Logic Simulation on a Cellular Automata Machine

by

Ruben Agin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Electrical Engineering

and

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

© Ruben Agin, MCMXCVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

OCT 29 1997

LIBRARIES

Author ....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by ....
Norman H. Margolus
Research Affiliate
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Logic Simulation on a Cellular Automata Machine

by

Ruben Agin

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Electrical Engineering
and
Master of Engineering in Computer Science and Engineering

Abstract

CAM-8, a cellular automata machine, is a SIMD array of virtual processors which are time-shared among parallel, uniform lookup-tables (LUT) for updating and utilizes framebuffer-like bit-planes with variable addressing for data transport. The CAM-8 lends itself well to gate array simulations of sequential logic, and a method has been devised for doing this: Spacetime circuitry is utilized as a technique for optimizing these simulations by scheduling circuit operations to coincide with the time-slices of the LUT updates, so that combinational latency is reduced to one update of the entire array. Many circuits, including an 8-bit microprocessor, have already been manually laid out and simulated. A logic synthesis program has been designed and implemented for automatically laying out spacetime circuits. Given an initial netlist and an optional state transition table, generated using standard logic CAD tools, along with topology information about the CAM-8 simulator, the program generates a bit pattern representing the circuit, an appropriate lookup-table, and boundary scan information appropriate for STD simulation and verification. Results of the synthesis and simulation of various benchmark circuits are presented.

Thesis Supervisor: Norman H. Margolus

Title: Research Affiliate

Acknowledgments

I would like to thank Norman Margolus for taking time out of his busy schedule to supervise this thesis. He is the one who suggested this project and originated most of the ideas in here, which I simply took the time to implement. I am grateful to him for all the time he took to explain things to me and to give me a new perspective on how things should work. I also thank him for all the time he took to do favors for me and to take me out to dinner with his lovely wife, Carol. He provided a second family to me at MIT, which helped me get by.

I have enjoyed my stay here at MIT, due in large part to the great experience I had as a UROP in the Information Mechanics Group. I was introduced to concepts that I had never before thought about and probably would never have thought about had I not been exposed to them. I believe the knowledge I have gained will affect my beliefs for the rest of my life. I will miss all the great people I had the pleasure of knowing in the group. To Mike Biafore, I am indebted to his efforts at finding me a summer job, and for being a good source for career advice. To Mark Smith, a virtual fountain of knowlege, I am grateful for helping me out on many of my problem sets at MIT, and explaining many of the things in physics and math which I did not understand. To Raissa D'Souza, I am thankful for providing a cheerful environment in which to work, for helping me out on my project, providing useful feedback on my talk, and providing useful advice. To Harris Gilliam, the great "hacker one", I am grateful for much useful programming help, and for providing comic relief and friendship. I am thankful to all the other members of the IM Group who I have had the pleasure of knowing: Sasha Woods, Jason Quick, Rebecca Frankel, Todd Kaloudis, Dan Risacher, Surya Ganguli, and Pierluigi Pierini. Last but not least, I'd like to thank Tommaso Toffoli for hiring me and having faith in my abilities.

A special thanks to my inspiration, my mother. Her unwavering love and support in the face of difficult obstacles made it possible for me to make it through the days at MIT. She provided the foundation for my life and career. I'd also like to thank my lifelong best friend, Glen Hurst, for keeping me grounded and sane by never letting

me forget about great times and old friends.

I'd like to thank Russ Tessier for providing the useful pointers to me on logic synthesis which got the last part of my project going. I am also grateful to Tom Knight for his generous financial support of this project and for useful feedback on my proposal and talk. Support for this work comes from DARPA contract DABT63-95-C-0130 as part of the MIT AI Laboratory's Reversible Computing Project.

Contents

1 Introduction **12**

1.1 Cellular Automata 12

1.1.1 Definition 13

1.1.2 History 13

1.1.3 Classifications 14

1.2 CAM-8 16

1.2.1 Data Update 17

1.2.2 Data Transport 17

1.2.3 Topology 18

1.2.4 Programming 18

1.2.5 Visualization and Other Features 19

1.3 Logic Simulation and Emulation 19

1.3.1 Logic Synthesis 20

1.3.2 Programmable Hardware 20

1.4 Thesis Objective and Organization 21

2 Parallel Gate Array Emulation **23**

2.1 Motivation 23

2.1.1 Theoretical Equivalence 23

2.1.2 An Alternative Architecture for Nanoscale Computation . . . 24

2.1.3 Gate-Array Like Quality 25

2.1.4 Architectural Proximity 26

2.2 PGA Architecture 28

2.3 Initial Implementation of a Logic Simulator 30
2.3.1 Signals 31
2.3.2 Configuration 32
2.3.3 Dynamics 35
2.4 Some Logic Simulation Examples 36

3 Spacetime Circuitry **39**
3.1 Spacetime Computation 39
3.1.1 Spatial Pipelining 40
3.1.2 Temporal Pipelining 41
3.2 Spacetime Circuitry 42
3.2.1 Implementation 47
3.2.2 Finite State Machines 49
3.3 A Brief History 50

4 Simulation of a Microprocessor **51**
4.1 The Virtual Gate Array Emulator 51
4.1.1 The RLSC Node Model 51
4.1.2 The Space 57
4.1.3 Transport and Interaction 57
4.1.4 Input/Output 58
4.2 Implementation 59
4.2.1 Control 62
4.2.2 PCMA 64
4.2.3 Register File 64
4.2.4 Putting It All Together 64
4.3 High-Level Implementation 67
4.4 Discussion 68

5 Logic Synthesis **69**
5.1 CAD/CAM-8 69

5.2 The Synthesis Environment 71
5.2.1 Technology Independent Logic Synthesis 71
5.2.2 The Cellular Automata Logic Synthesis (CALS) Program . . . 75

6 Results **83**
6.1 Technology Independent Synthesis 83
6.2 Layout Data 84
6.2.1 Placement 84
6.2.2 Routing 87

7 Conclusions and Future Research **93**
7.1 Conclusions 93
7.1.1 Spacetime Circuitry 93
7.1.2 Synthesis 94
7.2 Future Research 95

A CAM-8 Logic Simulator Code **98**

B Logic Synthesis Scripts and Code **137**

List of Figures

1-1 State transitions of two different transition functions 14
1-2 Illustration of sites, layers and kicks in CAM-8 16
1-3 LUT in CAM-8 is time-shared over a group of sites called a sector . . 17

2-1 XC6200 interconnect with nearest-neighbor wires and length-4 flyovers 29
2-2 XC6200 cell routing 29
2-3 XC6200 cell function unit 29
2-4 CAM-8 logic simulator summary 31
2-5 Routing, logic, storage cell 32
2-6 Routing cell 33
2-7 Logic cell 34
2-8 Storage cell 34
2-9 Illustration of a step for a parallel gate array simulation 35
2-10 A simple PLA on CAM-8 36
2-11 More complicated CAM-8 circuit 37
2-12 Microprocessor on CAM-8 37

3-1 A physical processor assuming the role of different virtual processors . 41
3-2 The levels of a 2-D circuit laid out on a spatial grid without spatial pipelining 43
3-3 Illustration of a step for a virtual gate array simulation 44
3-4 Comparison of normal and spacetime circuit simulation 45
3-5 A 3-bit spacetime adder 47
3-6 Configurations of the spacetime adder 47

3-7 The division of modules in the y-strip topology 48
3-8 An FSM 49
3-9 The scan of a spacetime circuit 49

4-1 The RLSC Node Model 52
4-2 A circuit format representation of a logic cell 56
4-3 In screen coordinates, the logic is laid out in the x dimension and data is pipelined with the scan through the t dimension 57
4-4 A schematic of the ALU from the original VLSI project 60
4-5 The CA layout of the ALU 60
4-6 The embedding of the multiplexor in the cellular space 61
4-7 A close-up of one of the 2-1 8-bit multiplexors in the actual CAM-8 circuit layout 61
4-8 A schematic of the control 62
4-9 The CA layout of the control 63
4-10 Generic embedding of a PLA in cellular space 63
4-11 A schematic of the PCMA 64
4-12 The CA layout of the PCMA 65
4-13 A schematic of the register file 65
4-14 The CA layout of the register file 66
4-15 A schematic of the sexium 66
4-16 The CA layout of the sexium 67

5-1 Logic synthesis for the CAM-8 simulator 72
5-2 Node and its net in a netlist 72
5-3 A BLIF netlist 73
5-4 CALS cell model in d dimensions 75
5-5 Assignment of paths to edges illustrating the scheduling constraint in the placement phase 77
5-6 Four compatible cell configurations in a 2-D gate array 80

6-1 The number of nodes versus the number of levels after placement . . 86

6-2 The number of levels versus the depth of the circuit 86
6-3 Spatial/temporal distance versus the number of nodes 86
6-4 Number of routing iterations versus number of nodes 89
6-5 Number of nodes versus number of used cells 89
6-6 Fraction of space used versus the number of cells 90
6-7 Fraction of space used versus the number of nodes 90
6-8 Speed of the circuit versus the number of nodes 91
6-9 Speed of the circuit versus the number of cells 91
6-10 The number of configurations versus the number of cells 92

List of Tables

4.1 A possible routing table. In this case, outgoing east(00) comes from north (01), north (01) comes from west (10), etc. 53

4.2 Truth table for XOR 54

4.3 All possible 2-1 totalistic logic functions with the rows enumerated by function and the columns by the sum of the inputs and the cells giving the output for that sum 54

4.4 A possible logic site LUT shown in *table* format. Takes the AND of 1 and 2 and outputs to 2 and 3 and takes input 2 and routes to 1. The circuit diagram for this is shown in Figure 4-3. Note that there are three directions and hence three different layers used for data transport. The kick of each layer proceeds at the end of each subscan in the directions indicated at the top of the figure 55

6.1 MCNC Placement Info 85

6.2 MCNC Routing and Configuration Info 88

Chapter 1

Introduction

Cellular automata (CA) provide a good architecture on which to emulate logic circuits. CA are arrays of identical, locally interconnected processors, called cells. Thus, they are essentially fixed gate arrays. CA are also well-suited to the constraints imposed at the quantum level. Semiconductor industry experts are already predicting the need for CA-like architectures at the nanometer scale[20].

CAM-8 is a scalable machine optimized for fast, efficient, and flexible simulation of CA. Therefore, CAM-8 is ideal for use as a logic simulation engine. CAM-8 can be programmed to simulate a gate array with just about any architecture. Thus, logic designs can be synthesized for these gate arrays and simulated on CAM-8. Hence, it provides an extremely general-purpose test engine for gate-array architects, digital logic designers, and other scientists and engineers exploring new types of gate arrays and logic circuits. CAM-8 simulates CA in a virtual manner. This provides a way to do fast and efficient logic simulation using a technique called spacetime circuitry.

1.1 Cellular Automata

It is hard to pin down the terminology of CA exactly and it is usually not worth the time or the effort because the definitions vary so much in any case. The definition of CA is given here for completeness and for reference. This is the definition in the paper by Arthur Burk, detailing von Neumann's Self-Reproducing Automata. In addition,

I use this definition and terminology in order to more easily explain how the CAM-8 works, why it works that way, and how it can be used to implement logic simulations efficiently. I only use the CA definition and terminology to introduce the CAM-8. The rest of the thesis uses the terminology of the CAM-8 in conjunction with gate arrays.

1.1.1 Definition

A cellular automaton is an n-dimensional Euclidean space of points, called *cells*, all of which can assume discrete states that get updated according to a *transition function*[9]. All cells are update synchronously and in parallel. In addition, there is a *neighborhood relation* which defines, for each cell, a set of cells in the space, called the *neighborhood.* The transition function gives the state of the cell at time t+1 as a function of its own state and the state of its neighbors at time t. The state of the cellular automaton is the set of states of all its cells.

In most CA, the neighborhood relation and transition function are uniform over the whole space. However, in the most generic case, the neighborhood relation and transition function of a cell need only be computable from its coordinates so that the state of the CA at time t+1 is computable from its state at time t.

1.1.2 History

The notion of a CA was originally conceived and developed by John von Neumann and Stanislaw Ulam, who believed that they would lead to a new way of understanding the biological concepts of construction and replication [45]. This was followed by a large amount of theoretical research done in the 1960's [9, 10]. The results of this research stayed mostly theoretical due to the practical constraint that actually simulating CA efficiently was very hard with the technology at that time. With more advanced technology, interest resurged in the 1980's. In particular, new physical and computational CA models and paradigms were introduced and new hardware on which to simulate these models were built [43].

1.1.3 Classifications

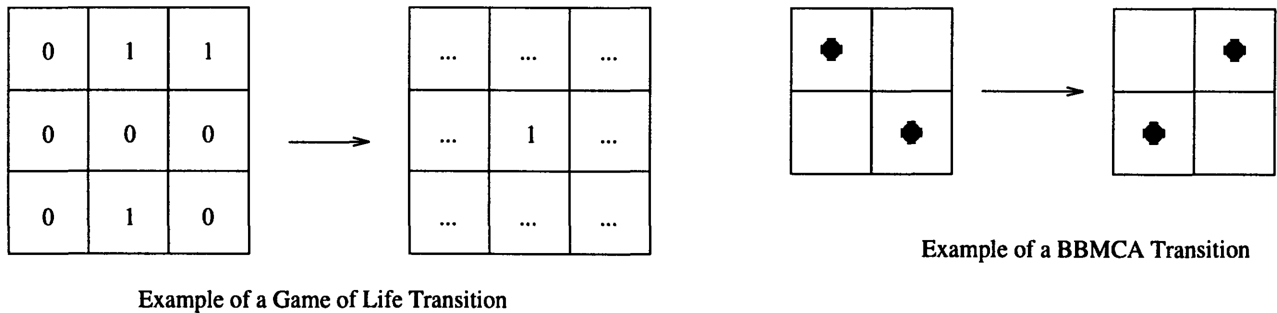

Figure 1-1: State transitions of two different transition functions. The Game of Life is a von Neumann CA and BBMCA is a partitioning CA. Both are capable of universal computation.

Traditionally, most CA have used non-disjoint, time-invariant neighborhoods, with only one transition function. These are known as the von Neumann CA, named after the inventor of CA[6, p. 3]. The most famous example of such a CA is the Game of Life, popularized by John Conway in the late 1960's[14, pp. 271-276]. In this particular CA, each cell is in one of two states, alive or dead: call these states 1 and 0. In each three-by-three neighborhood there is a center cell and eight adjacent cells. The new state of a cell is determined by counting the number of adjacent 1's–if exactly two adjacent cells contain a one, the center is left unchanged. If three are ones, the center becomes a one (Figure 1-1). In all other cases, the center becomes a zero. It has already been shown that patterns of bits the Game of Life can act like arbitrary logic circuitry and thus simulate a computer.

Partitioning cellular automata (PCA) (also known as lattice-gas automata) are CA with time-dependent, disjoint neighborhoods[7, p. 134]. At each time step, the space of cells is partitioned into disjoint clusters, all of the same size. The cells of each cluster transition to a new state dependent only upon the current state of the cells in the cluster. Thus, all the cells in that cluster have that cluster as their neighborhood. The space gets repartitioned during the next time step. If the function is different at differing time steps, the functions usually apply periodically in phases [6, p. 4]. In a PCA, the function referred to is the function that is applied to the clusters. If

a cluster is composed of m cells, each cell has a function that is m to 1 so that the function of the cluster is m to m. PCA have been shown to be useful for simulating lattice gases as well as being good devices for nanoscale computation. This is the theoretical model underlying the operation of CAM-8, the newest cellular automata machine [30]. An example of a PCA is the Billiard Ball Model CA (BBMCA)[27]. This is a model of computation which was originally devised to show how digital logic could be implemented reversibly, where very little or no power would be dissipated if this form of logic is used as a basis for making circuitry.[1] The BBMCA is a reversible CA, meaning that it can run forwards and backwards with time.[2] The BBMCA can be used to implement reversible digital logic using CA[29]. Each cell is in either a 0 or 1 state. If it is in a 1 state, it represents a ball in space at that coordinate and if it is a 0, then it represents empty space. The balls move around and collide. The movement of balls is done by repartitioning the clusters appropriately. The collisions occur at each cluster and are modelled according to a function consisting of 6 state transitions and their rotations (Figure 1-1) [27, p. 14].

PCA were found useful for modelling gases and studying their properties. Researcher's in the 1980's realized the value of using CA to model physical systems (or even to model entirely new "universes"[43]). The motivation is that it is possible to get complex global behavior from simple local dynamics such as that provided by cellular automata. This phenomenon is similar to physical theories based on a reductionist principle where far-reaching laws and theories are deduced from simple assumptions such as the existence of atoms. Rather than starting from the results of these theories to do numerical simulations based on the equations, the CA approach is to start from the simple assumptions (at the microscopic scale) and use the massive computational power provided by cellular automata to get the results of those assumptions[42]. A discussion of using CA for physical modelling is beyond the scope of this thesis but is treated thoroughly elsewhere[40]. Obviously, to get any results from this approach, the CA simulations must be fast and large. Normal computers

[1] This obviously has important consequences for building ultra-dense computers, where issues of power dissipation are very important[37].

[2] The same goes for reversible logic circuitry or reversible anything.

are very inefficient at doing this, and for this reason, researchers decided to build special-purpose hardware, the so-called cellular automata machines (CAM).

1.2 CAM-8

The CAM project was started in the early 1980's by researchers in the MIT Laboratory for Computer Science, Information Mechanics Group, devoted to the study of Physics and Computation. Several versions of CAM were built before the first one became commercially available, CAM-6[43]. Much was learned up to that point and incorporated into the CAM-6. However, with the rapid advances in hardware, along with new ideas about how to build CAM's, the CAM-6 hardware soon became obsolete. Next-generation ideas and hardware were incorporated into the CAM-8, the latest-generation CAM[30, 29].

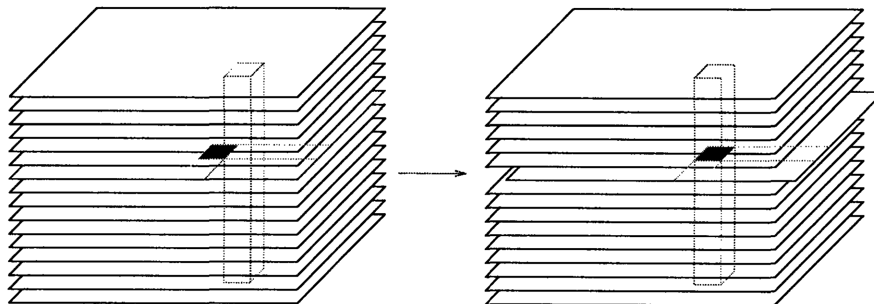

Figure 1-2: Illustration of sites, layers and kicks in CAM-8. On the left is the initial arrangement with a cross-section through the slices representing a particular site. On the right is the arrangement after a kick showing how the composition of the site has changed as a result.

The CAM-8 was designed to take advantage of hardware speed and parallelism to execute large cellular automata programs quickly and efficiently. The CAM-8 uses a PCA scheme to more efficiently simulate CA and LGA experiments. Each cluster is referred to as a *site*. Each bit of a site can be considered to be in a different bit plane, called a *layer*, similar to the arrangement in a frame-buffer, each site being analogous to a pixel (Figure 1-2). Each layer is stored in a different fast-access DRAM. There are two basic operations in CAM-8: data update and data transport.

1.2.1 Data Update

The m-bit state of each site is updated by fetching the appropriate m bits from DRAM and sending them through an SRAM lookup table (LUT) which maps them to another m bits which are then placed back into memory at the same location.In the current-generation CAM-8, m=16 by default, but can be extended by having virtual bits in what are called *subcells*. This requires extra work though and will slow down the speed of the simulation. Since the sites are disjoint, the updates are atomic, so that both sequential and parallel update over the set of sites are identical. The LUT is programmed by the user to implement a site function. There is only one logical active LUT in CAM-8 at any given time. However, the CAM-8 is capable of applying any sequence of LUT's so that the function need not be periodic. Switching LUT's is also useful for operating on subcells.

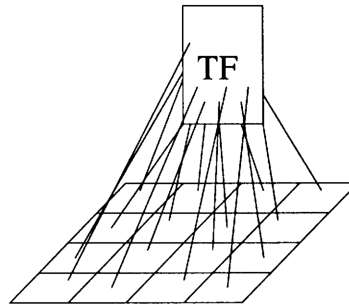

Figure 1-3: LUT in CAM-8 is time-shared over a group of sites called a sector

Obviously, the fastest way to do a CA simulation is to have one physical LUT per site so that all the sites are updated in parallel. However, such an arrangement would be too costly in terms of hardware. Not only that, but it would be inefficient for certain types of computations (Chapter 3). For this reason, the CAM-8 only has a few physical LUT's each of which is time-shared over a group of sites, called a sector (Figure 1-3).

1.2.2 Data Transport

The method by which CAM-8 implements a repartitioning operation is called data transport. The different layers can be moved in parallel, independently of one another,

by programmable vectors, by simply changing an offset which is added to the DRAM addresses of bits fetched from that layer. A single layer movement is referred to as a *kick* operation. This operation can be pictured as the different layers being "sheets" that move across the space in their respective planes, with wraparound (Figure 1-2). Thus, the bits in each bit plane are moved to different sites. This is equivalent to saying that the cells get repartitioned.[3] An atomic update of a set of sites (one without any kicks) is known as a *scan* operation. The sequence of these operations is also programmable in CAM-8.

1.2.3 Topology

The CAM-8 is indefinitely extensible in three dimensions, with more internal dimensions possible. In order to accomplish this, the space of sites is divided into modules which can be glued together in three dimensions.[4] The scan operations occur in parallel among the modules. Within each module the speed of the update is fixed by the size of that module. The current generation CAM-8 can update about 25 million sites per second and each module has 2^{24} sites. Thus, the more modules there are, the more cells which can be updated in parallel, so that the speed of update actually goes up with the size of the space. Currently, a typical CAM-8 has about 8 modules, so that there are actually about 200 million sites updated per second. The modules can be connected or *glued* together in three dimensions with a mesh topology. More internal dimensions are possible but they wraparound in the modules. The boundary conditions of all dimensions can be fixed or periodic.

1.2.4 Programming

An arbitrary sequence of the scan and kick operations, along with other useful peripheral operations can be grouped into a procedure called a *step*, which can be considered the unit of time in the CA program. This is programmed in Forth, the assembly lan-

[3]Think about it.

[4]A module is just the hardware that implements a sector.

guage of CAM-8[5, 19, 43](see Appendix A). The implementation of these operations is handled by the Space Time Event Processor (STEP) chip, each one of which is devoted to a different layer, within a module. The lookup-table function is also usually programmed in Forth, although this is not always the case.[5]

1.2.5 Visualization and Other Features

The CAM-8 can be interfaced to various external media, the two most important being a host and a monitor. The host is mainly responsible for I/O routines which can be programmed in Forth. The host is also directly responsible for controlling the CAM-8. The monitor can be used to visualize what is going on with the CA with a colormap which can also be programmed in Forth. More details about the CAM-8 can be found elsewhere[32].

1.3 Logic Simulation and Emulation

Logic simulation is important for verifying the functionality of a given logic circuit before dedicating hardware to it. A logic designer can use a logic simulator to isolate faults and correct them before hardware is actually used to implement the circuit, thereby saving time and money in the design cycle. Most logic circuits are too large to check by visual inspection. Formal verification tools are generally not sophisticated enough to handle most classes of circuits[4, p. 16]. Thus, logic simulators are important tools for VLSI and other hardware designers.

There is a large spectrum of platforms for logic simulation. This ranges from software simulators to full hardware implementations of the logic. Simulators can also be classified by the level of simulation [33]. That is, there are simulators ranging from low-level device simulation (SPICE, for instance), to high-level behavioral simulation (VHDL). In addition to logic simulation, emulation is becoming increasingly prevalent in the industry with the rapidly falling costs of programmable devices (PD's).

[5] In particular, the LUT was synthesized in my logic layout program as will be seen later on.

1.3.1 Logic Synthesis

Before logic circuits can be simulated they must be laid out. Logic layout can be done manually, partially automatically, or fully automatically. Beyond very small circuits, it is very hard to layout circuits by hand. Automatic logic layout is referred to as logic *synthesis.*

Pure logic design is mostly a software design task these days. Typically, designs are specified in high-level design languages (HDL's) such as Verilog and VHDL[41, 36]. With HDL's, it is possible to specify very large logic systems in a coherent way. These are then synthesized down to lower levels of abstraction. This is analogous to how a high-level programming language, like C, is synthesized down to assembly and then machine language. The information at these different levels can be used to configure a logic simulator or emulator. There are many different formats for specifying logic not only at different levels, but also between levels. It is usually possible to convert between levels. There are also a variety of different tools for the logic designer to choose from for logic synthesis, ranging from academic to commercial software. Many of them include simulators as well. I will return to the problem of logic synthesis in chapter five, particularly in regards to the CAM-8 logic simulator outlined in the next few chapters.

1.3.2 Programmable Hardware

Programmable devices (PD's) are used to emulate logic designs by actually implementing the logic functionality at the hardware level. They are distinguished from direct hardware implementations in that they can be quickly reconfigured, thus being more flexible[4, p. 19]. Like hardware implementations, they can be used as prototypes of the system being implemented, since they run at hardware speeds. Programmable devices usually come in two flavors: PLD's and PGA's[34].

Programmable Logic Devices

Programmable Logic Devices (PLD's) are PD's that are based on the two-level logic of PLA's[8]. Two-level logic usually has an AND plane and an OR-plane. They are usually implemented as networks of PLA's on a single chip. The logic behind PLD's is well-understood and it is usually used to implement the control logic in logic circuits, if such a function is necessary. They are usually synthesized from a truth-table or sum-of-products format.

Programmable Gate Arrays

Programmable Gate Arrays (PGA's) are PD's that can be programmed to implement multi-level logic[34]. PGA's are essentially cellular arrays of functional elements much like cellular automata. In fact, the idea of using cellular arrays for logic emulation grew directly out of the research on CA[4, p. 30]. I discuss the architecture of PGA's in more detail in the next chapter.

1.4 Thesis Objective and Organization

The purpose of this thesis is to explore reconfigurable logic simulation on CAM-8 and determine ways to make this simulation efficient. There is also the side goal of implementing some exploratory synthesis tools for this type of simulation. The CAM-8 can be considered to be an indefinitely scalable virtual gate array. As such it is good for exploring the types of large circuits and gate arrays which may be implemented in the future. This should provide some insight into how to arrange large-scale cellular logic computations and how circuits may be laid out on such cellular gate arrays. This is already done with FPGA's but the CAM-8 has a different type of architecture than most FPGA's which may actually be more suitable for doing logic simulations and even emulations than most normal gate array architectures. In addition, the CAM-8 has a completely programmable transition function so that it could emulate just about any possible gate array architecture. This would be useful for gate array architects who want to test their designs. Current synthesis tools may be sufficient but some

modifications are explored for synthesizing logic circuits to the CAM-8 simulator in a more natural way.

This thesis continues in chapter 2, with motivation for why cellular automata and CAM-8 can and should be used for logic simulation. I draw an analogy between PGA's and cellular automata. Then, a logic simulation based on emulating a parallel gate array on CAM-8 is explained. In particular, the underlying hardware abstraction of the gate array is explained and compared with the sample FPGA shown before. Some logic simulations of example circuits are then given. In particular, it is explained why parallel simulation of a gate array is not the best way to simulate digital logic. This leads into chapter 3, which discusses spacetime circuitry as a way to make the logic simulation more efficient. I discuss how to make computation more efficient in space and time with virtual parallel hardware and how to do this on the CAM-8. Chapter 4 goes into graphic detail about a specific implementation of a spacetime circuit, an 8-bit microprocessor. This serves to illustrate the concept of spacetime circuitry and the details of how circuits are laid out for simulation on CAM-8. This large circuit provides motivation for making automated synthesis tools which are discussed in Chapter 5. Chapter 6 gives some data and results about the synthesis and simulation of some spacetime circuits, the MCNC benchmark. I conclude in Chapter 7 and explain how this work can be carried forward.

Chapter 2

Parallel Gate Array Emulation

2.1 Motivation

Using cellular automata for doing digital logic simulations has already been reported elsewhere[43, pp. 136-138]. There are several reasons for using CA to perform conventional digital logic (DL) simulation and emulation.

2.1.1 Theoretical Equivalence

As abstract computing machines, CA are based on a parallel model of computation. Spatial arrays of finite automata are computing their states at the same time[10]. In contrast, normal computers like the average workstation or PC operate according to a sequential model of computation, the Turing model[18].[1] It has been shown, theoretically, that a CA with an infinite number of cells and a required blank cell state is equivalent to a Turing machine, which has an infinite tape[10]. Hence, a CA, like a Turing machine is computation universal, which means it can compute any computable function. This trait is also shared by DL, another model of computation equivalent to a Turing machine. Because these are all equivalent models, they can simulate each other with only polynomial time overhead. Hence, we can use CA to simulate DL efficiently, with the right tools.

[1] Normal computers, like PC's and workstations, have enough memory to "look" like a Turing machine, although they are really large finite automata.

2.1.2 An Alternative Architecture for Nanoscale Computation

Cellular automata have been proposed as alternative architectures for nanoscale computers. Wires will be hard to fabricate at this scale creating a need for wireless architectures like CA[15]. Beyond the fabrication problems already apparent as we go to smaller integration scales, the onset of "quantum" effects,[2] like tunneling at about 0.01μ linewidth and below will result in a breakdown of the assumptions underlying the operation of transistors. We will have to find a way to use quantum mechanics to our advantage rather than as an impediment to be avoided. Anyway one looks at it, this will have to happen eventually.

New devices and architectures will be needed. Some possible devices include resonant tunneling diodes (RTD's), quantum wires, and quantum dots.[3] Quantum dots, in particular, which are essentially artificial atoms, have been proposed as the cells in a CA, making a so-called quantum CA[44]. From a more fundamental perspective, quantum mechanical interactions are local and reversible (unitary) [39]. Cellular automata map well to these interactions because they update their state in a local manner. There are also reversible CA rules such as the BBMCA alluded to in the first chapter. These ideas have led some to propose a nanoscale CAM[6]. Such a device would be useful not only for implementing a nanoscale computer but also for exploring extremely large-scale simulations of physical systems such as those that the CAM's are currently used for, at a scale which is much closer to the actual scale. In addition, quantum mechanical states, and their associated operators can be (and some would argue, always are) discrete, so that physical states could be mapped in a direct way onto logical states[28].

All of these considerations lead to the conclusion that the CAM-8 can be used to preview and explore the kinds of things these nanoscale computers based on CA would be good for. In particular, this project could be seen as an exploration of a direct way to implement computation using CA. As such, the ideas in this thesis

[2]Those effects which can be explained *only* with quantum mechanics.

[3]Electron confinement in one, two and three dimensions respectively.

could be a first step towards making the synthesis tools for these architectures as well as seeing how the cells in a CA could be arranged to yield an efficient emulation of digital logic using CA. And emulating digital logic in CA is a direct way of performing a known computation using CA.

2.1.3 Gate-Array Like Quality

Cellular automata are essentially fixed gate arrays. Cellular automata were the predecessors to modern logic cell arrays, such as those used in FPGA's. This means that they can be programmed, at the hardware level, to simulate or emulate logic circuitry. Thus, design can be reduced to a software task, with the result being compiled directly to hardware. The architecture of gate arrays is discussed in more detail in the next section.

There are several arguments for using a cellular architecture to simulate or emulate digital logic[26]. Designing and optimizing a single, repeatable cell is preferable to having to redesign and refabricate a new circuit for every different application. It puts all the design constraints into software and leaves the hardware for the cell designer who can use whatever lithographic techniques are necessary to make the best possible cell. For the same reasons we can get good DRAM integration, we should also be able to get good cellular logic integration.

In addition, a cellular architecture is generally more flexible than other architectures. For instance, PLA's are generally limited to implementing combinational logic or sequential logic with only one level of logic. More course-grained architectures like dedicated processors hardware simulation or emulation limit the functionality of the design necessary to get efficient simulation. For instance, an architecture made up of programmable ALU's would not be a good architecture for mapping control logic[12]. In other words, a cellular architecture provides no arbitrary levels of abstraction on which a logic designer or synthesizer must work around to obtain a simulation or an emulation. Cellular architectures essentially provide a "canvas" on which can be painted any logic circuit, regardless of what its modules at higher levels of abstraction are meant to be. A cellular architecture is completely flattened, so that it can

be programmed with a hardware level machine language.

