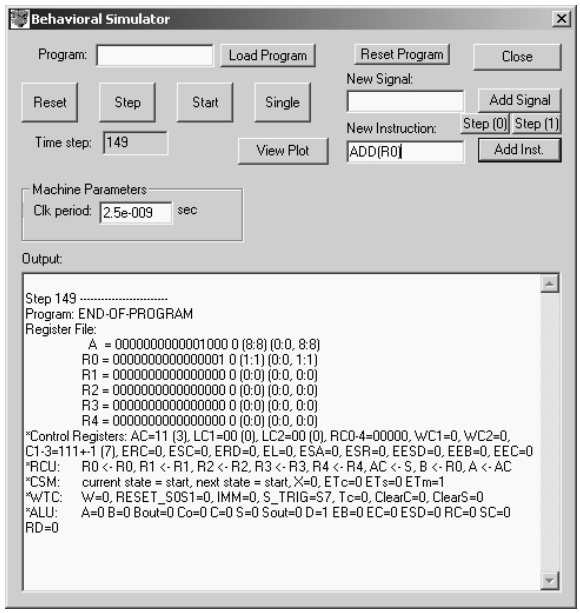

Figure 9.31. Interface for the behavioral simulator.

Figure 9.32 illustrates the model used for each architectural unit described in chapter 8. The register file, RCU, ALU, control registers, CSM, and WTC were all implemented as modules following the structure of figure 9.32.

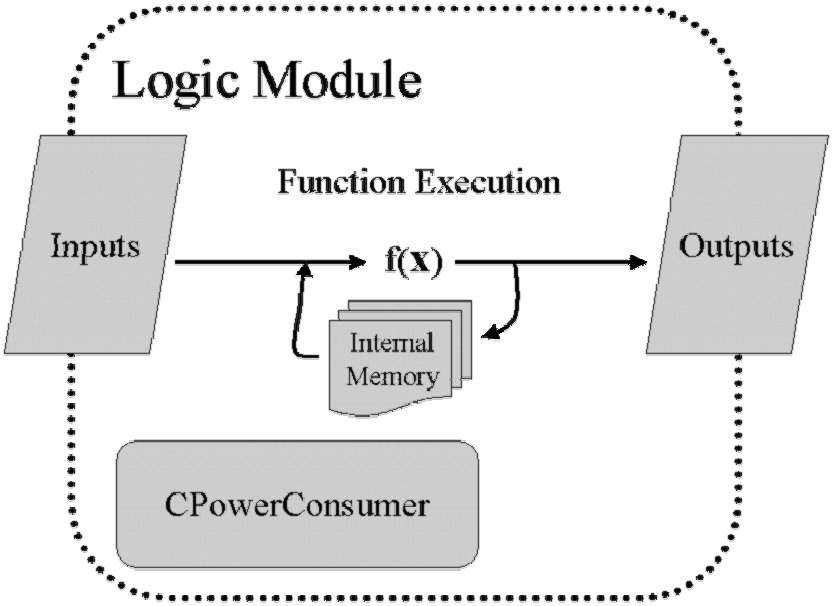

Figure 9.32. Basic logic module structure. Each logical unit from the architecture was cast into this structure.

The interconnectivity of each module has already been described in the implementation details of chapter 8. To implement the connections between units, the behavioral simulator "binds" outputs from one unit to the inputs of another. Binding, in this context, is simply a copying of the various bit values produced and consumed by each unit.

For every time step, the behavioral simulator binds the units together and then executes each unit. The results of each execution are stored in the output variables of each module. During the next time step, the output variables will be bound to the input variables of the next module.

A time step is defined as the time period over which the clock signal transitions from a steady-state zero through a steady-state one and ends just as the signal goes back down to zero. The total power dissipation during a time step is calculated by summing the worst-case

power-delay product for each logic circuit that underwent a transition during the time step, as described in section 8.3, and dividing it by the clock period. A running total of the energy consumption (power-delay product) is also kept so that a running average power dissipation can be calculated per time step.

Each module has a "CPowerConsumer" object that records how much energy it consumes per time step. The CPowerConsumer class is initially told how many of the primitive logic circuits, outlined in section 8.3, each module uses and subsequently told to increment counters for each time step. The particular numbers of each primitive circuit in each module are given in table 9.2.

Module	NOT	NAND	D-Latch	MUX2	MUX4	3:8 Decoder	Register bit	Full adder
Register file	68	0	0	0	0	0	102	0
RCU	1	0	0	5	3	0	0	0
ALU	0	0	4	0	0	0	0	1
Control registers	0	0	0	0	0	0	26	0
CSM	2	7	2	0	0	0	0	0
WTC	4	37	4	1	1	1	0	0

Table 9.2. Primitive circuit counts for each behavioral module.

The order in which modules are bound to each other is important to consider because it captures the asynchronous behavior of the circuitry. In the case of the register file and RCU, the values of the inputs to the multiplexers in the RCU are determined by the values of the output from the register file. A topological sorting must be followed to get the proper inputs to the proper outputs before execution. Figure 9.33 illustrates the binding and execution order used by the behavioral simulator.

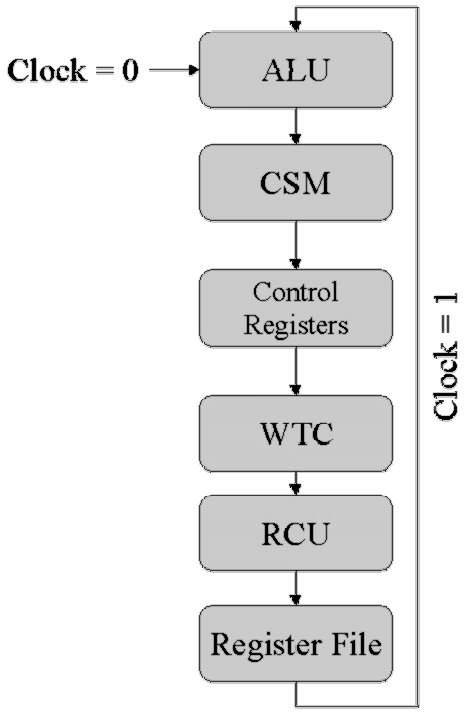

Figure 9.33. The binding and execution order used by the behavioral simulator.

9.4.2 Results

The behavioral simulator takes a series of one or more instructions and (using the assembler described later) converts these instructions into a serial stream of bits that control the DAMP. Since the DAMP has identical processors, it is sufficient to simulate a single processor. The state of the processor is reported after every clock cycle and a running plot of energy dissipation can be viewed. Figures 9.34 to 9.39 illustrate the energy dissipation for the integer operations ADD, ADDI, ADDC, NOT, INC, and DEC running at a clock period of 2.5 ns, or 400 MHz. A more detailed description of instruction energy dissipation is given in chapter 11.

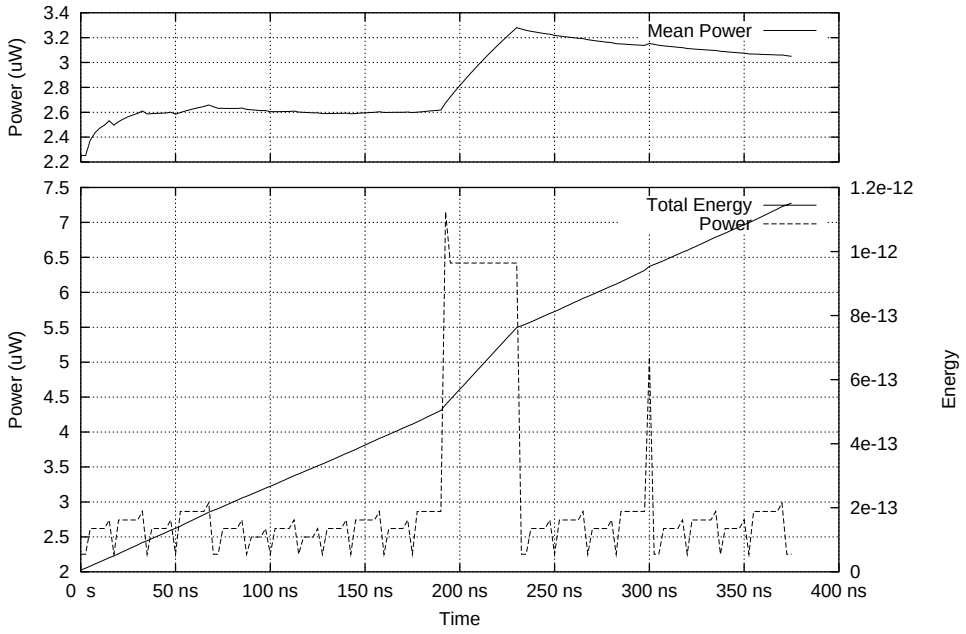

Figure 9.34. Energy dissipation of the ADD instruction.

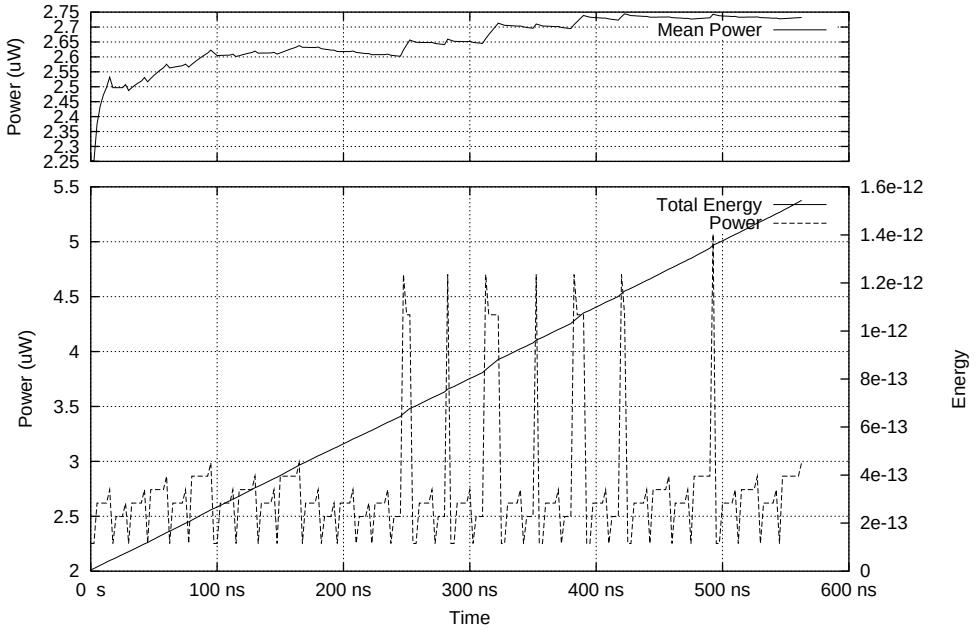

Figure 9.35. Energy dissipation of the ADDI instruction.

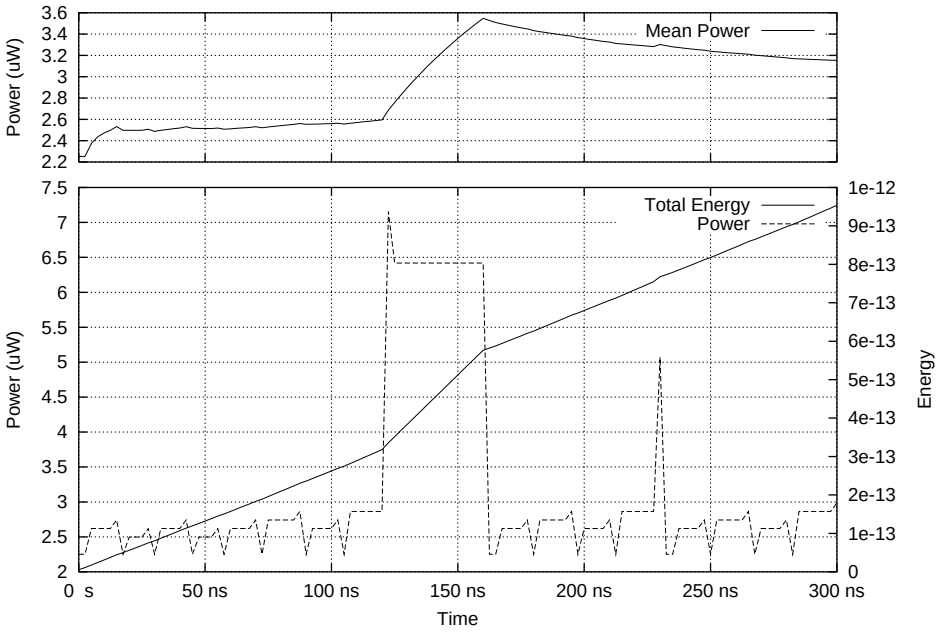

Figure 9.36. Energy dissipation of the ADDC instruction.

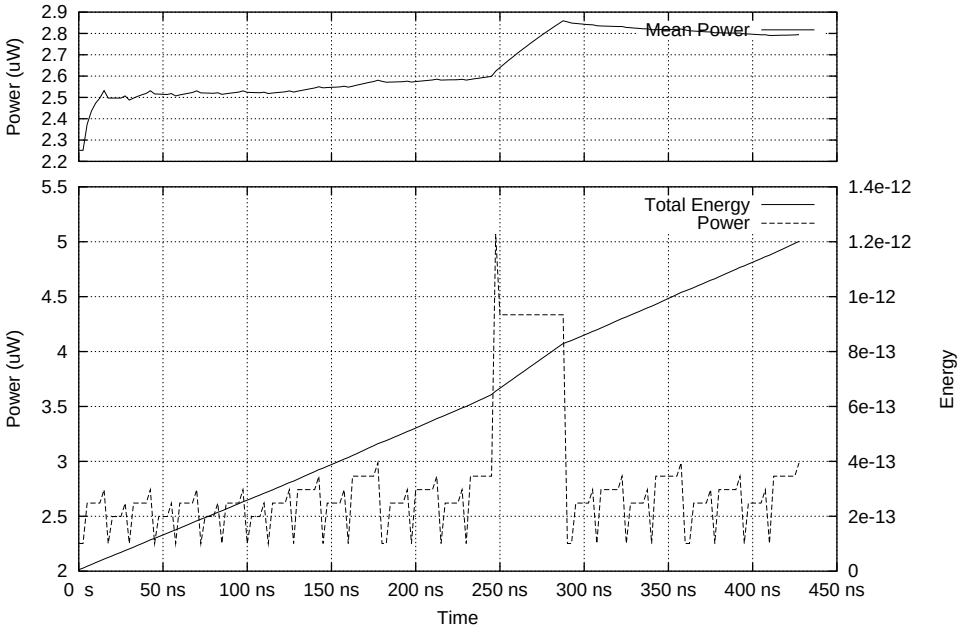

Figure 9.37. Energy dissipation of the NOT instruction.

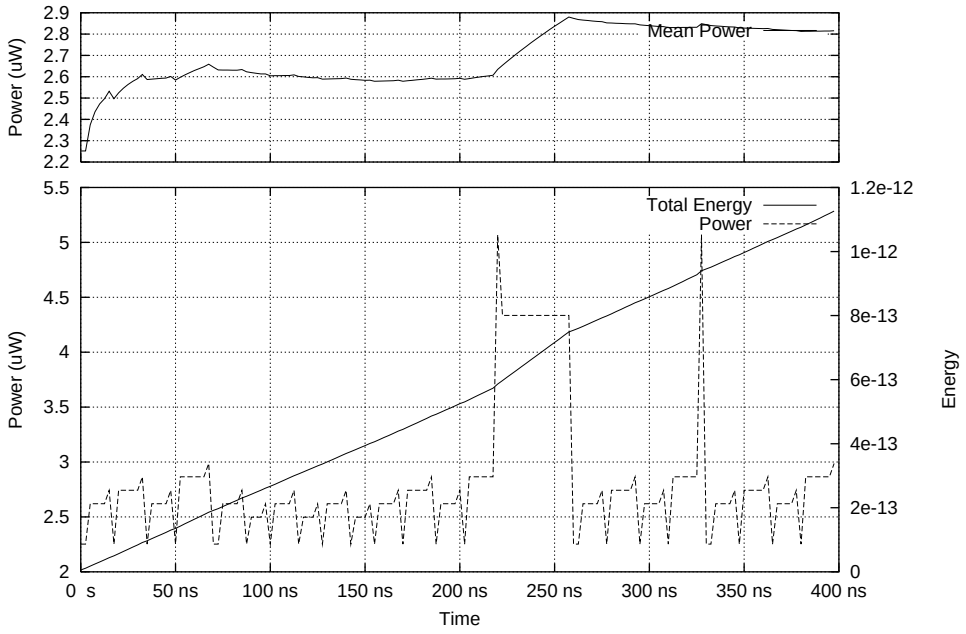

Figure 9.38. Energy dissipation of the INC instruction.

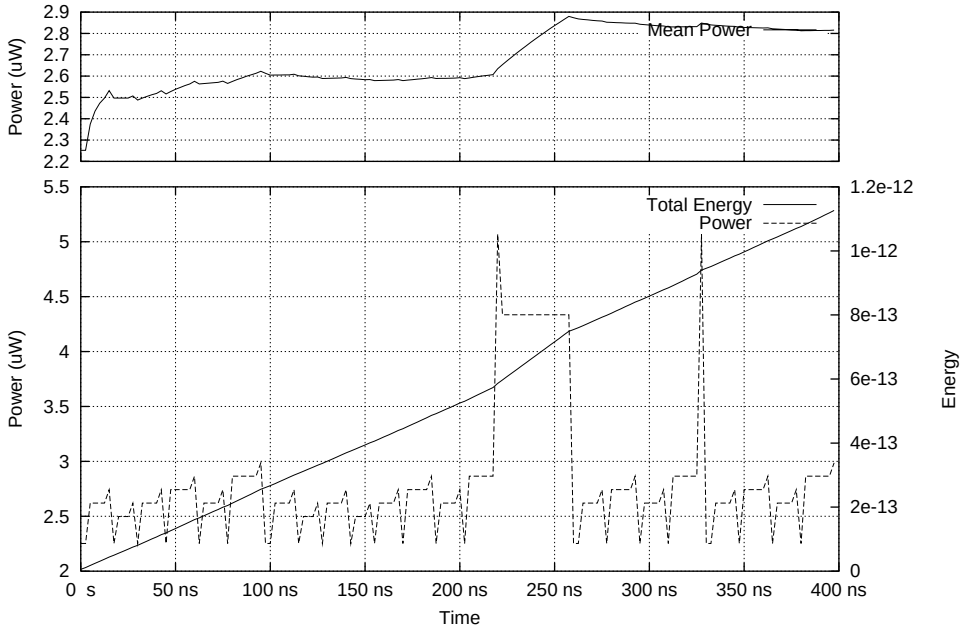

Figure 9.39. Energy dissipation of the DEC instruction.

9.4.3 Instruction assembler

As mentioned earlier, the assembler used by the behavioral simulator takes instructions and turns them into a serial stream of control signals for the DAMP. These control signals are the signals that would be directly applied to the P0 and P1 electrodes of the DAMP's processor nodes, as described in chapter 6. Since each of the processors in the DAMP is a serial machine that has no micro-code there is no complete set of instructions except all bit strings. It is unproductive to enumerate all possible instructions, so the assembler has support for instructions needed for the present evaluation and a few others. The complete list of instructions can be found in chapter 8. Since the control of a processor is fully exposed to the assembler, it is possible to implement new instructions as the need arises. Such will be the case if integer multiplication and division are required, or if an application program needs floating-point arithmetic.

The assembler included with the behavioral simulator described earlier uses a simple instruction prefix search and argument parsing algorithm to translate instructions. It was written in C++ and uses no optimization techniques to reduce the size of the final control signal stream.

Chapter 10. Thermal evaluation

The large numbers of transistors in the DAMP makes it important to evaluate the thermal effects of electrical transitions on each processor and the thermal behavior of the larger machine they compose. The SPICE and behavioral simulations described in chapter 9 produce power and energy dissipation estimates based on the drift-diffusion simulation of electrons and holes through the silicon nanorods. We can apply these results to a hypothetical processor housing and evaluate the thermal situation during the operation of the DAMP. Figure 10.1 illustrates a partial stack of processor nodes. Each node, as described in chapter 8, has 2^{28} functioning processors that are assumed to be uniformly distributed across the processor substrate among the processors that did not form properly.

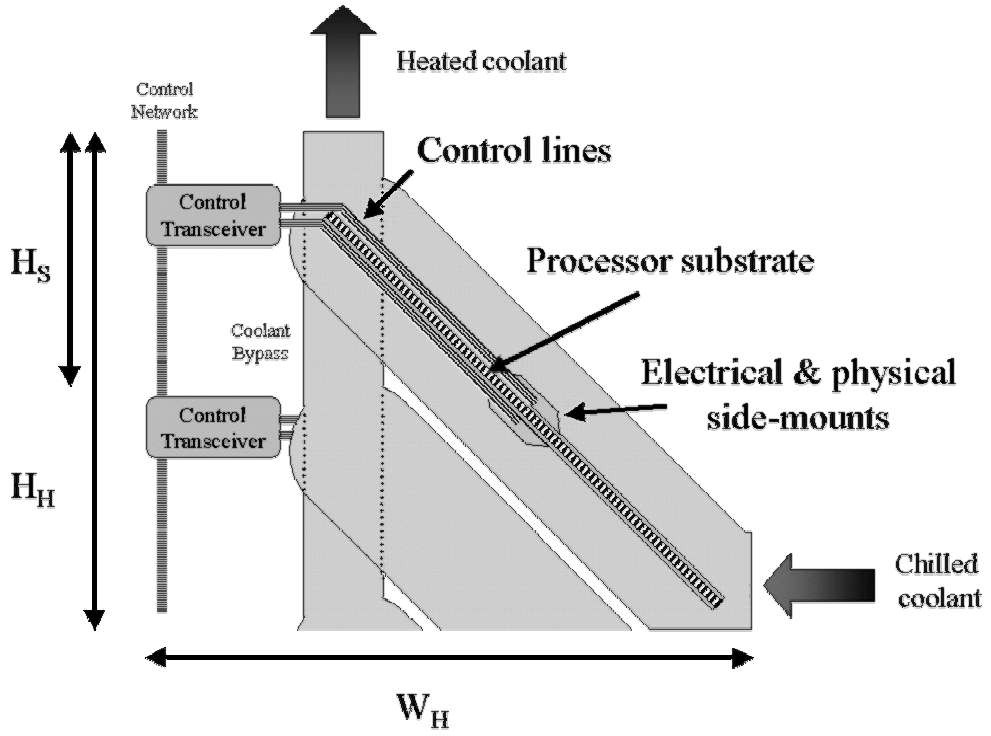

Figure 10.1. Side view of a partial stack of processor nodes. The processor substrate is suspended in a cooling jacket by side-mounts. Chilled coolant enters at the right side and rises upward due to convective flow and pool boiling.

Research into microelectronic energy removal and cooling techniques has shown that a quiescent perfluorinated liquid[11] can remove as much as 45 W/cm^2 in horizontal pool boiling configurations up to 80°C [Watwe, 1997]. The passive nature of this technique is attractive because of its simplicity, as illustrated in figure 10.1. Given that convective flows will also be circulating in the housing, 45 W/cm^2 is likely to be a lower bound on the heat removal rate.

10.1 Steady-state dissipation

The steady-state heat removal from a processor on the substrate must be greater than the amount of heat being deposited into the substrate by the electrical switching activity of

[11] The term 'quiescent perfluorinated liquid' refers to a class of fluorine containing liquids acting as a coolant that is not pumped or forced to flow around an object.

the processor for the substrate temperature to be stable. The treatment of the steady-state power dissipation presented here is crude at best because the materials used during the self-assembling process (see chapter 3) are not crystalline but amorphous. Amorphous materials can require much longer times to cool than pure crystalline materials due to *stretched exponential* cooling. For example, polycrystalline silicon has a thermal conductivity of ~14 W/m·K or about an order of magnitude less than single-crystal silicon (149 W/m·K) [Uma, 2000]. This means that the following evaluation is an approximation of the lower bound for the heat transfer rate.

The equation for heat transfer by thermal conduction through a slab of material is shown in equation (10.1).

$$\frac{Q}{t} = \frac{\kappa \cdot A \cdot \Delta T}{d} \qquad (10.1)$$

Where Q is the energy transferred through the material in t seconds, κ is the thermal conductivity of the material, A is the cross-sectional contact area between the hot and cold sides, ΔT is the temperature difference between the two sides, and d is the thickness of the material.

Heat can be dissipated either through the material in between the rods or through the rods themselves. It is likely that the high thermal conductivity of the silicon rods will dominate the heat transfer process. Since the material in-between the rods is a thermoplastic (amorphous) any estimates of the heat transfer through it will only be useful as a crude guideline for the reasons mentioned above. The heat transfer through the silicon rods is more ideal since the rods are crystalline, making the estimates shown here more accurate.

Heat transfer through SU-8

The material that is most prevalent in the space between the nanorods in each processor is the photoresist material used during the modular assembly. Some typical physical and thermal properties for a thermoplastic photoresist (SU-8) are listed in table 10.3.

Property	Symbol	Value
Thermal conductivity	κ	0.2 W/m·K
Specific heat	C_p	1.3 kJ/kg·K
Density	ρ	1300 kg/m^3

Table 10.3. Thermal and physical properties of SU-8 photoresist.

There are several assumptions we can make to estimate a lower bound on the steady-state heat flow out of this system. These assumptions will be discussed with the illustration from figure 10.2 in mind. The dashed line in the figure represents a thermal reflector, or insulator, and is a simplification that limits the direction of heat flow from the system. In reality, this constraint is overly restrictive because, as illustrated in figure 10.1, both sides of the processor substrate are in thermal contact with the cold bath. It seems reasonable to assume that all the energy being deposited into the processor is largest at the center of the structure. This simply averages the length of the thermal path and given the reflecting thermal barrier, appears to be a safe assumption.

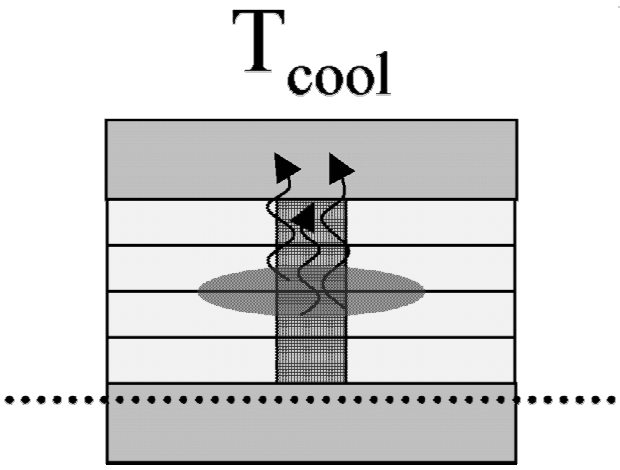

Figure 10.2. Worst-case flow of heat through the processor substrate to the cold bath. The processor is schematically represented here by four modules, when in fact a processor requires nearly 250 modules. The dashed line is a thermal barrier and all energy deposited into the processor is assumed to be largest at the center of the stack.

The cross-sectional thermal contact area of the processor to the cold bath can be approximated by taking the area of the smallest processor spacing parameters, $X_p * Y_p$, or

$(4.5\ \mu m)^2$. This will undoubtedly lead to a conservative estimate of the total heat flow because it ignores lateral heat flow within the substrate through areas where there are no active processors. The thickness of the material slab is approximately the number of modules (250) times the thickness of each module (~0.5 µm) divided by two, or 62.5 µm. The operating temperature range of the processor can be safely estimated as 20˚C to 80˚C, or the operating range of the perfluorinated coolant [Watwe, 1997]. Using these assumptions and evaluating (10.1) for *Q/t*, we get 3.8×10^{-6} J/s.

That is, using the assumptions described above, the processor substrate will support a maximum heat flow of 3.8×10^{-6} J/s. The average energy consumption of an integer instruction, as estimated in chapter 7 to be 1.1×10^{-12} J, can be used to determine the maximum integer instruction rate that the processor substrate will support. Using these figures, the rate is 3.45×10^6 instructions per second or a clock rate of 545 MHz for integer instructions.

This clock rate is the not-to-be-exceeded speed limit for the processor due to the material properties of the substrate. For this to a be a useful bound, however, the heat extraction due to pool boiling must meet or exceed the 3.8×10^{-6} J/s heat flow that the substrate can support. Otherwise, heat will build up inside the processor and the temperature will exceed the operating temperature range.

The previous estimate for the extractable heat flow from pool boiling, ~30W/cm^2, can be used to estimate the extractable heat flow from pool boiling over the processor. The minimum X_p and Y_p processor area, $(4.5\ \mu m)^2$, permits the pool boiling heat flow to be 6.075×10^{-6} W. This estimate is ~1.5 times the maximum heat flow supported by the material of the substrate, indicating that pool boiling of this type can extract as much heat as can be conducted away from the processor.

Heat transfer through silicon rods

The heat transfer rate through the silicon rods can be calculated in the same way as the SU-8 calculation. Since each rod has a diameter of 50 nm and there are 54 rods per footprint (see chapter 8) the total surface area of the silicon rods in contact with the cold bath is 1.05×10^{-13} m^2. The thermal conductivity of crystalline silicon is ~140 W/m·K and using

the same distance and temperature difference as before (d = 62.5 μm, ΔT = 60 ºC) the maximum heat transfer rate is 14 x 10^{-6} J/s, or 3.6 times greater than through SU-8.

Conclusion

Since the method used here to estimate the heat transfer through the SU-8 does not accurately model the stretched exponential cooling seen in amorphous materials, it is only useful as a crude guideline. Although the estimate for the heat transfer rate through the silicon rods may be more accurate, it is difficult to be certain about which method will dominate the transfer process. Since the estimates for both materials support a heat transfer rate within an order of magnitude of each other, an approximate lower bound of 3.8 x 10^{-6} J/s will be used as the pessimistic estimate. This will limit the DAMP and Oracle implementations to a clock rate of ~400 MHz.

10.2 Burst-mode computation

A class of applications exists where exceedingly high performance is required for a brief period in a small package. For example, an on-board computer of an anti-missile weapon or other munitions. For such applications a much denser packaging of a DAMP like computer is possible, as well as a far higher clock speed limited only by electrical considerations. This section explores the limits of this usage mode.

The heat that can be extracted by pool boiling scales linearly with the thermal contact area between the substrate and the coolant. That is, only the surface of the processor can be cooled. This is an especially important issue if the processor has little surface area compared to its volume. The processor substrate designed here has a relatively large surface area to volume ratio. If there were more processors stacked on top of each other for each "footprint" on the substrate, this would not be the case. Even though the total extractable heat flow would be the same, the amount of energy being deposited into the substrate increases with increasing number of processors.