2.1.4 Architectural Proximity

As was pointed out in the last chapter, logic simulators come in many flavors. Software simulators provide the most flexible platforms for simulating logic[4]. They can usually simulate logic circuits at all different levels. Higher levels of logic simulation, such as behavioral simulation, are more efficient than low-level simulations on a normal computer. This is because a lot of the low-level details are being left out. Also, high-level behavior is better simulated by the complex operations and high data diversity at which more conventional processors excel at[12, p. 6]. Moreover, these simulations are generally word-oriented unless the simulations are of machines which are bit-grained architectures, few and far-between these days. It is usually behavioral models that designers are interested in these days in any case, since lower levels are usually abstracted out and can be completely synthesized without errors. Very low-level simulations can also be accommodated more easily, but this is getting down to the level of physical simulation. Software simulators are also the cheapest. It generally only costs time to program up a simulation on a computer. Software simulators are portable meaning that they can be moved to different hardware platforms.

But there is penalty to be paid in efficiency. First, computers are based on the sequential von-Neumann architecture, in which both data and program are stored in memory and accessed in a sequential fashion as the computation progresses. DL, like CA, has some inherent parallelism, which can't be taken advantage of. The problem is less severe for DL than CA but it still exists. This is known as the von Neumann bottleneck. Second, the datapath in most computer processors is word-oriented, so that bit operations are generally inefficient[12, p. 30]. Computers are not optimized for the types of operations inherent in CA and DL. In fact, this was the major motivation for building the CAM's and is the motivation behind building many of the dedicated hardware simulators and emulators of logic[4, 43]. Third, there is a high degree of overhead associated with running a program on a computer. Programs are usually implemented in a high-level language, which is generally compiled into

code which is less efficient than it could be. Even a fully-optimized program at the machine-level would have to contend with other operations using up processor time such as the operating system. Stand-alone machines, like the CAM-8 and dedicated hardware simulators and emulators, which are not time-shared, provide advantages here. Fourth is that software simulators are generally not scalable. They are usually limited in time and space resource by the machine on which they are being executed. This problem can sometimes be solved by the portability advantage but this is usually a hard task.

Specialized hardware simulators and emulators provide advantages over software in those three respects. However, hardware has several important disadvantages as well. Hardware is generally less flexible than a computer. The logic designer is limited to simulations at the level provided by the hardware simulator. For instance, if the hardware simulator only simulates two-input, one-output logic functions, the designer is limited to compiling the logic to two-input, one-output functions. This is fine if that is what the goal is, but in the electronic design automation industry, technical objectives may change on short notice due to competitive developments. Indeed, hardware itself is evolving at such a rapid pace, that the hardware in most simulators becomes obsolete soon after it is created. The inflexibility of the hardware simulator may also render it obsolete in a short time span. Hardware is becoming a commodity these days, but it is still expensive to simulate next-generation state-of-the-art machines by building hardware using current-generation technology.

The CAM-8 combines the efficiency of a hardware simulator with the flexibility of a software simulator. Because CAM-8 is based on cellular automata, it is well-suited to performing the bit-level operations necessary for efficient simulation of logic. Since it is based on a virtual-processor architecture, it is indefinitely scalable. Thus, arbitrarily large circuits could, in principle, be simulated by CAM-8. CAM-8 has some inherent parallelism. Of course, data movement is completely parallel as was pointed out in Chapter 1. Some of the updates are done in parallel as well, particularly across modules, and the update simulates a parallel operation. But CAM-8 is also programmable enough to simulate the sequential operations of logic circuits as well.

Because an arbitrary sequence of kicks and scans can be performed on CAM-8, it is possible to make the most efficient use of the resources in CAM-8 to simulate digital logic. This fact is taken advantage of in *spacetime circuitry* which is introduced and discussed in the next chapter. In addition to the functionality and computation flow, the topology of CAM-8 is also reconfigurable. This means that the space of sites in CAM-8 can have any dimensionality and regular connection so that it could, in principle, simulate any FPGA architecture, and do it in something other than two dimensions which most FPGA's are limited to. This could be useful to gate array architects who want to simulate their designs. There has always been talk of going to three dimensions to make IC's and some serious work has been done in this direction[17]. CAM-8 could be used to simulate these three-dimensional circuits and even some multi-dimensional circuits which don't scale. Thus, there is a high degree of flexibility inherent in CAM-8 which should be utilized and explored in performing logic simulations.

2.2 PGA Architecture

PGA's are basically cellular automata, except that they have semisystolic processors: the processors are not necessarily latched at the output so that the cell updates are not synchronous[24, 23]. In fact, unlatched connections through cells can travel arbitrarily large distances so that the architecture is not scalable like CA are. A PGA consists of an array of these semisystolic processors, called *cells*, all of which are identical and have uniform interconnect[34]. These cells are usually referred to as *logic blocks* but I will refer to them as cells for ease of notation and to carry over some of the terminology from CA. From now on, the terms gate array, PGA, and CA are used interchangeably with the terminology meaning the same thing for all of them and with the difference in synchrony being ignored except where needed. Any connection between two cells is referred to in this thesis as a *wire*.

A good example of a PGA is the Xilinx XC6200 FPGA. The XC6200 has nearest-neighbor 2-D mesh routing along with length-4 flyovers to simulate a 3-D interconnect

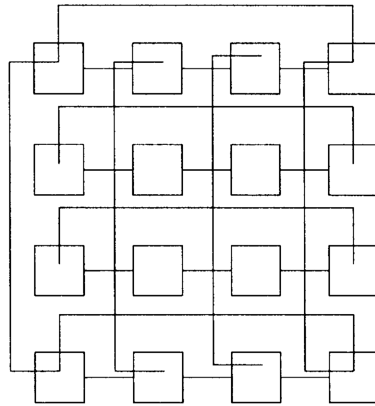

Figure 2-1: XC6200 interconnect with nearest-neighbor wires and length-4 flyovers

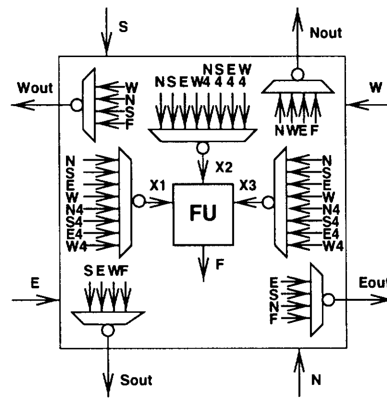

Figure 2-2: XC6200 cell routing

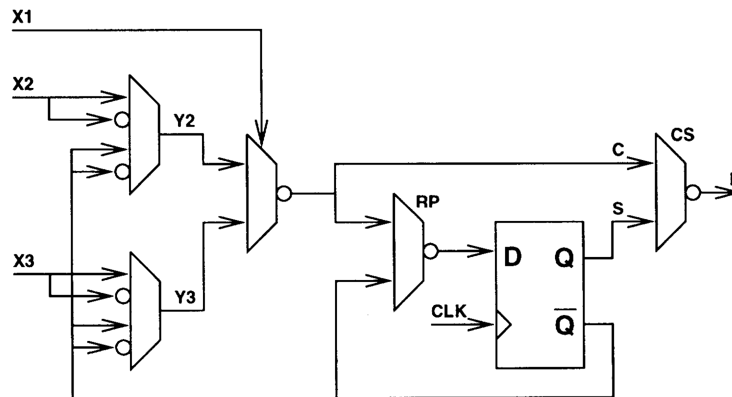

Figure 2-3: XC6200 cell function unit

pattern (Figure 2-1). The cell has some basic routing capabilities and a function unit (Figure 2-2). The function unit emulates a 2-input MUX/2-input LUT with an optional DFF at the output (Figure 2-3). This type of FPGA is similar to the hardware abstraction for the initial CAM-8 logic simulator.

2.3 Initial Implementation of a Logic Simulator

The CAM-8 was initially programmed to emulate a fully parallel and systolic gate array in order to do logic simulation[24, 23]. A fully parallel gate array is one in which all the cells can get updated in parallel. That is, each cell has its own hardware, so that it can perform its update independently of the other cells. The emulation is inherently systolic because the CAM-8 emulates a systolic network. As will be seen later, this parallel gate array model is an inefficient way to do logic simulation on the CAM-8. Later, I'll talk about a technique that uses the virtual processors of CAM-8 efficiently. An arbitrary two-dimensional gate array model was invented which was very similar to the Xilinx XC6200. Each site emulates a single cell. Like the von Neumann architecture, there is an arbitrary division in each cell between data and configuration. The data defines the logical signals which flow through the circuit which is being simulated. The configuration defines the circuit functionality. They can be pictured as two planes of information, one of which is static (static field), and one which is dynamic (dynamic field). For this reason, they map well to the layers of CAM-8. Groups of layers in CAM-8 are referred to as fields. The static field is called the configuration field and the dynamic field is called the signals field. It is also easy to think about in terms of the layout (Chapter 5). Within a site, a certain number of bits are dedicated to logical signals in the circuit which are referred to as the *signals.* The rest of the bits are dedicated to the *configuration.* Figure 2-4 shows the logic simulation implementation for a different gate array than the one described here. Nonetheless, it is a good illustration of how logic simulation is implemented on the CAM-8.

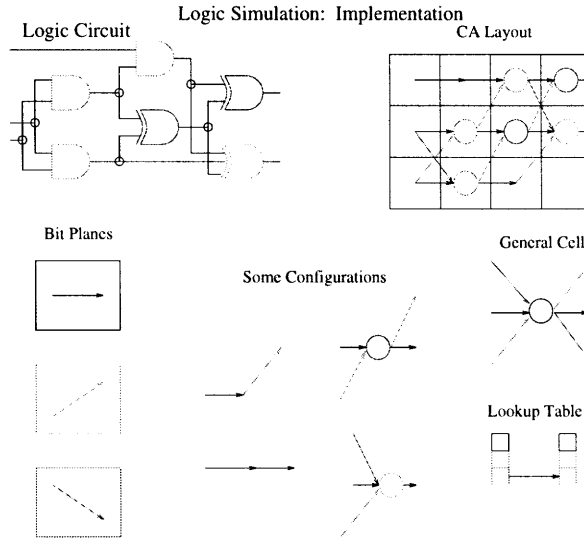

Figure 2-4: CAM-8 logic simulator summary

2.3.1 Signals

The signals must be able to travel between sites in order to emulate the wires of a gate array. The logical interconnect between cells was implemented through the data transport mechanism in CAM-8. The wires of the gate array are implemented through the kicks in CAM-8 (Figure 2-4). So each signal at a site is represented by a bit. Some signals are not moved but are stored in the site up to a certain point, thus emulating a latch or flip-flop in a cell. A global clock signal is implemented in CAM-8 by having two planes of configuration, which have one bit, called the clock bit, which is different. The way this works is discussed in the next section.

The gate array has short and long wires. The short wires are the nearest neighbor interconnects: north, east, west, and south (NEWS). The long wires are in the same directions (NEWS), but have length 10. That is, the long wires can transport signals from a site to a site 10 positions over in the north, east, west, or south, similar to the length-4 flyover wires in the XC6200. Again, the magnitude and direction of these wires is completely configurable in the CAM-8. Because of the parallel operation of the kick, all signals are transported simultaneously.

2.3.2 Configuration

During a site update, the data bits are changed as a function of both the data bits and the configuration bits. The configuration bits are static meaning that they are invariant under both data transport and data update. This is because the CAM-8 is emulating an FPGA whose function on the data is constant with time. The CAM-8 could also be programmed to implement an FPGA that changes its configuration with time in a dynamic way. Such an emulation has not been attempted yet although it would be straightforward to implement and possibly quite interesting. But since this is simply an abstraction, it doesn't really matter for our purposes. We do not have enough configuration bits in a site to model the entire functionality of most real gate arrays. The XC6200, for instance, requires 24 bits just for configuration, as can be seen by looking at Figure 2-2 and Figure 2-3. There are only 16 bits available in CAM-8 not to mention the bits that are already used to transport the signals. There is a way to get around this in CAM-8 called subcells. With subcells, each site can have more virtual bits, but it requires more time to update each site. This introduces some added complications. So, we choose a simpler set of cell configurations for our gate array.

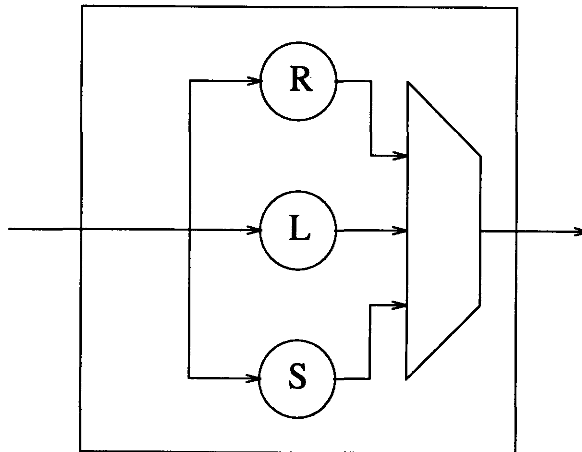

Figure 2-5: Routing, logic, storage cell

In our example, there are three main types of cell configuration: routing, logic, and storage (Figure 2-5). They simply specify what type of function is performed on the data. It is similar to the RLSC node model and is described in more detail in Chapter

4 as part of the virtual gate array model used to simulate a microprocessor. The only difference is that these function types are mutually exclusive. These abstractions, though not strictly necessary, make it easier to see how the CAM-8 emulates a normal gate array, like the XC6200 and are easier to program. The data bits in the layers that implement short wires are called slow bits, and the bits in the layers that implement long wires are called fast bits.

Routing

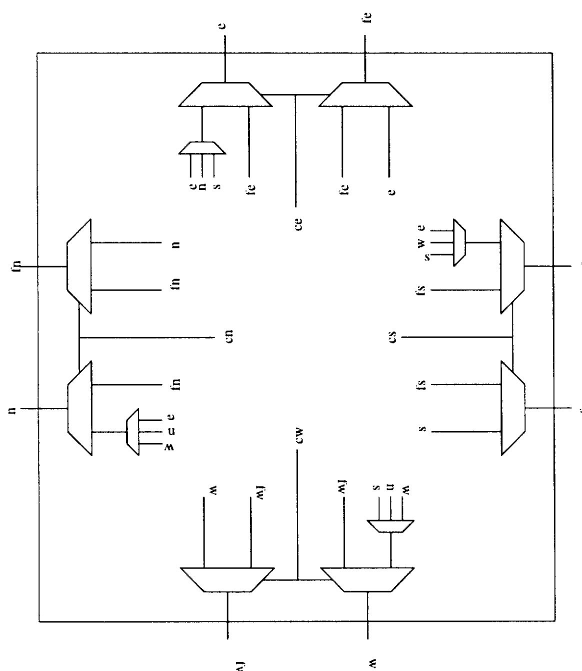

Figure 2-6: Routing cell

If the configuration specifies that a routing function should be performed upon the data bits, then some bits are just moved and possibly copied to other layers. There aren't enough bits in a non-virtual site to implement a full crossbar, so we opt for a simpler scheme which is similar to the XC6200 routing resources. In our example, slow bits can go to any other of the short wires, except the one in the opposite direction from which they came. In addition, a slow bit can go to a long wire in the same direction and a fast bit can go to a short wire in the same direction. In other words, either slow bits are routed to short wires or slow bits and fast bits are

swapped (Figure 2-6). There are also sites which provide constant sources or ones or zeros although this is, strictly speaking, logic and not routing.

Logic

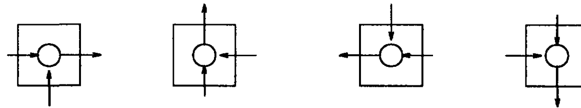

Figure 2-7: Logic cell

The configuration can specify that the site should perform a certain logical function on some of the data bits. Just as in the XC6200, we multiplex a subset of the logical signals into the function. In our example, the subset can be one of four possibilities with the output direction being determined for each of these four cases, called *orientations*. We output to only one signal, let one of the inputs flow through and zero out the rest of the signals. Figure 2-7 shows how the logic works.

Storage

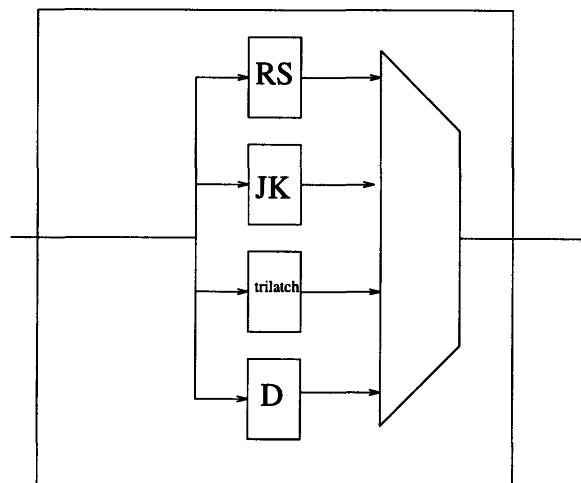

Figure 2-8: Storage cell

If the configuration specifies that the site implements a storage function, then there are four types of latches which may be specified: an RS latch, a trilatch, a JK flip-flop, or a D flip-flop. The storage functions, like the routing functions, have orientations

which specify what directions are used for input and output. An RS flip-flop has two inputs and one output. The output toggles if both the inputs are high, otherwise, it stays the same. A trilatch has two external signals, read and write. When read is high, the value of the stored bit is put onto another one of the short wires. If write is high, then one of the slow bits is stored in the storage bit. A toggle flip-flop toggles the stored bit whenever the clock signal is high. The stored bit is also constantly read out onto one of the slow bits. A D flip-flop, whenever the clock bit is high, puts the value of one of the slow bits into the storage bit, and moves the previous storage bit into one of the other slow bits. Figure 2-8 shows how the storage works.

2.3.3 Dynamics

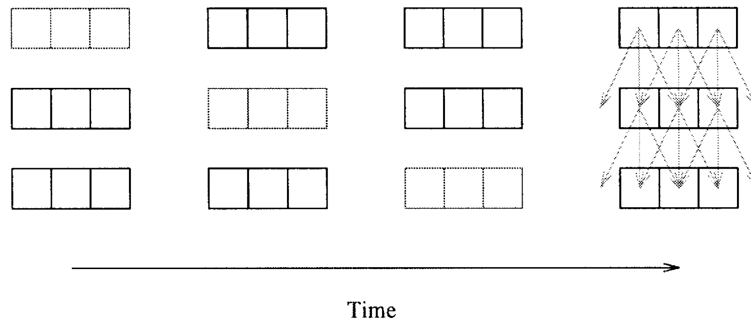

Figure 2-9: Illustration of a step for a parallel gate array simulation

In order to emulate a fully parallel gate array, all the sites which are emulating a cell must be scanned during a data update. Then the data transport is done to move data. So, a single step in this program is a full scan of the space followed by the kicks (Figure 2-9). Also, the gate array implementation is systolic which means that the signals are all synchronized. Signals that emanate from storage sites and constant sites are repeated so that the signal can actually be a worm-like chain of bits crawling down the wires. It is only really important that the right signals interact at the right places. Hence, the signals can be implemented with one bit but the placement of gates and registers is constrained by this condition.

2.4 Some Logic Simulation Examples

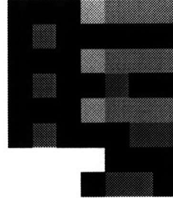

Figure 2-10: A simple PLA on CAM-8

Using the model of a gate array outlined in the previous section, several circuits were implemented. A simple PLA is shown in Figure 2-10. This is actually a majority gate which has been tested and works. PLA's like this are relatively easy to synthesize because of their very regular structure. However, they are inefficient in terms of area, because of the simple components of which they are made. Multi-level logic is much better suited to simulation on CAM-8. Furthermore, using only two types of logic gates is a waste of the resources that CAM-8 provides. That is, because CAM-8 is a LUT-based machine, it is wasteful to use the lookup table to just implement two functions. That is not to say that the PLA's don't come in useful. They are still useful for generating control circuitry because this circuitry is often specified in canonical form which is directly translatable to a PLA[8].

A more complicated circuit is shown in Figure 2-11. This is an 8-bit random number generator feeding an 8-1 multiplexor being selected from a 3 bit counter. It has been used for many demos because it looks very much like an electron micrograph image of a circuit which is actually being run. In both this and the previous circuit, each of the pixels is one site. Because of the colormap, many of the contiguous sites have the same color and hence are indistinguishable.

The most complicated circuit which was attempted using this gate array scheme was an 8-bit microprocessor (Figure 2-12). This circuit was found to function way

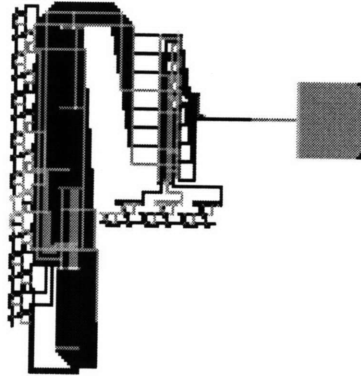

Figure 2-11: More complicated CAM-8 circuit

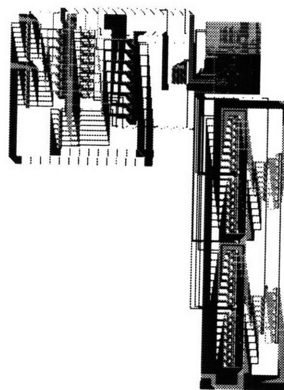

Figure 2-12: Microprocessor on CAM-8

too slowly to be of any use. It functioned at around 0.2 Hz. In the next chapter, spacetime circuitry is introduced as a way to make the simulation go much faster and run more efficiently on CAM-8 by taking advantage of its virtual-processor feature. In Chapter 4, a spacetime circuitry implementation of this same circuit is shown which runs at around 200 Hz, approximately 3 orders of magnitude faster! In Chapter 5, I discuss the automated synthesis tools which make this circuit and other large circuits a whole lot easier to lay out.

Chapter 3

Spacetime Circuitry

Spacetime circuitry is a technique in which logic simulation is made much more efficient on CAM-8. The basic idea is that if only a limited number of resources are needed at any one time, then the entire resources of a fully-parallel architecture are not needed to have a simulation go nearly as fast as the fully-parallel architecture. This is the case in combinational logic in which only a portion of the logic circuitry is active at any one time.

3.1 Spacetime Computation

During a computation, it is generally desirable that a minimal number of resources are left idle so that maximum use of the space is made during the computation. Similarly, if only a limited number of resources are needed at any one time during a computation, then only that number of resources need be utilized during the entire computation. In general, in order to have a maximally efficient computation, it is desirable to minimize the spacetime volume. This involves finding the point on the spacetime tradeoff curve that is optimal. The exact definition of efficiency ultimately varies from situation to situation as resources and needs vary. For instance, it may be more important to have a faster computation than to use up less space.

One definition of efficiency, E, comes from the theory of parallel algorithms[24, 23].
$E = S/P$

where

$S = T_S / T_P$

where the T_S is the time the algorithm would take on a sequential computer and T_P is the time on a parallel computer with P processors. S is the speedup of the algorithm on the parallel computer as compared to that on the sequential computer.

If one computation depends on another, then it is obvious that those computations must be done sequentially. If these computations are each done on two separate processors, then during each of the two time-steps, one processor would be idle. In other words, since the total computation is sequential, the full parallelism is not taken advantage of, and there is some obvious inefficiency. Its best to do such a computation on one processor.

This generalizes to any computation with sequential and parallel parts. The most efficient way to do such a computation is to map the parallel parts onto parallel resources and the sequential parts onto sequential resources. There are several ways of doing this, one of which is *pipelining.*

3.1.1 Spatial Pipelining

In *spatial pipelining*, a fixed number of resources (call them processors) are available at one time and it is desired to make full use of those resources. If an input set does not require all processors be used at the same time, and there is more than one input set, then the object of spatial pipelining is to use the processors to work on more than one input set at the same time. For instance, an input set could be input on one side of a mesh, pushed through one column, then the next input set could come in and so on.

This is a common practice in digital logic to increase the total throughput without increasing the latency[46]. But this technique is only useful if there is no feedback involved, as there is in a finite state machine (FSM)[12, p. 181]. That is, when there is a sequential dependence between input sets, the entire circuit must be evaluated for one input set before the next input set is input. Thus we are *latency limited* here.

3.1.2 Temporal Pipelining

Temporal pipelining is a technique which is useful where the number of parallel resources is limited. If the number of resources available at one time is less than the total number of resources needed throughout the computation, then those resources are going to have to be reused. In other words, the resources have to be time-shared over the computations that need them. In situations like these, the scheduling of the computations is called *temporal pipelining.*

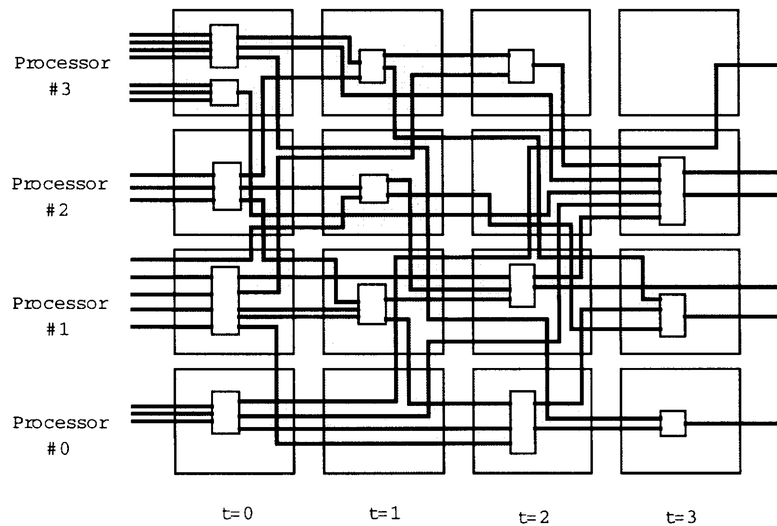

Figure 3-1: A physical processor assuming the role of different virtual processors

Physical processors which get reused in this way are essentially emulating *virtual processors.* These are processors which exist in time rather than space (Figure 3-1). At each time-step, each physical processor can assume the identity of a different virtual processor. This idea has been used in several different contexts.

The CAM-8 operates in this manner. As was mentioned in Chapter 1, each LUT is time-shared over a chunk of space in CAM-8. Each site, by itself just holds the state of that site. Instead of each site having its own LUT, the sites share a single LUT with other sites. The LUT is considered to be the hardware that makes each site into a *virtual processor*, rather than just a memory cell.

This idea has also been used to create a class of PGA's which can change their configuration with time. These are known as DPGA's and are based on LUT's[11,

12]. Each configurable logic block has a local instruction cache which stores several different configurations which are swapped into the active configuration at different time steps. Thus, each CLB can act like a different "processor" at different time steps.

It has also been used to reduce the routing constraints between different partitions of a multi-partition logic circuit. But in this case, the resources are not active elements, like sites or CLB's, but pins on an FPGA. The partitions of the circuit are each mapped to a different FPGA. If the FPGA's are pin-limited, so that they generally have less pins than needed to connect different partitions, then the number of CLB's which are usable on the FPGA is reduced. By taking advantage of the fact that the pins can be strobed much faster than the logic circuitry in the chip, it is possible to time-share the pins among the different resources that need them within the chip, thus increasing the usable pin count. This technique is known as *virtual wires*[4, 3]. This is similar to the problem encountered in the gate-array simulation outlined in the last chapter. As will be seen in the next chapter, we can time-share both the LUT and the bit-plane data movement among the sites to increase the efficiency of our logic simulation tremendously.

3.2 Spacetime Circuitry

In our original gate-array simulations, we were scanning the entire space of sites before we did any kicks. This is a very inefficient way to perform the computation for several reasons.

First of all, scanning simulates a parallel update of all the sites. But, since there is some logical dependency between the different sites, a fully parallel update is not really necessary. That is because the sites are simulating the gates and wires of a logic circuit. So, gates that depend on each other, which are mapped to different sites, can be simulated in a sequential fashion. Following the logic signals through the circuit during the simulation leads to the conclusion that only a few sites are doing any useful work during a single full scan (Figure 3-2). This set of sites varies

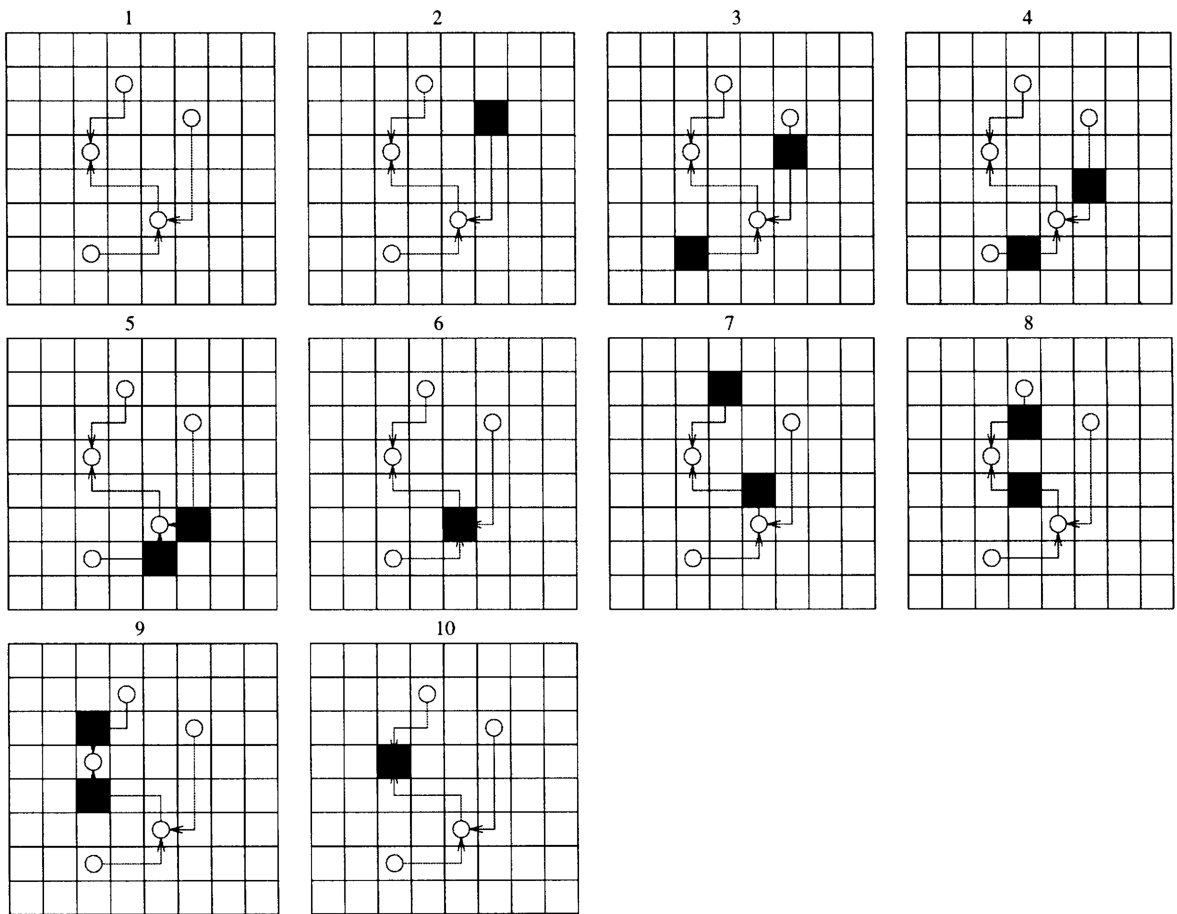

Figure 3-2: The levels of a 2-D circuit laid out on a spatial grid without spatial pipelining

through time and each such set is called a *level*, analogous to the levels in the logic circuit. If this were a fully-parallel simulation of sites, then this would already be inefficient. If S is the average size of a level, T is the number of levels, P is the total number of sites, and $f = S * T/P$ is the total fraction of sites which do useful work throughout the simulation, then the efficiency would be

E=$((S * T)/T)/P = S/P = f/T$

If $S_{max} = r * S$ were the maximum number of sites in a single level, and $P = S_{max}$, then E=$S/P = S/S_{max} = 1/r$

Assuming that $f*r < T$, then there is definitely a gain in efficiency by using a fixed number of sites equal to the maximum number of sites needed in any level. Because I am assuming a linear speedup in going from the sequential to the parallel run of the simulation, we are using T times as many processors than are really necessary.

Translating to the virtual processor architecture of CAM-8, this means that the simulator is running about T times slower than it needs to. This is because, in virtual space, there are T times as many virtual processors than are necessary and each virtual processor takes up one unit of time.

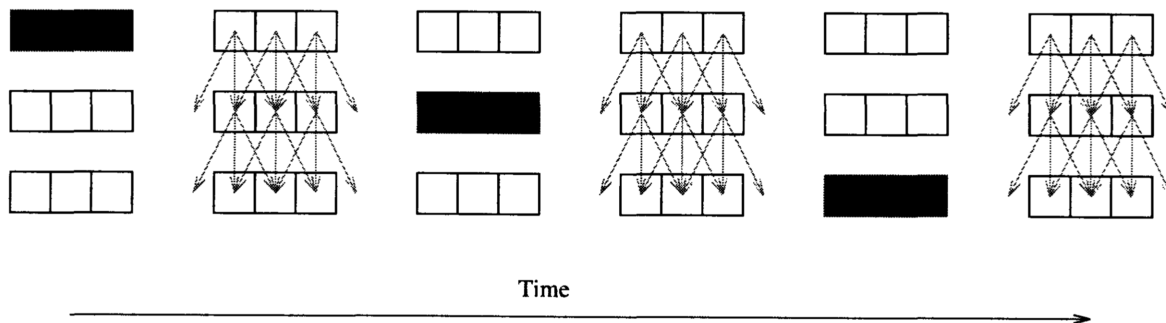

Figure 3-3: Illustration of a step for a virtual gate array simulation. Each row of cells, which is a level, is updated once and then data is kicked to the next row.

It is possible to eliminate this inefficiency by reducing the number of sites that are being simulated during a single scan. This allows us to scan only those sites that are necessary to simulate a single level. Between each of these reduced scans, we can kick the values to the next level that needs them and then perform the scan over

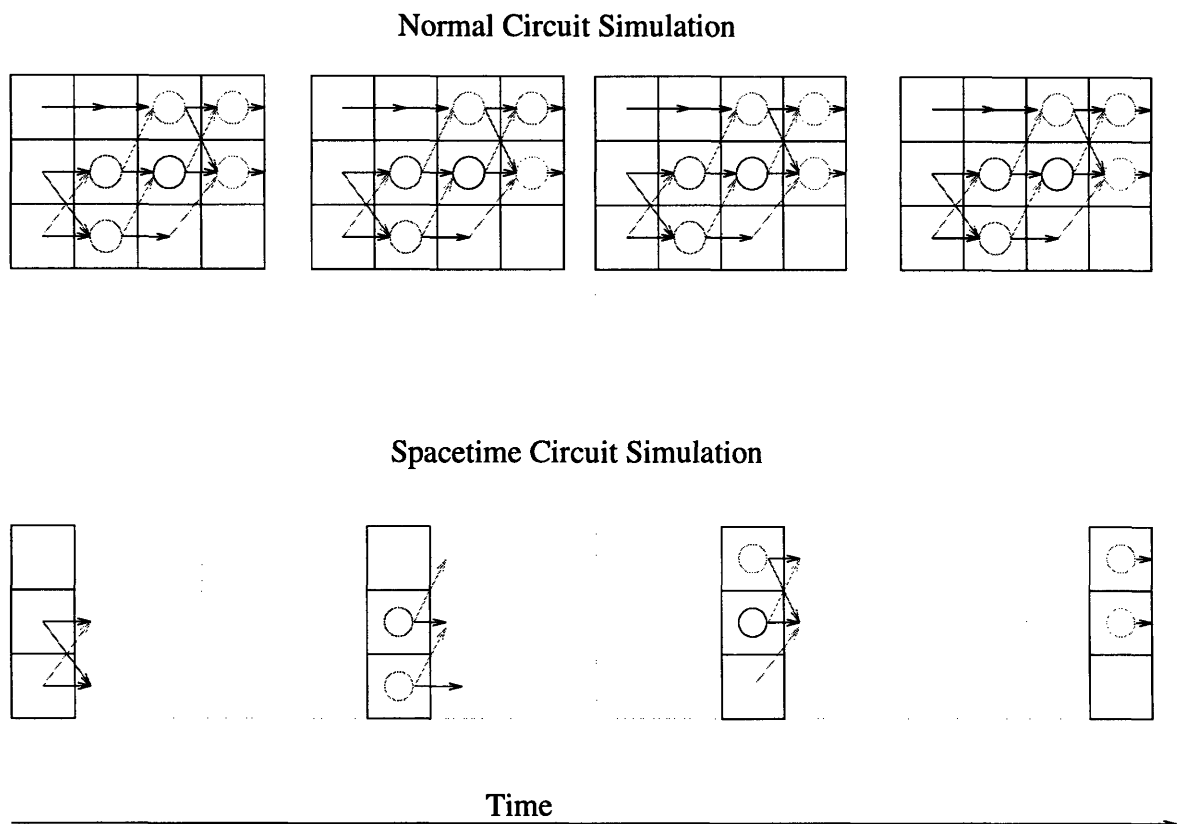

Figure 3-4: Comparison of normal and spacetime circuit simulation

them (Figure 3-3). Said another way, we are actually moving the signals along with the update, thus reducing the latency of the circuit being simulated to one complete scan of the space! This is what is known as spacetime circuitry.

Figure 3-4 gives an illustration of how the circuit simulation is sped up by a factor of T. In the normal circuit simulation, we are simulating a parallel gate array. So all the sites get updated in one step. In Figure 3-4, this is shown at the top of the figure. The top figure shows a circuit with a latency of 4, so that 4 scans of the entire space are needed to move the data all the way through it. In the spacetime circuit simulation, one column of the gate array gets updated at a time and then data is moved to the next column. Each column is essentially a level. So, the data moves all the way through the circuit in one time step, so that the combinational latency is one complete scan of the space as claimed before. Since we only have to scan the entire space once rather than T times to get the results of one combinational cycle of the circuit simulation, the speedup is T. Because the kicks are parallel and implemented with a simple pointer operation, the overhead of adding T of these is negligible compared to the factor of T in the speedup of the scan, which is a much slower operation than the kick.

What we are really doing is using the virtual processor architecture to actually perform a virtual processor emulation of the gate array. Before, we were not actually dividing up the virtual processors, which were already there, among the levels in order to perform temporal pipelining of the gate array for maximal efficiency. In CAM-8, we get this feature for free.

Our design was also pin-limited. If we look at the wires between the different levels, we could only access these wires after one scan of the entire space. But, since only one level was really active, only that level could transport signals to the next level. This is wasteful since we could just transport the signals after a scan of any single level. This is analogous to time-sharing a single bus which connects all the levels. Rather than putting the signals from the current level on the bus after each full scan of the space, we can put them on the bus after only a scan of one level.

3.2.1 Implementation

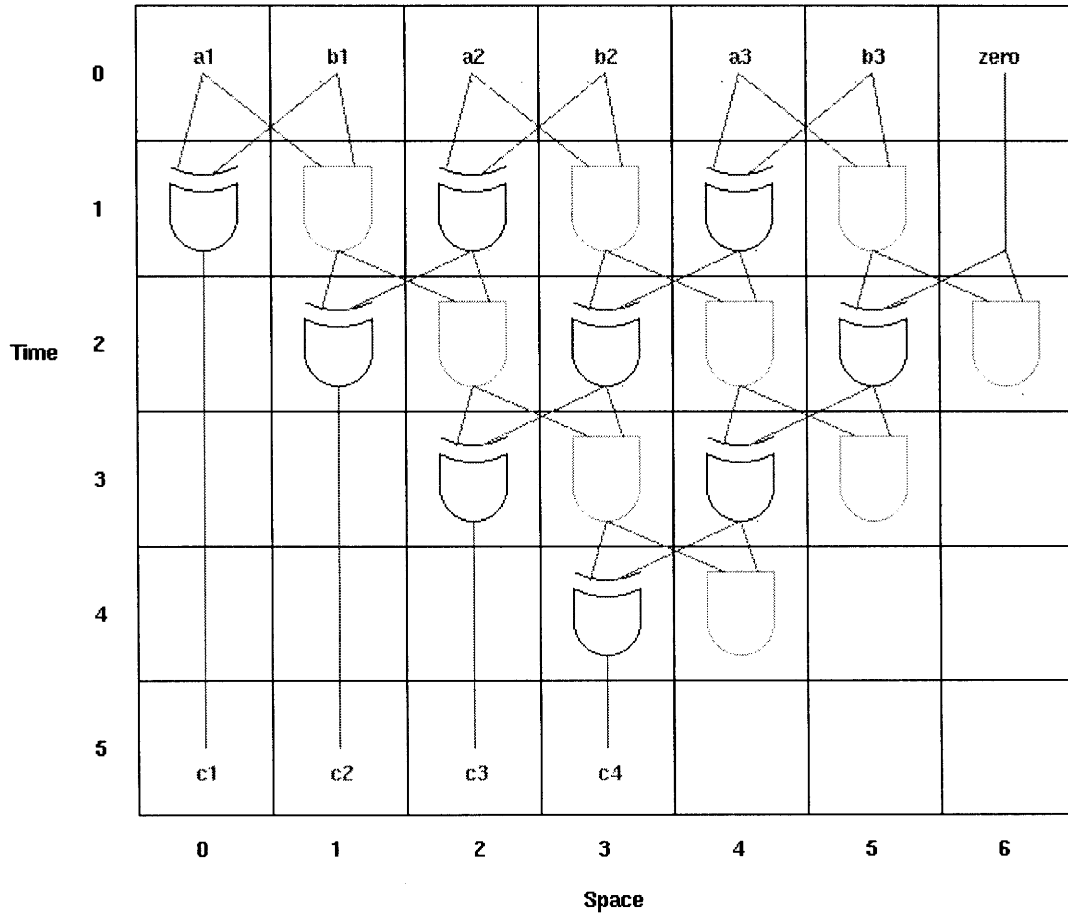

Figure 3-5: A 3-bit spacetime adder

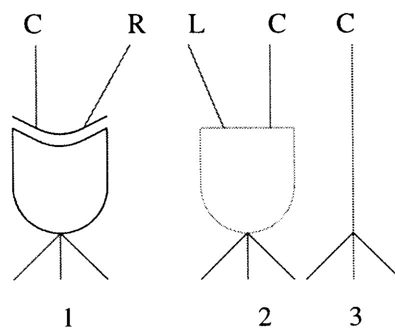

Figure 3-6: Configurations of the spacetime adder

Spacetime circuitry can be pictured as a circuit which is laid out in space and time. The space is the space of processors that are doing work at the same time, thus implementing a level. At each time step a different level is implemented in the space. Figure 3-5 shows a spacetime circuit, a 3-bit adder, with one space and one time dimension. Figure 3-6 shows the different types of cells which would be needed to implement this adder. As will be seen in Chapter 5, because these are the only configurations of cells needed, they are the only ones which need be implemented as configurations in the simulation, thus saving configuration space. This is another way in which only those resources that are needed are actually used.

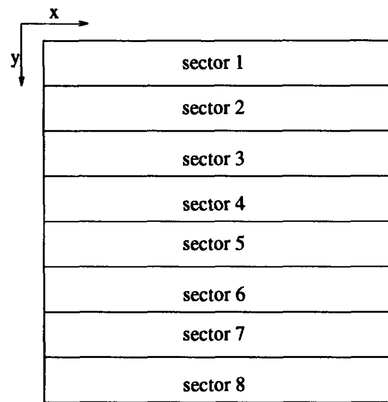

Figure 3-7: The division of modules in the y-strip topology. The y dimension should be the parallel dimension and the x dimension, the pipelined dimension in a 2-D spacetime simulation

In CAM-8, the space and time dimensions are each different dimensions in the space. We refer to the space as the parallel dimension(s) and the time as the pipelined dimension. For instance, the spacetime circuit shown in Figure 3-5 would be implemented in two dimensions. The time dimension maps onto one of the spatial dimensions in CAM-8. For instance, x could be the parallel dimension and y could be the pipelined dimension. There are ways to optimize this situation in CAM-8. For instance, the parallel dimension should be mapped onto the hardware parallelism as much as possible. As was mentioned in Chapter 1, the modules in CAM-8 can be arranged in an arbitrary topology. The default in CAM-8 is called the y-strip topology in which the sectors divide up the space in the y dimension only (Figure 3-7). Thus,

the parallel dimension should be the y dimension in this case. Or if there is more than one parallel dimension, one of them should be the y-dimension. Different sector topologies would have different dimension mappings.

3.2.2 Finite State Machines

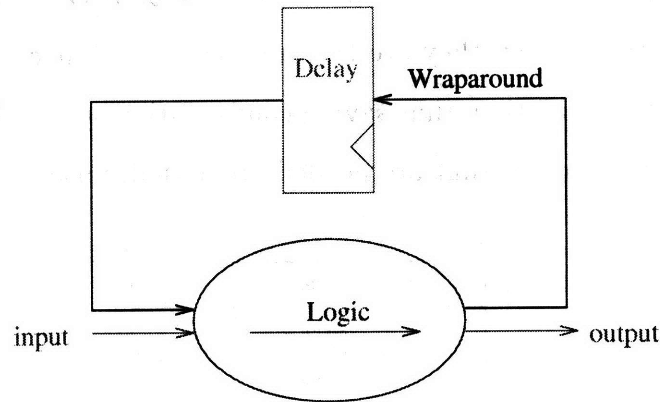

Figure 3-8: An FSM

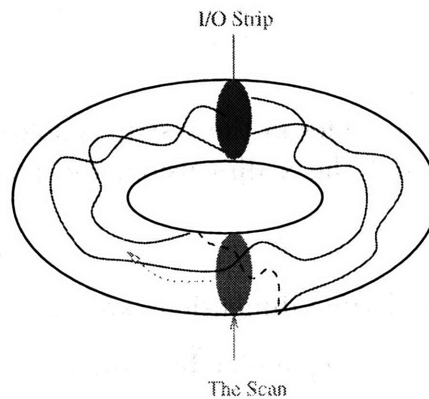

Figure 3-9: The space in CAM-8 is best visualized as a torus because of the wraparound at the edges of a dimension imposed by periodic boundary conditions

Spacetime circuitry can be used to implement sequential logic. Since the dimensions can have periodic boundary conditions, they can wrap around. Thus, the temporal dimension can be wrapped around providing feedback. Since the entire simulation is systolic, there is no need for explicit latches as was done in the parallel gate array

implementation in Chapter 2[1]. The combinational logic is just the entire space. If there are multiple spatial pipeline stages, the registers between combinational blocks can be ignored as well. Thus, the data simply flows through the combinational circuit in one time step. The first and last level are identical so that IO can be performed on just this level. The state wraps around, providing the feedback. Figure 3-8 shows a finite state machine and Figure 3-9 shows how the CAM-8 could implement such an FSM using spacetime circuitry. In Figure 3-9, the scan proceeds from the IO edges all the way around the torus back to the IO edge carrying the signals along with it so that the latency of the entire circuit is one update of the space.

3.3 A Brief History

Doing levelized logic simulation is not a new idea. In fact, the idea can be traced back to the Yorktown Simulation Engine (YSE) and its predecessor, the Logic Simulation Machine[4, pp. 29-30]. The YSE was a dedicated hardware simulator which could simulate one million gates at over two billion gate simulations per second[13].[2]. It took advantage of the same fact that is used in spacetime circuitry, i.e., that processors could be scheduled to run in a virtual way without incurring any penalty since the circuits themselves often have sequential dependency.

Levelized logic simulation is still fairly common in most parallel simulators. Scheduling circuit levels precisely so that only the nodes in that level are simulated has often been thought to be a task most suitable for MIMD computers[33]. This is because all the processors which need to do work currently can just be sent the right information. However, it is possible that CAM-8, which is really simulating a SIMD array of processors, can be made to do this same thing by appropriately *permuting* the scan. This idea has not been attempted, but is suggested as a future avenue of research.

[1]They weren't needed there either if the logic was arranged in a synchronous way.

[2]Comparable to the capabilities of the CAM-8 as shown in Section 1.2.3 considering each site as a gate. In reality, a gate is usually a 2-input NAND gate for obvious marketing reasons so that the CAM-8 can be considered even better in terms of gate updates per second (13 trillion gates/second with this metric).

Chapter 4

Simulation of a Microprocessor

To illustrate the efficiency of using spacetime circuitry for simulating large digital logic circuits at high clock rates on CAM-8, an 8-bit microprocessor, affectionately called the Sexium, was implemented. The Sexium microprocessor was designed by Bill Dally and David Harris for a VLSI chip design class, in which this was actually designed and fabricated[16]. It has already been emulated using a single Xilinx XC4005 and hence, it seemed a reasonable candidate for simulation on CAM-8. The circuit was successfully laid out using CAD/CAM-8, a layout tool described in the next chapter, and simulated as a two-dimensional spacetime circuit. One dimension was designated as the parallel dimension and the other was designated as the pipelined dimension.

4.1 The Virtual Gate Array Emulator

This section describes the implementation of the two-dimensional virtual gate array which was emulated on CAM-8 to simulate the microprocessor.

4.1.1 The RLSC Node Model

The model of a site used in the CA simulations is referred to as the RLSC Node Model. This stands for the routing, logic, storage, and configuration node model. These terms describe the functionality of a site, called a *node* and are represented as

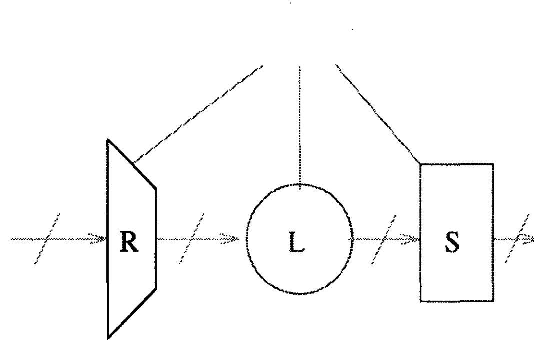

Figure 4-1: The functionality of each node in the lattice is shown above. The control element labelled C controls the mapping of the inputs through routing (R), logic (L), and storage (S).

shown in Figure 4-1. It is an abstraction of a circuit which could be used to implement this cell.

The configuration module controls the actual function of the node, i.e., whether the node will be a routing, logic, or storage element. In a routing element, each output is assigned a certain input. Routing elements are necessary to take the place of the arbitrary wires which would be use to connect logic gates in a circuit. A logic element performs a logic function on the actual logical values of the input signals. Examples of logic functions include AND, XOR, NAND, NOT, and other familiar ones. A storage element latches the values of inputs and holds them for a certain number of time steps. It can be used to model the functionality of a memory element, such as a latch, flip-flop, or register.

In this case, the static data is divided into three components, each representing an RLSC module: routing, logic, and storage. There are two bits which represent whether the site is background (i.e., no functionality) routing, logic, or storage. These two bits represent the configuration. Branching can be implemented by using condition bits which set the configuration. If a certain condition bit is 1, then all the rest of the configuration bits represent information about that condition. This is a way of reusing the bits, to represent different things based upon the value of the condition bits. In this case, the condition bits might be called `routing?` and `logic?`. The bits

	east	north	west	south
out	0 0	0 1	1 0	1 1
in	0 1	1 0	1 1	0 0

Table 4.1: A possible routing table. In this case, outgoing east(00) comes from north (01), north (01) comes from west (10), etc.

represent routing if the first condition bit is 1 and logic if the second is 1. In this case, there is one more bit for routing configuration than for logic configuration since the logic? configuration bit is only active if routing? is 1. If both condition bits are 0, then the site would represent a storage site. The I/O for each site is obviously the signals. The inputs are represented by dynamic data which has just been kicked into a site for updating. The outputs are represented by the dynamic data after the update of the site. The outputs get kicked out after the update.

Routing

Each site can have a different routing table and this table is represented by the bits in the site. There are a number of ways of using these bits to represent the routing information. In the method used here, the routing bits are divided into equal segments. The total number of segments is the number of outputs. Each segment is a number that represents the input that is mapped to that output. For instance, if there are four inputs and four outputs then there would be $4\log_2(4) = 8$ routing bits as shown in Table 4.1.

The size of this routing table grows as $O(n\log(n))$ where n is the number of directions. In our case, n is linearly proportional to d, so the routing table grows as $O(d\log(d))$.

Logic

The logic element can be specified by a truth table. An example is the XOR function, which outputs a 0 if both inputs are equal and a 1 otherwise. The truth table is shown in Table 4.2.

The truth-table is analogous to a LUT where the output is addressed by the 2-bit

input 1	input 2	output
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.2: Truth table for XOR

	0	1	2
zero	0	0	0
nor	1	0	0
xor	0	1	0
nand	1	1	0
and	0	0	1
xnor	1	0	1
or	0	1	1
one	1	1	1

Table 4.3: All possible 2-1 totalistic logic functions with the rows enumerated by function and the columns by the sum of the inputs and the cells giving the output for that sum

input. This is the way it is implemented in CAM-8. A number of bits equal to the number of possible outputs is allocated. These bits are enumerated by the inputs and the CAM LUT is set up accordingly. For instance, a 2-input, 1-output function has four possible outputs and so four bits are used to specify the function. There are obviously 16 such functions. If the number of inputs is i and the number of outputs is o, then the total number of output elements in a fully specified truth table, which is also the number of bits which need to be allocated per site in CAM-8, is $n = o2^i$ and the total number of truth tables is 2^n.

Due to bit constraints imposed by the 16-bit address space of the LUT's, only totalistic logic elements were used for this experiment. A totalistic logic function is a function of the sum of the inputs. For instance, an XOR logic gate is a totalistic logic function which is 1 when the sum of its inputs is odd and 0 otherwise. Likewise, most other logic gates in circuits are totalistic. Table 4.3 summarizes all 2-1 totalistic logic functions.

The three bits in each row correspond to three bits in the LUT which are used to

In			Out		
1	2	3	1	2	3
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	1	1

Table 4.4: A possible logic site LUT shown in *table* format. Takes the AND of 1 and 2 and outputs to 2 and 3 and takes input 2 and routes to 1. The circuit diagram for this is shown in Figure 4-3. Note that there are three directions and hence three different layers used for data transport. The kick of each layer proceeds at the end of each subscan in the directions indicated at the top of the figure

configure a particular site as a particular logic element. For instance, if a site is going to be an AND function, then the three bits would be 001. Notice that using totalistic logic reduces the total number of bits used for configuration from $O(2^n)$ to $O(n)$ where n is the number of inputs. This is because many logic functions, which are really redundant in the sense that they are routing functions, are eliminated. Notice that the symmetry in the table above is ultimately due to the fact that the logic function is invariant with respect to the direction of the inputs. While ultimately, logic and routing can be considered to be one in the same, making the distinction makes it easier to think about the circuitry. Ultimately, a pure computational substrate would be automatically programmed with no such "human" abstractions necessary. While this would be the ideal, it would be hard to understand what's going on. This ideal computational substrate is really the pure CAM-8 itself and further research should be done into automating the rule-making procedure. This was partially done in the synthesis described in the next chapter but not for the pure CAM-8, but a minimal abstraction which still divides data and configuration.

Some additional information corresponding to which inputs are used in the logic function needs to be given by orientation bits. Careful consideration should be given to this since the directions in the cellular space used here are discrete. In fact this

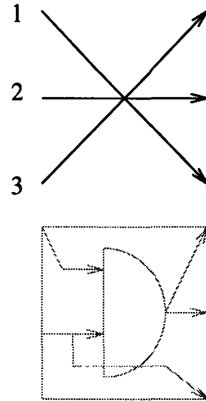

Figure 4-2: A circuit format representation of a logic cell in a rule with three dynamic data directions defined as shown above. This circuit takes the AND of the data in 1 and 2 and outputs to 2 and 3. It also takes input 2 and routes to 1. This is shown in LUT format in Table 4.4.

is really just routing in conjunction with logic as is represented by the multiplexor in the RLSC illustration. If the total number of inputs is n, then the total number of combinations of 2 inputs is $O(n^2)$, so that the number of bits needed to represent them is $O(log(n))$. All possible combinations should be used to accommodate the symmetry of the totalistic logic functions. There are still a number of bits left over for the routing to outputs. This is handled by routing the inputs to any combination of outputs in conjunction with the outputs of the logic function. In this way, wires can be passed "over" logic functions. This turns out to be a useful feature. Overall, the number of bits used for totalistic logic is $O(n)$. Figure 4-2 gives an example of a logic site. The table of all transformations (i.e., the CAM-8 LUT composition) is given in Table 4.4 for this particular logic site configuration.

Storage

There is no need for storage elements since each storage elements are just wires in time. But we already have this implicitly since each wire travels in both space and time and the entire circuit is synchronous so that hard register boundaries are not necessary.

4.1.2 The Space

The first simulation was done using the horizontal (x) dimension as the pipelined dimension and the vertical (y) dimension as the parallel dimension. The space is, by default, divided into sectors in the vertical dimension. Then, if the space is 512 by 512, and there are 8 modules in the CAM-8, the sector updated by each module is 512 by 64. Thus, the parallel logic is laid out in the vertical dimension. Different logic stages cannot go into different modules because they will be updated at the same time. Hence, this must be laid out in the horizontal dimension. However, it is generally faster to do the memory accesses sequentially within a row than within a column, arguing the need for an x-strip topology. This would speed up the circuit by several orders of magnitude. The first simulations, performed using the original y-strip topology, operated at around 5 Hz. This was sped up to about 200 Hz, using the x-strip topology. The actual circuit, once laid out, consumed 1024 sites in the pipelined dimension and 256 sites in the parallel dimension. Hence, 1024 subscans of size 256 each were performed during a single step.

4.1.3 Transport and Interaction

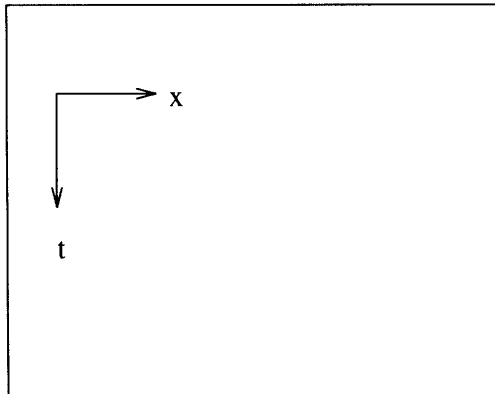

Figure 4-3: In screen coordinates, the logic is laid out in the x dimension and data is pipelined with the scan through the t dimension

Dynamic data (signals) constituted three bits, representing each of three possible directions. At each time-step, after update, the data in each of these three bits are

kicked in corresponding directions with all being kicked to the next stage. That is, in the x-strip topology, the kick vector for straight is (0,+1), for left (-1,+1), and for right (+1,+1). These coordinates are relative screen coordinates, which are used by the CAM-8 site addressing scheme. That is, the y-direction is positive downward and the x-direction is positive to the right as viewed on a two-dimensional screen (see Figure 4-3). The configuration was represented by the other 13 bits in a 16-bit site. These bits are programmed according to the rule recipe laid out in the CAM-8 section 4.1. Two bits are used for classifying the site as routing, logic, storage, or background. Six bits are used to specify the routing function exactly as outlined before. Logic is implemented the same way with three bits used to specify the totalistic LUT, two bits to specify which combination of two inputs are used, and five bits to specify which outputs are used. Storage is implemented as explained before as well. In addition, a single bit was used for tracing the signal for ease in viewing the evolution of signals on a screen.

4.1.4 Input/Output

A single edge was denoted as the I/O port. Here, individual sites were singled out as either input or output sites. The input sites were read to determine what signals they were currently holding and output sites were written with data which was sent from the host machine that controls the CAM-8. The I/O was the same as the logic I/O pins on the actual Sexium microprocessor. This consisted of an 8-bit data bus for reading and writing to and from memory, a 16-bit address bus for addressing into a memory location, and a read/write flag to tell the memory whether to read or write. The job of the host was simply to emulate a random-access memory for the Sexium simulation. This was done by allocating a 64 kilobyte chunk of memory in the host and providing some primitive operations for accessing them, akin to the addressing logic in memory chips. The 64KB chunk of memory provided the full memory address space needed by the Sexium since each memory location was one byte. In addition, a halt instruction was added to the Sexium's instruction set for convenience.

4.2 Implementation

The Sexium microprocessor was originally divided into four subcircuits: arithmetic-logic unit (ALU), program-counter/memory-address (PCMA), control, and register file. This hierarchical scheme was used here as well to lay out each of these subcircuits independently of one another on CAD/CAM-8 and then stitch them together using interface wires which connected to the input and output ports of each of these modules. These input and output ports were designated sites at the edges of each module. The input ports were on the first edge of each module as measured in the pipelined dimension and likewise the output ports were on the last edge. Thus, a bounding box was imposed for each module. Individual sites within an I/O edge were designated as individual signal pins on which interface wires were to be connected. In addition, each module was tested in isolation from the others to ensure that interface specifications could be met. The rest of this section details the implementation and testing of each individual module with the last section describing how the system was brought together to make the functional microprocessor. In the illustration of each module are described individual submodules which could be considered to be the building blocks for a module library.

ALU

The ALU consists of two registers, three 2-1 multiplexors, an adder, an ander, a neger, a shifter, one 4-1 multiplexor, and one 5-1 multiplexor (see Figure 4-4). The final circuit is illustrated in Figure 4-5. Each of these submodules was laid out and tested independently. The mux's are each taken from the same mold, shown in Figure 4-6. Figure 4-7 shows a close up of one of these mux's as it appears in the actual circuit. The adder is the same one illustrated before, for 8 bits. The ander takes two bytes as inputs and outputs the bitwise "and" of them. The neger takes one byte as input and outputs the bitwise inverse. The shifter takes one byte of input and shifts it by 1. In addition, it takes a rollflag from the control as input which tells it whether to wrap around the end bit during a shift. Register's are different from the storage element's

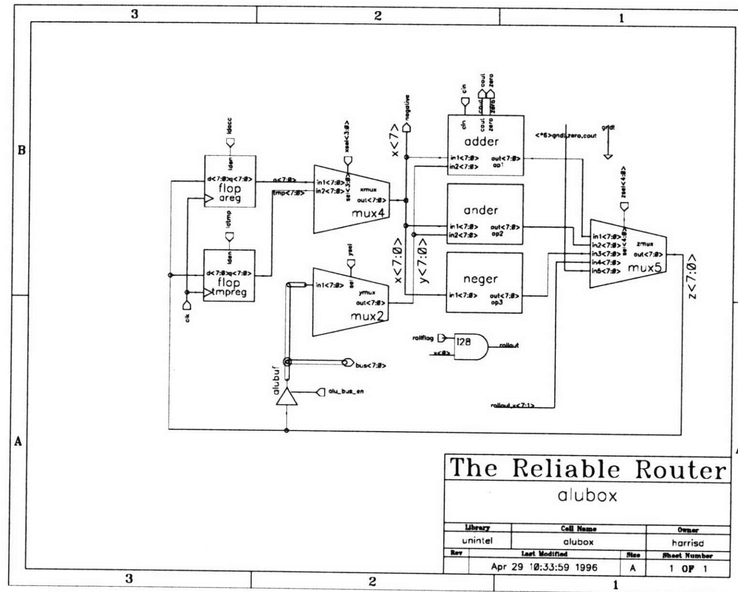

Figure 4-4: A schematic of the ALU from the original VLSI project

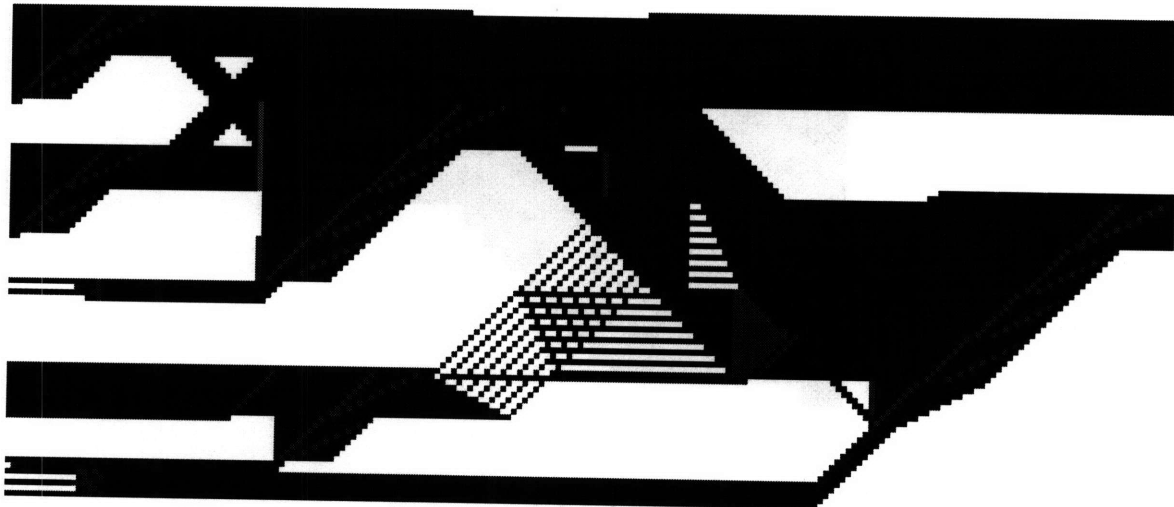

Figure 4-5: The CA layout of the ALU

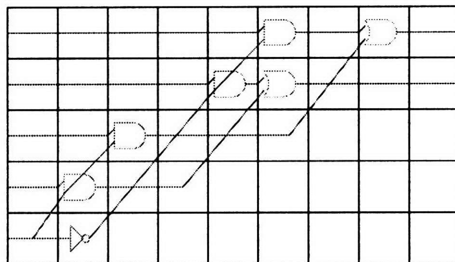

Figure 4-6: The embedding of the multiplexor in the cellular space. The configuration information for each cell can be obtained by enumerating the properties of each type of cell as was done for the 3-bit adder before. Inputs a and b are 2 bits each and if select is 0 then o gets a, else o gets b

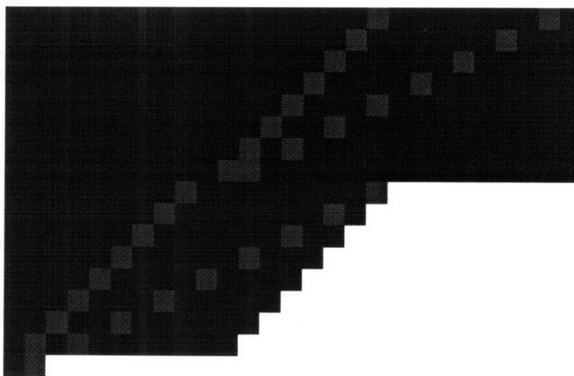

Figure 4-7: A close-up of one of the 2-1 8-bit multiplexors in the actual CAM-8 circuit layout. Notice the similarities with the 2-bit mux from the previous figure.

mentioned before. These register's have read/write signals. They are simply group's of 8 wires which go all the way around the torus. Signal's can be read to or written from them just as in any normal register. They hold their value indefinitely since signals just go around and around the space until replaced by something else during a write.