Under these circumstances, the heat builds up more quickly than it can be transferred to the coolant. Therefore, heat flow in the system is confined to small local regions around where the electrical switching energy is being deposited into the substrate. Figure 10.3 illustrates this idea.

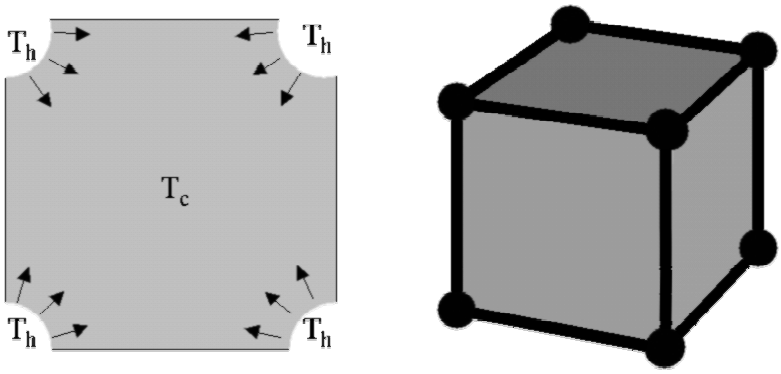

Figure 10.3. Locally confined heat flow. The interior of each unit cell absorbs the electrical switching energy.

Burst-mode computation is a way of running the processors such that even though heat is building up and would eventually push the processor's temperature beyond the operational temperature range, computation is paused before the upper limit of the temperature range is reached. Since the substrate will only support a limited heat flow to cool the processors, the cool-down period may be very long compared to the active period, but if the active period was long enough to solve an important problem, the ratio did not matter.

The relationship between the number of executed instructions and processor temperature can be derived by starting with the simple heat capacity equation.

$$Q = C_p \cdot m \cdot \Delta T \ , \qquad (10.2)$$

where C_p is the specific heat of the heat conducting media, m is the mass of the media, and ΔT is the difference in temperature across the media.

Since Q is the amount of energy deposited into the system, we can use this to model the execution of instructions. Earlier estimates say that the average integer operation takes E_{op}, or 1.1×10^{-12} J. We assume that the time scale for the instruction execution is much smaller than the heat diffusion time scale, so that $Q = n \cdot E_{op}$, where n is the number of instructions to be executed. If we also assume that the entire processor (or at least the portions of it that can absorb heat) is the structural photoresist then we can say that m =

$\rho \cdot V_{proc}$ and C_p =1.3 kJ/kg·K, where ρ is the density of the photoresist (1300 kg/m^3 in this case), and V_{proc} is the processor volume which is approximately (4.5 µm x 4.5 µm x 125 µm), or 2.53 x 10^{-15} m^3. Using these assumptions we can rewrite equation (10.2) as the following:

$$E_{OP} \cdot n = C_p \cdot \rho \cdot V_{PROC} \cdot \Delta T \qquad (10.3)$$

and solving (10.3) for ΔT,

$$\Delta T = \frac{E_{OP} \cdot n}{C_p \cdot \rho \cdot V_{PROC}} \qquad (10.4)$$

Evaluating (10.4) yields ΔT = n · 2.57 x 10^{-4} ˚C/instruction. Adopting a 60˚C operating range and assuming the processor temperature starts at the low end of the range, as many as 233 x 10^3 instructions (~90 ms @ 400 MHz) can be executed before the processors reach the upper temperature limit. This instruction count has the interesting property that is does not change with the total number of processors in the DAMP. The cool-down period will increase with increasing numbers of processors.

It is interesting to note that the ratio n / V_{proc} is a constant that depends, almost entirely, on the material properties of the DAMP processors. If we say that the minimum n that can be productively used is about 1000 (DES decryption, chapter 11) then we have a lower bound on the volume of the processor of about 1 x 10^{-17} m^3. This bound is only true if the specific heat and thermal conductivity of the photoresist material is not increased. If this volume is divided over the 27 x 10^3 unit cells found in a DAMP processor, and each unit cell is considered to be a cube, the cell will have a minimum edge length of about 74 nm (the entire DAMP cube would have an edge length of 0.15 m). The designs presented here and in chapter 8 use a cubic unit cell edge length of 500 nm, or about 7 times larger than the minimum under the burst-mode computation model.

Again, these arguments hold true if there is only local heat flow from the processor to the substrate material. The limits calculated above can be overcome if this restriction is lifted and a volume-scaleable heat extraction method is employed, such as flowing a non-conductive fluid thorough the processors themselves.

The calculation of the cool down period under this mode of computation depends on the processor arrangement. If we assume, as in section 6.2, a monolithic cubic structure and that the cooling takes place from only one side, the simulation of the cool down period is straightforward using the finite element method. First, as an approximation, equation (10.1) can be used to estimate the maximum heat flow (P_{max}) that the substrate can support through one face of the processor cube, or 1.8 J/s. Taking equation (10.2) as the total amount of heat that must be extracted from the cube,

$$Q_t = C_p \cdot N_P \cdot \rho \cdot V_{PROC} \cdot \Delta T \qquad (10.5)$$

where N_p is the total number of processors in the DAMP, or 2^{40}. The 60˚C temperature range (ΔT) yields $Q_t = 348.4 \times 10^3$ J. An approximate cooling time can be calculated by dividing Q_t by P_{max}, or 53 hours. This approximation can be tested against a FEM simulation of a 0.15 m cube being cooled by one face [FEMLAB, 2003]. The cube is initially at 80˚C and the cooling face is kept at a constant temperature of 20˚C. Figure 10.4 is the plot of temperature of a cross-section through the middle of the cube in time.

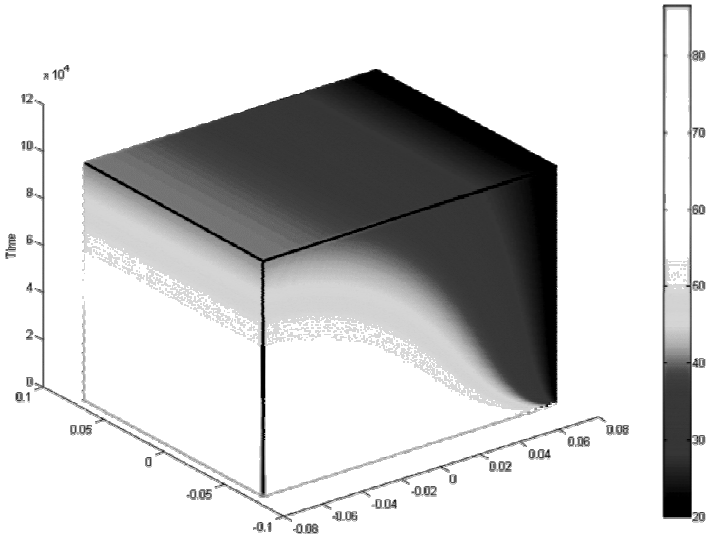

Figure 10.4. Plot of a cross-section of the processor cube as it cools in time (˚C).

The X and Y-axes are the position along the cross-section and Z is the time step (seconds.)The highest temperature spot in the cube (the far edge from the cooling face) is

within 10% of the cool side after about 53 hours. Therefore, the cube will require greater than 53 hours to completely cool. After this cool down period is over, the machine can be run again. Again, this estimate is pessimistic since the silicon rods of the structure can transfer ten times as much heat as the thermoplastic support used here.

Chapter 11. Applications and performance

11.1 The DAMP

The details of the basic instruction set can be found in chapter 8. This section provides a detailed accounting of the cycles, execution time, total energy consumed, and maximum sustainable clock rate (for the DAMP with 10^{12} processor and a 3.5 MW power budget) for several example instructions that can be found in table 11.1.

Instruction	Cycles	Execution time @ 400 MHz (µs)	Total energy consumption (J)	Max. clock rate (MHz)
ADD	159	0.3975	1.2e-12	464
ADDC	131	0.3275	1e-12	459
ADDI	170	0.425	1.2e-12	496
ANDI	136	0.34	9.75e-13	488
ASR	116+N	0.29 + N * 0.0025	8e-13 + N * 7.5e-15	502 (16 bits)
CLEARB	77	0.1925	6e-13	449
CLEARC | D	55	0.1375	3.75e-13	513
CMP	131	0.3275	9.5e-13	483
CMPI	307	0.7675	2e-12	537
CMPI8	211	0.5275	1.4e-12	528
COPY	92	0.23	7.5e-13	429
COPYH	215	0.5375	1.6e-12	470
COPYL	154	0.385	1.2e-12	449
COST	120	0.3	9.5e-13	442
CSR(N)	88+N	0.22 + N * 0.0025	6e-13 + N * 9.375e-15	485 (16 bits)
CYCLE(N)	N	N * 0.0025	~N * 1.125e-14	311 (16 bits)
DEC	159	0.3975	1.1e-12	506
GRAB(N)	157	0.3925	1.1e-12	500
INC	159	0.3975	1.1e-12	506
LOAD	280	0.7	1.9e-12	516
LSR	88+N	0.22 + N * 0.0025	6e-13 + N * 9.375e-15	485 (16 bits)
LSRC	66+N	0.165 + N * 0.0025	4.5e-13 + N * 9.375e-15	478 (16 bits)
MCOPY	103	0.2575	8.5e-13	424
NOT	171	0.4275	1.2e-12	499
ORI	202	0.505	1.4e-12	505
RANDOM	55	0.1375	4e-13	481
RESUME	6	0.015	4.5e-14	467
RINGOFF	11	0.0275	7.2e-14	535
RINGON	11	0.0275	7.5e-14	513
SET*	55	0.1375	3.75e-13	513
SETB	77	0.1925	5e-13	539
SETCREG	6	0.015	4.5e-14	467
SETC	55	0.1375	3.75e-13	513
SETSREG	5	0.013	4.2e-14	417
STORE	109	0.2725	9e-13	424
STOREH	171	0.4275	1.3e-12	460
STOREL	109	0.2725	9e-13	424
WAIT*	12	0.03	8e-14	525
XOR	148	0.37	9.25e-13	560

Table 11.1. Basic instructions and cycle counts, execution time at 400 MHz, energy consumed, and estimated maximum sustainable clock rate for 10^{12} processors operating with a power budget of 3.5 MW.

The data for table 11.1 were derived from simulation runs for each instruction using the behavioral simulation described in chapter 9. Figure 11.2 illustrates a typical output plot from the simulator. (The list of instructions in table 11.1 is not comprehensive. Other instructions are possible by using SETSREG and SETCREG as described in chapter 8.) Each operation, if it is not a variable bit-length operator, operates on the 16-bit accumulator with or without a single 16-bit register input. Instructions ending with the letter "I" indicate that an immediate value is used. The suffix letters "GE", "NGE", "LT", and "NLT" signify greater-than, not greater-than, less-than, and not less-than, respectively. The SET* instruction must have a suffix of one of the inequality operators. The WAIT instruction must have a suffix from the following list: B, NB, C, NC, D, ND, S, NS, GE, NGE, LT, or NLT. The semantics for this instruction are described in chapter 8.

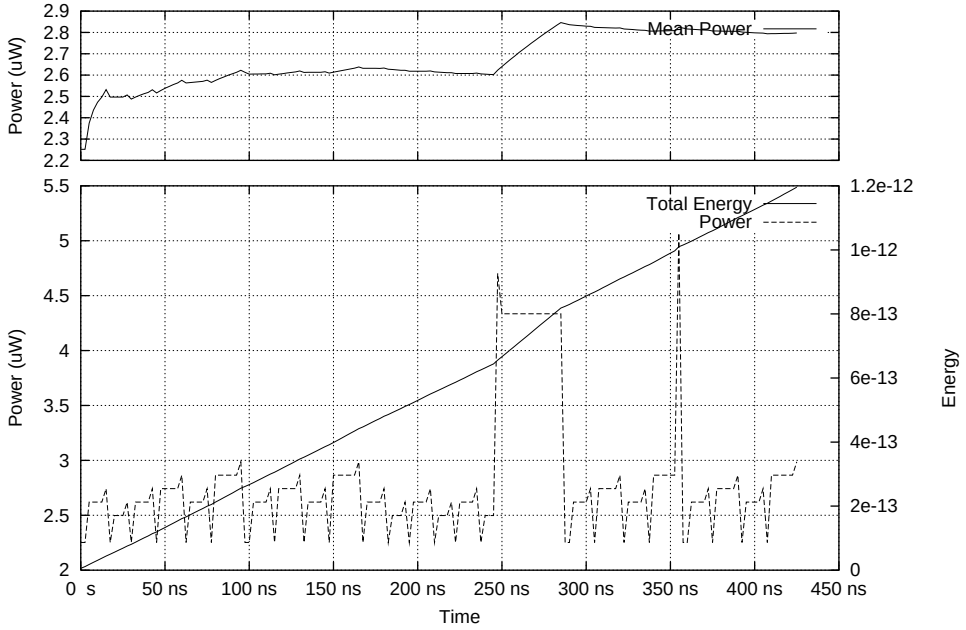

Figure 11.1. Output plot from the behavioral simulator for the ADDI instruction.

The average number of cycles in a DAMP integer operation was obtained by averaging the number of cycles used by the ADD, ADDI, ADDC, NOT, INC, and DEC. The average obtained using this instruction distribution is 158 cycles, or 0.395 µs per integer operation at a clock rate of 400 MHz. This instruction distribution consumes an average of

1.1 x 10^{-12} J per instruction. Therefore, the maximum clock rate for integer operations under a 3.5 MW power budget is 502 MHz. This is well above the conservative 400 MHz clock rate used throughout.

The results of many computations with in the DAMP need to be communicated to the node controller. A simple binary search can be used to find the identity (i.e. the random constant) of a processor that has calculated a value of interest (e.g. an extrema.) The following code implements a binary search for the identity of the processor with the largest value in its 16-bit accumulator.

```
i = 32768
n = 2
do {
      1: CMPI(i)  // Compare the ACC value to i (the current pivot)
      2: WAITLT   // Any processor less than the pivot should wait
      3: RINGON   // Used to detect the presence of processors
                  //  with ACC values greater than i
      4: RINGOFF
      If the RINGER signal was detected during step 3
       i = i + (32768 / n), n = n * 2
      Else (no RINGER signal detected during step 3)
       5: RESUME  // Need to wake all the processors (i is too big)
       i = i - (32768 / n), n = n * 2
} while n < 32768 && 0 < i < 65536
// At this point, i-1 is the largest ACC value in the DAMP
6: RESUME
7: CMPI(i - 1)
8: WAITLT         // Only the largest ACC holder(s) remain active

// Time to determine the constant... (destroys ACC, R0, R1 value)
9: CLEARB         // The B-bit will be used later to single out PEs
10: LOAD          // Load the constant
11: CSR(1)        // Copy the first bit into the S-bit
12: WAITNS        // If the bit is a 0, wait.
13: RINGON
```

```
14: RINGOFF
// If RINGER signal detected during step 13
  // Record possible 1 in current bit position
15: RESUME
16: WAITS         // If the bit is a 1, wait.
17: RINGON
18: RINGOFF
// If RINGER signal detected during step 17
  // Record possible 0 in current bit position

// Multiple unique processors with the same ACC value
//  may respond and make both a 0 and a 1 the likely
//  current bit.  That is, the RINGER signal will be
//  detected both times.  Just pick one (0 or 1) if both are likely.

19: REsuME
// If the 0 is the current bit value (step 17's signal > step 13's)
20: WAITNS  // The 0 bit PEs wait
21: SETB    // The B-bit represents "do not participate"

22: RESUME
// If the 1 is the current bit value (step 13's signal > step 17's)
23: WAITS   // The 1 bit PEs wait
24: SETB    // The "do not participate" flag

25: RESUME
26: WAITB   // Any PE with the B-bit set will wait

// Append the current bit (whichever value was chosen) to
//  the accumulated "inferred" constant

// Repeat from step 11 for the remaining 15 bits
//

// Repeat this entire procedure twice more for R0 and R1
```

Steps 1 through 5 of the search for the largest ACC value require (in the worst-case) 260 cycles per loop. The loop is executed 16 times for a total of 4,160 cycles. Steps 11 through 26 require 355 cycles per loop. Since this loop must be executed once for each random bit (48 times in total) the total number of cycles for just this loop is 17,040. The entire procedure requires 21,200 cycles, or 42.4 μs using a 400 MHz DAMP.