4.2.1 Control

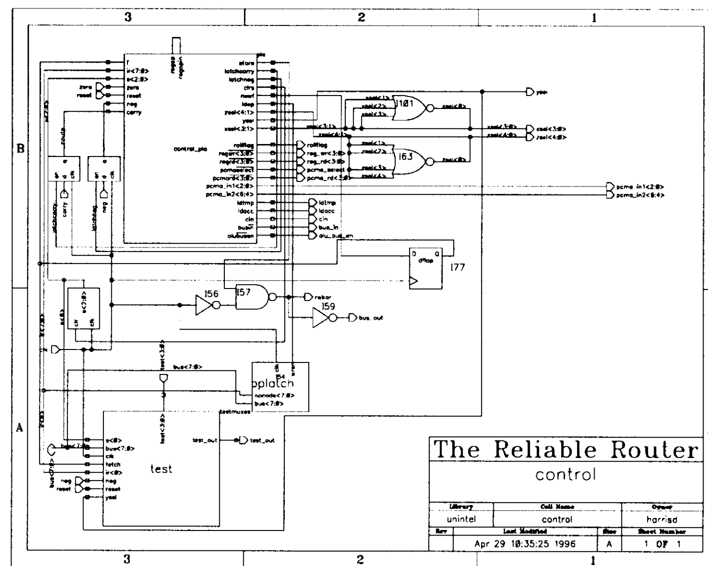

Figure 4-8: A schematic of the control

The control circuitry was responsible for converting opcodes in conjunction with various internal flags into control signals for each of the other modules. This is the "brains" of the microprocessor. The control was automatically generated using a PLA generator in CAD/CAM-8. The control was specified by a PLA file which expressed each output as an equation in canonical form. This was read in by CAD/CAM-8 to form the logic-array using AND's and OR's just like a normal PLA. Each input signal and their inverses are placed in adjacent positions in the parallel dimension. A wire is dragged out from each input. For each canonical equation product term, an AND is placed on the input line and redirected to another line which does the OR of all the products for a particular output. This is illustrated in Figure 4-10. The control

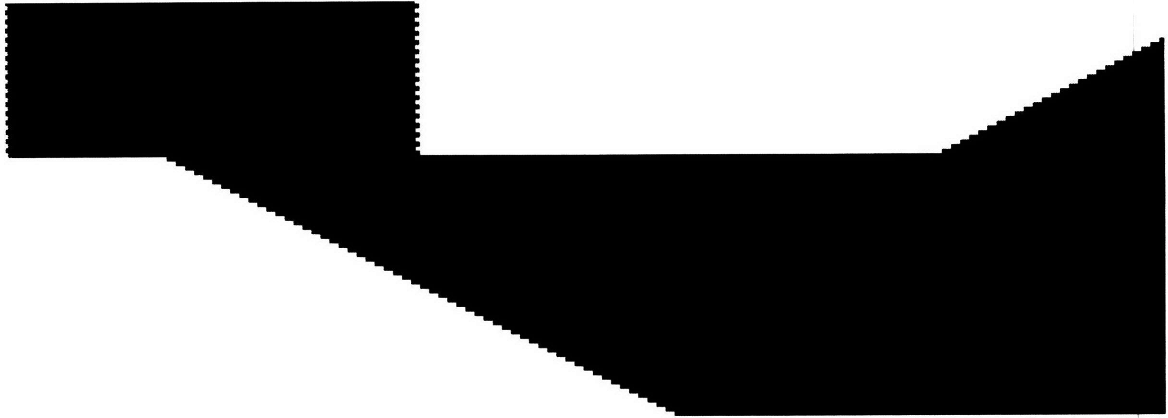

Figure 4-9: The CA layout of the control

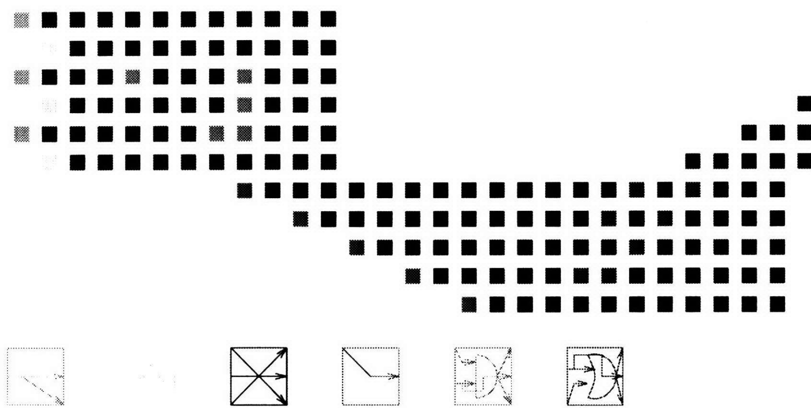

Figure 4-10: Generic embedding of a PLA in cellular space

is shown in Figure 4-8 and Figure 4-9.

4.2.2 PCMA

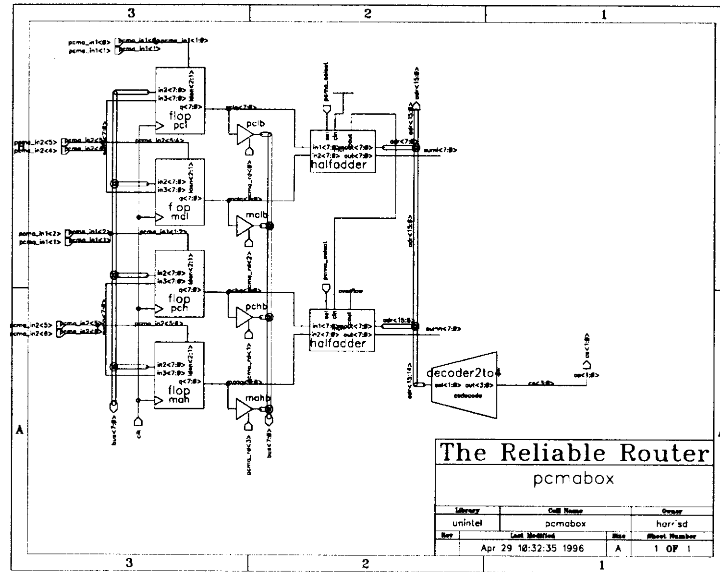

Figure 4-11: A schematic of the PCMA

The PCMA consists of 4 3-1 mux's, 4 registers, two 2-1 mux's, two halfadder's and a 2-4 decoder. This is illustrated in Figures 4-11 and 4-12.

4.2.3 Register File

The register file consists of four general purpose byte registers used for internally storing temporary values when needed for quick access. These registers are laid out as was explained above. The register file is illustrated in Figures 4-13 and 4-14.

4.2.4 Putting It All Together

Each of the submodules was individually tested using a probe program written in Forth for reading and writing test vectors to CAM-8. The probe inputs were read from probe input files (PIF's) and the outputs were written to probe output files (POF's). Comparing the POF's with the expected outputs for various known test

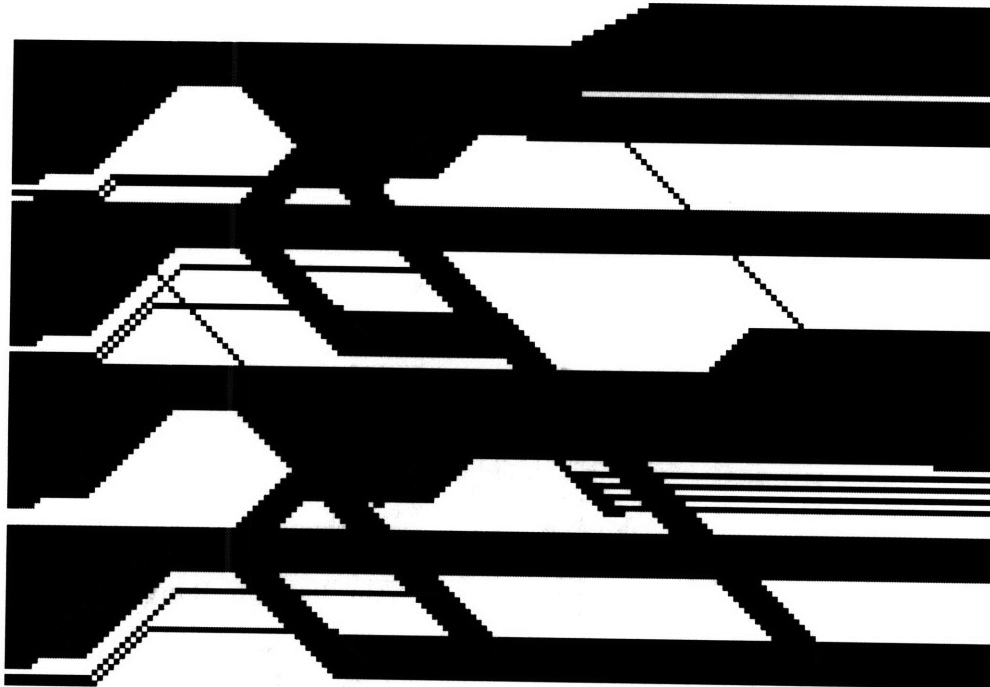

Figure 4-12: The CA layout of the PCMA

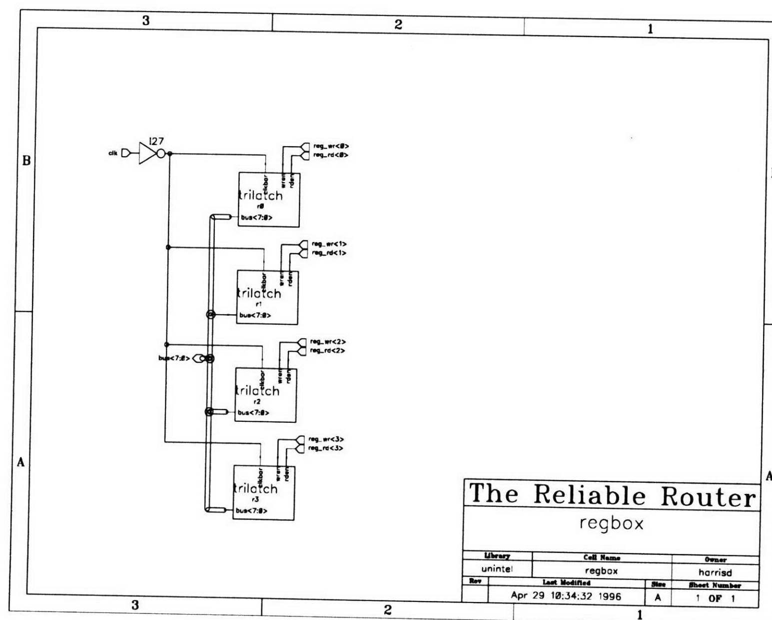

Figure 4-13: A schematic of the register file

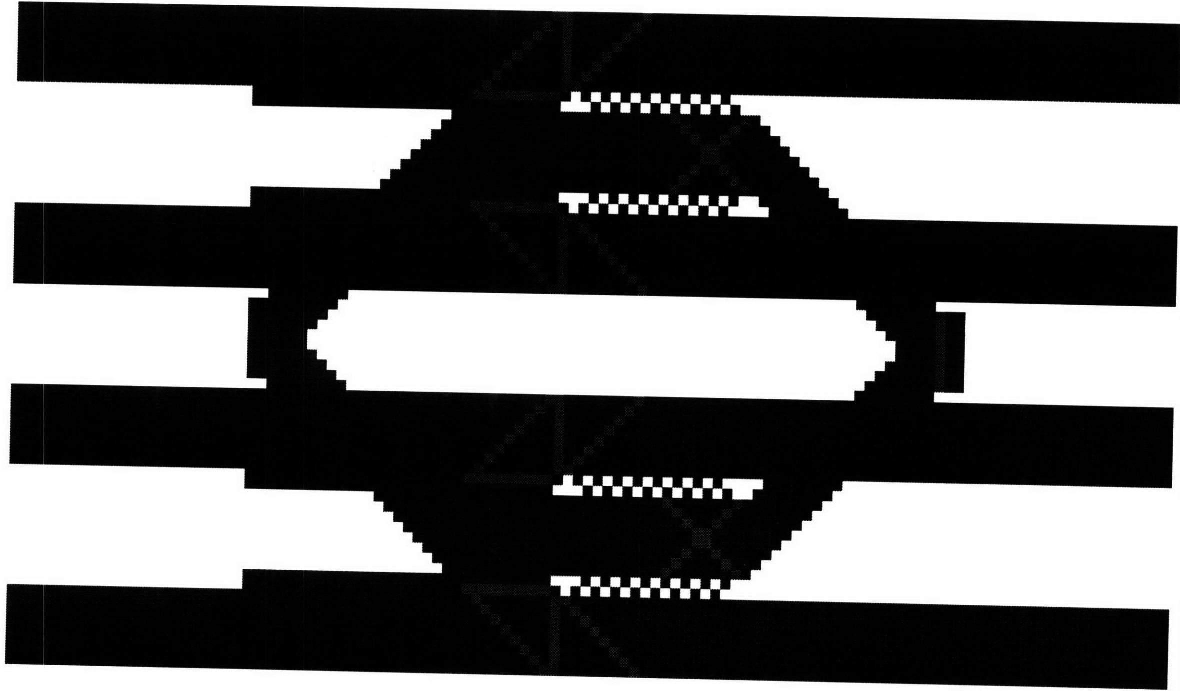

Figure 4-14: The CA layout of the register file

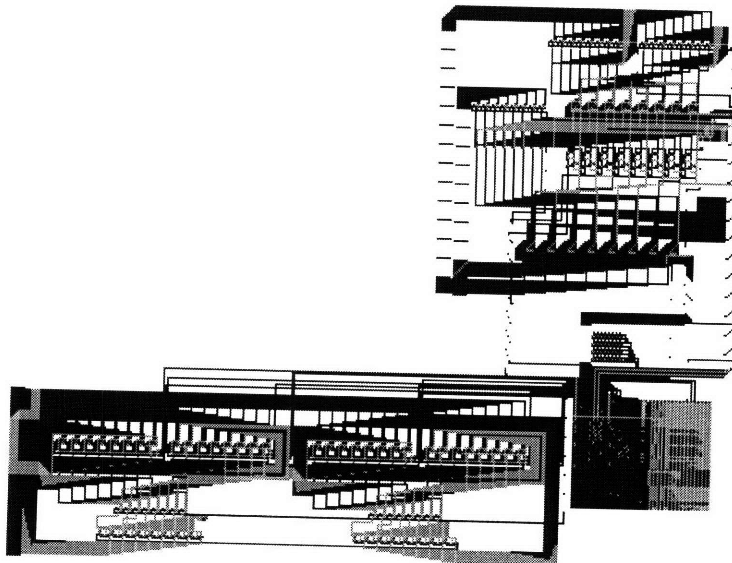

Figure 4-15: A schematic of the sexium

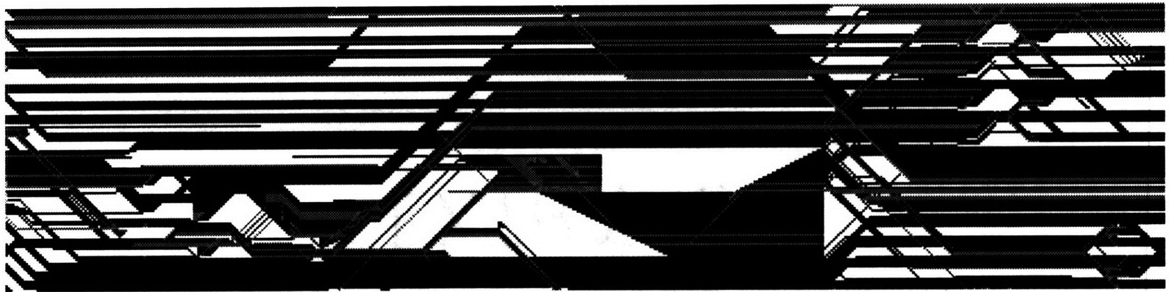

Figure 4-16: The CA layout of the sexium

vector inputs provided the necessary verification. The PIF's specified I/O ports for each submodule in terms of the coordinates of those designated sites. Using the information in the PIF's, intersubmodule connections were made using wires which were terminated on these ports. The entire circuit was then tested in this same way. After verification, the circuit was now ready for higher-level testing. The final Sexium circuit is illustrated in Figures 4-15 and 4-16.

4.3 High-Level Implementation

After verification of test vectors, this detail was abstracted to the assembly level for higher-level implementation. An assembler was implemented in Forth for compiling assembly language files to the opcodes for CAM-8. The assembler interpreted the opcodes and placed them into the designated program memory. The I/O was then implemented as explained in the Input/Output section above. The address specified which location in memory to read or write next. If a location was being written, then the data taken off the data bus was put into that location. If a location was being read, the host sent the information in that memory location back onto the bus for subsequent processing by the Sexium.

4.4 Discussion

Three programs were implemented on the Sexium for demonstration. The first program was called `sum`. This program read four adjacent memory locations specified by the program and summed them putting the sum into the next two adjacent locations in memory. The program was supplemented by a Forth program which allowed the user to input the four numbers from the terminal. The next program was called `bubble`. This program sorted an arbitrary number of locations in memory using the famous bubble sort algorithm. Again, a supplemental Forth program was written to allow the user to specify the numbers to be sorted and placed them in memory for the Sexium. The third program was `Tic-Tac-Toe`. This program did just what its name implies: it played a perfect game of tic-tac-toe. It was supplemented by a Forth program which drew the board and got moves as input from the user.

All of these programs demonstrated the "high" speeds at which the Sexium simulation could be executed. Depending on whether or not each update was displayed on the CAM-8 monitor, speeds of up to 200 Hz were achieved. A typical Verilog simulation of the same-size circuit would probably run at under 0.1 Hz. This points the way towards using CAM-8 as a digital logic simulation tool for the IC industry. More importantly, it demonstrates how the CAM-8 can be used to model a particular class of logic circuits well. The CAM-8 demonstrates the feasibility of using cellular automata as building blocks for digital logic.

Chapter 5

Logic Synthesis

There were two approaches taken to the logic synthesis in this thesis. Initially, a logic CAD tool, called CAD/CAM-8 was implemented in order to lay out the circuits by hand. Of course, manual layout of large circuits, was too tedious. For this reason, logic synthesis for the CAM-8 logic simulator was explored.

5.1 CAD/CAM-8

Layout was originally performed using a tool called CAD/CAM-8, written with the Tcl/Tk Toolkit [35]. This tool has the ability to represent an arbitrary gate array architecture, which is specified by an input file called the *model.* This model specifies an arbitrary dimensionality, functionality, topology, and temporal partitioning of the space. The space is made up of discrete cells, which map to the sites in CAM-8. The dimensionality simply specifies the number of dimensions and the size of each dimension. In the programmer's model, the dimensions are orthogonal so that the space of cells is a Cartesian lattice. The dimensions include both space and time. The functionality is the cell configurations that are available. Each cell has a value which determines its' configuration. The model of abstraction used for the cells is the RLSC node model, discussed in Section 4.1[1]. The topology specifies the data transport or interconnect of the cells, which can be considered as a Bravais lattice of a crystal [2]. This Bravais lattice is always embedded in the Cartesian lattice, where

the Bravais lattice is simply an abstraction.

These are all the main parameters for CAM-8. Once these are specified, the cells can be laid out by hand by simply specifying the cell functionalities by pull-down menus. The cells are laid out on a discrete canvas of cells, each of which is several pixels long on each side. By clicking the mouse button over a distinct cell coordinate, where the coordinate is given in terms of the multidimensional spacetime Cartesian lattice, and then specifying the functionality and interconnect of the cell, that cell is configured. Included is the ability to lay out circuits hierarchically using functional modules which are collections of cells with specific interfaces which could be placed anywhere in the spacetime discretum. Thus, functional units of supercells can be created. These supercells can also be functionally transformed into the input-state space. In other words, the tool is capable of functionally abstracting the modules so that the functionality of multiple cells can be compressed into single cells for use in higher-level simulation. Other cellular operations include translation, rotation, inversion, magnification, and scaling. Many other mundane capabilities are included as well like printing, cut-and-paste, saving, etc. Some synthesis operations are included like PLA synthesis. Automatic level placement and routing of multi-level logic netlists was not included.

The CAD/CAM-8, while initially a very useful tool for rapid prototyping of gate array architectures and logic simulations, was limited in capability. Because it was based on a simple scripting language, it was relatively inefficient. For instance, it was hard to layout large circuits because of memory constraints. Also, as was mentioned in Chapter 1, large circuits are hard to lay out. However, making and using CAD/CAM-8 provided some very useful lessons and ideas which could be applied to making a "real" logic synthesis tool for automatic layout of logic circuits.

5.2 The Synthesis Environment

The first lesson learned from CAD/CAM-8 was that we needed to implement a tool for automatic rather than manual circuit layout. An initial version of such a layout tool was created for this thesis, called the CALS program. The CALS program was used in conjunction with other standard logic synthesis tools in order to create an initial synthesis environment for the CAM-8 logic simulator.

When logic simulation is being done by emulating a gate array, it is possible to perform logic synthesis using standard tools. It would be a simple matter, for instance, to program the CAM-8 to emulate a standard FPGA architecture, such as the XC6200, and then use some standard tool, such as Synopsys to synthesize down to the architecture. Then, it would just be a simple matter of emulating the gate array to simulate the logic. This approach has not been tried yet, but it would not be hard to do. On the other hand, with a made-up gate array, such as those discussed in Chapters 2 and 4, the logic synthesis tools have to be configured to layout the circuit onto those gate arrays.

The approach taken here is summarized in Figure 5-1. The Berkeley SIS tools are used for technology independent logic synthesis and optimization. From this, a netlist is created. The netlist has the functionality of the circuit embedded in it. The topology and functionality of the gate array are specified separately in the form of a model file as outlined in the last section on CAD/CAM-8. With the netlist and model file, the CALS program is able to generate an initial pattern of site information[1], which has only the configurations, and a lookup table which are then input to the CAM-8 simulator. Optionally, if a state transition diagram is also input to the CALS simulator, probe input and output files are created for testing the CAM-8 circuit simulation using test vectors. The details are discussed in the next two subsections which describe in more detail the two main phases of the logic synthesis for CAM-8.

5.2.1 Technology Independent Logic Synthesis

[1]Which essentially serves the same purpose as a configuration bitstream for an FPGA

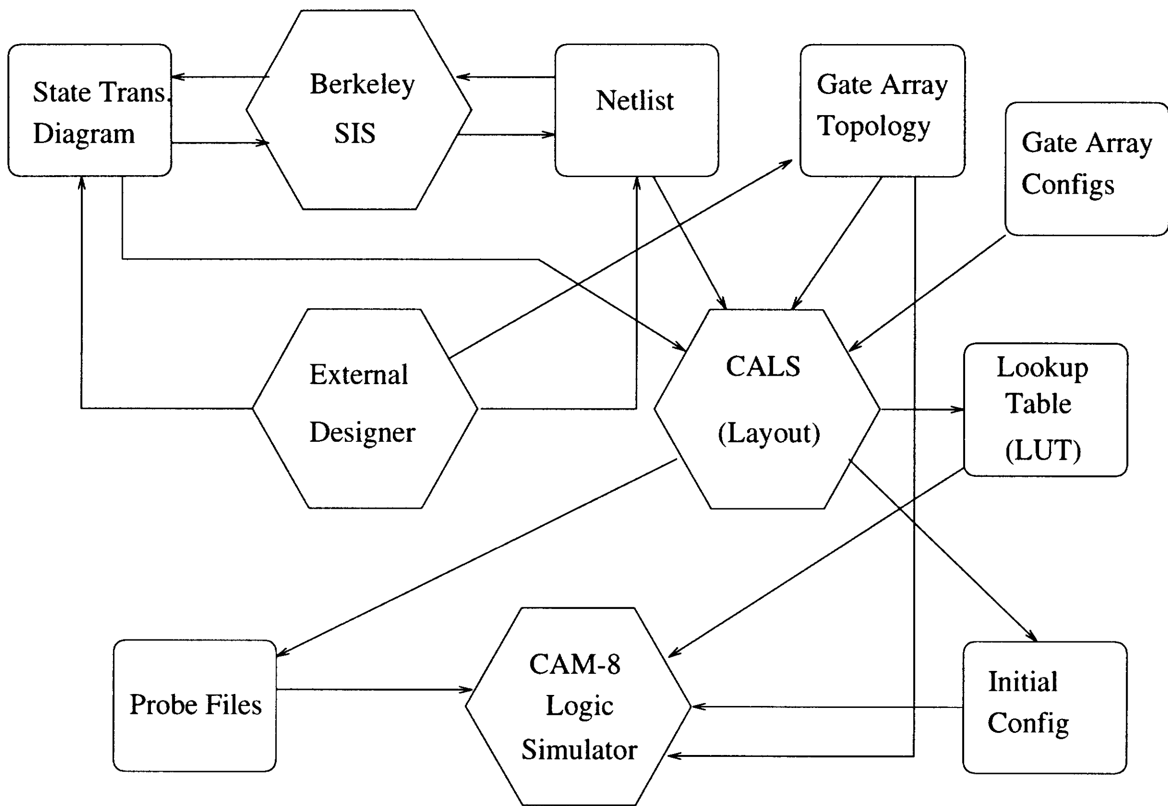

Figure 5-1: Logic synthesis for the CAM-8 simulator

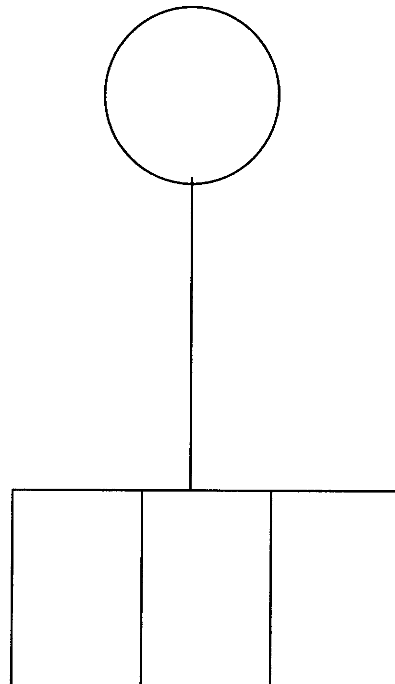

Figure 5-2: Node and its net in a netlist

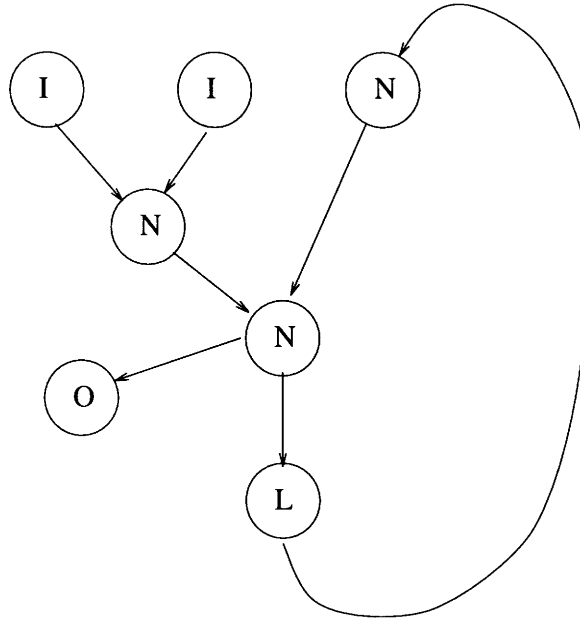

Figure 5-3: A BLIF netlist: I - inputs, N - internal nodes, L - latch, O - output

The goal of this part of the logic synthesis is to create a logical circuit description in the form of a *netlist*. A netlist describes a logic *network* which is basically a list of *nodes* and the connections between them, which are called *nets*. This network completely describes the logical circuit. The netlist format chosen was the Berkeley Logic Interchange Format (BLIF)[pp. 33-42][38]. In this format, each node is an input port, output port, latch, or an internal node. Input ports are the entry points for logical signals into the circuit, so that they are always in the first level. Output ports are the exit points for logical signals. Any feedback path in the circuit must have a latch in it. Internal nodes are all other nodes (Figure 5-3).

Each node has a function associated with it which maps its n inputs to its 1 output. The maximum number of inputs to a node in a network is known as the *support*, m. Every node, except for possibly some outputs, has a net associated with it. The net is the logical output of that node. The net may fan out to other nodes, to which it is an input (Figure 5-2).

This standard format was chosen for several reasons. First of all, the logic gates are completely specified. This means that each node has associated with it a list of *cubes* which are the values of the inputs which give an output of logical one.

These values can be zero, one, or don't care. That is, each node has its function completely specified in the form of a compressed truth table. This precludes the need to have to have additional input files to the CALS program in the form of a gate library, which would otherwise specify the gate functions. Second, the format is completely technology independent. Third, the format does not have any extraneous information, such as drawing information. Fourth, the format specifies latches, inputs and outputs which are necessary bits of information for the CALS program to layout the circuit. Fifth, it has the capability for hierarchical circuit layout. Sixth, there are some programs which convert high-level circuits descriptions to BLIF format. The disadvantage is that it is limited to n-to-1 functions. Thus, other gates, like reversible gates can't be specified. This is supposed to change in future versions of the format specification. Other formats may be more ideal for other situations. For instance, if the CAM-8 were emulating a Xilinx XC4005 FPGA, the most appropriate format would have been the Xilinx Netlist Format (XNF).

There are many circuits already specified in BLIF and KISS format which can be obtained from any number of sources. The Berkeley SIS tools can be used to generate BLIF files from state transition tables (STT's) specified in KISS format (see Appendix B). Berkeley SIS is a suite of different smaller tools such as state encoders, state minimizers, technology mappers, PGA synthesis tools, state extractors, optimizers, etc.[38] Berkeley SIS is basically capable of state transition graph (STG) and netlist translations and manipulations. The input is either an STT in KISS format or a netlist in the Berkeley Logic Interchange Format (BLIF). If an input is an STT, Berkeley SIS produces a BLIF netlist with the STT in it along with the coding of the circuit in relation to the STT. In particular, the mapping of inputs, outputs, and latches to the data in the STT is provided in the BLIF netlist.

There are two approaches which could be used for the technology-independent logic synthesis. SIS has the capability to map circuits using a library which is given as input. SIS also has the capability to synthesize circuits for for both multiplexor and table-lookup based PGA's. In fact, it doesn't matter which approach is used to do the logic synthesis in this part of the project. This is because, the only result that is

gotten here is a BLIF netlist which contains all the information that is needed about the circuit functionality and connectivity. However, it is generally desirable that this netlist be optimized. Optimization would depend on one or several cost functions. In our case, what is desirable is a minimal number of nodes, circuit depth and width. These are all interrelated of course, but it is desirable to minimize everything because this leads to circuits with less area which also means that they will be simulated faster. The number of nodes can be reduced by using block count minimization and the depth of the circuit can be reduced by retiming the circuit.

5.2.2 The Cellular Automata Logic Synthesis (CALS) Program

The main contribution to synthesis in this thesis was to design and implement a program which performed the circuit layout, the CALS program. For the purposes of this project, the CALS program assumes a virtual gate array so that it lays out spacetime circuitry. A more general-purpose CALS program would be able to layout circuits on a non-virtual gate array as well.

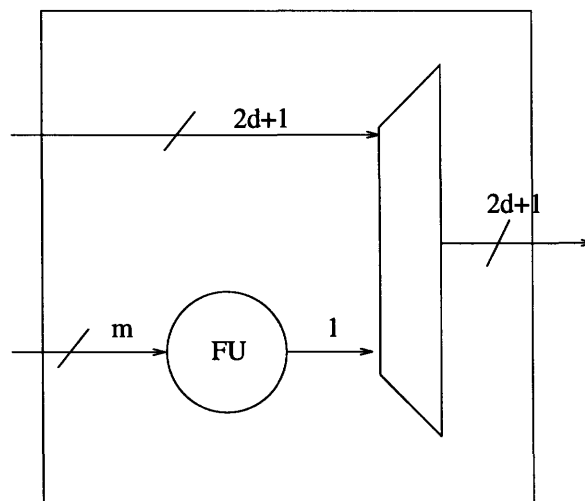

Figure 5-4: CALS cell model in d dimensions

Basically, the CALS program accomplishes three main tasks: placement, routing, and lookup table generation. The specification for the program is that it is given

a BLIF netlist and model file, which is in the same format as that outlined in the previous section on CAD/CAM-8. So, internally, it has a representation of the circuit that is to be laid out, and the gate array on which the circuit is to be laid out. However, due to the time constraints of the project, the CALS program that was implemented cannot take a model file as input. Hence, it assumes a default internal representation of the gate array with nearest-neighbor interconnect only. Thus, there are $2d+1$ inputs to a cell in d *spatial* dimensions: the nearest cells and the cell itself. The cell can route any input to any output as long as an output is only assigned to at most one input. This is used for routing. In addition, there can be a maximum of one node at any cell. The node can take up to m of any of the cell inputs. Obviously, $m \leq 2d+1$ since the number of inputs to a node must be less than or equal to the number of inputs to a cell. We are also limiting ourselves in this default model since in many cases, up to s nodes can fit into a cell, where $sm \leq 2d+1$. But this, again, could be specified by a model file. The only real constraint is that there is no conflict at an output. The model of a cell is shown in Figure 5-4. It is very similar to the XC6200 cell architecture shown in Chapter 2 for the 2-D case, except that it does not include the flyover routing. But the program can still lay out a circuit in one, two, or three dimensions. It also assumes wrap-around at the boundaries. The output of the CALS program is a pattern file and a LUT for CAM-8. Optionally, the CALS program can produce probe input and probe output files for probing the CAM-8 simulation with test vectors.

Terminology

In order to explain how the CALS program works, the terminology must be clear to avoid confusion. Cells and wires refer to the gate array with the same definitions (Chapter 2, Section 1). A level still refers to a time-slice of a virtual gate array (Chapter 3). Likewise, nodes and nets refer to the netlist with the same definitions. An edge is a single connection between two nodes, which is a portion of a net. A path is a group of connected wires in the gate array. The routing portion of the program essentially maps edges onto paths as will be seen.

Placement

The purpose of placement is to assign the nodes to cells in a compact way in space and time. That means that all of the nodes should be as close as possible to each other, so that the paths which implement edges will be as short as possible.

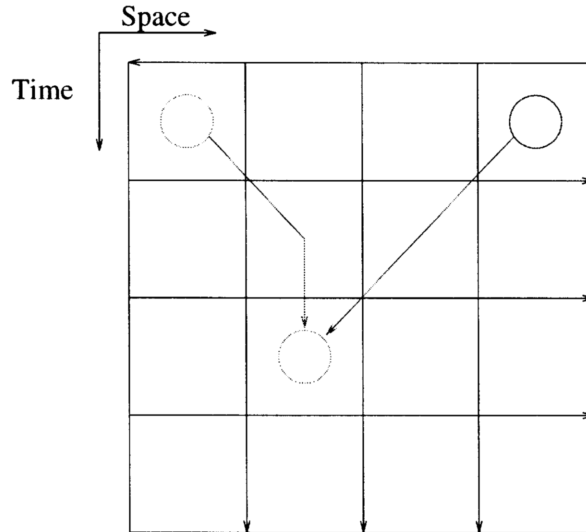

Figure 5-5: Assignment of paths to edges illustrating the scheduling constraint in the placement phase

The levels of nodes must obey certain constraints in order for them to be routable (see next section). Assume that node A depends directly upon node B. Node A is assigned to a cell s_1 on level t_1 Node B is assigned to a cell s_2 on level t_2. Then, the constraint is that $d(s_1, s_2) \leq t_2 - t_1$ where $d(a, b)$ is the distance from a to b, which is the minimum number of wires to get from a to b. That is, node B should be assigned to a later level than node A and a signal from node A to node B must be able to get to node A in that time difference, given that it can use any of the wires available. Of course, this is a necessary but insufficient condition for an edge from node A to node B to be assignable to a path. Figure 5-5 shows an illustration of this for a nearest-neighbor 2-D spacetime mesh.

Initially, the nodes are randomly assigned to cells. The input nodes are placed first so they are assigned to level 0. Since there is no input constraint to the inputs, they remain on level 0. Latches are also assigned to level 0. The inputs to latches

are temporally wrapped around, thus simulating the feedback. Then a force-directed algorithm is applied to the nodes. This algorithm simulates a network in which the nets are springs or gravitational forces between the nodes. The nodes that are connected are thus "attracted" to each other[25]. Note that this can only be done assuming that there is some distance metric $d(a, b)$. In the case of nearest-neighbor interconnect $d(a, b)$ is the taxicab distance[21].

Routing

Given a placement, the next thing the CALS program does is routing. The routing assigns edges to paths, thus connecting the cells to which connected nodes were assigned. For the case of having a generic gate-array specification, complete with the topology, the best way to do the routing would have been to use a maze-routing algorithm such as Lee's algorithm[22]. This would have ensured that any possible routing was found. This could have been done using the normal sequential algorithm or possibly any of the parallel variants implemented on CAM-8 itself. Because of the time constraints of the project, only a simple annealing algorithm was implemented.

Since an internal representation of the gate array based on nearest-neighbor routing is assumed, the path from one cell to another is determined by the taxicab route. Therefore, the routing is done using a greedy annealing algorithm. The algorithm routes each node sequentially. It starts out by going through all the output nodes of the node, and finds the shortest path that is not *blocked*, to the output node. A path is blocked if a particular wire is already used by another path emanating from a different node. It does this in a greedy way, so that the first path it finds is the path that is used to connect the two nodes. If the routing encounters blockage, then with a probability $P = e^{-1/tN}$ where t is the temperature in the annealing loop and N is the number of unrouted nodes, the entire net which is causing the blockage is unrouted. An unrouted node is a node whose net has not been routed. Each annealing pass goes through all unrouted nodes sequentially and multiplies the temperature t by a fixed factor $f < 1$ so that the temperature is exponentially decreasing.

Table and Pattern Generation

Often times, there are not enough bits at a site to fully model a particular gate array cell. For instance, there are not enough bits per site to implement the XC6200 cell as was shown in Chapter 2. In the cell used in the model gate array in CALS, we can estimate an upper bound on the number of configurations that are really necessary. Of course, there are only 2d+1 connections to a cell with nearest-neighbor connect: where d is the number of dimensions. So, there are $(2d+1)2^{2^{(2d+1)}}$ functions of inputs to outputs, i.e., configurations. This bound can be made tighter by taking into account the number of actual configurations that the underlying cell is capable of in our default gate array model. As can be seen in Figure 5-4, the number of cell configurations is $(2d+2)^{(2d+1)} + \sum_{i=0}^{m} \binom{2d+1}{i} 2^{2^i}$ where m is again the support. It is obvious that the configuration space increases exponentially with the number of dimensions and the support so that the number of bits per site becomes insufficient to model a cell beyond a small number of dimensions and support without resorting to subcells.

There are ways to get around this. One way is to model the entire cell using more than one site. Then the gate array takes up more space but it is fully functional. This solution is also basically equivalent to using subcells since that is the way that they work. Another way is to not model the cell entirely. In particular, for any circuit that is laid out using a particular gate array model, there will always be some configurations of a gate array that are unused. If this is the case, there is no reason to emulate these configurations and thus, the configuration state space can be reduced. This second approach is the one used in the CALS program.

As was mentioned before, the functionality and topology of the gate array are fully specified to the CALS program in the netlist and model files. Since, in this implementation, the model file was left out, only functionality info is variable. In any case, the CALS program has a fully internal representation of the gate array. However, the CAM-8 does not implement this gate array in its entirety.

Rather, the CAM-8 can be programmed with a particular topology in the usual

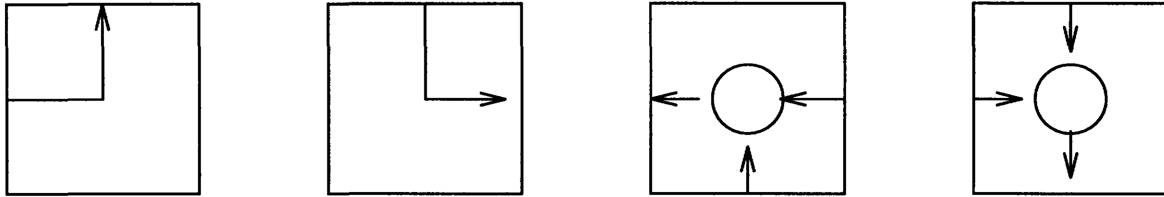

Figure 5-6: Four compatible cell configurations in a 2-D gate array

way, thus mimicking the topology specified in the model file. The LUT for CAM-8 is externally generated using the CALS program. The CALS program does the layout first, i.e., the place and route. With this information, it can then count the number of configurations that get used. These are enumerated as separate states. A recursive state minimization algorithm is then applied to these states. This algorithm applies state minimization to fixed size sets of states, reducing the number of states in each recursive pass until the states can be reduced no further. The state minimization was done using standard techniques implemented in the Stamina library which came with Berkeley's SIS. The enumeration eliminates unused configurations, while the state minimization makes it possible to have even more configurations by combining "compatible" configurations which aren't combined in the gate array model (Figure 5-6). The final states are again enumerated. Then, the LUT is generated by going through all the states, and cycling through all possible input signals and generating the output signals for that configuration. The signals are of course assigned to the data bits. The state number is placed in the configuration bits of a site with that configuration, with the appropriate invariance in the LUT. That is, the configuration of the circuit is put into the static configuration plane(s).

Minimizing the number of configurations is done for the same reasons that we use spacetime circuitry: to minimize the number of resources that are used. In particular, we only use the resources that are necessary. In the case of spacetime circuitry, only those functions which were done in parallel were actually simulated to be in parallel. In this case, only the configurations that are needed are used. This is analogous to block count minimization, the process of minimizing the number of nodes or cells

used to implement a circuit. We are actually minimizing the number of configuration bits per site logarithmically. This would not seem to provide much benefit, but it actually does make it possible to synthesize to a gate array with more configurations than would otherwise be allowed by the number of configuration bits in a fixed size cell.

As was mentioned in Chapter 1, CAM-8 has the ability to time-share its lookup-table among transition functions, so that CAM-8 has the capability to change its transition function with time. Another way to look at it is that the state space is being time shared over the LUT'S so that configurations mean different things with different LUT's. This is another way to use the state space in a conservative fashion, and one that should be explored. In particular, if the number of cell configurations is more than is possible in the CAM-8, even after minimization, then using a virtual LUT approach might help. It could be incorporated into the CALS program so that more than one table could be generated. However, it was not attempted in this thesis due to time constraints.

Test Vector Generation

For circuits whose STT is not known *a priori*, test vector generation is up to the logic designer. If nothing is known about the circuit, brute force is probably necessary. This would involve extracting an STT from the circuit by putting in all possible vectors and getting out the state and output. This still involves knowing where the input, output, and latch nodes are. Since the BLIF netlist has at least this information, this is still plausible. If some things are known about the circuit, such as the separation of the instruction control and data lines into a microprocessor, along with its instruction set, then more specific testing can be done. This is perhaps one of the greatest strengths of CAM-8 over software simulators. Since the CAM-8 is capable of simulation of most circuits in the KHz-MHz range, it should be possible to do very fast extraction of data about the circuit. This would be useful for such things as reverse engineering.

Since the CALS program has information about the input, output, and latch nodes from the netlist along with information about their placement from the layout, this

information can be used to specify the IO probe points for a circuit probe. Thus, it is easy to specify these points whether or not an STT is known. If an STT is known, then it is also possible to provide a probe input and probe outfile file (PIF and POF). This has the same information as the STT in the netlist, except that it includes the specific placement of the entry nodes so that a probe program can probe the circuit. In particular, the coordinates of the IO and latch cells along with the layers in which the data are to be written to or read from for each of those cells, is given. This is obviously layout-dependent information. These probes were used to verify the functionality of the synthesis and simulation programs (See Appendix B).

Chapter 6

Results

Early results were obtained for the logic synthesis using the gate array model outlined in the previous chapter. These results were obtained by synthesizing circuits in the standard MCNC benchmark. The results show that the CAM-8 synthesis environment is working correctly in its current state.

6.1 Technology Independent Synthesis

In order to verify the circuit layout functionality, circuits whose state-transition table was known *a priori* were used. A script was used to map the STT onto a LUT-based PGA architecture (Appendix B). It mapped it to a library of all possible gates with support m. It also performed various optimization routines such as block count minimization and retiming. The script made it possible to specify the support for the network.

For the purposes of this project, a support of 2 was used in order to ensure that the circuit could be laid out given the simple model of a gate array assumed by the CALS program. Once the CALS program is able to accept a model file, it should be able to experiment with larger values of the support, particularly since then we can assume that there is more than nearest neighbor interconnect in the gate array, which is a pretty limiting assumption. Nevertheless, with this assumption, results are obtained which show much promise for the future of the CALS program.

6.2 Layout Data

Layout was attempted for one and two-dimensional spatial circuits with the extra temporal dimension. Because of the nature of the greedy algorithm and the limited connectivity of nearest-neighbor interconnect, it is generally hard to achieve the maximum support possible for given d. Data about the layout was taken to get an initial glimpse of what techniques will work here and what algorithms should be used. Data about each of the main phases of the layout were obtained. In addition, data about the simulation of each of these was taken. In particular, all the circuits were verified to be working properly by probing all the circuits with the vector information generated by the synthesis process.

6.2.1 Placement

The information gathered about placement is summarized in Table 6.1. n_n is the number of nodes, n_f is the number of force-directed iterations applied to the randomly placed circuit, l_i is the number of levels before force-directed placement, d_i is the average spatial distance between nodes before fd placement, l_f is the number of levels after fd placement, d_f is the average distance between nodes after fd placement, and h is the depth of the circuit.

It is evident that the force-directed placement does indeed compact the spacetime significantly from looking at the difference in average distances and number of levels before and after the application of fd placement. However, this generally was not done often because the compaction was so good that it was very hard to route these circuits with our simple routing algorithm.

It is evident from Figure 6-1 that the number of nodes is a pretty good indicator of the number of levels in the layout. The number of levels is actually smaller which is good because it means that some of the nodes are being evaluated in parallel. It is desirable to minimize the slope of this curve as much as possible to maximize the parallelism of the stages in the circuit.

Figure 6-2 shows that the number of levels roughly corresponds to the depth

Name	n_n	n_f	l_i	d_i	l_f	d_f	h
bbara	44	100	50	3.933333	16	1.253333	13
bbara_bbtas	93	0	96	4.023392	96	4.023392	22
bbsse	111	0	98	4.086294	98	4.086294	22
bbtas	23	0	54	4.594595	54	4.594595	10
beecount	26	1	38	4.473684	18	2.131579	9
cse	165	0	109	4.078689	109	4.078689	30
dk14	82	1	116	4.053333	64	2.026667	29
dk15	59	2	137	4.257143	62	1.790476	28
dk16	188	1	240	4.065934	130	2.733516	50
dk17	50	0	99	3.800000	99	3.800000	24
dk27	22	100	29	3.945946	15	1.648649	7
dk512	47	100	56	3.670588	30	1.235294	19
ex1	198	0	215	8.254237	215	8.254237	25
ex4	73	0	63	6.148760	63	6.148760	9
ex6	75	100	63	3.976744	31	1.713178	17
f208	96	1	70	4.122699	44	2.558282	16
keyb	167	0	121	2.974441	121	2.974441	39
kirkman	161	0	84	4.038328	84	4.038328	18
lion	14	0	32	4.095238	32	4.095238	9
mark1	92	0	74	4.223776	74	4.223776	17
mc	27	1	36	4.179487	17	2.025641	8
opus	75	1	67	4.284615	42	2.500000	16
pma	188	0	133	4.138329	133	4.138329	36
s1	153	0	100	3.996416	100	3.996416	28
s208	96	1	70	4.122699	44	2.558282	16
s27	33	1	41	3.981481	24	2.462963	10
s386	103	0	62	3.955801	62	3.955801	16
s420	89	2	56	4.157895	29	2.533835	12
s510	277	0	224	7.838966	224	7.838966	24
s820	261	0	123	5.913043	123	5.913043	18
s832	285	0	157	5.990157	157	5.990157	21
sse	111	0	98	4.086294	98	4.086294	22
styr	375	0	303	8.073919	303	8.073919	39
tav	27	100	23	4.256410	15	2.128205	6
tbk	176	0	209	6.174174	209	6.174174	38

Table 6.1: MCNC Placement Info

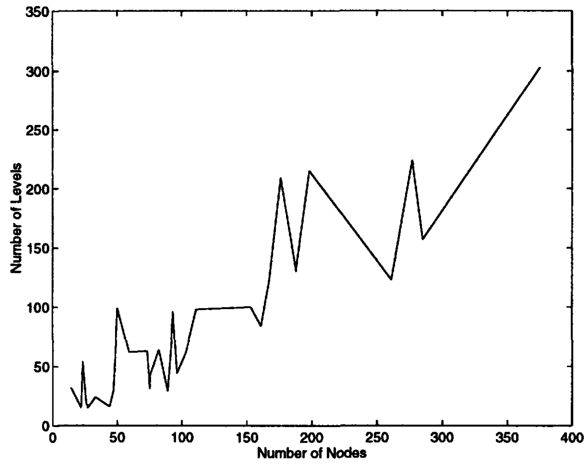

Figure 6-1: The number of nodes versus the number of levels after placement

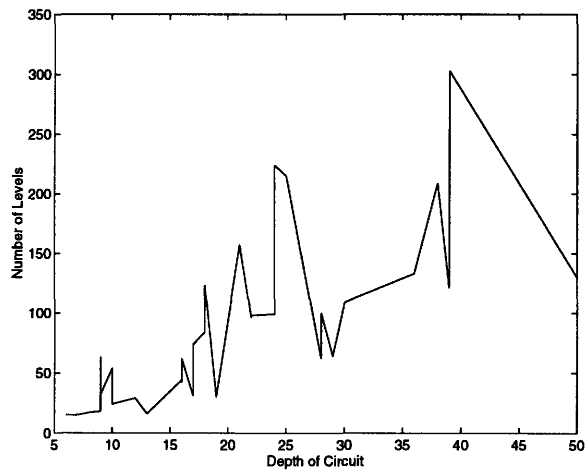

Figure 6-2: The number of levels versus the depth of the circuit

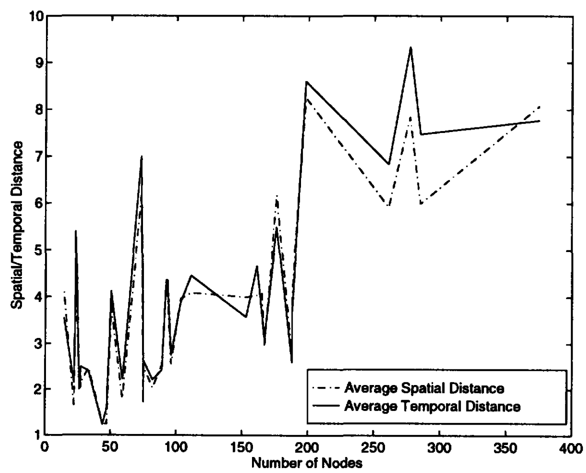

Figure 6-3: Spatial/temporal distance versus the number of nodes

of the circuit. The number of levels is on average, 3.975 times the depth of the circuit, which is not bad. That ratio should be minimized as much as possible. That roughly corresponds to the average "average spatial distance",d_f of about 3.7653 of all the circuits in the benchmark. The average distance measures the average spatial distance, and the number of levels divided by the depth of the circuit measures the average temporal distance. As can be seen from Figure 6-3, these numbers are about equal, demonstrating that the scheduling of the placed nodes in the spacetime circuits was not only feasible, but it was nearly optimal. That is, the temporal distance was about equal to the spatial distance so that signals could get from one node to the next with nearly minimal time for the given spatial placement. There were no more levels than were needed to route the circuits. This can easily be seen from the code in which nodes are scheduled at the minimum level at which they can be routed from their inputs. The average ratio of space to time was 0.9546 which is expected because of the constraint pointed out in Section 5.2.2. Of course, the scheduling of some nodes is determined by the input node which is furthest away spatially so that some of the input nodes which are closer will have a space/time ratio less than 1, contributing to making the average factor less than 1. It should be as close to 1 as possible.

6.2.2 Routing

The results of the routing are summarized in Table 6.2. n_c is the total number of used cells, n_s is the number of cell configurations, f is the fraction of used cells in the space, x and y are the respective dimensions, i is the number of routing iterations used to route the circuits, and s is the projected speed of the circuit simulation of this circuit.

As can be seen from Figure 6-4, the number of routing iterations necessary to route a circuit seems to grow roughly as the square root of the number of nodes in that circuit. This makes sense from Rent's Rule which states that the number of pins is roughly $O(n_n^p)$, where p is usually a fraction less than one and usually around 0.5. Assuming that this exponent is 0.5, then each of the levels should be on the same order as the number of pins,$O(\sqrt{(n_n)})$ Since routing takes place between different levels,

Name	n_c	n_s	f	x	y	i	s
bbara	133	37	0.172852	8	8	9	195312.500000
bbara_bbtas	1856	142	0.317220	8	8	12	32552.083333
bbsse	2239	162	0.374681	8	8	12	31887.755102
bbtas	418	32	0.127604	8	8	1	57870.370370
beecount	104	24	0.112847	8	8	2	173611.111111
cse	3355	313	0.504587	8	8	28	28669.724771
dk14	993	161	0.262451	8	8	7	48828.125000
dk15	687	95	0.188004	8	8	16	50403.225806
dk16	3798	485	0.479087	8	8	68	24038.461538
dk17	1373	83	0.224590	8	8	2	31565.656566
dk27	91	21	0.117708	8	8	4	208333.333333
dk512	285	65	0.172917	8	8	27	104166.666667
ex1	9062	346	0.168241	16	16	4	3633.720930
ex4	1430	97	0.186384	8	16	2	24801.587302
ex6	351	84	0.214718	8	8	32	100806.451613
f208	659	134	0.268111	8	8	15	71022.727273
keyb	2549	368	0.701446	4	8	86	51652.892562
kirkman	2339	279	0.465030	8	8	22	37202.380952
lion	170	17	0.089844	8	8	1	97656.250000
mark1	1985	134	0.438556	8	8	7	42229.729730
mc	143	19	0.156250	8	8	2	183823.529412
opus	720	96	0.295759	8	8	9	74404.761905
pma	4497	388	0.550399	8	8	13	23496.240602
s1	2635	240	0.435625	8	8	11	31250.000000
s208	659	134	0.268111	8	8	15	71022.727273
s27	192	36	0.146484	8	8	15	130208.333333
s386	1431	143	0.386593	8	8	4	50403.225806
s420	461	85	0.296336	8	8	12	107758.620690
s510	12714	553	0.226545	16	16	15	3487.723214
s820	6741	525	0.444741	8	16	47	12703.252033
s832	8376	595	0.430981	8	16	16	9952.229299
sse	2245	180	0.375638	8	8	11	31887.755102
styr	22076	934	0.289436	16	16	24	2578.382838
tav	123	22	0.156250	8	8	2	208333.333333
tbk	5864	303	0.225778	8	16	6	7476.076555

Table 6.2: MCNC Routing and Configuration Info

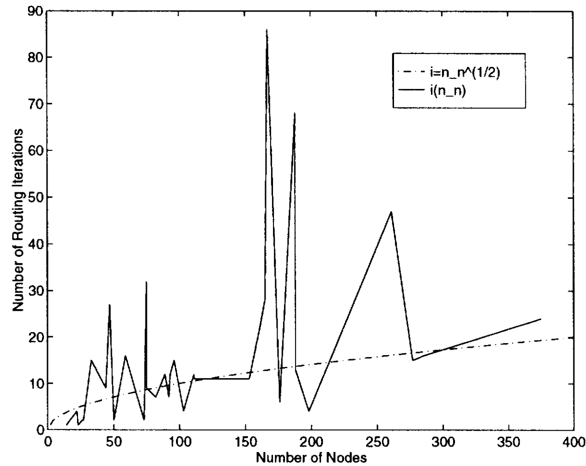

Figure 6-4: Number of routing iterations versus number of nodes

blockage occurs between those levels. The number of routing resources between two levels will be $O(\sqrt(n_n))$ so that the number of routing iterations will go up accordingly.

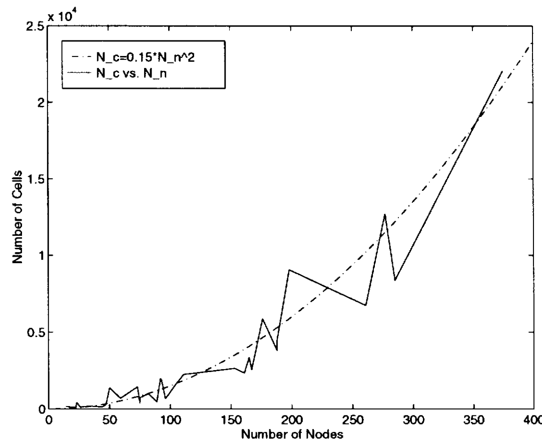

Figure 6-5: Number of nodes versus number of used cells

Figure 6-5 shows that the number of used cells grows quadratically with the number of nodes. This is not a good result. The average distance between nodes is $\sqrt{\sqrt{n_n}}$. This is because each level has $O(\sqrt(n_n))$ nodes by Rent's Rule. And a mesh with P cells, as each level is, where the nodes are placed randomly on that level will have average distance $\sqrt{P}$ to nodes in the next stage of the circuits[12, p. 66]. Thus, the number of cells should grow as $(n_n)^{5/4}$ which is closer to what is to be expected[12, p. 71]. Note that force-directed placement also makes the average distance go down by some factor when it is used, so we should be able to do even better. In fact, common

sense dictates that this factor should be linear since, for a given density of circuitry, the number of wires scales linearly with the number of nodes for a fixed support. Each of these wires should be about the same average distance long. Also note that since the size of a level is $O(\sqrt{(n_n)})$ the actual size of the space in a virtual circuit will grow by that amount as well. This is opposed to a normal circuit where the size of the space could scale linearly or even worse.

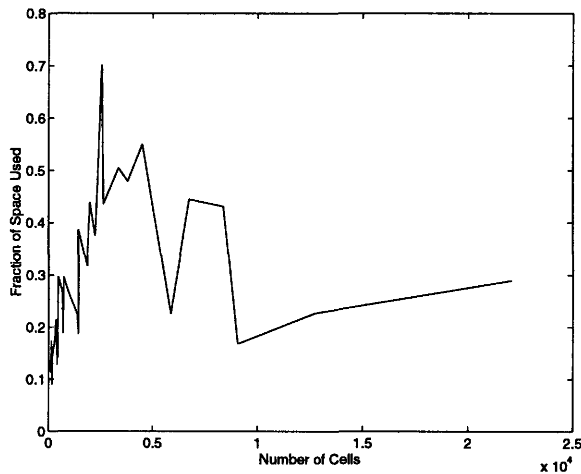

Figure 6-6: Fraction of space used versus the number of cells

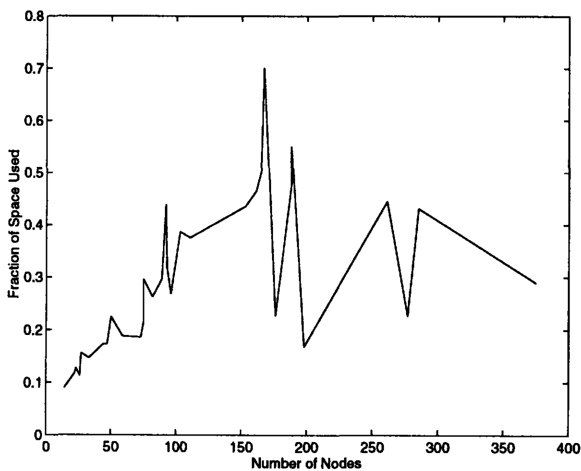

Figure 6-7: Fraction of space used versus the number of nodes

Note that the fraction of used cells in the space is relatively constant with the number of cells (or nodes) (Figure 6-6 and Figure 6-7). Thus, the size of the space really grows with the number of nodes and cells, so the number of cells grows with

the number of nodes. The average fraction of the space consumed is 29.15%. This fraction would probably increase with better topologies.

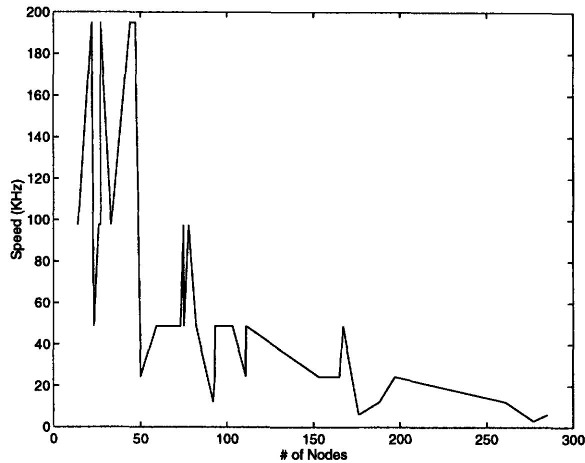

Figure 6-8: Speed of the circuit versus the number of nodes

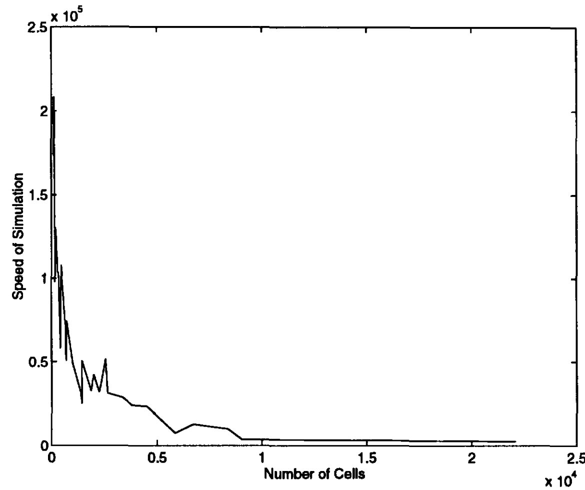

Figure 6-9: Speed of the circuit versus the number of cells

The speed of the simulation varied from 2.58 KHz - 0.21 MHz. As expected, the speed is inversely proportional to the number of nodes and cells (Figure 6-8 and Figure 6-9).