Machine comparisons

The number of integer operations per second, memory size, power consumption, and volume of each machine can be used to draw a simple comparison among machines. Such a comparison gives a qualitative feel for how the machine may perform on simple tasks. Table 11.2 lists several simple metrics for some comparison machines and the DAMP (with 1 x 10^{12} processors.)

Machine	16-bit integer op./sec	Norm. integer op./sec	Memory size (bytes)	Power budget (W)	Volume (m^3)	Energy / op. (J/op)
DAMP	2.53×10^{18}	1.00	1.0×10^{13}	3.5×10^{6}	12	1.3×10^{-12}
IBM BlueGene /L	2.88×10^{15}	1×10^{-3}	7.0×10^{13}	$\sim 3 \times 10^{6}$	533	1.04×10^{-9}
NEC Earth Simulator	3.28×10^{14}	1×10^{-4}	1.0×10^{12}	12.8×10^{6}	13 000	3.9×10^{-8}
SETI@home	2.4×10^{14}	9×10^{-5}	$> 1.4 \times 10^{12}$	508.3×10^{6}	312 800	2.1×10^{-6}
HP ASCI Q	6.2×10^{13}	2×10^{-5}	1.3×10^{13}	3×10^{6}	11 300	4.8×10^{-8}
IBM ASCI White	5.6×10^{13}	2×10^{-5}	2.6×10^{13}	1.2×10^{6}	6 600	2.1×10^{-8}
Thinking Machines CM-200	4.5×10^{10}	1×10^{-8}	8.59×10^{9}	28×10^{3}	4.16	6.2×10^{-7}
Intel Pentium 4	3.44×10^{10}	1×10^{-8}	$< 2 \times 10^{9}$	~150	0.08	4.36×10^{-9}
MasPar MP-1	1×10^{10}	3×10^{-9}	2.68×10^{8}	3.7×10^{3}	0.95	3.7×10^{-7}

Table 11.2. Comparison of several machines with respect to integer operation rate, memory size, power consumption, volume, and energy per operation.

The data for table 11.2 were taken, and in some cases extrapolated, from several architectural overviews and surveys, chiefly [BlueGene, 2002], [Dongarra, 2002], [Intel, 2003], [MacDonald, 1992], [SETI@home, 2003], [Top500, 2003], and [Warren, 2002]. The

16-bit integer performance for the Intel Pentium 4 was calculated using its sustainable memory to processor transfer rate of 4.3 GB/s [Intel, 2003].

Another interesting way to look at the gap between the DAMP and other machines is to consider that, on the embarrassingly parallel problems to which the DAMP is well-suited, it is about 15 years ahead of the nearest (yet to be completed) supercomputer, the IBM BlueGene /L with a similar power budget. This trend of performance doubling every 18 months is common in other areas of computing and has been shown to hold, on average in the past, in supercomputing performance trends [BlueGene, 2002].

The ACM has the SETI@home project on record as the owner of the largest computation ever (being) performed with approximately 5.16×10^{21} 64-bit integer operations [SETI@home, 2003]. That number translated into the equivalent number of 16-bit operations, listed for each machine in table 11.3, puts the scale of self-assembled computing into perspective.

Machine	Time to beat largest computation on record	Normalized time
DAMP	2 hrs, 13 min	1.0
IBM BlueGene /L	83 days	800
NEC Earth Simulator	1 year, 11 months, 28 days	7,000
SETI@home	3 years	10,000
HP ASCI Q	10 years, 6 months, 20 days	40,000
IBM ASCI White	11 years, 8 months, 7 days	~40,000
Thinking Machines CM-200	14,544 years	50,000,000
Intel Pentium 4	19,025 years	70,000,000
MasPar MP-1	65,449 years	200,000,000

Table 11.3. Time to beat the largest computation on record, as of February 2003.

11.2 Blind decryption of the Data Encryption Standard (DES)

The data encryption standard (DES) has long been used as a secure method of encrypting sensitive information. Recently, the growing number of computing machines that can complete brute force attacks on secured data has put the strength of this form of

encryption into question. For this reason alternative encryption standards are being investigated.

The purpose of this comparison is to show how the DAMP would perform on a particular problem of interest to a wide community. The DES has been heavily investigated since the late 1970's and its implementation is well known [Feldmeier, 1989]. Fast bit-serial implementations have also been developed in an attempt to improve the throughput of DES encryption and decryption modules and to test the security strength of the standard [Biham, 1997].

The particular bit-serial DES algorithm implemented in [Biham, 1997] uses approximately 17,000 bit-wise operations per DES decryption. This count includes the comparison operation that tests whether or not the decryption was successful. The usual completion criterion for the algorithm is to successfully decrypt a single known ciphertext to the proper plaintext. In this case, the algorithm is being employed to perform a blind search through the entire DES key space to test the algorithm's security rather than perform a complete data stream decryption.

The DAMP can be used to perform this search by assigning each processor a key from the 2^{56} bit key space. On average this input space will require 2^{55} decryption steps. Since the DAMP has only 40 assembly-time bits, the remaining 15 bits must be obtained at runtime. The instruction set described in section 8.1 includes all of the logic operations that are needed in the DES decryption process. The most common instructions in the algorithm are the XOR and CSR instructions. On average, the logic instructions in the DAMP require 10 cycles per bit-wise operation. Using this estimate, a single DES decryption will require 170,000 cycles on the DAMP.

The simulations and performance estimates presented in chapter 9 used a clock rate of 400 MHz to determine the total running time for various programs and instructions. Using this same clock rate yields a DES decryption rate of 2,352 decryptions per second per processor. The aggregate performance of the DAMP, with 2^{40} processors, is approximately 2.5×10^{15} DES decryptions per second. At this decryption rate the DAMP can cover the entire 55 bit input space in about 15 seconds. Table 11.4 lists the estimated performance of several machines on the blind DES decryption problem.

Machine	Bit-wise op./sec	Average DES decryption time	Normalized time
DAMP	4×10^{19}	15 sec.	1
IBM BlueGene /L	4.6×10^{16}	3 hours, 41 min.	870
NEC Earth Simulator	5.25×10^{15}	32 hours, 24 min.	7,600
SETI@home	3.84×10^{15}	1 day, 20 hours, 18 min.	10,400
HP ASCI Q	9.9×10^{14}	7 days, 3 hours, 51 min.	~41,000
IBM ASCI White	8.96×10^{14}	7 days, 21 hours, 53 min.	~45,000
PixelFlow (CipherFlow)*	1.04×10^{13}	1 year, 10 months, 10 days	~3,840,000
Thinking Machines CM-200	7.2×10^{11}	26 years, 11 months, 21 days.	~56,000,000
Intel Pentium 4	5.5×10^{11}	35 years, 3 months.	~73,000,000
MasPar MP-1	1.6×10^{11}	121 years, 4 months.	~254,000,000

Table 11.4. Comparison of blind DES decryption times for various machines. * Data taken from [Kedem, 1999].

11.3 Global optimization

The DAMP can also be applied to a much more broadly useful class of problems. Many science and engineering problems can be posed as global optimization problems, which seek to find the largest or smallest value for an objective function over a domain. The challenge in solving these problems comes from the large number of variables and multiple local minima that deceive simple search algorithms. Consider the hypothetical objective function shown in figure 11.3. This function has many local minima and the global minima, indicated by the black arrow, has a very narrow "opening" for the search algorithm to find.

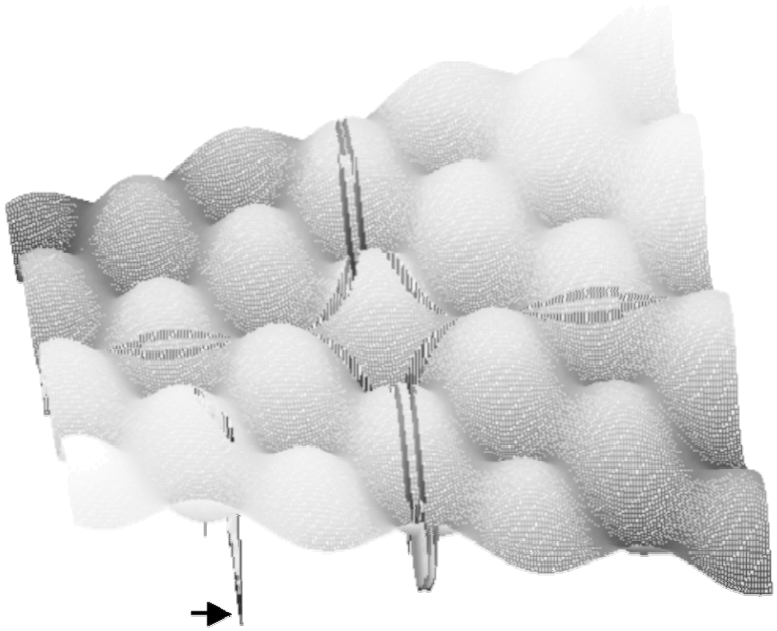

Figure 11.2. A difficult, constrained objective function with many local minima. The black arrow indicates the global minimum for this region.

Stochastic global optimization is a method of sampling an objective function at random points in the problem space and comparing the results at each point. Since the time required to exhaustively search the problem space at a resolution sufficient to be useful is far too large using conventional machines, the best local minimum is chosen from local searches

starting at a random set of starting locations. A new set of random points is selected that concentrates the search around the best-found solution. That is, the search continues but focuses on a few of the last-best answers. Typical calculations for each sample include the objective function and sometimes numerical derivatives (gradients) at the point. If the objective function has a well-behaved and computable gradient, this can be used as a local indicator of how to choose the next best solution to minimize the objective since the objective is decreased along that direction. This *gradient descent* approach is very sensitive to numerical instability because it uses the gradient to choose the next sample point. It is also very susceptible to getting trapped in local minima.

Parallel pattern search (PPS) has emerged as another technique used to optimize difficult objective functions [Hough, 2000]. This method uses a search along each dimension of the problem space to find the global minimum. The starting point for each iteration of the search after the first is the optimal point from the last round of evaluations. This technique has provable convergence to the minimum as long as certain rules are followed for adjusting the step size along each dimension and for comparing objective values are compared.

This approach to global optimization has been applied to continuous and mixed-variable problems. The typical continuous-variable optimization problem is formulated as follows.

$$\begin{aligned} &\text{minimize} && y = F(x) \\ &\text{where} && x = (x_1, x_2, x_3, \ldots, x_N) \in \mathbf{X}, \text{ and} \\ & && y \in \mathbf{Y}. \end{aligned}$$

The formulation above can be re-stated as a minimization of a function *F* that is subject to input and output constraints. That is, x_1 through x_N must belong to an allowable set of inputs $\boldsymbol{X}$ and the output *y* must belong to an allowable set of outputs $\boldsymbol{Y}$. Generally, *N* is less than fifty and several thousand iterations are required to converge to the best-known minimum [Audet, 2000; Hough, 2000; Zitzler, 2000; Fieldsend, 2002].

The DAMP can be used to solve continuous variable minimization problems that are much larger in dimensionality than those solvable today. The pseudo code below can be used with 32-bit fixed-point variable optimization problems. The vector x_k is the best-known

solution after each step. The problem space is spanned by a positive spanning set[12] **D**, where d_i is a unit vector from **D** along the i^{th} dimension of the problem space. The functions $C_x(x_i)$ and $C_y(y)$ are used to verify that the input and output vectors, respectively, satisfy the problem constraints.

The program listed below is run at each processor node. Since there are 2^{28} processors per node, the random number generated at each processor has only 28 bits of significance. This means that to cover a 32-bit random number space each processor must run the program 16 times with a new 4-bit low order value each time. The value, Δk, is simply incremented between loops. Each processor takes its Δk and uses it to compute a new input vector. The particular dimension that the processor searches (d_i) along is specific to the processor node. The new input vector is checked against the input constraints and if they are satisfied the function (F) is evaluated. The output from the objective function is checked against the output constraints and if they are satisfied the processor participates in a minimization query, or MIN-QUERY. This query is conducted by the processor node and it searches, bit by bit, for the smallest objective function value found by any of its processors. More details on the MIN-QUERY operation are given below the pseudo-code.

```
For each processor node j (PNj),
  For each processor at node PNj, k, (k is a random 28-bit integer)
    1: Δk = k << 4;
    2: evaluate and verify Cx(xk + dj·Δk)
    3: evaluate y = F(xk + dj·Δk);
    4: evaluate and verify Cy(y)
    5: participate in MIN-QUERY(y, xk + dj·Δk)
    6: Δk = Δk + 1;
    7: repeat 16 times from step 2.
```

The first *for* statement must be executed sequentially on the DAMP. Once the program has been run, the X_k solution vector that best minimizes the objective function is chosen for the next round.

[12] A positive spanning set is a set of vectors that can be combined using non-negative scalars to form all possible vectors in a constrained space.

The second *for* statement in the program can be distributed in parallel to the 4,096 processor nodes available on the DAMP. The third *for* statement can be run in parallel on all the processors within a node because they each use the same d_i vector in calculating a new X_k vector. This means that the DAMP can, in parallel, optimize a 4,096 dimension (M = 4096) 32-bit fixed-point problem per round of the program above.

The following discussion provides an analysis of the program's execution time.

1: $\Delta k = k << 4;$

Step 1 first requires the random integer *k* to be loaded. The LOAD instruction can be used for this. The multiplication can be implemented by either performing 4 logical left-shifts, or a multi-word circular right-shift by 28 bits followed by a logical AND with 0xFFF0. This step will take no more than 2,000 cycles.

2: evaluate and verify $c_x(x_k + d_j \cdot \Delta k)$

3: evaluate $y_i = f_i(x_k + d_j \cdot \Delta k);$

4: evaluate and verify $c_y(y)$

The constraint functions and objective function need to be preprocessed before being executed on each processor within a node to fit within the memory limitations. Since each processor node is responsible for a single search dimension, it is possible to precompute the value of all terms involving variables other than X_j. The other variables can be combined to form a function of the single variable X_j since all the other variables will remain constant at the j^{th} processor node. If we consider the precomputation of the single-variable form of the constraint and objective functions to be part of the problem description, then we need only consider the complexity of the resulting functions to estimate the execution time of step 2.

If the constraint and objective functions can be decomposed into a sum of terms that does not exceed the number of variables in the problem, *M*, then we can use this as an estimate for the evaluation complexity. Each term will have a precomputed multiplier that represents the contribution of the other variables and will only need to have the contribution of X_j calculated. Each processor must perform this calculation since X_j is incremented by Δk, which is a processor-specific value. The calculation $X_j = X_j + \Delta k$ will not require more than 300 cycles. If each term requires sixty-four 32-bit additions (316 cycles each) and multiplications (10,112 cycles each) then the entire function (4,096 terms) will require 2.73 x

10^9 cycles, or 6.8 seconds if the DAMP is run at a 400 MHz clock rate. That is, each constraint function, C_x and C_y, and the objective function, $F(x)$, has an estimated execution time of 6.8 seconds. Therefore, steps 2 to 4 will require 3 x 6.8 ≈ 20 seconds on the DAMP.

```
5: participate in MIN-QUERY(y, xk + dj·Δk)
```

MIN-QUERY (a, b) is a routine that first queries the processors within a processor node for the minimum value, *a*, and then collects the argument, *b*, that generated the minimum value. The DAMP has enough register memory to store both the *y* value and the $x_k + d_j \cdot \Delta k$ argument and participate in a 32-bit MIN-QUERY. The value being minimized, *y*, is first loaded into the accumulator and R0. This pair is then used in a 32-bit value binary search (see section 11.1) that requires ~44,000 cycles.

```
6: Δk = Δk + 1;
```

Step 6 is a simple 32-bit increment of the current Δk, which will not require more than 340 cycles.

```
7: repeat 16 times from step 2.
```

The total program from step 2 up to step 7 requires ~2.7 x 10^9 cycles to execute, or 4.3 x 10^{10} cycles after looping 16 times. Therefore, the DAMP can completely sample an objective function with 4,096 dimensions at 32-bit resolution and return the best solution in 110 seconds.

Example problem

Solving mixed-variable optimization problems is an important application of these optimization techniques to real engineering problems. The thermal intercept problem is one such application [Audet, 2000; Kokkolaras, 2000]. The brief definition of this problem, illustrated in figure 11.3, goes as follows.

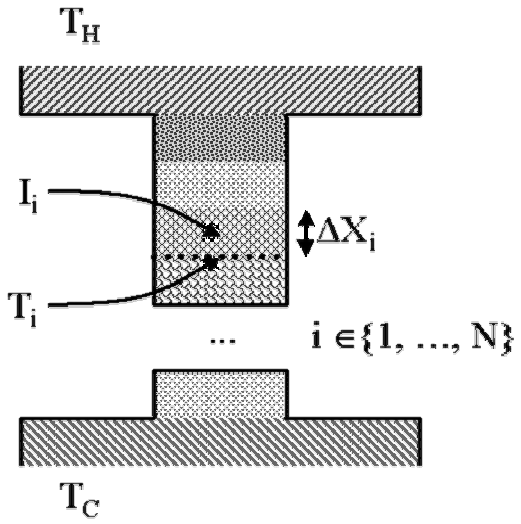

Figure 11.3. Thermal intercept problem. N shields are placed between a hot (T_H) and cold (T_C) side. Each shield is padded with a slab of insulating material, I_i, ΔX_i thick and must maintain an interface temperature with the next shield of T_i.

The number, thickness, composition, and refrigerated interface temperature of a set of thermal intercepts, or heat shields with insulation, put within a thermal gradient (T_H to T_C) is to be determined by minimizing a set of objective functions. The objective functions are typically power and entropy change through the stack of intercepts, with the cross-sectional area of the shields being fixed [Kokkolaras, 2000].

The insulating material is chosen from a library of materials making this a mixed-variable problem. That is, the composition of the intercept is determined by an integer that has no meaning in the problem except as an index for choosing material properties [Audet, 2000]. Typical problems must limit the size of the material library to about 3 materials because of the exponential growth of the problem difficulty as the library grows. For example, a thermal intercept problem with *N* possible materials and up to *M* shields has N^M different configurations. For large *N* this problem becomes extremely difficult to solve.

We can calculate the performance of the DAMP on this problem by considering the power equation (11.1) used in the optimization objective function.

$$P_i = A \cdot C_i \cdot \left(\frac{T_H}{T_i} - 1 \right) \cdot \left(\frac{\int_{T_i}^{T_{i+1}} \mathrm{K}_i(T) \cdot dT}{\Delta X_i} - \frac{\int_{T_i - 1}^{T_i} \mathrm{K}_i(T) \cdot dT}{\Delta X_{i-1}} \right) \quad (11.1)$$

The summation of equation (11.1) over all heat intercepts forms the complete objective function shown in equation (11.2).

$$f(\Delta X, T) = \sum_{i=1}^{M} P_i \quad (11.2)$$

This problem can be distributed over the processor nodes of the DAMP with the first 4,096 possible material configurations each handled by one node. That is, of the N^M possible configurations, each processor node is responsible for $N^M / 4096$ configurations. This can be thought of as splitting the first $\log_N(4096)$ levels of the N-ary (N branches at each node) configuration tree among all the processor nodes. The remaining levels of the configuration tree are handled by the random distribution of processors within each node. That is, each processor must evaluate equation (11.2) for one of the $N^M / 4096$ remaining configurations from its processor node, or the next $\log_N(N^M / 4096) = M - \log_N(4096)$ levels of the configuration tree. These configurations can be chosen randomly by using $\log_2(N) \cdot (M - \log_N(4096))$ of the 28 unique bits from each processor in the node. The next $\log_2(M)$ bits of the remaining random bits can be used to choose one of the $2 \cdot M$ dimensions to search.

This leaves a remainder, $R = 28 - \log_2(N) \cdot (M - \log_N(4096)) - \log_2(M)$ random bits for use as the particular value along the dimension selected by the processor. If the desired precision, *P*, is greater than *R* then $2^{(P-R)}$ iterations must be executed, incrementing the variable between loops, to cover the entire *P*-bit dimension as was done above. Figure 11.5 illustrates the distribution of processors for N=8 and M=8.

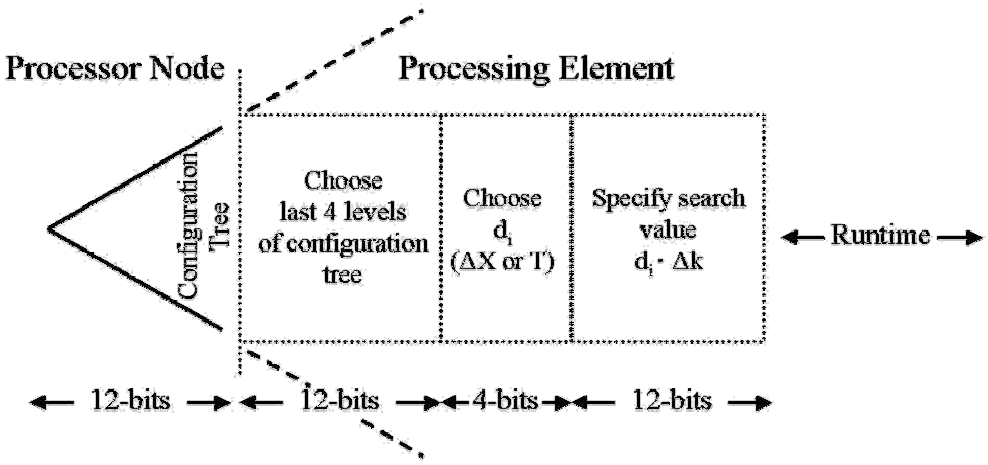

Figure 11.4. Distribution of processing elements to solve the N=8, M=8 thermal intercept problem.

If *P*=16, the program will need to be run 2^4, or 16, times to cover the space since there are only 12 bits dedicated to a specific search value along the dimension selected by the processor. Each processor chooses a dimension to search along by using the $\log_2(M)$ bits (as before) to select what variable in equations (11.1) and (11.2) to exchange for its specified search value, Δk. The values for the other variables (that are not the selected variable) are taken from the last best answer. These are lumped together and precomputed to make a unique "compressed" equation for each dimension. That is, $2 \cdot M$ equations can be made from equations (11.1) and (11.2) by extracting a single dependent variable and treating all others as constants. Each processor waits until it is time to compute the value of the function it has chosen by selecting a dimension to search along, so that at any time $\sim((2 \cdot M - 1) / 2 \cdot M)$ of the processors are idle during this period.

The material specific properties of the insulating material, $K_i(T)$ in equation (11.1), must be broadcast to all the processors within a node because they do not have sufficient memory to store the data locally. The integration of the table of data can be precomputed in discrete intervals and sent to the processors for accumulation. That is, if a processor needs to calculate the $K_i(T)$ integral from (11.1) because it is searching along T_i, it can use its integration limits to control when it starts accumulating values and when it stops, since all values of the $K_i(T)$ integral will be broadcast. Since an interval will not necessarily start or

end at the same T_i value as the processor's integration limits, a correction needs to be made to the accumulated sum because it will be an overestimate if the interval ended after the particular upper integration limit. Since only one of the limits will be searched by any single processor, the other limit (a constant in this case) can be used to adjust the precomputed integration values so that the first (or last) interval begins (or ends) at the constant limit, evenly. Considering the case where the lower limit of integration is fixed, and assuming that the K_i values are constant within each interval we can use the correction *C*,

$$C = \frac{\int_A^B K_i(T) \cdot dT}{B - A}, \text{ where } A \le T_i \le B.$$

The overestimate in the accumulated sum can be corrected by subtracting $C \cdot (B - T_i)$, from the accumulated sum.

The look-up-table and integration step is implemented as follows.

```
ACC = Si,   // Current integrated sum, zero initially
R0 = Ti+1,  // The upper limit of the integration
R1 = C      // Correction factor for the interval in
            //  which this processor's integration ended.
R2 = S      // The integrated sum, so far. (Zero initially)
1: ADDI(Si) // Accumulate the next interval
2: STORE(R2) // Save it…
3: LOAD(C)  // Load the correction factor for this interval
4: COST(R0, R1) // Save the correction, load the specific Ti value
5: ADDI(-T) // Subtract the current interval's end value (T)
6: WAITNLT  // any processor that didn't end the integration at T...
7: SETB     // all processors that DID end at T, set the B bit...
8: RESUME   // all-aboard
9: WAITB    // any processor that's done goes back to sleep...
LOOP from 1 to 9 for each entry in the table
```

There are a total of 1,100 cycles per entry for each integrated-table as implemented above, or 281.6 x 10^3 cycles for a 256-entry table. Figure 11.6 illustrates the power consumption during a single loop through the program above. Since the program consumes

less power per instruction than the average integer instruction, the clock rate can be increased 500 MHz and still remain under the 3.5 MW power budget. All other instructions will be run with a 400 MHz clock rate. To simplify comparisons, the discrepancy in the two clock rates can be accounted for by reducing the cycle count per look-up-table entry by the ratio of the two clock rates, or 1.25, to yield the same execution time at the faster clock rate, or 2.2 μs per entry. That gives an effective cycle count of 880 cycles per table entry, or 225.28 x 10^3 cycles for a 256-entry table.

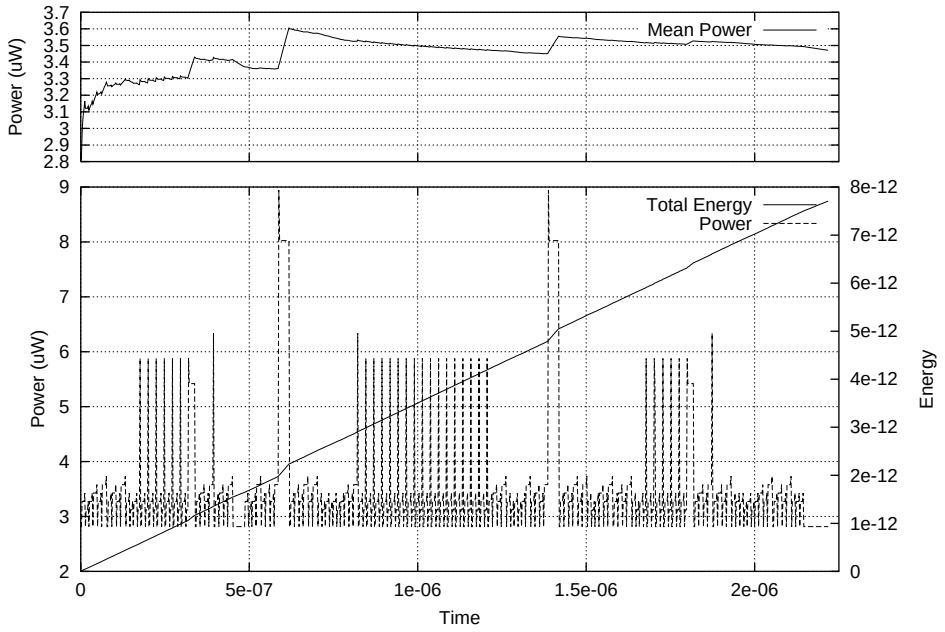

Figure 11.5. Power consumption of one look-up-table iteration. This program must be executed for each entry in the table.

The correction step can be taken after all tables have been integrated and can be executed in parallel. Table 11.5 lists the estimated number of cycles for calculating (11.1) and (11.2). The functions that involve the look-up-table integration, used by processor's searching a T_i variable, are more complex than the simpler ΔX_i search functions (since they do not use integrated material data) and will be used as an upper bound on the execution time.

Equation (11.1)		
Instruction class	**Count**	**Total DAMP cycles**
additions - subtractions	2	316
multiplications - divisions	6	30336
256-entry look-up-table & integration	2 * N	$450.56 \times 10^3 * N$
Correction step (2 adds & multiplies)	1	61304
Total		$T_{11.1} = 450.56 \times 10^3 * N + 91956$
Equation (11.2)		
Instruction class	**Count**	**Total DAMP cycles**
equation 11.1	M	$M * T_{11.1}$
additions - subtractions	M	M * 158
Total		$M * (158 + T_{11.1})$
Total after $2^{(P-R)}$ iterations		$2^{(P-R)} * M * (158 + T_{11.1})$

Table 11.5. Estimated cycle counts for evaluating the heat shield objective function with at most *M* thermal intercepts and *N* possible insulating materials. All instruction counts are for 16-bit operands except for multiplications and divisions.

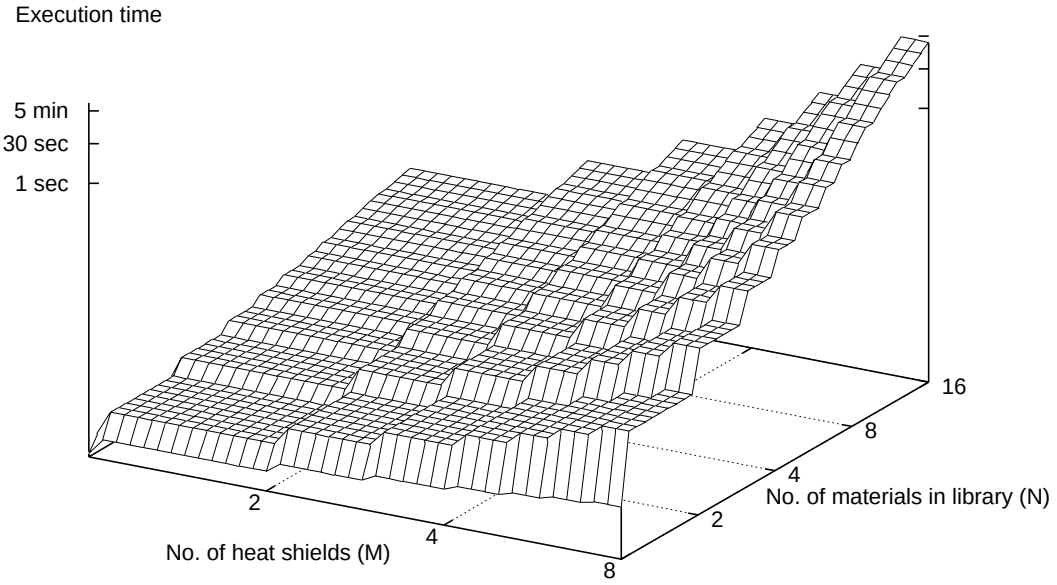

Figure 11.6. Tradeoff between library size and maximum number of heat shields. The surface depicts the execution time on a 400/500 MHz DAMP.