The number of configurations grows with the number of cells as expected (Figure 6-10). In all the circuits, the number of configurations fits into the number of configuration bits available. In this 2-D case, the number of configurations must be less than 1024 since there were only 10 bits available in our site. 5 bits were used

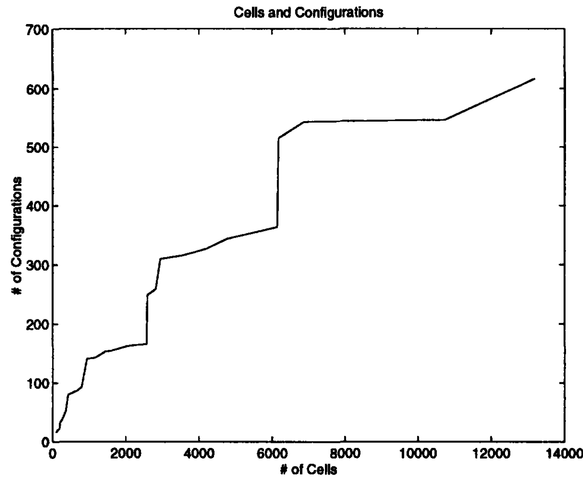

Figure 6-10: The number of configurations versus the number of cells

for data and one bit was used for tracing signals. The number of configurations will eventually saturate for large enough circuits in this case and in any case in which the number of bits available for configuration is fixed. In particular, from the equation given in Section 5.2.2, we find that the number of configurations possible in the underlying gate array model in this case is about 43 billion which requires about 36 bits. This would definitely require subcells which would slow down the simulation significantly. By reducing the number of configurations, we are also keeping the speed of update from being decreased.

Chapter 7

Conclusions and Future Research

7.1 Conclusions

The purpose of this thesis was to explore cellular automata logic simulations using CAM-8. This is essentially no different than using gate arrays for logic simulation and hence no new conclusions were reached in regards to this. There were several new ideas introduced in this thesis to increase the efficiency of logic simulation on CAM-8. All these ideas come from variations on standard techniques for making computation more efficient. Spacetime circuitry was born from combining the virtual processor approach with the notion of efficient parallel computation. It resulted in much faster logic simulations on CAM-8. The idea of minimizing the number of configurations resulted in being able to emulate cells with less configuration bits per site. These techniques are generalizable to any cellular processing architecture with resources analogous to those in CAM-8.

7.1.1 Spacetime Circuitry

The idea of trading space for time in spacetime circuitry was shown to be a good idea for doing logic simulations on CAM-8 fast and efficiently. With the reasonable assumption that the number of levels is about the same in a fully-parallel simulation and spacetime circuitry simulation, the speedup of the spacetime simulation was

directly proportional to the number of levels. The number of levels was found to be about directly proportional to the number of nodes in the circuit, which means that the speedup was directly proportional to the size of the circuit.

7.1.2 Synthesis

For the most part, traditional synthesis tools are sufficient for generating circuits for CAM-8 simulation, particularly for a fully parallel gate array emulation. Some modifications were obviously required for the virtual gate array approach which were implemented in the CALS program. Placement and scheduling were found to be interdependent. In addition, it was found that force-directed placement was not very helpful. This may have been due to the simple router which was implemented. Force-directed placement results in extremely dense circuits which are generally unroutable and hence is not widely used for the placement of nodes into a gate array unless the interconnect is very flexible like a crossbar. In our case, the added benefit of the spacetime circuitry was that it was generally easier to route circuits because the functionality of cells is changing in time to accommodate the circuits, thus providing an extra virtual dimension for routing. While there was some benefit here, the force-directed placement still produced circuits which were too dense to route in many cases. This could simply be a result of the nearest-neighbor interconnect again. However, what is really important is the size of the space. In many cases, it was simply good enough to just layout the circuit on a fixed-size space randomly without doing any force-directed placement. In this case, there is some natural compaction which comes from the limited size of the space. However, there is still a large variance in the distance between nodes which could result in long critical paths that will increase the number of levels beyond the optimum.

We were successfully able to reduce the number of configurations significantly, thus making the simulation more efficient. The number of configuration states for cells was found to grow linearly as the size of the circuits increased. This is only to be expected as larger random circuits are surely going to explore larger numbers of configurations in the state space when laid out. Thus, it is desirable that future

versions of CALS should have the ability to generate multiple tables for use as virtual tables in CAM-8, so that the number of configurations in the gate array simulation can be supported.

7.2 Future Research

Initial results are promising but there is still much work to be done. The results in the previous section are for the very limited case that there is only nearest-neighbor interconnect. The most immediate thing that needs to be done is to extend the capability of the CALS program to take a model file as input. That way it can have an arbitrary representation of a gate array rather than the one that it was limited to for this project. Also, the routing capabilities of CALS should also be improved. In particular, a maze router, such as Lee's algorithm[22], should be used rather than the annealing greedy algorithm used here. A maze router would guarantee a routing if there is one. The only reason it wasn't used here was due to time constraints. It is much easier to make an annealing greedy algorithm with the assumption made about the gate array architecture, in particular since distances and paths are taxicab in this instance. That is, there are no complicated interconnects. This would change with the model file input capability requirement. Also, in these experiments, the force-directed placement was not that useful. In future versions of CALS, with the added capability of accepting a model of a gate array, it should be possible to experiment with force-directed placement again on different topologies and see how results improve. In addition, force-directed placement can still be used for spacetime circuit compaction with the size of the space being adapted to fit afterwards. This was not investigated due to time constraints. Compacting the size of the space to fit the circuit compaction may be complicated by the fact that the space is wrapped around. This is a good problem to look into. As was stated before, techniques should be investigated for finding ways to represent the configurations efficiently. The CAM-8 simulator should be set up with subcells so that there can be more configuration bits and the CALS program should be adopted to fit by creating multiple lookup tables.

The CAM-8 can be used as a good logic simulation engine in its current state. But there is room for improvement. The CAM-8 is based on 1990 technology, so its parts are outdated. This is a more general problem and can be alleviated by simply updating the parts or making a next-generation CAM. A next-generation CAM has been discussed for some time. New types of FPGA's specialized for operation on a CAM machine have been proposed[31]. These FPGA's are based on the virtual processing feature of CAM-8. They would basically take the place of the LUT's and STEP chips with many added capabilities. In particular, they have a reconfigurable logic core which could take the place of the functionality of the LUT, providing much larger numbers of bits per site that could be updated with one pass through the logic core. Thus, it would be possible to emulate a gate array with much greater functionality of a cell, or more configuration states with less hassle. In addition, they have the ability to access fast synchronous DRAM's for quick virtual processing of sites. This would generically be useful for making large-scale logic array computations such as cellular automata possible. The techniques in this thesis would carry over.

As was mentioned in chapter 2, using a cellular automata architecture at the nanometer scale may be a good idea. FPGA's are already chip architectures based on cellular automata. We will eventually have to turn to nanotechnology for many things, including computation. Hence, it seems quite possible that nanochips will have reconfigurable logic cell arrays as their cores. The techniques in this thesis would carry over.

There is really no need to emulate a particular gate array architecture on CAM-8. In particular, CAM-8 is its own gate-array architecture, more generic than any emulation of a gate array that can be thought of. This was partially the idea behind the minimization of configuration states. That is, we don't have to fully simulate the functionality of a gate array, if only part of its functionality was needed for that circuit, thus saving the number of configuration states in state space. The more generic idea is to use CAM-8 itself as a gate array and just synthesize to that. This is essentially the problem of using CAM-8 as a general-purpose computer and we delve into more theoretical issues here. In particular, the problem is now to synthesize to the CAM-8

hardware itself, and this turns into the generic compiler problem[1]. We are now asking how do we do an arbitrary computation with a partitioning cellular automaton, in particular, CAM-8. This is an interesting project in its own right, but is beyond the scope of this thesis. Finally, the CAM-8 has many IO capabilities built in that have never been used. These could be used to interface CAM-8 logic simulations directly to prototype systems. This may or may not work depending upon the required speed. That way the CAM-8, like the Yorktown Simulation Engine, would have the capability to be used as a logic emulator as well as a logic simulator.

[1]Which was the Masters thesis of another student

Appendix A

CAM-8 Logic Simulator Code

```
new-experiment

\ CAM-8 parallel gate array simulator

10 constant F

512 by 512 by 2 space

\ *************************** Bit fields ********************************

0 3 == slow.signals
0 0 == slow.east  1 1 == slow.north  2 2 == slow.west  3 3 == slow.south

4 4 == routing.site?

5 5 == storage.site?

6 6 == logic.site?

7 7 == background.site?
```

```
5 8 == fast.signals
5 5 == fast.east  6 6 == fast.north  7 7 == fast.west  8 8 == fast.south

9 15 == routing.type

\ If not a routing site, then

9 10 == orientation \ directions 0 1 2 3 are: east north west south

12 12 == stored.bit
13 13 == clock
14 15 == storage.fn

\ If not storage, then its logic

12 15 == logic.fn

\ **************************** Rule ***********************************

0 value ctl.east
0 value ctl.north
0 value ctl.west
0 value ctl.south

: do-routing

routing.type 81 <
```

```
if
routing.type
3 /mod swap is ctl.east
3 /mod swap is ctl.north
3 /mod swap is ctl.west
3 /mod swap is ctl.south drop

ctl.east  {{ slow.east  slow.south slow.north }} -> slow.east
ctl.north {{ slow.north slow.east  slow.west }} -> slow.north
ctl.west  {{ slow.west  slow.north slow.south }} -> slow.west
ctl.south {{ slow.south slow.west  slow.east }} -> slow.south
then
routing.type 81 112 between
if
routing.type 81 -
2 /mod swap is ctl.east
2 /mod swap is ctl.north
2 /mod swap is ctl.west
2 /mod swap is ctl.south  ( direction )

                if

ctl.east  {{ fast.east  slow.east  }} -> fast.east
ctl.north {{ fast.north slow.north }} -> fast.north
ctl.west  {{ fast.west  slow.west  }} -> fast.west
ctl.south {{ fast.south slow.south }} -> fast.south

else

ctl.east  {{ slow.east  fast.east  }} -> slow.east
```

```
ctl.north {{ slow.north fast.north }} -> slow.north
ctl.west  {{ slow.west  fast.west  }} -> slow.west
ctl.south {{ slow.south fast.south }} -> slow.south
ctl.east  {{ fast.east  slow.east  }} -> fast.east
ctl.north {{ fast.north slow.north }} -> fast.north
ctl.west  {{ fast.west  slow.west  }} -> fast.west
ctl.south {{ fast.south slow.south }} -> fast.south

then
then
routing.type 126 =
if
15 -> slow.signals \ essentially power
then
routing.type 127 =
if
0 -> slow.signals \ essentially ground
then
;

\ For the moment we utilize four types of flip-flops for storage.

: do-storage

         storage.fn 0 =                \ R-S flip-flop
        if

orientation {{ slow.east slow.north slow.west slow.south }}
orientation {{ slow.north slow.west slow.south slow.east }}
        =
```

```
        if
         orientation {{ slow.north slow.west slow.south slow.east }}
1 =
if
stored.bit not -> stored.bit
\ just toggles if both inputs are high
then
else
         orientation {{ slow.north slow.west slow.south slow.east }}
  -> stored.bit
then
then


         storage.fn 1 =           \ a trilatch
        if
orientation {{ slow.north slow.west slow.south slow.east }}
clock not and
if
      orientation {{ slow.east slow.north slow.west slow.south }}
-> stored.bit
then
orientation {{ slow.south slow.east slow.north slow.west }}
if
stored.bit
      orientation {{ slow.east slow.north slow.west slow.south }} !!
then
then



   clock
```

```
if
        storage.fn 2 =           \ toggle flip-flop
 \ can be used as a clock
if
stored.bit not -> stored.bit
        else                                 \ D flip-flop
      orientation {{ slow.east slow.north slow.west slow.south }}
-> stored.bit
        then
  then

storage.fn 1 = not
if
0 -> slow.east
0 -> slow.north
0 -> slow.west
0 -> slow.south
         stored.bit
        orientation {{ slow.east slow.north slow.west slow.south }} !!
then
;

\ For the moment, do all possible 2-input function with inputs at east
\ and north, result at east, and rotations.

: do-logic

logic.fn

orientation {{ slow.east slow.north slow.west slow.south }} 2*
```

```
orientation {{ slow.north slow.west slow.south slow.east }} +
>> 1 and
orientation {{ slow.east slow.north slow.west slow.south }} !!
\ A logic gate normally has one output, but we are allowing the direction
\ which is not the output direction to be the other input direction so
\ that that input just goes straight through.  This allows for easier
\ implementations of PLA's.
        0 orientation {{ slow.west slow.south slow.east slow.north }} !!
        0 orientation {{ slow.south slow.east slow.north slow.west }} !!

;

: do-nothing
0 -> slow.signals
0 -> fast.signals
;

: logic-rule

routing.site?
if
do-routing
else
storage.site?
if
do-storage
else
logic.site?
if
do-logic
```

```
else
do-nothing
then
then
then
;


create-lut logic-table  ?rule>table logic-rule logic-table
lut-data logic-table  switch-luts


\ ****************************** Step ************************************

: logic-step  (s z.kick -- )

site-src lut
lut-src site

kick slow.north field -1 y
slow.south field  1 y
slow.east  field  1 x
slow.west  field -1 x
clock      field   z
run free
;

: ptoc-step  (s z.kick -- )

site-src lut
```

```
lut-src site

kick slow.north field -1 y
slow.south field  1 y
slow.east  field  1 x
slow.west  field -1 x
clock      field  1 z
run free
;

: stay-step  (s z.kick -- )

site-src lut
lut-src site

kick slow.north field -1 y
slow.south field  1 y
slow.east  field  1 x
slow.west  field -1 x
clock      field  0 z
run free
;

: ctop-step  (s z.kick -- )

site-src lut
lut-src site

kick slow.north field -1 y
slow.south field  1 y
```

```
slow.east  field  1 x
slow.west  field -1 x
clock      field -1 z
run free
;

0 value green
0 value red
0 value blue

: logic-map

slow.north slow.south slow.east slow.west + + + 3 min
255 3 * 4 / * 255 min is green

routing.site?
if
255 is blue
else
storage.site?
if
green 127 + 255 min is green
red   127 + 255 min is red
orientation 127 4 / * is blue
else
logic.site?
if
green 127 + 255 min is green
blue  127 + 255 min is blue
orientation 127 4 / * is red
```

```
else
background.site?
if
0 is green
0 is blue
0 is red
then
then
then
then

red >red  green >green  blue >blue
;

colormap logic-map

\ ******************** running the experiment ***************************

133 value #propagates/clock
133 value #p-left

: gatesim-step

#p-left 1- is #p-left  stay-step

#p-left 0<=
if
#propagates/clock is #p-left
ptoc-step ctop-step
then
```

```
;
this is update-step

: Clock-period arg? if arg is #propagates/clock  arg is #p-left then
;
press C "Set the clock period."

: Initialize
source-pat file>cam xvds
;  this is when-starting
press I "Initializing..."

load probe.fth

10 steps/display
"" gatesim5.pat file>cam xvds


new-experiment

\ sexium.exp
\ This was the file that implemented the gate array for the Sexium

1 K by 256 space

\ *************************** Bit fields ********************************

\ The signals are the bits that flow through the wires in any one of five
\ directions.  All signals go in those directions and then travel through
\ a second dimension, either up or down.
```

```
0 2 == signals

        0 0 == straight
1 1 == slow.north
2 2 == slow.south

\ There are four different types of sites: background, routing,
\ storage, and logic sites.

3 4 == site.type

\ A residue is used to keep track of where a signal has been for
\ viewing purposes.

5 5 == residue

\ The routing type cosists of a routing mechanism which allows for
\ any signal source to travel to any combination of five different
\ destinations as given by the signals.

6 11 == routing
6 7 == src.straight
8 9 == src.north
  10 11 == src.south

\ If not routing, then storage or logic
\ Orientation is given as four bits which are used by both storage
\ and logic sites for different purposes.

6 12 == orientation
```

```
6 7  == inputs
8 12 == dests

\ Storage sites consists of a one field for the storing a bit and
\ a storage function telling from which signal field to grab the
\ bit

13 13 == stored.bit

14 15 == storage.fn

\ Logic sites consist of a logic function which defines the truth
\ table results of an arbitrary two-input function as four bits.

13 15 == logic.fn


\ ***************************** Rule ***********************************

\ Background rule

: do-nothing
       0 -> signals
;

\ Routing rule

: do-routing

        src.north {{ 0 straight slow.north slow.south }}
```

```
                 -> slow.north

        src.straight {{ 0 straight slow.north slow.south }}
                 -> straight

        src.south {{ 0 straight slow.north slow.south }}
                 -> slow.south

;

\ Storage rule

: do-storage

      inputs {{ 0 straight slow.north slow.south }}
-> stored.bit

0 -> signals

stored.bit
if
        stored.bit
         storage.fn {{ 0 straight slow.north slow.south }} !!
then
;

\ Logic rule

0 constant inpa
0 constant inpb
```

```
0 constant out
0 constant dstraight
0 constant dslow.north
0 constant dslow.south

: do-logic


logic.fn
inputs {{ 0 straight straight slow.north }} dup
is inpa
inputs {{ 0 slow.north    slow.south    slow.south }} dup
is inpb
+ >> 1 and
is out


dests
3 /mod swap is dstraight
3 /mod swap is dslow.north
3 /mod swap is dslow.south
drop


dslow.north {{ out inpa inpb }} -> slow.north
dstraight {{ out inpa inpb }} -> straight
dslow.south {{ out inpa inpb }} -> slow.south


;


\ Main rule


: rlogic-rule
```

```
0 -> residue

                site.type 0 =
        if
                do-nothing
        then
site.type 1 =
if
do-routing
then
site.type 2 =
if
do-storage
then
site.type 3 =
if
do-logic
then

        straight slow.north slow.south or or -> residue

;


create-lut rlogic-table  ?rule>table rlogic-rule rlogic-table
lut-data rlogic-table  switch-luts


\ ****************************** Step ************************************
```

```
\ This is the main step.  The rule is similar to the light rule in
\ anneal.exp.  The update is down from top to botton and the signals
\ should travel all the way to the bottom or to a flip-flop in one
\ step.

: rlogic-step

1 by V subsector

scan-index

site-src lut
lut-src site
kick

subsectors/sector 0 ?do
run free
        kick
                         slow.north  field 0 1 - y
                        slow.south  field 1 y
signals    field 1 x
loop

site-src site
run
;
this is update-step

\ An arbitrary color map
```

```
: cals-map
0 >color

signals
if
255 >green
then

residue
if
127 >green
then

site.type 0 =
if
0 >blue
0 >green
0 >red
then

site.type 1 =
if
255 >blue
then

site.type 2 =
if
255 >red
then
```

```
site.type 3 =
if
127 >red
127 >blue
then

;

colormap cals-map

\ ******************** probing the simulation ***************************



\ Returns -1 if the pif is at the end of the file and 0 otherwise
: eof-probe-input-file?
ifd @ ftell ifd @ fsize =
;

\ variable arrays and constants used in conjunction with the probe procedure
\ below
variable pif 40 allot
variable pof 40 allot
variable ibuff 40 allot
variable temp 40 allot
variable vectors 8 allot
variable outputs 8 allot
variable empty 8 allot
variable dc 2 allot
0 constant pass
0 constant #propagates/clock
```

```
0 constant #clocks
0 constant #clocks/probe
0 constant seek-inputs
0 constant seek-vectors
0 constant seek-outputs
0 constant vsize
0 constant numv
0 constant numo
0 constant counter
0 constant count
0 constant tmp
0 constant cntr

\ The following procedure performs the probing of a digital logic simulation
\ by reading a special probe input file (.pif) which provides information
\ about how the probing is to proceed and outputs the results of the probing
\ into a file called a probe output file (.pof).  See probe.doc for more
\ information on the syntax and semantics of these files and about how to
\ generate a report (.rpt) from these files.


: Probe
pif 40 erase
pof 40 erase
ibuff 40 erase
temp 40 erase
vectors 8 erase
outputs 8 erase
dc 2 erase
0 is pass
```

```
0 is #clocks
0 is #clocks/probe
0 is seek-inputs
0 is seek-vectors
0 is seek-outputs
0 is vsize
0 is numv
0 is numo
0 is counter
0 is count
0 is tmp
0 is cntr
cr ." Please specify a probe input file "
pif [""] .pif filename:
pif read-open
." Please specify a probe output file "
pof [""] .pof filename:
pof new-file
ibuff 40 erase temp 40 erase
ibuff ifd @ getword
[""] #propagates/clock dup temp swap cstr cstrlen 1+ cmove
temp 40 compare not
if
ibuff 40 erase
ibuff ifd @ getword number is #propagates/clock
." #propagates/clock is " ibuff 40 type cr
else
." Error in probe-input-file, need #propagates/clock" cr
ifd @ fsize ifd @ fseek
exit
```

```
then
ibuff 40 erase temp 40 erase
ibuff ifd @ getword
[""] #clocks dup temp swap cstr cstrlen 1+ cmove
temp 40 compare not
if
ibuff 40 erase
ibuff ifd @ getword number is #clocks
." #clocks is " ibuff 40 type cr
else
." Error in probe-input-file, need #clocks" cr
ifd @ fsize ifd @ fseek
exit
then
ibuff 40 erase temp 40 erase
[""] #clocks/probe dup temp swap cstr cstrlen 1+ cmove
ibuff ifd @ getword
temp 40 compare not
if
ibuff 40 erase
ibuff ifd @ getword number is #clocks/probe
." #clocks/probe is " ibuff 40 type cr
else
." Error in probe-input-file, need #clocks/probe" cr
ifd @ fsize ifd @ fseek
exit
then
ibuff 40 erase temp 40 erase
[""] inputs dup temp swap cstr cstrlen 1+ cmove
ibuff ifd @ getword
```

```
temp 40 compare not
if
ibuff 40 erase
ifd @ ftell is seek-inputs
." Reading input probe points..." cr
else
." Error in probe-input-file, need inputs" cr
ifd @ fsize ifd @ fseek
exit
then
outputs 8 erase
[""] outputs dup outputs swap cstr cstrlen 1+ cmove
vectors 8 erase
[""] vectors dup vectors swap cstr cstrlen 1+ cmove
dc 2 erase
[""] x dup dc swap cstr cstrlen 1+ cmove
begin
ibuff 40 erase
ibuff ifd @ getword
vectors 8 compare eof-probe-input-file? not and
while
3 0 do
ibuff 40 erase
ibuff ifd @ getword drop
loop
vsize 1+ is vsize \ incr size of vector
repeat
ifd @ ftell is seek-vectors
." Reading vectors..." cr
begin
```

```
ibuff 40 erase
ibuff ifd @ getword
outputs 8 compare eof-probe-input-file? not and
while
vsize 0 do
ibuff 40 erase
ibuff ifd @ getword drop
loop
numv 1+ is numv \ incr number of vector
repeat
ifd @ ftell is seek-outputs
." Reading output probe points..." cr
begin
ibuff 40 erase
ibuff ifd @ getword
empty 8 compare eof-probe-input-file? not and
while
3 0 do
ibuff 40 erase
ibuff ifd @ getword drop
loop
numo 1+ is numo \ incr number of output vectors
repeat
numv 0 do
0 is counter
seek-vectors ifd @ fseek
ibuff 40 erase
ibuff ifd @ getword 40 type \ get and print out tag
vsize 0 do
ibuff 40 erase
```

```
ibuff ifd @ getword dup dup \ get component
1 spaces 40 type \ print out
dc 2 compare not
if
drop -1
counter 1+ is counter \ counter 1 bit more
else
number
then
loop
cr
0 is cntr
ifd @ ftell is seek-vectors
1 counter << 0 do
begin-line-io
seek-inputs ifd @ fseek
0 is count
vsize 0 do
depth reverse dup
ibuff 40 erase
ibuff ifd @ getword drop \ get, prn out tag
ibuff 40 erase
ibuff ifd @ getword  \ field-name
"compile field \ compile
dup -1 =
if
drop
cntr count >> 2 mod \ current x
count 1+ is count \ given current count
then
```

```
1 spaces dup .
ibuff 40 erase
ibuff ifd @ getword          \ x-coordinate
number \ leave on stack
ibuff 40 erase
ibuff ifd @ getword  \ y-coordinate
number  \ leave on stack
write-point \ write point with stack constants
is tmp
depth reverse
tmp
loop
cr
end-line-io
#clocks #clocks/probe / 0 do
#propagates/clock #clocks/probe * steps
seek-outputs ifd @ fseek
begin-line-io
numo 0 do
file-output
ibuff ifd @ getword drop
ibuff ifd @ getword  \ field
"compile field \ compile
ibuff 40 erase
ibuff ifd @ getword  \ x-cor
number \ leave on stack
ibuff 40 erase
ibuff ifd @ getword  \ y-cor
number  \ leave on stack
read-point . \ with stack params
```

```
[ hidden ] unsave-output
loop
file-output cr [ hidden ] unsave-output
end-line-io
loop
cntr 1+ is cntr
loop
clear
loop
close-files
." done."
;
press P "Beginning Probe Sequence ...."

\ ******************** running the experiment ***************************


\ Circuit pattern loaded


"" rlog.pat file>cam xvds


\ stc2d2.exp
\ This rule runs a three-dimensional spacetime circuitry simulation.
\ Two spatial dimensions, one temporal dimension



new-experiment  centering-hd off


16 by 16 by 512 space   \ load step-anag.fth  8 is rend0/rend1  0 set-logmag


0 0 == residue
```

```
1 5  == sg
1 1 == pn
2 2 == px
3 3 == nx
4 4 == py
5 5 == ny
6 15 == config

create-lut cals-lut
cals-lut "" styr3.tab load-buffer
lut-data cals-lut switch-luts

303 value L

define-step logic-step
U by V by 1 subsector

scan-index

site-src lut
lut-src site

kick
run
kick
px field 1 x
nx field -1 x
py field 1 y
ny field -1 y
sg field 1 z
```

```
run

L 2 do run repeat-kick loop

site-src site
kick
px field 1 x
nx field -1 x
py field 1 y
ny field -1 y
sg field Z L - 1 + z
run
end-step
this is update-step

: scan-space
   L 0 do 1 2  zero-space-shift space-shift ! perform-space-shift show loop
;
press S "Scan Space"

: cals-map
0 >color
config
if
255 >blue
sg if 255 >red then
then
residue if 255 >green then
;
```

```
colormap cals-map

load sliceio.fth
load probe-slice.fth

"" styr3.pat file>cam show


\ probe-slice.fth :  the forth code used to probe circuits using the PIF
\ as input and generating a POF as output

\ Returns -1 if the pif is at the end of the file and 0 otherwise
: eof-probe-input-file?
ifd @ ftell ifd @ fsize =
;

\ variable arrays and values used in conjunction with the probe procedure
\ below
variable pif 40 allot
variable pof 40 allot
variable ibuff 40 allot
variable temp 40 allot
variable vectors 8 allot
variable outputs 8 allot
variable empty 8 allot
variable dc 2 allot
0 value pass
1 value #propagates/clock
0 value #clocks
0 value #clocks/probe
0 value seek-inputs
```

```
0 value seek-vectors
0 value seek-outputs
0 value vsize
0 value numv
0 value numo
0 value counter
0 value count
0 value tmp
0 value cntr

\ The following procedure performs the probing of a digital logic simulation
\ by reading a special probe input file (.pif) which provides information
\ about how the probing is to proceed and outputs the results of the probing
\ into a file called a probe output file (.pof).  See probe.doc for more
\ information on the syntax and semantics of these files and about how to
\ generate a report (.rpt) from these files.

: Probe
pif 40 erase
pof 40 erase
ibuff 40 erase
temp 40 erase
vectors 8 erase
outputs 8 erase
dc 2 erase
0 is pass
0 is #clocks
0 is #clocks/probe
0 is seek-inputs
0 is seek-vectors
```

```
0 is seek-outputs
0 is vsize
0 is numv
0 is numo
0 is counter
0 is count
0 is tmp
0 is cntr
cr ." Please specify a probe input file "
pif [""] .pif filename:
pif read-open
." Please specify a probe output file "
pof [""] .pof filename:
pof new-file
ibuff 40 erase temp 40 erase
ibuff ifd @ getword
[""] #propagates/clock dup temp swap cstr cstrlen 1+ cmove
temp 40 compare not
if
ibuff 40 erase
ibuff ifd @ getword number is #propagates/clock
\ ." #propagates/clock is " ibuff 40 type cr
else
." Error in probe-input-file, need #propagates/clock" cr
ifd @ fsize ifd @ fseek
exit
then
ibuff 40 erase temp 40 erase
ibuff ifd @ getword
[""] #clocks dup temp swap cstr cstrlen 1+ cmove
```

```
temp 40 compare not
if
ibuff 40 erase
ibuff ifd @ getword number is #clocks
\ ." #clocks is " ibuff 40 type cr
else
." Error in probe-input-file, need #clocks" cr
ifd @ fsize ifd @ fseek
exit
then
ibuff 40 erase temp 40 erase
[""] #clocks/probe dup temp swap cstr cstrlen 1+ cmove
ibuff ifd @ getword
temp 40 compare not
if
ibuff 40 erase
ibuff ifd @ getword number is #clocks/probe
\ ." #clocks/probe is " ibuff 40 type cr
else
." Error in probe-input-file, need #clocks/probe" cr
ifd @ fsize ifd @ fseek
exit
then
ibuff 40 erase temp 40 erase
[""] inputs dup temp swap cstr cstrlen 1+ cmove
ibuff ifd @ getword
temp 40 compare not
if
ibuff 40 erase
ifd @ ftell is seek-inputs
```

```
." Reading input probe points..." cr
else
." Error in probe-input-file, need inputs" cr
ifd @ fsize ifd @ fseek
exit
then
outputs 8 erase
[""] outputs dup outputs swap cstr cstrlen 1+ cmove
vectors 8 erase
[""] vectors dup vectors swap cstr cstrlen 1+ cmove
dc 2 erase
[""] x dup dc swap cstr cstrlen 1+ cmove
begin
ibuff 40 erase
ibuff ifd @ getword
vectors 8 compare eof-probe-input-file? not and
while
3 0 do
ibuff 40 erase
ibuff ifd @ getword drop
loop
vsize 1+ is vsize \ incr size of vector
repeat
ifd @ ftell is seek-vectors
." Reading vectors..." cr
begin
ibuff 40 erase
ibuff ifd @ getword
outputs 8 compare eof-probe-input-file? not and
while
```

```
vsize 0 do
ibuff 40 erase
ibuff ifd @ getword drop
loop
numv 1+ is numv \ incr number of vector
repeat
ifd @ ftell is seek-outputs
." Reading output probe points..." cr
begin
ibuff 40 erase
ibuff ifd @ getword
empty 8 compare eof-probe-input-file? not and
while
3 0 do
ibuff 40 erase
ibuff ifd @ getword drop
loop
numo 1+ is numo \ incr number of output vectors
repeat
numv 0 do
0 is counter
seek-vectors ifd @ fseek
ibuff 40 erase
ibuff ifd @ getword 40 type \ get and print out tag
vsize 0 do
ibuff 40 erase
ibuff ifd @ getword dup \ get component
dup 1 spaces 40 type \ print out
dc 2 compare not
if
```

```
drop -1
counter 1+ is counter \ counter 1 bit more
else
number
then
loop
cr
0 is cntr
ifd @ ftell is seek-vectors
1 counter << 0 do
\ begin-line-io
begin-slice0
seek-inputs ifd @ fseek
0 is count
vsize 0 do
depth reverse dup
ibuff 40 erase
ibuff ifd @ getword drop \ get, prn out tag
ibuff 40 erase
ibuff ifd @ getword  \ field-name
"compile field \ compile
dup -1 =
if
drop
cntr count >> 2 mod \ current x
count 1+ is count \ given current count
then
1 spaces dup .
ibuff 40 erase
ibuff ifd @ getword          \ x-coordinate
```

```
number \ leave on stack
ibuff 40 erase
ibuff ifd @ getword  \ y-coordinate
number  \ leave on stack
write-field \ write point with stack values
is tmp
depth reverse
tmp
loop
cr
\ end-line-io
end-slice0
#clocks #clocks/probe / 0 do
#propagates/clock #clocks/probe * steps
seek-outputs ifd @ fseek
\ begin-line-io
begin-slice0
numo 0 do
file-output
ibuff ifd @ getword drop
ibuff ifd @ getword  \ field
"compile field \ compile
ibuff 40 erase
ibuff ifd @ getword  \ x-cor
number \ leave on stack
ibuff 40 erase
ibuff ifd @ getword  \ y-cor
number  \ leave on stack
read-field . \ with stack params
[ hidden ] unsave-output
```

```
loop
file-output cr [ hidden ] unsave-output
\ end-line-io
end-slice0
loop
cntr 1+ is cntr
loop
clear
loop
close-files
." done."
;
press P "Beginning Probe Sequence ...."
```