The same instruction counts listed in table 11.5 can be used to determine the performance of the nearest competitor to the DAMP, the NEC EarthSimulator with a few modifications to make the program more efficient on that machine. To simplify the comparison, N=16 and M=8 will be used to calculate execution times. As shown in figure 11.7, the DAMP can solve this problem in ~5 minutes.

The NEC EarthSimulator (ES) has 640 nodes with 8 processors per node, 16 of the total processors are devoted to supervisory roles leaving 5,104 processors for problem computation. Since each processor has ample memory space to store material specific data, the look-up-table and integration step can be reduced to about 10 instructions (ignoring address calculations) per entry in the table. If we use a 256-entry table the total maximum instruction count for the look-up and integration is 2,560 instructions since each entry must be broadcast to the processors (i.e. each entry *might* be used.) This leads to 5,128 instructions to evaluate equation 11.1. The combination of objective functions in equation 11.2 requires $8 \cdot 5128 + 8 = 41{,}032$ instructions. That is, 41,032 instructions are required to evaluate a single configuration of heat shields for a single instance of the continuous variables (T_i and X_i).

Since the heat intercept problem is a $2 \cdot M$ dimensional problem and M=8, or 16 dimensions, each processor can choose one dimension to search along for a total of 5104 / 16 = 319 processors devoted to a single dimension. Since there are 16^8 configurations of insulating materials (an 8-level 16-ary tree), each of the 319 processors searching in one dimension must handle 13.46×10^6 unique configurations. Also, in order to cover the entire 16-bit input space along a single dimension for each configuration, every processor must run the program 2^{16} times, incrementing its Δk value between loops.

The total instruction count is therefore 41032 instructions / configuration **X** 13.46×10^6 configuration · dimensions / processor **X** 2^{16} processor · cycles / dimension = 5.52×10^{11} instruction · dimensions / processor **X** 2^{16} processor · cycles / dimension = 3.62×10^{16} instruction · cycles. Since each processor of the ES can execute ~6.8×10^{10} 16-bit integer instructions per second, the entire problem will take 3.62×10^{16} instructions ÷ 6.8×10^{10} 16-bit instructions per second = 5.32×10^5 seconds, or ~6 days. That is, the DAMP covers the same space at the same resolution ~2,000 times more quickly.

Appendix A. DNA-functionalized single-walled carbon nanotubes

[Published as "DNA-functionalized single-walled carbon nanotubes", Dwyer, et al., 2002]

Prologue

The use of carbon nanotubes in a self-assembling fabrication process appears, at first, to be quite practical. The novel electronic properties and controllable doping of carbon nanotubes makes this seem even more plausible. However, the single molecular shells that make a single-walled carbon nanotube are by virtue susceptible to point defects that can dramatically alter the electronic behavior of the nanotube. Doping of carbon nanotubes has been demonstrated but most use vapor-phase donor or acceptor ions that must adsorb to the nanotube surface. Doped carbon nanotubes have even been used in depletion mode field-effect-transistors but are still extremely sensitive to environmental conditions. Any molecules, like the donor or acceptor ions, that adsorb to the nanotube sidewall will alter the electronic behavior of the system. It is for these reasons that we use silicon nanorods in place of carbon nanotubes for use in self-assembled computer fabrication.

Introduction

We present here the use of amino-terminated DNA strands in functionalizing the open ends and defect sites of oxidatively prepared single-walled carbon nanotubes, an important first step in realizing a DNA-guided self-assembly process for carbon nanotubes.

The unique electrical properties of single-walled carbon nanotubes (SWNTs) make them good candidates for a self-assembling process that can controllably form electronic circuitry. Control over the assembly process may be derived from the selective binding of complementary DNA strands as in [Mbindyo, 2001]. This work represents a step toward the DNA-guided assembly of carbon nanotubes by demonstrating that the well-known chemical pathway already discovered to attach amino-terminal compounds to carbon nanotubes is also compatible with DNA functionalization [Liu, 1998]. Previous work in the field of nanotube-

DNA self-assembly has focused on either non-covalent associations between the nanotubes and DNA molecule [Guo, 1998] or the self-organizational properties of a carbon nanotube / DNA system [Buzaneva, 2002]. Other studies have explored the use of DNA self-assembly and frayed wire systems [Batalia, 2002]. While these studies illuminate our understanding of self-organizing systems, our work focuses on developing a controllable assembly system. We hope that one day a high level of control will be possible by using the hybridization of covalently bound DNA strands on carbon nanotubes.

The single-walled carbon nanotube material, as formed by a laser ablation method, is first purified in nitric acid and then oxidized in a sulfuric and nitric acid mixture as described by Liu [Liu, 1998]. The product of this purification is a solution of open-ended nanotubes with terminal carboxylic acid groups. The carboxylic acid groups can be reacted with primary amine compounds by any of several condensation reactions [Hendrickson, 1970]. The reactions couple the amine compound to the nanotube by way of an amide bond [Wong, 1998]. Figure B.1 illustrates the basic chemical pathway we have used in this work.

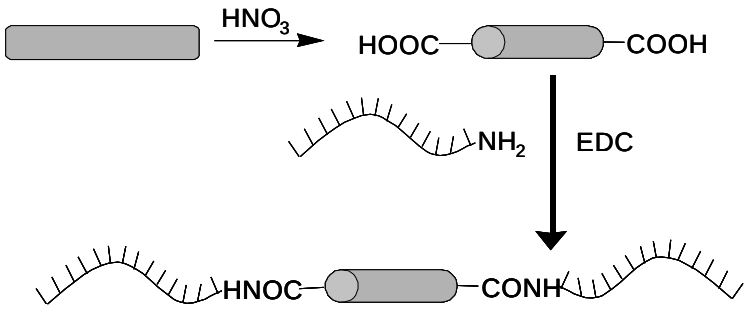

Figure A.1. DNA/nanotube reaction scheme. Capped nanotubes are oxidatively opened and then reacted with amine-terminated single-stranded DNA.

A.1 Materials and Methods

Linking DNA strands to the nanotube requires specially prepared DNA strands. Amino-terminated DNA strands can be purchased from commercial suppliers.[13] For the first

[13] Operon Technologies, Inc., Alameda, CA, http://www.operon.com

set of experiments described here, we used 10 μM amino-terminated DNA[14,15] which had been purified using polyacrylamide gel electrophoresis (PAGE) by the vendor. Our second experiment used 51.1 μg/mL of lambda-DNA extracted from bacteria[16].

We have imaged the functionalization of the as prepared single-walled carbon nanotube material (diluted in DMF) and the amino-terminated DNA strand oligo 1 and carboxylic-terminated DNA strand oligo 2 by ^{32}P radioisotope PAGE. Oligo 2 serves as our control to determine that the DNA does not non-specifically interact with the nanotube material. We expect the low mobility of the SWNTs in the polyacrylamide gel to prevent any bound DNA from migrating at its normal rate through the gel. DNA strands migrate through a gel because their charged phospho-diester backbone interacts with the applied electric field. Shielding effects limit the force that can be applied to any strand and this means that strands will move at a rate that is inversely proportional to their length. The longer a strand is the greater the number of interactions that strand will have with the gel, thus slowing its progress through the pores of the gel.

The DNA strands were first labeled with ^{32}P-γ-ATP using a T4 kinase enzyme[17]. Each of the following were added to a 0.5 mL microcentrifuge tube: 1 μL of oligo 1 or oligo 2 (10 μM), 10 μL ^{32}P-γ-ATP (3.3 μM, 3000 Ci/μL), 1 μL T4-polynucleotide kinase (10 units / μL), 6 μL 5x T4 reaction buffer (as described in the enzyme specification sheet, with the exception that no DTT (dithiothreitol) or Tris-HCl was used), 12 μL nanopure H_2O (18.2 MΩ). The replacement of the Tris-HCl with PBS (phosphate buffered saline) from the T4 kinase reaction buffer was required to prevent side reactions between the amine groups on Tris with the carbon nanotubes. Since Tris is smaller and more mobile in solution than the DNA strands, it will dominate the competition for carboxylic acid sites on the nanotubes. DTT was omitted from the reaction protocol to eliminate unnecessary additives. The labeling reaction was incubated at 37°C for 2 hours and then heat killed for 3 minutes at

[14] Oligo 1 sequence: 5'-NH_2 - ATG GTG GAT AGG CGA CTC AAG GGC-3'

[15] Oligo 2 sequence: 5'-TTT TTT TTT TTT TTT TT-COOH-3'

[16] Lambda-DNA cl857 Sam7 isolated from *E. coli* strain W3350, 48,502 bp in length, Promega Corp., USA.

[17] T4-polynucleotide kinase and ^{32}P-γ-ATP, Amersham Pharmacia Biotech.

>65°C. Following the heat kill, each DNA strand was purified from the kinase reaction using a phenol extraction.

The ^{32}P labeled oligo 1 and diluted SWNT material were added to 50 mM EDC[18] and incubated at room temperature for 24 hours. Eight reactions were prepared using 1x, 2x, 4x, 8x, 16x, 32x, 64x, and 128x dilutions of 50μg/ml of SWNT in DMF and labeled as reaction A. Two control reactions were performed with one having no EDC and 50μg/ml SWNT (A9) and the other having no SWNT material (A10). The second set of reactions, reaction B, was identical to reaction A but used oligo 2 instead of oligo 1. Oligo 2 should have no primary amine groups for reaction with the SWNT material due to its sequence (Poly-thymine has no primary amine groups.)

The products of reaction A and reaction B were loaded on to a 10% denaturing polyacrylamide gel using a loading buffer (30% glycerol, 25 mM EDTA, and 0.01% bromophenol blue and xylene cyanol.) In addition to the previously described reactions, a sample of the purified oligo 1 and oligo 2 (A11 and B11) was loaded as well as the kinase reaction product (A12 and B12) from which they were each purified. Each reaction was heated to 100ºC for 3 minutes before loading on the gel to denature any non-specific DNA binding. The gel was pre-run for 1 hour using TBE, pH 8.9, buffer (Tris-HCl, borate, EDTA). Following the pre-run, the gel was loaded with samples and run for 3 hours at ~900 V, 80 mA, with a closed-loop controller maintaining a constant power of 65 W by varying the plate voltage. The gel slab was then imaged by exposure to a radioisotopic imaging screen for 15 hours. A screen scanner was then used to acquire a digital image of the gel.

We performed the second experiment to verify the reactivity of DNA with the SWNT material. We reacted lambda-DNA in place of the short oligos under the same conditions described above. Lambda-DNA will react with the carboxylic acid groups on the SWNTs because of the primary amines found on the many A, G, and C nucleotides the DNA contains. Approximately 10 mg EDC, 100 μL lambda-DNA, 300 μL SWNT material, and 600 μL nanopure H_2O were mixed and left to react at room temperature for 1 hour. The relative insolubility of the SWNT material in H_2O compared to the modest solubility of DNA in pure H_2O reduces the chance for non-specific adsorption of the DNA to the SWNT

[18] 1-Ethyl-3-(3-dimethylaminopropyl)carbodiimide Hydrochloride, Sigma-Aldrich.

material. 10μL of the reaction product was deposited on UV cleaned (6 minute exposure, rinsed with nanopure H_2O, repeated twice) silicon with a native oxide layer. The silicon sample was then rinsed with nanopure H_2O and dried under a stream of dry N_2. The sample was then immediately placed into the load lock chamber of a Hitachi S-4700 SEM and pumped down to approximately 10^{-5} torr. The sample was then imaged using typical SEM parameters for non-conducting samples. Figures A.4 and A.5 show that the lambda-DNA will readily form clusters with varied attachment points to the SWNTs indicating that there may be multiple carboxylic acid binding sites at the end and along the sidewalls of the SWNTs.

A.2 Results

As expected from the dilution series of A1-A9, we see in figure A.2 that there is a steady decline in the amount of ^{32}P-labeled DNA retained at the top of the lanes. Since we expect the reaction to greatly, if not completely, reduce the DNA mobility in the polyacrylamide gel we expect to see a correlation between the mobility and SWNT concentration. Accordingly, as we reduce the concentration of SWNT material we see greater amounts of labeled DNA run down the gel lanes indicating that less DNA is being immobilized. Lane A10 demonstrates the non-specific immobilization "background" of the series since it has no SWNT material. Lane A11 and A12 visualize the cleaned and original kinase labeling products. In particular, lane A12 indicates the importance of removing the by-products of the kinase reaction by the large degree of variation in strand mobility. This variation is likely due to any number of binding events between the kinase and DNA strands that reduces the strands mobility through the porous gel.

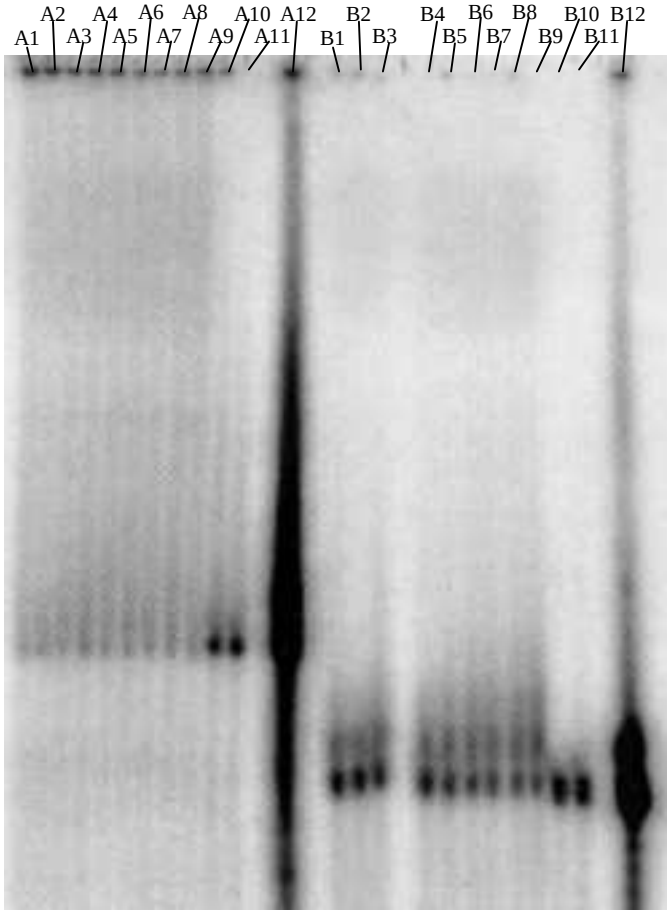

Figure A.2. Polyacrylamide gel of reactions A1-A12 and B1-B12. The material bound at the top of A1-A9 indicates successful SWNT-DNA linking.

Because the oligo 2 used in reaction B has no primary amine groups, we do not expect it to react with the SWNT material. Further, since oligo 2 is shorter than oligo 1 (17 nucleotides versus 24 nucleotides) we expect oligo 2 have higher mobility in the gel. Lanes B1-B9 demonstrate this by a large amount of DNA moving through the gel with very little being immobilized at the top of the lanes. Lane B10 was used to characterize the background immobilization, near zero in this reaction, and B11 and B12 were used to demonstrate the quality of the cleaned and original kinase products using oligo 2.