Appendix B

Logic Synthesis Scripts and Code

```
# Script used to generate BLIF from KISS file using SIS

read_kiss %:3.kiss2
state_minimize stamina
state_assign jedi
extract_seq_dc
source script.rugged
xl_part_coll -m -g 2 -n %:2
xl_coll_ck -n %:2
xl_partition -m -n %:2
simplify
xl_imp -n %:2
xl_partition -t -n %:2
xl_cover -e 30 -u 200 -n %:2
xl_coll_ck -k -n %:2
write_blif %:4.blif
print_stats
```

```
# bbtas.kiss2 : an example KISS files showing an STT

.i 2
```

```
.o 2
.p 24
.s 6
00 st0 st0 00
01 st0 st1 00
10 st0 st1 00
11 st0 st1 00
00 st1 st0 00
01 st1 st2 00
10 st1 st2 00
11 st1 st2 00
00 st2 st1 00
01 st2 st3 00
10 st2 st3 00
11 st2 st3 00
00 st3 st4 00
01 st3 st3 01
10 st3 st3 10
11 st3 st3 11
00 st4 st5 00
01 st4 st4 00
10 st4 st4 00
11 st4 st4 00
00 st5 st0 00
01 st5 st5 00
10 st5 st5 00
11 st5 st5 00

# A BLIF netlist generated by the KISS file bbtas.kiss2 using SIS

.model bbtas.kiss2
```

```
.inputs IN_0 IN_1
.outputs OUT_0 OUT_1
.latch    [370] LatchOut_v2   0
.latch    [371] LatchOut_v3   0
.latch    [372] LatchOut_v4   0
.start_kiss
.i 2
.o 2
.p 24
.s 6
.r st0
00 st0 st0 00
01 st0 st1 00
10 st0 st1 00
11 st0 st1 00
00 st1 st0 00
01 st1 st2 00
10 st1 st2 00
11 st1 st2 00
00 st2 st1 00
01 st2 st3 00
10 st2 st3 00
11 st2 st3 00
00 st3 st4 00
01 st3 st3 01
10 st3 st3 10
11 st3 st3 11
00 st4 st5 00
01 st4 st4 00
10 st4 st4 00
```

```
11 st4 st4 00
00 st5 st0 00
01 st5 st5 00
10 st5 st5 00
11 st5 st5 00
.end_kiss
.latch_order LatchOut_v2 LatchOut_v3 LatchOut_v4
.code st0 000
.code st1 001
.code st2 100
.code st3 101
.code st4 111
.code st5 011
.names [505] [506] [370]
1- 1
-1 1
.names [478] [479] [371]
1- 1
-1 1
.names [480] [481] [372]
1- 1
-1 1
.names [398] [507] OUT_0
11 1
.names [398] [508] OUT_1
11 1
.names IN_0 IN_1 [397]
1- 1
-1 1
.names LatchOut_v2 LatchOut_v4 [398]
```

```
11 1
.names [397] [398] [478]
01 1
.names LatchOut_v3 [397] [479]
11 1
.names LatchOut_v4 [397] [480]
01 1
.names LatchOut_v2 [371] [481]
1- 1
-1 1
.names LatchOut_v2 [372] [504]
1- 1
-0 1
.names [397] [504] [505]
11 1
.names LatchOut_v3 [371] [506]
01 1
.names IN_0 LatchOut_v3 [507]
10 1
.names IN_1 LatchOut_v3 [508]
10 1
.exdc
.inputs LatchOut_v4 LatchOut_v3
.outputs [370] [371] [372] OUT_0 OUT_1
.names LatchOut_v4 LatchOut_v3 dc[391]
01 1
.names dc[391] [370]
1 1
.names dc[391] [371]
1 1
```

```
.names dc[391] [372]
1 1
.names dc[391] OUT_0
1 1
.names dc[391] OUT_1
1 1
.end
```

```
/* blifscan.lex */
/* Scanner for BLIF netlist reader */


%{
#include <string.h>
#include "structures.h"
#include "blifparse.h"
%}


%%
\.model     { yylval.string=strdup(yytext); return MODEL; }
\.inputs    return INPUTS;
\.outputs   return OUTPUTS;
\.latch     return ALATCH;
\.start_kiss      return SKISS;
".i "[0-9]+ {
sscanf(yytext,"%s %d",&yylval.string,&yylval.num);
return KISSI;
    }
".o "[0-9]+ {
sscanf(yytext,"%s %d",&yylval.string,&yylval.num);
return KISSO;
    }
```

```
".p "[0-9]+ {
sscanf(yytext,"%s %d",&yylval.string,&yylval.num);
return KISSP;
    }
".s "[0-9]+ {
sscanf(yytext,"%s %d",&yylval.string,&yylval.num);
return KISSS;
    }
\.r return KISSR;
\.end_kiss return EKISS;
\.latch_order return LORDER;
\.code return CODE;
\.names return NAMES;
\.exdc return EXDC;
\.end return END;
^\#.*           /* eat one-line comments */
[ \t\n\\]+ /* eat whitespaces and slashes */
[012-]+ { yylval.string=strdup(yytext); return BITS; }
[^\\ \t\n\.\#\-][^ \t\n\\]* { yylval.string=strdup(yytext); return STR; }
```

```
/* blifparse.y */
/* Parser for BLIF netlist reader */

%{
#include "structures.h"
extern FILE *outfile;
#define YYPRINT(file, type, value) yyprint(file, type, value)
  void yyerror(const char *);
  int yylex(void);
%}                                                              10
```

```
%union {
  String string;
  int num;
  Node node;
}

%token MODEL
%token INPUTS
%token OUTPUTS                                                    20
%token ALATCH
%token SKISS
%token KISSR
%token EKISS
%token LORDER
%token CODE
%token NAMES
%token EXDC
%token END
%token EOF_TOK                                                    30

%token <string> STR
%token <string> BITS
%token <num> KISSI
%token <num> KISSO
%token <num> KISSP
%token <num> KISSS
%type <node> cubes
%type <node> name
                                                                  40
%%
input:
model_rec inputs outputs latches kisses encoders names exdc

model_rec:
MODEL STR { fprintf(outfile,"Reading %s...\n",$2); }
;
```

```
inputs: INPUTS
| inputs STR { make_node($2,INPUT); }                                    50

outputs: OUTPUTS
| outputs STR {
  Node latch;
  String name=(String)malloc(1);
  sprintf(name,"*%s*",$2);
  add_segment(make_node($2,OUTPUT),latch=add_node(name));
  latch->latch=1;
  an->nodes[LATCH-1]=add_el(an->nodes[LATCH-1],latch);
}                                                                        60

latches: /* empty */
| latches ALATCH STR STR BITS {
  Node latch;
  add_segment(add_node($3),latch=add_node($4));
  latch->latch=1;
  latch->init=*$5;
  latch->cubes=add_el(latch->cubes,strdup("1"));
  an->nodes[LATCH-1]=add_el(an->nodes[LATCH-1],latch);
  fsm->num_latches++;                                                    70
}
| latches ALATCH STR STR STR STR BITS {
  Node latch;
  if (def_clk) {
    make_latch(add_node($3),latch=add_node($4),$5,add_node($6));
  } else {
    add_segment(add_node($3),latch=add_node($4));
    latch->cubes=add_el(latch->cubes,strdup("1"));
    an->nodes[LATCH-1]=add_el(an->nodes[LATCH-1],latch);
  }                                                                      80
  latch->latch=1;
  latch->init=*$5;
  fsm->num_latches++;
```

```
}

kisses: /* empty */
| skiss kissi kisso kissp kisss kissr stds ekiss lorder

skiss: SKISS
kissi: KISSI { fsm->num_in=$1; }                                   90
kisso: KISSO { fsm->num_out=$1; }
kissp: KISSP {
  fsm->terms=(String **) calloc(sizeof(String *),$1);
  fsm->num_terms=0;
}
kisss: KISSS {
  fsm->states=(String **) calloc(sizeof(String *),$1);
  fsm->num_states=0;
}
kissr: KISSR STR {                                                100
  fsm->reset_state=strdup($2);
}
| KISSR BITS {
  fsm->reset_state=strdup($2);
}

stds: /* empty */
| stds std

std: BITS STR STR BITS {                                          110
  String term[4]={$1,$2,$3,$4};
  add_stt(term);
}
| BITS BITS BITS BITS {
  String term[4]={$1,$2,$3,$4};
  add_stt(term);
}
| BITS BITS STR BITS {
  String term[4]={$1,$2,$3,$4};
```

```
  add_stt(term);                                                120
}
| BITS STR BITS BITS {
  String term[4]={$1,$2,$3,$4};
  add_stt(term);
}

ekiss: EKISS

lorder: LORDER
| lorder STR {                                                  130
  fsm->latch_order=add_el(fsm->latch_order,node_lookup($2));
}

encoders: /* empty */
| encoders CODE STR BITS {
  String state[2]={$3,$4};
  encode(state);
}
| encoders CODE BITS BITS {
  String state[2]={$3,$4};                                      140
  encode(state);
}

names: /* empty */
| names cubes
| names ALATCH STR STR BITS {
  Node latch;
  add_segment(add_node($3),latch=add_node($4));
  latch->latch=1;
  latch->init=*$5;                                              150
  latch->cubes=add_el(latch->cubes,strdup("1"));
  an->nodes[LATCH-1]=add_el(an->nodes[LATCH-1],latch);
  fsm->num_latches++;
}
| names ALATCH STR STR STR STR BITS {
```

```
  Node latch;
  if (def_clk) {
    make_latch(add_node($3),latch=add_node($4),$5,add_node($6));
  } else {
    add_segment(add_node($3),latch=add_node($4));                    160
    latch->cubes=add_el(latch->cubes,strdup("1"));
    an->nodes[LATCH-1]=add_el(an->nodes[LATCH-1],latch);
  }
  latch->latch=1;
  latch->init=*$5;
  fsm->num_latches++;
}

name: NAMES name { $$=$2; }
| STR name { add_segment(add_node($1),$$=$2); }                      170
| STR { $$=add_node($1); }
| NAMES STR BITS { $$=add_node($2); $$->cubes=add_el($$->cubes,strdup($3)); }

cubes: name { $$=$1; }
| cubes BITS BITS { ($$=$1)->cubes=add_el($1->cubes,strdup($2)); }

exdc: EXDC
| END
| exdc INPUTS
| exdc STR                                                           180
| exdc BITS
| exdc OUTPUTS
| exdc NAMES
| exdc END

%%

static void
yyprint (file, type, value)
    FILE *file;                                                      190
    int type;
```

```
   YYSTYPE value;
{
  if (type == MODEL) printf (" %s\n", value.string);
}
```

```
/* fsm.c */

#include "structures.h"
#include "struct.h"
#include "global.h"

extern String fn;
extern int rtv;

void add_stt(String term[4]) {                                                        10
  /* add another entry to the state transition table */
  int i;
  fsm->terms[fsm->num_terms]=(String *) malloc(4*sizeof(String));
  for (i=0;i<4;i++) {
    fsm->terms[fsm->num_terms][i]=strdup(term[i]);
  }
  fsm->num_terms++;
}

void encode(String state[2]) {                                                        20
  /* encoding of the states */
  int i;
  fsm->states[fsm->num_states]=(String *) malloc(2*sizeof(String));
  for (i=0;i<2;i++) {
    fsm->states[fsm->num_states][i]=strdup(state[i]);
  }
  fsm->num_states++;
```

```
}

int fprintcor(FILE *fp,int nc) {
  int i;
  for (i=1;i<st->num_dim;i++) {
    fprintf(fp,"%d ",nc%(st->num_cor[i]/st->num_cor[i-1]));
    nc=nc/(st->num_cor[i]/st->num_cor[i-1]);
  }
  fprintf(fp,"%d ",nc%(st->num_cor[i]/st->num_cor[i-1]));
  return nc;
}

void write_pif() {
  /* write a probe input and output files using the information about placement,
     encoding, latch order, and STT */
  Node node;
  el it,it2;
  int i,j,k,n,num;
  String nbs[num_pi];
  FILE *pif,*pof;
  pif=fopen(strcat(fn,".pif"),"w");
  fn=strtok(fn,".");
  pof=fopen(strcat(fn,".pof"),"w");
  fn=strtok(fn,".");
  nbs[0]="pn";
  nbs[1]="px";
  nbs[2]="nx";
  nbs[3]="py";
  nbs[4]="ny";
  nbs[5]="pz";
  nbs[6]="nz";
  fprintf(pif,"#propagates/clock 1\n");
  fprintf(pif,"#clocks 1\n");
  fprintf(pif,"#clocks/probe 1\n");
  fprintf(pif,"\ninputs\n");
  for(it=an->nodes[INPUT];it->next;it=it->next) {}
```

```
for(;it;it=it->prev) {
  node=it->key;
  fprintf(pif,"%s pn ",node->name);
  fprintcor(pif,node->coor);
  if(st->num_dim==1) fprintf(pif,"%d",node->level);
  fprintf(pif,"\n");
}                                                                    70
for(it=fsm->latch_order;it->next;it=it->next) {}
for(;it;it=it->prev) {
  node=it->key;
  for(it2=node->inets;it2->next;it2=it2->next) {}
  fprintf(pif,"%s ",node->name);
  fprintf(pif,"%s ",nbs[neighbor(it2->key)]);
  fprintcor(pif,node->coor);
  if(1==st->num_dim) fprintf(pif,"%d",node->level);
  fprintf(pif,"\n");
}                                                                    80
fprintf(pif,"\nvectors\n");
for(i=0;i<fsm->num_terms;i++) {
  fprintf(pif,"term%d ",i);
  num=1;
  for(j=0;j<strlen(fsm->terms[i][0]);j++) {
    if(rtv) {
        if(fsm->terms[i][0][j]=='-') fprintf(pif,"%d ",round(ran3(&idum)));
        else fprintf(pif,"%c ",fsm->terms[i][0][j]);
    } else {
        fprintf(pif,"%c ",(fsm->terms[i][0][j]=='-') ? 'x' : fsm->terms[i][0][j]);    90
    }
    if((fsm->terms[i][0][j]=='-')&&(!rtv)) num<<=1;
  }
  for(j=0;j<fsm->num_states;j++) {
    if(!strcmp(fsm->states[j][0],fsm->terms[i][1])) {
        for(k=0;k<strlen(fsm->states[j][1]);k++) {
          if(rtv) {
            if(fsm->states[j][1][k]=='2') fprintf(pif,"%d ",round(ran3(&idum)));
            else fprintf(pif,"%c ",fsm->states[j][1][k]);
```

```
          } else {                                                              100
            fprintf(pif,"%c ",(fsm->states[j][1][k]=='2') ? 'x' : fsm->states[j][1][k]);
          }
        }
    }
    if(!strcmp(fsm->states[j][0],fsm->terms[i][2])) {
        n=j;
    }
  }
  fprintf(pif,"\n");
  for(j=0;j<num;j++) {                                                          110
    for(k=0;k<strlen(fsm->states[n][1]);k++) {
        fprintf(pof,"%c ",(fsm->states[n][1][k]=='2') ? '-' : fsm->states[n][1][k]);
    }
    for(k=0;k<strlen(fsm->terms[i][3]);k++) {
        fprintf(pof,"%c ",fsm->terms[i][3][k]);
    }
    fprintf(pof,"\n");
  }
}
fclose(pof);                                                                    120
fprintf(pif,"\noutputs\n");
for(it=fsm->latch_order;it->next;it=it->next) {}
for(;it;it=it->prev) {
  node=it->key;
  for(it2=node->inets;it2->next;it2=it2->next) {}
  fprintf(pif,"%s ",node->name);
  fprintf(pif,"%s ",nbs[neighbor(it2->key)]);
  fprintcor(pif,node->coor);
  if(1==st->num_dim) fprintf(pif,"%d",node->level);
  fprintf(pif,"\n");                                                            130
}
for(it=an->nodes[OUTPUT];it->next;it=it->next) {}
for(;it;it=it->prev) {
  node=it->key;
  fprintf(pif,"%s ",node->name);
```

```
    for(it2=node->outputs;it2->next;it2=it2->next) {}
    node=it2->key;
    fprintf(pif,"%s ",nbs[neighbor(node->inets->key)]);
    fprintcor(pif,node->coor);
    if(1==st->num_dim) fprintf(pif,"%d",node->level);                    140
    fprintf(pif,"\n");
  }
  fclose(pif);
}

void pofcomp(String tpofn,String epofn) {
  /* compare to probe output files. In particular, this is used to compare the
     probe output file produced by the CAM-8 simulator to the probe output file
     generated from the STT in the above procedure */
  FILE *tpof,*epof;                                                      150
  char tc,ec;
  int cnum=0;
  int diff=0;
  tpof=fopen(tpofn,"r");
  epof=fopen(epofn,"r");
  while((ec=fgetc(epof))!=EOF) {
    tc=fgetc(tpof);
    cnum++;
    if(tc!='-') {
      if(ec!=tc) {                                                       160
        printf("%d:  (%c,%c)\n",cnum,tc,ec);
        diff=1;
      }
    }
  }
  if(diff) printf("pof's are not the same.\n");
  else printf("pof's are the same.\n");
  fclose(tpof);
  fclose(epof);
}                                                                        170
```

```c
/* layout.c */

#include "structures.h"
#include "struct.h"
#include "global.h"

int debug=DEBUG;
extern int merging;                                                    180
long idum=-1;
extern float temp;
extern FILE *outfile;

void convert(int nc,int *ncor) {
  /* converts an integer to an n-dimensional coordinate */
  int i;
  for (i=1;i<st->num_dim+1;i++) {
    ncor[i-1]=nc%(st->num_cor[i]/st->num_cor[i-1]);
    nc=nc/(st->num_cor[i]/st->num_cor[i-1]);                           190
  }
}

int revert(int *cor) {
  /* converts an n-dimensional coordinate to an integer */
  int j;
  int nc=0;
  for (j=0;j<st->num_dim;j++) {
    nc+=st->num_cor[j]*cor[j];
  };                                                                   200
  return nc;
}

int round(double a) {
  return (a-floor(a)>0.5 ? (int) a+1 : (int) a);
}
```

```
int min(int a,int b) {
  return (a<b ? a : b);
}                                                                  210

int sdist2(int src,int dest,int dim) { return dest-src; }

int sdist1(int src,int dest,int dim) {
  /* returns the taxicab distance from coord src to dest in the dimension dim,
     taking into account wraparound at the edges */
  int k1,k2,k3,dim_size;
  dim_size=st->num_cor[dim+1]/st->num_cor[dim];
  k1=dest-src;
  k2=(-k1+dim_size)%dim_size;                                      220
  k1=(k1+dim_size)%dim_size;
  if (st->lwires) {
    k3=st->nbhd[2*(dim+st->num_dim+1)-1]/st->num_cor[dim];
    k1=round((double) k1/k3)+min((k1%k3+k3)%k3,((-k1)%k3+k3)%k3);
    k2=round((double) k2/k3)+min((k2%k3+k3)%k3,((-k2)%k3+k3)%k3);
  }
  if (k1<=k2) return k1;
  else if (k1>k2) return -k2;
}
                                                                   230
int sdist(int source, int dest) {
/* returns the minimum distance between two points and the sequences of kicks
   which accomplish that minimum distance */
  int i,dist=0;
  static int *srccor;
  static int *destcor;
  static int init=0;
  if (!init) {
    srccor=(int *) malloc(sizeof(int)*(st->num_dim));
    destcor=(int *) malloc(sizeof(int)*(st->num_dim));            240
    init=1;
  }
  convert(source,srccor);
```

```
  convert(dest,destcor);
  for (i=0;i<st->num_dim;i++) dist+=abs(sdist1(srccor[i],destcor[i],i));
  return dist;
}

int get_num_levels() {
  /* finds the maximum level */                                        250
  int ml=0,sd,i,j;
  el it,it2;
  Node node,node2;
  for (i=INT_NODE;i<NUM_TYPES;i++) {
    for (it=an->nodes[i];it;it=it->next) {
      node=it->key;
      if (node->level+1>ml) {
            ml=node->level+1;
      }
    }                                                                  260
  }
  for (it=an->nodes[LATCH-1];it;it=it->next) {
    node=it->key;
    for (it2=((Node)it->key)->inputs;it2;it2=it2->next) {
      node2=it2->key;
      sd=sdist(node2->coor,node->coor);
      if (node2->level+sd>ml) ml=node2->level+sd;
    }
  }
  return ml;                                                           270
}

void fix_latches() {
  /* adjusts levels according to latch positions */
  el it;
  for (it=an->nodes[LATCH-1];it;it=it->next) {
    fix_levels(it->key);
  }
}
```

```
                                                                    280
void fix_levels(Node node) {
  /* fixes the levels of all the nodes that depend on node node */
  el it;
  Node node2;
  int level,i;
  for (it=node->outputs;it;it=it->next) {
    node2=it->key;
    if (node2->placed) {
      level=schedule(node2,node2->coor);
      for (i=level;;i++) {                                          290
            if (compatible(node2,node2->coor,i)) {
              level=i;
              break;
            }
      }
      if (level!=node2->level) {
            place_node(node2,node2->coor,level);
      }
    }
  }                                                                 300
}

int compatible(Node node,int coor,int level) {
  /* returns whether or not a node can be placed at coordinate coor on level
     level given the requirements of compatibility */
  el it;
  for (it=st->tess[coor]->nodes;it;it=it->next) {
    if (debug) printf("compat:  %s %d %d %s\n",node->name,coor,level,((Node)it->key)->name);
    if ((((Node)it->key)->level==level)&&(it->key!=node)) {
      /* leave merging off for now, it's just too complicated */     310
      if (merging) {
      } else {
            return 0;
      }
    }
```

```
  }
  return 1;
}

int printcor(int nc) {                                                    320
  /* prints the n-dimensional coordinate representation of an integer */
  int i;
  fprintf(outfile,"(");
  for (i=1;i<st->num_dim;i++) {
    fprintf(outfile,"%d ",nc%(st->num_cor[i]/st->num_cor[i-1]));
    nc=nc/(st->num_cor[i]/st->num_cor[i-1]);
  }
  fprintf(outfile,"%d) ",nc%(st->num_cor[i]/st->num_cor[i-1]));
  return nc;
}                                                                         330

void place_node(Node node,int coor,int level) {
  /* places a node */
  node->coor=coor;
  node->level=level;
  node->placed=1;
  st->tess[coor]->nodes=add_el(st->tess[coor]->nodes,node);
  fix_levels(node);
}
                                                                          340
int schedule(Node node,int coor) {
  /* schedules a node given its spatial placement */
  int level,maxlevel;
  el it;
  Node node2;
  level=maxlevel=0;
  for (it=node->inputs;it;it=it->next) {
    node2=it->key;
    if (node2->placed) {
      if (level=sdist(node2->coor,coor)) {                                350
          level+=node2->level;
```

```
      } else {
        level=node2->level+1;
      }
      if (node->latch) {
        level=0;
      }
      if (debug) printf("%d ",sdist(node2->coor,coor));
      if (level > maxlevel) maxlevel=level;
    }                                                                    360
  }
  return maxlevel;
}

int seq_rand_coor() {
  /* returns a random coordinate that has not been generated before */
  static int cnt=0;
  static int *coors;
  static int init=0;
  int i,nc;                                                            370
  if (!init) {
    coors=(int *)malloc(st->num_cor[st->num_dim]*sizeof(int));
    for (i=0;i<st->num_cor[st->num_dim];i++) {
      coors[i]=0;
    }
    init=1;
  }
  if (!coors[nc=st->num_cor[st->num_dim]*ran3(&idum)]) {
    coors[nc]=1;
    cnt++;                                                             380
    return nc;
  } else if (cnt==st->num_cor[st->num_dim]) {
    fprintf(outfile,"All coordinates exhausted\n");
    return -1;
  } else { return rand_coor(); }
}
```

```
int rand_coor() {
  /* returns a random (spatial) coordinate */
  return st->num_cor[st->num_dim]*ran3(&idum);                                390
}

void rand_place(Node node) {
  /* randomly places a node */
  int coor,level,i;
  if (debug) printf("Randomly placing %s\n",node->name);
  level=schedule(node,coor=rand_coor());
   if((node->latch)||(!node->num_in)) {
   while(!compatible(node,coor,level)) level=schedule(node,coor=rand_coor());
  } else {                                                                     400
   for (i=level;;i++) {
    if (compatible(node,coor,i)) {
         level=i;
         break;
    }
   }
  }
  place_node(node,coor,level);
}
                                                                               410
void rand_place_all() {
  /* randomly places all nodes */
  el it;
  int i;
  for (i=INPUT;i<NUM_TYPES;i++) {
   for (it=an->nodes[i];it;it=it->next) {
     if (debug) printf("type %d\n",i);
     rand_place(it->key);
   }
  }                                                                            420
}

void stitch_place_all() {
```

```
  /* random placement without scheduling */
  el it;
  int i;
  Node node;
  for (i=INPUT;i<NUM_TYPES;i++) {
    for (it=an->nodes[i];it;it=it->next) {
      node=it->key;                                                    430
      if (debug) printf("type %d\n",i);
      node->coor=rand_coor();
    }
  }
}

int force_direct(Node node) {
  /* force-directed iteration on a node */
  el it;
  static int *cor;                                                     440
  static int *cor2;
  static int *sum;
  static int init=0;
  int i,level,coor,j;
  Node node2;
  if (!init) {
    cor=(int *) malloc(sizeof(int)*(st->num_dim));
    cor2=(int *) malloc(sizeof(int)*(st->num_dim));
    sum=(int *) malloc(sizeof(int)*(st->num_dim));
    init=1;                                                            450
  }
  for (i=0;i<st->num_dim;i++) sum[i]=0;
  for (it=node->inputs;it;it=it->next) {
    node2=it->key;
    convert(node2->coor,cor2);
    for (i=0;i<st->num_dim;i++) sum[i]+=sdist1(cor[i],cor2[i],i);
  }
  for (it=node->outputs;it;it=it->next) {
    node2=it->key;
```

```
    convert(node2->coor,cor2);                                                          460
    for (i=0;i<st->num_dim;i++) sum[i]+=sdist1(cor[i],cor2[i],i);
  }
  for (i=0;i<st->num_dim;i++) {
    j=st->num_cor[i+1]/st->num_cor[i];
    cor2[i]=0;
    if(node->num_in+node->num_out) {
      cor2[i]=round(sum[i]/(node->num_in+node->num_out));
    }
    if (cor2[i]>0) cor[i]=(cor[i]+1)%j;
    else if (cor2[i]<0) cor[i]=(cor[i]-1+j)%j;                                          470
  }
  coor=revert(cor);
  if (node->coor!=coor) {
    if (compatible(node,coor,level=schedule(node,coor))) {
      if (debug) printf("%d %d %d %d\n",node->coor,coor,level,node->level);
      st->tess[node->coor]->nodes=del_el(st->tess[node->coor]->nodes,el_lookup(st->tess[node->coor]->r
      place_node(node,coor,level);
      return 1;
    }
  }                                                                                     480
  return 0;
}

int force_place_all() {
  /* force-directed placement without scheduling */
  el it;
  int i,change;
  change=0;
  for (i=INPUT;i<NUM_TYPES;i++) {
    for (it=an->nodes[i];it;it=it->next) {                                              490
      if (debug) printf("type %d\n",i);
      change+=force_direct(it->key);
    }
  }
  return change;
```

```
}

void place_all(int mode) {
  /* different methods of placement */
  switch (mode) {                                                    500
  case 0:
    rand_place_all();
    break;
  case 1:
    stitch_place_all();
    break;
  case 2:
    force_place_all();
  }
}                                                                    510

void init_layout() {
  /* initializes the space */
  int i,j;
  st->num_nbhd=2*st->num_dim+1;
  num_pi=num_po=st->num_nbhd;
  st->nbhd=(int *) malloc(sizeof(int)*st->num_nbhd);
  for (i=0;i<st->num_nbhd;i++) {
    if (!i) {
      st->nbhd[i]=i;                                                 520
    } else if (i<2*st->num_dim+1) {
      st->nbhd[i]=(int) pow(-1,i-1)*st->num_cor[(int) (i-1)/2];
    } else {
      j=i-2*st->num_dim-1;
      st->nbhd[i]=(int) pow(-1,j)*round(sqrt(st->num_cor[(int) j/2+1]/st->num_cor[(int) j/2]))*st->nu
    }
  }
  st->tess=(Cell *)malloc(sizeof(Cell)*st->num_cor[st->num_dim]);
  for (i=0;i<st->num_cor[st->num_dim];i++) {
    st->tess[i]=(Cell)malloc(sizeof(struct CellRecord));            530
    st->tess[i]->nodes=NULL;
```

```
    st->tess[i]->nets=NULL;
  }
}

int node_feasible(Node node) {
  /* returns whether or not a node is placed such that its input nodes are
     routable given the placement and scheduling constraint */
  Node node2;
  int sd,td,f=1;                                                        540
  el it;
  for (it=node->inputs;it;it=it->next) {
    node2=it->key;
    if (node->latch) {
      td=st->num_levels-node2->level+node->level;
    } else {
      td=node->level-node2->level;
    }
    if (!(sd=sdist(node2->coor,node->coor))) { sd=1; }
    if (sd>td) {                                                        550
      fprintf(outfile,"%s->%s is unfeasible.\n",node2->name,node->name);
      fprintf(outfile,"%d %d %d\n",sdist(node2->coor,node->coor),node2->level,node->level);
      f=0;
    }
  }
  return f;
}

void net_feasible(Net net) {
  /* tests to see whether the net has been correctly laid out */         560
  int cor[st->num_dim];
  int cor2[st->num_dim];
  el it;
  Net net2;
  if (net->parent_net) {
    for (it=st->tess[net->coor]->nets;it;it=it->next) {
      net2=it->key;
```

```
        if (net2!=net) {
            if (net2->parent_net) {
              if (net2->level==net->level) {                                    570
                if (net->dist==net2->dist) {
                  if ((!net->dist)||(net->dim==net2->dim)) {
                    fprintf(outfile,"\n");
                    printcor(net->parent_net->coor);
                    fprintf(outfile,"%d->",net->parent_net->level);
                    printcor(net->coor);
                    fprintf(outfile,"%d %d %d",net->level,net->dist,net->dim);
                    printcor(net2->parent_net->coor);
                    fprintf(outfile,"%d->",net2->parent_net->level);
                    printcor(net2->coor);                                       580
                    fprintf(outfile,"%d %d %d",net2->level,net2->dist,net2->dim);
                    fprintf(outfile,"unfeasible from nodes %s and %s.",net->parent_node->name,net2->
                  }
                }
              }
            }
        }
      }
    }
    for(it=net->child_nets;it;it=it->next) {                                    590
      net_feasible(it->key);
    }
}

int all_feasible() {
  /* returns whether or not all nodes are feasible, thus saying whether the
     placement is correct in terms of its routability */
  el it;
  int i,f=1;
  for (i=INPUT;i<NUM_TYPES;i++) {                                               600
    for (it=an->nodes[i];it;it=it->next) {
      if (!node_feasible(it->key)) {
          f=0;
```

```
      }
    }
  }
  return f;
}

double avg_dist() {                                                        610
  /* returns the average spatial distance between nodes */
  int i,ad,count;
  el it,it2;
  Node node;
  ad=count=0;
  for (i=INPUT;i<NUM_TYPES;i++) {
    for (it=an->nodes[i];it;it=it->next) {
      node=it->key;
      for (it2=node->inputs;it2;it2=it2->next) {
            ad+=sdist(((Node)it2->key)->coor,node->coor);                 620
            count++;
      }
      for (it2=node->outputs;it2;it2=it2->next) {
            ad+=sdist(((Node)it2->key)->coor,node->coor);
            count++;
      }
    }
  }
  return (double) ad/count;
}                                                                          630

void del_net(Net net) {
  /* deletes a net */
  el it,next;
  Node cnode;
  st->tess[net->coor]->nets=del_el(st->tess[net->coor]->nets,el_lookup(st->tess[net->coor]->nets,net));
  an->num_nets--;
  st->levels[net->level]=del_el(st->levels[net->level],el_lookup(st->levels[net->level],net));
  for (it=net->child_nets;it;it=next) {
```

```
    next=it->next;                                                          640
    del_net(it->key);
    free(it);
  }
  for (it=net->child_nodes;it;it=next) {
    cnode=it->key;
    if (cnode->level==net->level) {
      cnode->inets=del_el(cnode->inets,el_lookup(cnode->inets,net));