When the ratio of DNA bound at the top of a lane to the amount of DNA in the gel from reaction A is plotted, we should see the exponentially decreasing dilution series. Figure A.3 plots this ratio as calculated from the gel image in figure A.2. The general trend from A1 to A11 is clearly decreasing. The smeared-out background signal seen only in lanes A1-A9 indicates that the reaction conditions widely alter the mobility of the DNA strands. This may be caused by unwanted reactions of primary amines on the A, G, and C nucleotides of oligo 1 with the phospho-diester backbones of other strands. The many amino- and phosphate groups available for this reaction could explain the wide variation in the mobility of the products.

A.3 Conclusions

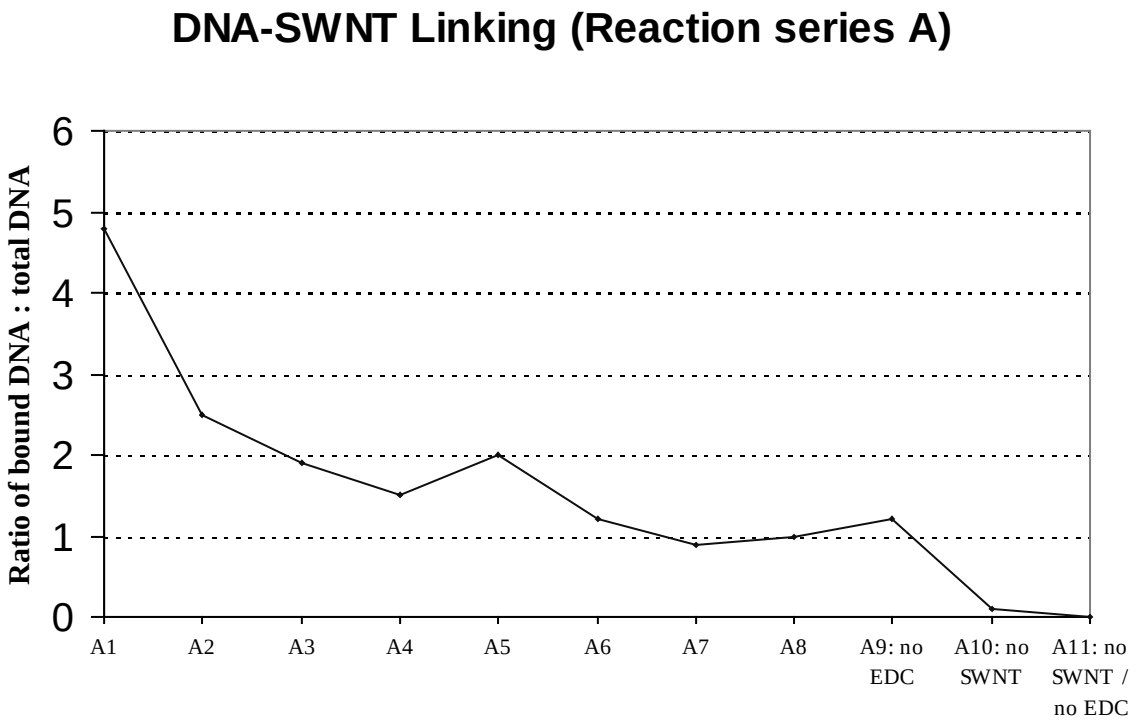

Figure A.3. Plot of the ratio of DNA immobilized at the top of a well to the total DNA found in the well from reaction A. Reaction B appears to have little bound DNA.

We have shown that SWNT material can be functionalized with modified DNA. This form of chemical modification is the first step toward implementing a DNA-guided self-

assembling process capable of directing the placement of SWNTs. The work presented here in concert with work performed by other groups in DNA metallization [Richter, 2001] and self-assembly [Mbindyo, 2001; Gracias, 2000] presents the start of a compelling argument for the feasibility of self-assembled molecular-scale electronic systems.

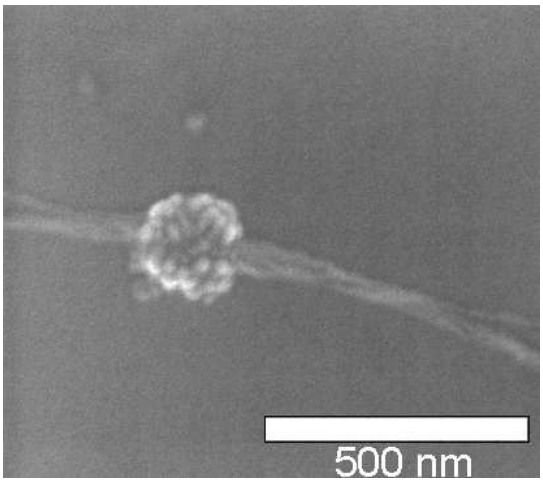

Figure A.4. Lambda-DNA cluster attached to defect sites and ends of a SWNT bundle.

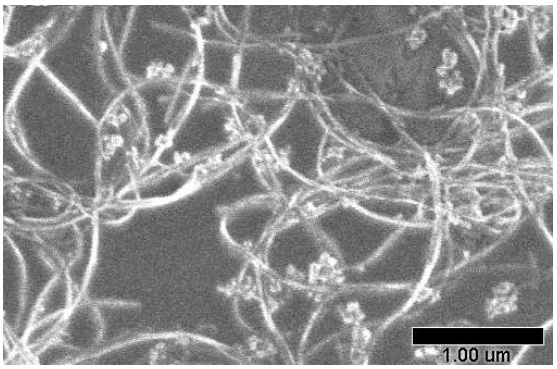

Figure A.5. Lambda-DNA clusters on SWNT bundles.

Appendix B. Source Code

All source code created in the support of this dissertation can be found on the accompanying compact disc, and on the dissemination page at www.cs.unc.edu/nano.

Appendix C. DAMP instruction set implementation

The following tables describe the implementation of the instructions built into the behavioral simulation described in chapter 9.

To save space, some conventions will be used to indicate how the processor is signaled. Each instruction will have two columns, the left column will indicate which signals from the control registers will be asserted. The second column will indicate how many cycles will be clocked using these values. For example:

ERC	1
ESA, ESR, EESD, EEC	16

This means that the "enable reset carry-bit" (ERC) is set in the control registers (last bit in S7) for one clock cycle, and that the enable - "shift accumulator" (ESA), "shift register" (ESR), "enable set D-bit" (EESD), and "enable carry-bit" (EEC) are set in the control register for 16 clock cycles. The "enable" signals are set in such a way that they are cleared if any others are set. That is, the second line in the box above removes the "ERC" signal simply because it does not appear. Multiplexers, on the other hand do not to be re-specified on each line (i.e. they retain their previous setting.)

There are several control register bits that select the output of multiplexers. These values are cumbersome to read and understand without using symbols. Since each multiplexer has only a single output, it is useful to describe each by the name given them in the implementation figures above. For example, LC2 can select the output from R1, R2, R3, or R4. Therefore, the notation "LC2 = R3" signifies the bits that would be set in the control register to make LC2 select the output of R3, or LC2a=1 and LC2b=0. This notation will be used with each multiplexer including LC1, LC2, AC, RC0, RC1, RC2, RC3, RC4, WC1, and WC2.

In the event that an argument to the instruction is needed to select the processor signaling (e.g. choosing a register to copy into the accumulator), the notation "if(argument-condition, then-clause)" where the argument-condition is a Boolean condition that if satisfied by a particular argument will cause the then-clause to be used to generate processor signals. For example, since the selection of a register output is common, "select(RX)" will be used to denote the following:

if(RX=0, LC1=R0)
if(RX=1 .. 4, LC1=LC2)
if(RX=1 .. 4, LC2=RX)
	(any number)

This means that the LC1 and LC2 multiplexers are set to present the output from any of the registers, R0 through R4, to one of the inputs (LC1) of the AC multiplexer. This notation will be used frequently with register-register operations.

It is occasionally necessary to select the bits from an immediate value provided as an argument to an instruction. The notation "imm[i]" will be used to indicate the i^{th} bit from the immediate value with the least significant bit starting at i=0.

Please refer to section 8.1 for the details about the semantics for each instruction below. Each of these instructions was implemented as shown and verified using the behavioral simulation described in chapter 9.

ADD(RX)	
ERC	1
select(RX) ESA, ESR, EESD, EEC	16
ESA	1

ADDC(RX)	
select(RX) AC=S ESA, ESR, EESD, EEC	16
ESA	1

ADDI(constant)	
AC=S ERC	1
LC1=LC2b LC2b=imm[i] i++ ESA, EESD, EEC	16
ESA	1

ANDI(constant)	

LC1=LC2b
LC2b=0
if(imm[i]=0, AC=ACC)
if(imm[i]=1, AC=LC1)
i++
ESA
	16

ASR(N)	
LC1=LC2b LC2b=0 EESD, ERC	1
AC=S ESA	N

CLEARB	
LC1=LC2b LC2b=0 EEB	1

CLEARC, CLEARD	
CLEARC: ERC CLEARD: ERD	1

CMP	
ERD, ERC	1
LC1=LC2b LC2b=0 AC=ACC ESA, EESD	16

CMPI(constant)	
AC=ACC LC1=LC2b LC2b=0 ESC, ERD	1

if(imm[i]=1, LC2b=0)
if(imm[i]=0, LC2b=1)
i++
ESA, ESC, EESD
	16

CMPI8(constant)	
AC=ACC LC1=LC2b LC2b=0 ESC, ERD	1

if(imm[i]=1, LC2b=0)
if(imm[i]=0, LC2b=1)
i++
ESA, ESC, EESD
	8

COPY(RX)	
AC=LC1 select(RX) ESA, ESR	16

COPYH(RX)	
AC=LC1 LC1=LC2b LC2b=0 ESA, ESR	8
ESA	1
select(RX) ESA, ESR, EESD	8
AC=S ESA	8

COPYL(RX)	
AC=LC1 select(RX) ESA, ESR	8
ESA	8
ESR	8

COST(RX1, RX2)	
AC=LC1 select(RX1) RC(X2)=ACC ESA, ESR	16
RC(X2)=RX2	0

CSR(N)	
AC=ACC ESA	N

DEC	
AC=S ERC	1
LC1=LC2b LC2b=1 ESA, EESD, EEC	16
ESA	1

GRAB(N)	
AC=ACC LC1=LC2b LC2b=0 ESA	N
ESA, EESD	1
ESA	15 - N

INC	
ESC	1
AC=S LC1=LC2b LC2b=0 ESA, EESD, EEC	16
ESA	1

LOAD(constant)	
AC=LC1 LC1=LC2b LC2b=imm[15] ESA	1
LC2b=imm[i] i++ ESA	16

LSR(N)	
AC=LC1 LC1=LC2b LC2b=0 ESA	N

LSRC(N)	
AC=C ESA	N

MCOPY(constant, RX)	

select(RX)
if(imm[i]=0, AC=ACC)
if(imm[i]=1, AC=LC1)
ESA, ESR
	16

NOT	
AC=S LC1=LC2b LC2b=0 ESC	1
ESA, ESC, EESD	17
ERC	1

ORI(constant)	

LC1=LC2b
LC2b=1
if(imm[i]=0, AC=ACC)
if(imm[i]=1, AC=LC1)
ESA
	16

RANDOM	
EL	1

RESUME	
C1=0, C2=0, C3=0	0

RINGOFF, RINGON	
RINGOFF: LC1 = LC2b RINGON: LC1 = !LC1	0

SETGE, SETLT, SSETNGE, SETNLT	
ETNGE, SETLT: ERC ETGE, SETNLT: ESC	1

SETB	
LC1=LC2b LC2b=1 EEB	1

SETCREG(constant)	
C1=imm[2] C2=imm[1] C3=imm[0]	-

SETC	
ESC	1

SETSREG(constant)	
SREG(C)=imm	-

SIGN	
AC=ACC LC1=LC2b LC2b=0 ESA	15
ESA, EESD	1

STORE(RX)	
AC=ACC RC(X)=ACC ESA, ESR	16
RC(X)=RX	0

STOREH(RX)	
AC=ACC ESR	8
RC(X)=ACC ESA, ESR	8
RC(X)=RX ESA	8

STOREL(RX)	
AC=ACC RC(X)=ACC ESA, ESR	8
RC(X)=RX ESA, ESR	8

WAIT*	
WAITB: WC1=B, WC2=WC1 WAITNB: WC1=B, WC2=!WC1 WAITC: WC1=C, WC2=WC1 WAITNC: WC1=C, WC2=!WC1 WAITD: WC1=D, WC2=WC1 WAITND: WC1=D, WC2=!WC1 WAITS: WC1=S, WC2=WC1 WAITNS: WC1=S, WC2=!WC1 WAITGE, WAITNLT: WC1=C, WC2=WC1 WAITNGE, WAITLT: WC1=C, WC2=!WC1	0

XOR(RX)	
AC=S select(RX) ERC	1
ESA, ESR, EESD, ERC	16
ESA	1

BIBLIOGRAPHY

[Adleman, 1994] Adleman L. M., "Molecular Computation of Solutions to Combinatorial Problems", Science, 266, 1021-1024, 1994.

[Alien, 2003] Alien Technology, Inc., Morgan Hill, CA, http://www.alientechnology.com, 2003.

[Audet, 2000] Audet C., Dennis J. E., “Pattern search algorithms for mixed variable programming", *SIAM Journal on Optimization*, 11 (3): 573-594, November 2000.

[Batalia, 2002] Batalia M. A., Protozanova E., Macgregor R. B., Erie D. A. , “Self-assembly of frayed wires and frayed-wire networks: Nanoconstruction with multistranded DNA”, *Nano Letters*, 2 (4): 269-274, April 2002.

[Biham, 1997] Biham E., "A Fast New DES Implementation in Software", *Technical Report CS0891*, Technion - Israeli Institute of Technology, 1997.

[BlueGene, 2002] Adiga N. R., Almasi G., Almasi G. S., Aridor Y., Barik R., Beece D., Bellofatto R., Bhanot G., Bickford R., Blumrich M., Bright A. A., Brunheroto J., Caşcaval C., Castaños J., Chan W., Ceze L., Coteus P., Chatterjee S., Chen D., Chiu G., Cipolla T. M., Crumley P., Desai K. M., Deutsch A., Domany T., Dombrowa M. B., Donath W., Eleftheriou M., Erway C., Esch J., Fitch B., Gagliano J., Gara A., Garg R., Germain R., Giampapa M. E., Gopalsamy B., Gunnels J., Gupta M., Gustavson F., Hall S., Haring R. A., Heidel D., Heidelberger P., Herger L. M., Hoenicke D., Jackson R. D., Jamal-Eddine T., Kopcsay G. V., Krevat E., Kurhekar M. P., Lanzetta A. P., Lieber D., Liu L. K., Lu M., Mendell M., Misra A., Moatti Y., Mok L., Moreira J. E., Nathanson B. J., Newton M., Ohmacht M., Oliner A., Pandit V., Pudota R. B., Rand R., Regan R., Rubin B., Ruehli A., Rus S., Sahoo R. K., Sanomiya A., Schenfeld E., Sharma M., Shmueli E., Singh S., Song P., Srinivasan V., Steinmacher-Burow B. D., Strauss K., Surovic C., Swetz R., Takken T., Tremaine R. B., Tsao M., Umamaheshwaran A. R., Verma P., Vranas P., Ward T. J. C., Wazlowski M., Barrett W., Engel C., Drehmel B., Hilgart B., Hill D., Kasemkhani F., Krolak D., Li C. T., Liebsch T., Marcella J., Muff A., Okomo A., Rouse M., Schram A., Tubbs M., Ulsh G., Wait C., Wittrup J., Bae M., Dockser K., Kissel L., Seager M. K., Vetter J. S., Yates K., "An Overview of the BlueGene/L Supercomputer", IEEE SC2002, 2002.

[Braich, 2002] Braich R. S., Chelyapov N., Johnson C., Rothemund P. W. K., Adleman L., "Solution of a 20-variable 3-SAT problem on a DNA computer", *Science*, 296 (5567): 499-502, April 2002.

[Braun, 1998] Braun E., Eichen Y., Sivan U., Ben-Yoseph G., “DNA-templated assembly and electrode attachment of a conduc silver wire”, *Nature*, 391 (6669): 775-778, February 1998.

[Brown, 2002] Brown K. R., Lidar D. A., Whaley K. B., "Quantum computing with quantum dots on quantum linear supports", *Physical Review A*, 65 (1): no. 012307, January 2002.

[Buzaneva, 2002] Buzaneva E., Karlash A., Yakovkin K., Shtogun Y., Putselyk S., Zherebetskiy D., Gorchinskiy A., Popova G., Prilutska S., Matyshevska O., Prilutskyy Y., Lytvyn P., Scharff P., Eklund P., "DNA nanotechnology of carbon nanotube cells: physico-chemical models of self-organization and properties", *Material Science Engineering C*, 19, 41, 2002.

[Clark, 2001] Clark T. D., Tien J., Duffy D. C., Paul K. E., Whitesides G. M., "Self-Assembly of 10-µm-Sized Objects into Ordered Three-Dimensional Arrays", Journal of the American Chemical Society, 123 (31): 7677-7682, July 2001.

[Clark, 2002] Clark T. D., Ferrigno R., Tien J., Paul K. E., Whitesides G. M., "Template-Directed Self-Assembly of 10-µm-Sized Hexagonal Plates", Journal of the American Chemical Society, 124 (19): 5419-5426, April 2002.

[Collier, 1999] Collier C. P., Wong E. W., Belohradsky M., Raymo F. M., Stoddart J. F., Kuekes P. J., Williams R. S., Heath J. R., "Electronically configurable molecular-based logic gates", *Science*, 285 (5426): 391-394, July 1999.

[Cui, 2000] Cui Y., Duan X. F., Hu J.T., Lieber C. M., "Doping and electrical transport in silicon nanowires", *Journal of Physical Chemistry B*, 104 (22): 5213-5216, June 2000.

[Dongarra, 2002] Dongarra J., "Notes on the Earth Simulator", Computer Science Department, University of Tennessee, 2002.

[Dwyer, 2002] Dwyer C., Taylor R., Vicci L., "Transport Simulation of a Nanoscale Silicon Rod Field-Effect Transistor", *Proceedings of the 2nd IEEE Conference on Nanotechnology*, Arlington, VA, 601-604, September 2002.

[Dwyer, 2002b] Dwyer C., Guthold M., Falvo M., Washburn S., Superfine R., Erie D., "DNA-functionalized single-walled carbon nanotubes", *Nanotechnology*, 13, 601-4, 2002.

[Dwyer, 2003] Dwyer C., Taylor R., Vicci L., "Performance Simulation of Nanoscale Silicon Rod Field-Effect Transistor Logic", *IEEE Transactions on Nanotechnology*, in press, 2003.

[Dujardin, 2001] Dujardin E., Hsin L.B., Wang C.R.C., Mann S., "DNA-driven self-assembly of gold nanorods", *Chemical Communications*, (14): 1264-1265, 2001.

[Feldmeier, 1989] Feldmeier D. C., "A High-Speed Software DES Implementation", Computer Communications Research Group, 1989.

[Fieldsend, 2002] Fieldsend J.E., Singh S., "A Multi-Objective Algorithm based upon Particle Swarm Optimisation, an Efficient Data Structure and Turbulence", Proceedings of the 2002 U.K. Workshop on Computational Intelligence, 37-44, Sept. 2002.

[FEMLAB, 2003] ComSol, Inc. http://www.femlab.com, 2003.

[Goldstein, 2002] Goldstein S.C., Rosewater D., "Digital Logic Using Molecular Electronics", IEEE International Solid-State Circuits Conference, 2002.

[Gracias, 2000] Gracias D. H., Tien J., Breen T. L., Hsu C., and Whitesides G. M., "Forming electrical networks in three dimensions by self-assembly", Science 289, 1170-1172, 2000.

[Gudiksen, 2002] Gudiksen M. S., Lauhon L. J., Wang J., Smith D. C., Lieber C. M., "Growth of nanowire superlattice structures for nanoscale photonics and electronics", *Nature*, 415, 617-620.

[Guo, 1998] Guo Z. J., Sadler P. J., Tsang S. C., "Immobilization and visualization of DNA and proteins on carbon nanotubes", *Advanced Materials*, 10, no. 9, pp. 701-3, June 18, 1998.

[Head, 2001] Head T., "Biomolecular Realizations of a Parallel Architecture for Solving Combinatorial Problems", *New Generation Computing*, 19, 302-312, 2001.

[Heath, 1998] Heath J. R., Kuekes P. J., Snider G. S., Williams R. S., "A defect-tolerant computer architecture: Opportunities for nanotechnology", *Science*, 280 (5370): 1716-1721, June 1998.

[Heath, 2000] Heath J. R., "Wires, switches, and wiring. A route toward a chemically assembled electronic nanocomputer", *Pure and Applied Chemistry*, 72 (1-2): 11-20, 2000.

[Hendrickson, 1970] Hendrickson J. B., Cram D. J., Hammond G. S., ***Inorganic Chemistry***, 3rd edition, edited by Z. Z. Hugus (McGraw-Hill, New York, 1970), Vol. 1, Chap. 12, p.468-471.

[Hough, 2000] Hough P. D., Kolda T. G., Torczon V. J., "Asynchronous Parallel Pattern Search for Nonlinear Optimization", *SIAM Journal on Scientific Computing*, 23 (1): 134-156, 2000.

[IES, 2001] IES Inc., COULOMB, 2001. (www.integratedsoft.com)

[Intel, 2003] Intel Corporation, www.intel.com, 2003.

[Jang, 1998] Jang S. L., Liu S. S., "An analytical surrounding gate MOSFET model", *Solid-State Electronics*, 42(5): 721-726, 1998.

[Kedem, 1999] Kedem G., Ishihara Y., "Brute Force Attack on UNIX Passwords with SIMD Computer", *Proceedings of the 8th USENIX Security Symposium*, 1999.

[Kokkolaras, 2000] Kokkolaras M., Audet C., Dennis J. E., "Mixed variable optimization of the number and composition of heat intercepts in a thermal insulation system", *TR00-21*, Department of Computational & Applied Mathematics, Rice University, Houston, 2000.

[Kovtyukhova, 2002] Kovtyukhova N. I., Mallouk T. E., "Nanowires as building blocks for self-assembling logic and memory circuits", *Chemistry-A European Journal*, 8 (19): 4355-4363, October 2002.

[Lauhon, 2002] Luahon L. J., Gudiksen M. S., Wang D., Lieber C. M., "Epitaxial core–shell and core–multishell nanowire heterostructures", *Nature*, 420, 57-61, November 2002.

[Lebowitz, 2002] Lebowitz J., Lewis M. S., Schuck P., "Modern analytical ultracentrifugation in protein science: A tutorial review", *Protein Science*, 1 (9): 2067-2079, September 2002.

[Lew, 2002] Lew K., Reuther C., Carim A. H., Redwing J. M., Martin B. R., "Template-directed vapor–liquid–solid growth of silicon nanowires", *Journal of Vacuum Science Technology* B, 20(1): 389-392, 2002.

[Lewin, 1997] Lewin B., ***Genes VI***, Oxford University Press, New York, 1997.

[Lieberman , 2002] Lieberman M., Chellamma S., Varughese B., Wang Y. L., Lent C., Bernstein G. H., Snider G., Peiris F. C., "Molecular Electronics II", *Annals of the New York Academy of Sciences*, 960: 225-239, 2002.

[Liu, 1998] Liu J., Rinzler A. G., Dai H., Hafner J. H., Bradley R. K., Boul P. J., Lu A., Iverson T., Shelimov K., Huffman C. B., Rodriguez-Macias F., Shon Y., Lee T. R., Colbert D. T., and Smalley R. E., "Fullerene Pipes", *Science*, 280, 1253-1256, 1998.

[Loakes, 2001] Loakes D., "The applications of universal DNA base analogues", *Nucleic Acids Research*, 29:2437-2447, 2001.

[MacDonald, 1992] MacDonald, Neil B., "An Overview of SIMD Parallel Systems: AMT DAP, Thinking Machines CM-200, & MasPar MP-1", Workshop on Parallel Computing, Quaid-i-Azam University, Islamabad, Pakistan, 26th–30th April 1992.

[Martin, 1996] Martin C. R., "Membrane-based synthesis of nanomaterials", *Chemistry of Materials*, 8 (8): 1739-1746, August 1996.

[Martin, 2002] Martin B. R., St. Angelo S. K., Mallouk T. E., "Interactions between suspended nanowires and patterned surfaces", *Advanced Functional Materials*, 12 (11-12): 759-765, December 2002.

[Melosh, 2003] Melosh N. A., A. Boukai, Diana F., Geradot B., Badolato A., Petrof P. M., Heath J. R., "Ultrahigh-Density Nanowire Lattices and Circuits", *Science Express Online*, March 2003.

[Mbindyo, 2001] Mbindyo J. K. N., Reiss B. D., Martin B. R., Keating C. D., Natan M. J., Mallouk T. E., "DNA-directed assembly of gold nanowires on complementary surfaces", *Advanced Materials*, 13 (4): 249-54, February 2001.

[Miyano, 1992] Miyano S., Hirose M., Masuoka F., "Numerical-Analysis of a Cylindrical Thin-Pillar Transistor (CYNTHIA)", *IEEE Transactions on Electron Devices*, 39(8): 1876-1881, 1992.

[Pena, 2002] Pena S. R. N., Raina S., Goodrich G. P., Fedoroff N. V., Keating C. D., "Hybridization and enzymatic extension of Au nanoparticle-bound oligonucleotides", *Journal Of The American Chemical Society*, 124 (25): 7314-7323, June 2002.

[Pinto, 1988] Pinto M. R., Rafferty C. S., Dutton R. W., Eldredge M. J., Yu Z., et al., PISCES-IIB 9009, Win32 port by J. Faricelli. See http://www-tcad.stanford.edu/tcad/programs/ftpable.html

[Qiagen, 2003] Qiagen, Inc. website, www.qiagen.com.

[Quarles, 1991] Quarles T., et al., SPICE3f5. See http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE.

[Richter, 2001] Richter J., Mertig M., Pompe W., Monch I., Schackert H. K., "Construction of highly conductive nanowires on a DNA template", *Applied Physics Letters*, 78 (4): 536-538, January 2001.

[Rollins, 1988] Rollins G. J., Choma J., "Mixed-Mode PISCES-SPICE Coupled Circuit and Device Solver", *IEEE Transactions on Computer Aided Design*, 7(8): 862-867, 1988.

[Roweis, 1998] Roweis S., Winfree E., Burgoyne R., Chelyapov N. V., Goodman M.F., Rothemund P.W.K., Adleman L.M., "A sticker-based model for DNA computation", *Journal of Computational Biology*, 5 (4): 615-629, 1998.

[Sano, 2002] Sano N., Hiroki A., Matsuzawa K., "Device Modeling and Simulations Toward Sub-10 nm Semiconductor Devices", *IEEE Transactions on Nanotechnology*, 1(1): 63-71, 2002.

[Schmid, 1996] Schmid A. W., Kessler T. J., Papernov S., Barone J., "Low-Surface-Energy Photoresist as a Medium for Optical Replication", LLE Review Quarterly Report, University of Rochester, Laboratory for Laser Energetics, vol. 66, January – March, 1996.

[Seeman, 2001] Seeman N. C., Mao C. D., LaBean T., Reif J. H., "XOR operations by algorithmic assembly of DNA tiles", *Biophysical Journal*, 80 (1): 45 Part 2, January 2001.

[SETI@home, 2003] http://setiathome.ssl.berkeley.edu/totals.html, 2003.

[Soto, 2002] Soto C. M., Srinivasan A., Ratna B. R., "Controlled assembly of mesoscale structures using DNA as molecular bridges", *Journal of the American Chemical* Society, 124 (29): 8508-8509, July 2002.

[Srinivasan, 2001] Srinivasan U., Liepmann D., Howe R. T., "Microstructure to substrate self-assembly using capillary forces", Journal *of Microelectromechanical Systems*, 10 (1): 17-24, March 2001.

[Takato, 1988] Takato H., "High-performance CMOS surrounding gate transistor (SGT) for ultra high density LSIS", *IEDM Technology Digest*, 222, 1988.

[Takato, 1992] Takato H., Sunouchi K., Okabe N., Nitayama A., Hieda K., Horiguchi F., Masuoka F., "Impact of Surrounding Gate Transistor (SGT) for Ultra-High-Density LSIS", *IEEE Transactions on Electron Devices*, 38, no. 3, 573-578, 1991.

[Taylor, 2002] Taylor R., et al., MASSMESH code, UNC-CH, 2002.

[Thompson, 1998] Thompson S., Packan P., Bohr M., "MOS Scaling: Transistor Challenges for the 21st Centry", *Intel Technology Journal Q3'98*, Intel Corporation, 1998.

[Top500, 2003] www.top500.org, Top 500 supercomputer sites, 2003.

[Uma, 2000] Uma S., McConnell A. D., Asheghi M., Kurabayashi K., Goodson K. E., "Temperature dependent thermal conductivity of undoped polycrystalline silicon layers", *Fourteenth Symposium on Thermophysical Properties*, Boulder, Colorado, June 2000.

[Warren, 2002] Warren M. S., Weigle E. H., Feng W. C., "High-Density Computing: A 240-Processor Beowulf in One Cubic Meter", IEEE SC2002, 2002.

[Watwe, 1997] Watwe A. A., Bar-Cohen A., "Enhancement of the Pool Boiling Critical Heat Flux Using a Binary Mixture of Dielectric Liquids," Proceedings of the Engineering Foundation Conference on Convective Flow and Pool Boiling, Irsee, Germany, 1997.

[Winfree, 2000] Winfree E., "Algorithmic self-assembly of DNA: Theoretical motivations and 2D assembly experiments", *Journal of Biomolecular Structure & Dynamics*, 263-270, Sp. Iss. S2. 2000.

[Wong, 1998] Wong S. S., Woolley A. T., Joselevich E., Cheung C. L., and Lieber C. M., *J. Am. Chem. Soc.* 120, 8557-8558, 1998.

[Yan, 2002] Yan H., Zhang X. P., Shen Z. Y., Seeman N. C., "A robust DNA mechanical device controlled by hybridization topology", *Nature*, 415 (6867): 62-65, January 2002.

[Zimmermann, 1997] Zimmermann R., Fichtner W., “Low-Power Logic Styles: CMOS Versus Pass-Transistor Logic”, *IEEE Journal of Solid-State Circuits,* vol. 32, no. 7, 1997.

[Zitzler, 2000] Zitzler E., Deb K., Thiele L., “Comparison of Multiobjective Evolutionary Algorithms: Empirical Results”, *Evolutionary Computation*, 8(2): 173-195, 2000.