    }
    next=it->next;
    free(it);                                                               650
  }
  free(net);
}

void unroute(Node node) {
  /* removes the layout of a nodes net */
  an->nodes[UNROUTED-1]=add_el(an->nodes[UNROUTED-1],node);
  an->num_unrouted++;
  if (node->net) del_net(node->net);
  node->net=NULL;                                                           660
}

Net blocked(Node pnode,int cor,int level,int dist,int dim) {
  /* returns whether a not a net can be routed through this coordinate
     and level given its origin */
  el it;
  Net net;
  for (it=st->tess[cor]->nets;it;it=it->next) {
    net=it->key;
    if ((net->level==level%st->num_levels)&&(net->parent_node!=pnode)) {    670
      if ((dim!=st->num_dim)&&(net->parent_node->net!=net)) {
            if (dist==net->dist) {
              if ((!dist)||(net->dim==dim)) {
                return net;
              }
```

```
          }
        }
      }
    }
    return NULL;                                                                680
  }

  Net greedy(Node nsrc,Node ndest,int cor[st->num_dim],int relcor[st->num_dim],int sd,int td,int level,int idist,
    /* greedy annealing router */
    int dim,dist,dim_size,num_try,i;
    int tried[st->num_dim];
    Net  net;
    int blk=0;
    Net net_blk[num_pi];
    Node node;                                                                  690
    if (td) {
      for (i=0;i<st->num_dim;i++) {
        tried[i]=0;
      }
      num_try=0;
      do {
        if (!tried[dim=ran3(&idum)*st->num_dim]) {
              dist=relcor[dim];
              if (dist) {
                if (dist>0) { dist=1; } else { dist=-1; }                       700
                dim_size=st->num_cor[dim+1]/st->num_cor[dim];
                cor[dim]=(cor[dim]+dist+dim_size)%dim_size;
                relcor[dim]-=dist;
                if (!(net=blocked(nsrc,revert(cor),level+1,dist,dim))) {
                  if (net=greedy(nsrc,ndest,cor,relcor,sd-1,td-1,level+1,dist,dim)) {
                    if (net->parent_net) {
                          return net->parent_net;
                    } else {
                          cor[dim]=(cor[dim]-dist+dim_size)%dim_size;
                          return add_net(nsrc,ndest,net,revert(cor),level,idist,idim);   710
                    }
```

```
        }
      } else {
        net_blk[blk++]=net;
      }
      cor[dim]=(cor[dim]-dist+dim_size)%dim_size;
      relcor[dim]+=dist;
    }
    tried[dim]++;
    num_try++;                                                        720
  }
} while (num_try<st->num_dim);
if (sd==td) {
  if ((ran3(&idum)<exp(-1/(temp*an->num_unrouted)))&&blk) {
      node=net_blk[blk=ran3(&idum)*blk]->parent_node;
        unroute(node);
        return greedy(nsrc,ndest,cor,relcor,sd,td,level,idist,idim);
  } else { return NULL; }
} else {
  if (!(net=blocked(nsrc,revert(cor),level+1,0,dim))) {              730
      if (net=greedy(nsrc,ndest,cor,relcor,sd,td-1,level+1,0,dim)) {
        if (net->parent_net) {
          return net->parent_net;
        } else {
          return add_net(nsrc,ndest,net,revert(cor),level,idist,idim);
        }
      }
  } else { net_blk[blk++]=net; }
}
if (sd==td-1) {                                                       740
  if((ran3(&idum)<exp(-1/(temp*an->num_unrouted)))&&blk) {
      node=net_blk[blk=ran3(&idum)*blk]->parent_node;
      unroute(node);
      return greedy(nsrc,ndest,cor,relcor,sd,td,level,idist,idim);
  } else { return NULL; }
} else {
  for (i=0;i<st->num_dim;i++) tried[i]=0;
```

```
    num_try=0;
    do {
      if (!tried[dim=ran3(&idum)*st->num_dim]) {                                       750
        dist=relcor[dim];
        if (dist>0) dist=-1;
        else if (dist<0) dist=1;
        dim_size=st->num_cor[dim+1]/st->num_cor[dim];
        tried[dim]=(dist ? 1 : 2);
        for (i=0;i<tried[dim];i++) {
          if (!dist) dist=-1;
          if (i) dist=1;
          cor[dim]=(cor[dim]+dist+dim_size)%dim_size;
          relcor[dim]-=dist;                                                           760
          if (!(net=blocked(nsrc,revert(cor),level+1,dist,dim))) {
            if (net=greedy(nsrc,ndest,cor,relcor,sd+1,td-1,level+1,dist,dim)) {
                if (net->parent_net) {
                  return net->parent_net;
                } else {
                  cor[dim]=(cor[dim]-dist+dim_size)%dim_size;
                  return add_net(nsrc,ndest,net,revert(cor),level,idist,idim);
                }
            }
          } else { net_blk[blk++]=net; }                                               770
          cor[dim]=(cor[dim]-dist+dim_size)%dim_size;
          relcor[dim]+=dist;
        }
        num_try++;
      }
    } while (num_try<st->num_dim);
    if ((ran3(&idum)<exp(-1/(temp*an->num_unrouted)))&&blk) {
      node=net_blk[blk=ran3(&idum)*blk]->parent_node;
        unroute(node);
        return greedy(nsrc,ndest,cor,relcor,sd,td,level,idist,idim);                   780
    } else { return NULL; }
  }
} else {
```

```
    net=add_net(nsrc,ndest,NULL,revert(cor),level,idist,idim);
    if (!((net->parent_net)&&(el_lookup(ndest->inets,net)))) {
      ndest->inets=add_el(ndest->inets,net);
    }
    return net;
  }
}                                                                                    790

Net route(Node node1,Node node2) {
  /* produces a path from node 1 to node 2 */
  static int *cor1;
  static int *cor2;
  static int *relcor;
  static int init=0;
  Net net;
  int i,sd=0;
  if (!init) {                                                                       800
    cor1=(int *)malloc(sizeof(int)*st->num_dim);
    cor2=(int *)malloc(sizeof(int)*st->num_dim);
    relcor=(int *)malloc(sizeof(int)*st->num_dim);
    init=1;
  }
  convert(node1->coor,cor1);
  convert(node2->coor,cor2);
  for (i=0;i<st->num_dim;i++) {
    relcor[i]=sdist1(cor1[i],cor2[i],i);
    sd+=abs(relcor[i]);                                                              810
  }
  if (net=greedy(node1,node2,cor1,relcor,sd,(node2->latch ? st->num_levels : node2->level)-node1->le
    node1->net=net;
  }
  return net;
}

int routeNet(Node node) {
  /* routes node's net */
```

```
  el it;                                                          820
  int f=0;
  for (it=node->outputs;it;it=it->next) {
    if (!route(node,it->key)) f++;
  }
  if (f) unroute(node);
  return f;
}

int routeAll() {
  /* routes all the nodes */                                      830
  int i,j;
  el it,next;
  int s=0;
  for (it=an->nodes[UNROUTED-1];it;it=next) {
    next=it->next;
    an->nodes[UNROUTED-1]=del_el(an->nodes[UNROUTED-1],it);
    an->num_unrouted--;
    s+=routeNet(it->key);
  }
  return s;                                                       840
}



/* lut.c */

#include "structures.h"
#include "util.h"
#include "struct.h"
#include "global.h"                                               850
#include "user.h"
#include "max.h"

NLIST **hash_initial();
STATE *install_state();
```

```
STATE **states;
EDGE **edges;
static int lut;
char b_file[1];
struct u user;                                                     860
long t_start;
static FILE *kiss;
struct statespace *ss;
int ctypes=0;
int nct;
int oct;
int olcfgs;
int mw;
extern String fn;
extern FILE *outfile;                                              870
extern int slut;
extern struct isomor *iso;
alloc_max_block(int);

void install_cube(Cube cube,int expand,int num) {
 /* installs a cube into an integer which implements a temporary LUT */
 int i;
 switch(cube[0]) {
 case '\0':
  lut|=1<<expand;                                                  880
  return;
 case '-':
  install_cube(cube+1,expand,num+1);
  install_cube(cube+1,expand+(1<<num),num+1);
  return;
 case '0':
  install_cube(cube+1,expand,num+1);
  return;
 case '1':
  install_cube(cube+1,expand+(1<<num),num+1);                      890
  return;
```

```
  }
}

static NLIST **state_hash;                /* a hash table stores state name */

int neighbor(Net net) {
  /* computes the neighborhood number of this net */
  if (net->dist) return 2*net->dim+1+((net->dist+1) ? 0 : 1);
  return 0;                                                              900
}

static int edge_index = 0;          /* a counter when saving edges */
static int hnum_st;

void install_fa(int snum,int stnum) {
  /* installs a state for minimization */
  EDGE *edge_ptr;                   /* pointer to EDGE structure */
  EDGE *search_edge;
  char pstate_name[5];     /* the name of present state */                 910
  STATE *pstate_ptr;       /* pointer to STATE structure */
  el it,it2,next,next2;
  (void) sprintf (pstate_name, "%d", stnum);
  pstate_ptr = install_state(pstate_name,state_hash,hnum_st);
  pstate_ptr->edge=NULL;
  if ( pstate_ptr == NIL(STATE) ) {
    panic("failed to install a state.");
  }
  it2=ss->outputs[snum];
  for (it=ss->inputs[snum];it;it=next) {                                   920
    next=it->next;
    next2=it2->next;
    if ( (edge_ptr = ALLOC (EDGE, 1)) == NIL(EDGE) ) {
      panic("ALLOC edge");
    }
    edge_ptr->next = NIL(EDGE);
    edges[edge_index] = edge_ptr;  /* save the address of new edge*/
```

```
  edge_index++;
  edge_ptr->input=it->key;
  FREE(it);                                                                     930
  edge_ptr->output=it2->key;
  FREE(it2);
  it2=next2;
  edge_ptr->n_star  = 0;
  edge_ptr->p_star = 0;
  edge_ptr->p_state = pstate_ptr;
  if (pstate_ptr->edge != NIL(EDGE)) {
    search_edge = pstate_ptr->edge;
    while (search_edge->next != NIL(EDGE))
          search_edge = search_edge->next;                                      940
    search_edge->next = edge_ptr;
  } else pstate_ptr->edge = edge_ptr;
  edge_ptr->n_state = pstate_ptr;
  if(debug) fprintf(kiss,"%s %d %d %s\n",edges[edge_index-1]->input,snum,snum,edges[edge_index-1]->
 }
 ss->inputs[snum]=ss->outputs[snum]=NULL;
}

void install_cell(int snum,int stnum) {
 /* installs a state for the first minimization pass */                         950
 EDGE *edge_ptr;                    /* pointer to EDGE structure */
 EDGE *search_edge;
 Net net;
 Node node=NULL;
 int nbin[num_pi],nbout[num_pi][num_po],nodein[num_pi],nodeout[num_pi];
 char pstate_name[5];       /* the name of present state */
 STATE *pstate_ptr;         /* pointer to STATE structure */
 el it,it2;
 int i,j,k,num_in,val,n;
 char value;                                                                    960
 (void) sprintf (pstate_name, "%d", stnum);
 pstate_ptr = install_state(pstate_name,state_hash,hnum_st);
 pstate_ptr->edge=NULL;
```

```
if ( pstate_ptr == NIL(STATE) ) {
  panic("failed to install a state.");
}
for(i=0;i<num_pi;i++) {
  nbin[i]=nodein[i]=nodeout[i]=-1;
  for(j=0;j<num_po;j++) nbout[i][j]=-1;
}                                                                          970
i=0;
for (it=ss->cfgs[snum];it;it=it->next) {
  net=it->key;
  if (net->parent_net) {
    nbin[i]=neighbor(net);
    j=0;
    for (it2=net->child_nets;it2;it2=it2->next) {
          nbout[nbin[i]][j++]=neighbor(it2->key);
    }
    i++;                                                                   980
  } else { node=net->parent_node; }
}
if (node) {
  for (it2=node->inets;it2;it2=it2->next) {
    net=it2->key;
    nodein[neighbor(net)]=el_num(node->inputs,net->parent_node);
  }
  j=0;
  if(node->net) {
    for (it2=node->net->child_nets;it2;it2=it2->next) {                    990
          nodeout[j++]=neighbor(it2->key);
    }
  }
  lut=0;
  for (it2=node->cubes;it2;it2=it2->next) {
    install_cube(it2->key,0,0);
  }
}
num_in=i;
```

```
for (i=0;i<1<<num_in;i++) {                                                    1000
  if ( (edge_ptr = ALLOC (EDGE, 1)) == NIL(EDGE) ) {
    panic("ALLOC edge");
  }
  edge_ptr->next = NIL(EDGE);
  edges[edge_index] = edge_ptr;   /* save the address of new edge*/
  edge_index++;
  if (!(edge_ptr->input =ALLOC ( char, num_pi + 1 ))) {
    panic("ALLOC input");
  }
  for(j=0;j<num_pi;j++) edge_ptr->input[j]='-';                                 1010
  edge_ptr->input[j]='\0';
  if (!(edge_ptr->output=ALLOC( char, num_po + 1))) {
    panic("ALLOC edge output");
  }
  for(j=0;j<num_po;j++) edge_ptr->output[j]='-';
  edge_ptr->output[j]='\0';
  val=0;
  for (j=0;j<num_in;j++) {
    edge_ptr->input[nbin[j]]=(k=(i>>j)&1) ? '1' : '0';
    if ((n=nodein[nbin[j]])+1) {                                                1020
         val+=k<<n;
    }
    k=0;
    while (((n=nbout[nbin[j]][k])+1)&&(num_pi>k++)) {
         edge_ptr->output[n]=edge_ptr->input[nbin[j]];
    }
  }
  if (node) {
    value=(lut>>val)&1 ? '1' : '0';
    j=0;                                                                        1030
    while (((n=nodeout[j])+1)&&(num_pi>j++)) {
         edge_ptr->output[n]=value;
    }
  }
  edge_ptr->n_star  = 0;
```

```
  edge_ptr->p_star = 0;
  edge_ptr->p_state = pstate_ptr;
  if (pstate_ptr->edge != NIL(EDGE)) {
    search_edge = pstate_ptr->edge;
    while (search_edge->next != NIL(EDGE))                                    1040
          search_edge = search_edge->next;
    search_edge->next = edge_ptr;
  } else pstate_ptr->edge = edge_ptr;
  edge_ptr->n_state = pstate_ptr;
  if(debug) fprintf(kiss,"%s %d %d %s\n",edge_ptr->input,snum,snum,edge_ptr->output);
 }
}

void install_net(Net net) {
 /* initializes the states of cells by cycling through the nets */              1050
 el it;
 Net net2;
 if(!net->snum) {
  ++ss->num_cfgs;
  for(it=st->tess[net->coor]->nets;it;it=it->next) {
    net2=it->key;
    if(net2->level==net->level) {
          net2->snum=ss->num_cfgs;
          ss->cfgs[ss->num_cfgs]=add_el(ss->cfgs[ss->num_cfgs],net2);
    }                                                                         1060
  }
 }
 for(it=net->child_nets;it;it=it->next) install_net(it->key);
}

void count_prod(int snum) {
 /* counts the number of products for state snum */
 el it;
 int i=0;
 for(it=ss->cfgs[snum];it;it=it->next) if(((Net)it->key)->parent_net) i++;     1070
 num_product+=1<<i;
```

```
}

void install_trans(String input,int stnum,String output) {
  /* stores a product for use in the next minimization pass or writing the LUT */
  if(slut) {
    install_tlu(input,stnum,output,0,0);
  } else {
    ss->inputs[stnum]=add_el(ss->inputs[stnum],strdup(input));
    ss->outputs[stnum]=add_el(ss->outputs[stnum],strdup(output));     1080
    ss->nump[stnum]++;
  }
}

void reassign(int st1,int st2) {
  /* reassigns state st1 to st2, which is done when states are minimized */
  el it,it2;
  Net net;
  if(ss->temp[st1]) it=it2=ss->temp[st1];
  else it=it2=ss->cfgs[st1];                                           1090
  for(;it->next;it=it->next) {
    net=it->key;
    net->snum=st2;
  }
  net=it->key;
  net->snum=st2;
  if(ss->temp[st2]) {
    it->next=ss->cfgs[st2];
    ss->cfgs[st2]->prev=it;
    ss->cfgs[st2]=it2;                                                 1100
  } else {
    ss->temp[st2]=ss->cfgs[st2];
    ss->cfgs[st2]=it2;
  }
}

void init_ss() {
```

```
/* initial state minimization pass */
int (**do_work)();
int say_solution();                                                   1110
extern int (*method1[])();
int i,j,ptotal=0;
el it,it2;
Node node;
ss=(struct statespace *)malloc(sizeof(struct statespace));
ss->num_cfgs=0;
ss->cfgs=(el *)malloc(an->num_nets*sizeof(el));
ss->temp=(el *)malloc(an->num_nets*sizeof(el));
ss->inputs=(el *)malloc(an->num_nets*sizeof(el));
ss->outputs=(el *)malloc(an->num_nets*sizeof(el));                    1120
ss->nump=(int *)malloc(an->num_nets*sizeof(int));
for(i=0;i<an->num_nets;i++) {
  ss->cfgs[i]=NULL;
  ss->temp[i]=NULL;
  ss->inputs[i]=NULL;
  ss->outputs[i]=NULL;
  ss->nump[i]=0;
}
for (it=an->nodes[0];it;it=it->next) {
  node=it->key;                                                       1130
  if(node->net) {
    for(it2=node->net->child_nets;it2;it2=it2->next) {
        install_net(it2->key);
    }
  }
}
for (i=INT_NODE;i<NUM_TYPES;i++) {
  for (it=an->nodes[i];it;it=it->next) {
    node=it->key;
    if(node->net) install_net(node->net);                             1140
  }
}
b_file[0]=0;
```

```
user.level=8;
user.oname="out.kiss";
user.opt.hmap=4;
user.opt.solution=0;
user.opt.verbose=0;
user.cmd.merge=0;
user.cmd.shrink=1;                                                    1150
user.cmd.trans=0;
hnum_st=num_st;
if ( (state_hash = hash_initial(num_st)) == NIL(NLIST *) ) {
  panic("ALLOC hash");
}

/***
*** allocate memory for **states, and **edges.
***/
                                                                      1160
if ( (states = ALLOC(STATE *, num_st)) == NIL(STATE *) ) {
  panic("ALLOC state");
}
max = (PRIMES) 0;
alloc_max_block(0);
if (!(iso=ALLOC(struct isomor,num_st)))
  panic("iso");
for(i=0;i<num_st;i++) {
  if (!(iso[i].list=ALLOC(int,num_st)))
    panic("iso_find3");                                               1170
}
for(oct=1;oct<=ss->num_cfgs;oct+=num_st) {
  if(ss->num_cfgs-oct+1<num_st) {
    num_st=ss->num_cfgs-oct+1;
  }
  if(debug) kiss=fopen("in.kiss","a");
  if(debug) fprintf(kiss,".i %d\n",num_pi);
  if(debug) fprintf(kiss,".o %d\n",num_po);
  num_product=0;
```

```
    for (j=oct;j<oct+num_st;j++) count_prod(j);                                    1180
    fprintf(outfile,"\nThere are %d states.\n",num_st);
    fprintf(outfile,"There are %d products.\n",num_product);
    if(debug) fprintf(kiss,".p %d\n",num_product);
    if(debug) fprintf(kiss,".s %d\n",num_st);
    if ( (edges = ALLOC(EDGE *, num_product )) == NIL(EDGE *) ) {
      panic("ALLOC edge");
    }
    edge_index=0;
    for (j=oct;j<oct+num_st;j++) {
      install_cell(j,j-oct+1);                                                     1190
    }
    if(debug) fclose(kiss);
    t_start=util_cpu_time();
    nct=ctypes+1;
    j=0;
    for (do_work=method1; *do_work; do_work++) {
      printf("%d\n",j++);
      (**do_work)();
    }
    for (i=0;i<num_st;i++) states[i]->assigned=0;                                  1200
    ptotal+=user.stat.product;
    for(j=0;j<num_product;j++) {
      FREE(edges[j]->input);
      FREE(edges[j]->output);
      FREE(edges[j]);
    }
    FREE(edges);
  }
  fprintf(outfile,"There were %d states\n",ss->num_cfgs);
  olcfgs=ss->num_cfgs;                                                             1210
  fprintf(outfile,"Now, there are %d states and %d products.\n",ctypes,ptotal);
  for(i=0;i<an->num_nets;i++) ss->temp[i]=NULL;
  ss->num_cfgs=ctypes;
  num_st=hnum_st;
}
```

```
void min_ss() {
  /* subsequent state minimization passes */
  int (**do_work)();
  int say_solution();                                          1220
  extern int (*method1[])();
  int i,j,ptotal=0;
  ctypes=0;
  for(oct=1;oct<=ss->num_cfgs;oct+=num_st) {
    if(ss->num_cfgs-oct+1<num_st) {
      num_st=ss->num_cfgs-oct+1;
    }
    if(debug) kiss=fopen("in.kiss","a");
    if(debug) fprintf(kiss,".i %d\n",num_pi);
    if(debug) fprintf(kiss,".o %d\n",num_po);                  1230
    num_product=0;
    for (j=oct;j<oct+num_st;j++) {
      num_product+=ss->nump[j];
      ss->nump[j]=0;
      states[j-oct]->assigned=0;
    }
    fprintf(outfile,"\nThere are %d states.\n",num_st);
    fprintf(outfile,"There are %d products.\n",num_product);
    if(debug) fprintf(kiss,".p %d\n",num_product);
    if(debug) fprintf(kiss,".s %d\n",num_st);                  1240
    if ( (edges = ALLOC(EDGE *, num_product )) == NIL(EDGE *) ) {
      panic("ALLOC edge");
    }
    edge_index=0;
    for (j=oct;j<oct+num_st;j++) {
      install_fa(j,j-oct+1);
    }
    fclose(kiss);
    t_start=util_cpu_time();
    nct=ctypes+1;                                              1250
    for (do_work=method1; *do_work; do_work++) (**do_work)();
```

```
    ptotal+=user.stat.product;
    for(j=0;j<num_product;j++) {
      FREE(edges[j]->input);
      FREE(edges[j]->output);
      FREE(edges[j]);
    }
    FREE(edges);
  }
  fprintf(outfile,"There were %d states\n",ss->num_cfgs);                          1260
  fprintf(outfile,"Now, there are %d states and %d products.\n",ctypes,ptotal);
  for(i=0;i<an->num_nets;i++) ss->temp[i]=NULL;
  ss->num_cfgs=ctypes;
  num_st=hnum_st;
}

static short tab[1<<NUM_LAYERS];

void install_tlu(String inp,int stnum,String out,int num,int val) {
  /* installs a product into the LUT */                                            1270
  int ol,nl,i,np=num_pi+1;
  switch(inp[0]) {
  case '\0':
    nl=0;
    ol=tab[(val<<1)+(stnum<<np)];
    for(i=0;i<num_po;i++) {
      if(out[i]=='-') nl+=(ol&1)<<i;
      else nl+=((out[i]=='1') ? 1 : 0)<<i;
      ol>>=1;
    }                                                                               1280
    for(i=0;i<2;i++) {
      tab[(val<<1)+(stnum<<np)+i]=(nl<<1)+(stnum<<np)+(nl ? 1 : 0);
    }
    return;
  case '-':
    install_tlu(inp+1,stnum,out,num+1,val);
    install_tlu(inp+1,stnum,out,num+1,val+(1<<num));
```

```
    return;
  case '0':
    install_tlu(inp+1,stnum,out,num+1,val);                         1290
    return;
  case '1':
    install_tlu(inp+1,stnum,out,num+1,val+(1<<num));
    return;
  }
}

void make_pat() {
  /* configures the initial pattern */
  short pat[st->num_cor[st->num_dim]];                              1300
  Net net;
  int i,j,width;
  el it;
  FILE *fpat;
  for(i=0;i<st->num_cor[st->num_dim];i++) pat[i]=0;
  fpat=fopen(strcat(fn,".pat"),"w+b");
  fn=strtok(fn,".");
  j=1;
  mw=0;
  for(i=st->num_levels-1;i;i>>=1) j<<=1;                            1310
  for(i=0;i<j;i++) {
    if(i<st->num_levels) {
      width=0;
      for(it=st->levels[i];it;it=it->next) {
            net=it->key;
            width++;
            pat[net->coor]=(net->snum)<<(num_pi+1);
      }
      if(width>mw) mw=width;
      fwrite(pat,sizeof(pat),1,fpat);                               1320
      for(it=st->levels[i];it;it=it->next) {
            net=it->key;
            pat[net->coor]=0;
```

```
    }
  } else { fwrite(pat,sizeof(pat),1,fpat); }
 }
 fclose(fpat);
}

void make_lut() {                                                     1330
 /* configures the LUT */
 int i,j,nc;
 el it,it2,next,next2;
 Net net;
 FILE *flut;
 for(i=0;i<1<<num_pi;i++) tab[(i<<1)]=((i&1) ? (1<<num_pi)-1 : 0)<<1;
 for(i=0;i<=ss->num_cfgs;i++) {
  it2=ss->outputs[i];
  for(it=ss->inputs[i];it;it=next,it2=next2) {
   next=it->next;                                                     1340
   next2=it2->next;
   install_tlu(it->key,i,it2->key,0,0);
   FREE(it->key);
   FREE(it2->key);
   FREE(it);
   FREE(it2);
  }
 }
 flut=fopen(strcat(fn,".tab"),"w+b");
 fn=strtok(fn,".");                                                   1350
 fwrite(tab,sizeof(tab),1,flut);
 fclose(flut);
}

void panic(msg)
char *msg;
{
      (void) fprintf(stderr,"Panic:  %s",msg);
      exit(1);
```

```
}                                                            1360
/* main.c */

#include "structures.h"
#include "struct.h"
#include "global.h"
#include "lut.h"
extern int yydebug;
extern FILE *yyin;
struct stt *fsm;
struct spacetime *st;                                        1370
struct AllNodes *an;
extern struct statespace *ss;
extern int olcfgs;
extern int mw;
int merging;
int num_st=25;
int num_product=0;
int num_pi=0;
int num_po=0;
int def_clk=0;                                               1380
int rtv=0;
int slut=0;
String fn;
float temp;
FILE *outfile=NULL;
int yyparse(void);

yyerror(const char *s)
{
  fprintf(stderr,"%s\n",s);                                  1390
}

main(int argc,char *argv[])
{
  int i,j,cnt,init;
```

```
int fnum=100;
int rnum=100;
float tfac=0.9;
double ad1,ad2;
int ins=0;                                                    1400
int num_ss=1;
int prout=0;
int do_sm=1;
int dop=1;
int nlbef;
el it;
yydebug=0;
if (argc==1) {
  print_usage();
  exit(0);                                                    1410
}
st=(struct spacetime *)malloc(sizeof(struct spacetime));
st->lwires=0;
st->num_pluts=8;
init=0;
for(i=1;i<argc;i++)
  {
    if(argv[i][0]=='-')
        {
          switch(argv[i][1])                                  1420
            {
            case 'a':
              pofcomp(argv[i+1],argv[i+2]);
              exit(0);
            case 'b':
              slut=1;
              break;
            case 'h':
              print_usage();
              exit(0);                                        1430
            case 'c':
```

```
  def_clk=1;
  break;
case 'D':
  debug=1;
  break;
case 'E':
  yydebug=1;
  break;
case 'f':                                                          1440
  fnum=atoi(argv[i]+2);
  break;
case 'l':
  st->lwires=1;
  break;
case 'd':
  st->num_dim=atoi(argv[i]+2);
  st->num_cor=(int *)malloc(sizeof(int)*(st->num_dim+1));
  for (j=0;j<st->num_dim+1;j++) {
        st->num_cor[j]=1<<(6*j);                                   1450
  }
  init=1;
  break;
case 'm':
  num_st=atoi(argv[i]+2);
  break;
case 'n':
  num_ss=atoi(argv[i]+2);
  break;
case 'o':                                                          1460
  prout=1;
  break;
case 'p':
  st->num_pluts=atoi(argv[i]+2);
  break;
case 'q':
  dop=0;
```

```
      break;
    case 'r':
      rnum=atoi(argv[i]+2);                                           1470
      break;
    case 's':
      do_sm=0;
      break;
    case 't':
      sscanf(argv[i]+2,"%f",&tfac);
      break;
    case 'v':
      rtv=1;
      break;                                                          1480
    case 'x':
      st->num_cor[1]=1<<atoi(argv[i]+2);
      for (j=2;j<st->num_dim+1;j++) {
            st->num_cor[j]=st->num_cor[j-1]*(1<<6);
      }
      break;
    case 'y':
      st->num_cor[2]=st->num_cor[1]*(1<<atoi(argv[i]+2));
      for (j=3;j<st->num_dim+1;j++) {
            st->num_cor[j]=st->num_cor[j-1]*(1<<6);                   1490
      }
      break;
    case 'z':
      st->num_cor[3]=st->num_cor[2]*(1<<atoi(argv[i]+2));
      break;
    }
  }
else if(!ins)
  {
   yyin=fopen(argv[i],"r");                                           1500
   fn=strtok(strdup(argv[i]),".");
   if((!yyin)||(!strlen(argv[i])))
     {
```

```
          fprintf(stderr,"cannot open %s for read\n",argv[i]);
          exit(1);
        }
      ins=1;
    }
  else
    {                                                          1510
      outfile=fopen(argv[i],"w");
      if(!outfile)
        {
          fprintf(stderr,"cannot open %s for write\n",argv[i]);
          exit(1);
        }
    }
  }
if(!outfile) {
  if(prout) outfile=stdout;                                    1520
  else {
    outfile=fopen(strcat(fn,".out"),"w");
    fn=strtok(fn,".");
  }
}
if (!init) {
  st->num_dim=3;
  st->num_cor=(int *)malloc(sizeof(int)*(st->num_dim+1));
  for (i=0;i<st->num_dim+1;i++) {
    st->num_cor[i]=1<<(6*i);                                   1530
  }
  init=1;
}
fsm=(struct stt *)malloc(sizeof(struct stt));
an=(struct AllNodes *)malloc(sizeof(struct AllNodes));
for (i=INPUT;i<NUM_TYPES+2;i++) { an->nodes[i]=NULL; };
an->max_stage=0;
an->num_nodes=0;
an->num_nets=0;
```

```
an->num_unrouted=0;                                                                1540
for(i=0;i<argc;i++) { fprintf(outfile,"%s ",argv[i]); }
fprintf(outfile,"\n");
/* read BLIF netlist */
yyparse();
merging=0;
/* placement phase */
init_layout();
fprintf(outfile,"Randomly placing nodes...\n");
place_all(0);
st->num_levels=get_num_levels();                                                   1550
if (all_feasible()) { fprintf(outfile,"Circuit network layout is feasible.\n"); }
fprintf(outfile,"Number of nodes is %d\n",an->num_nodes);
fprintf(outfile,"Number of levels is %d\n",st->num_levels);
nlbef=st->num_levels;
fprintf(outfile,"Average distance is %f\n",ad1=avg_dist());
fprintf(outfile,"Spacetime volume is %d\n",(st->num_cor[st->num_dim]<st->num_pluts ? 1 : st->num_co
fprintf(outfile,"Doing force-directed placement...\n");
for (i=0;i<fnum;i++) {
  cnt=force_place_all();
}                                                                                  1560
fprintf(outfile,"done\n");
st->num_levels=get_num_levels();
st->levels=(el *)malloc(st->num_levels*sizeof(el));
for(i=0;i<st->num_levels;i++) st->levels[i]=NULL;
fprintf(outfile,"Average distance is now %f\n",ad2=avg_dist());
if (all_feasible()) { fprintf(outfile,"Circuit network layout is feasible.\n"); }
fprintf(outfile,"Number of levels is now %d\n",st->num_levels);
fprintf(outfile,"Spacetime volume is now %d\n",(st->num_cor[st->num_dim]<st->num_pluts ? 1 : st->nu
depth();
fprintf(outfile,"Circuit depth is %d\n",an->max_stage);                            1570
/* routing phase */
fprintf(outfile,"Performing iterative routing of the nodes...\n");
cnt=0;
temp=1;
while (an->num_unrouted&&(cnt<rnum)) {
```

```
    routeAll();
    cnt++;
    fprintf(outfile,"Pass %d yielded %d unrouted nodes at temperature %f\n",cnt,an->num_unrouted,
    temp*=tfac;
  }                                                                                   1580
  if (an->num_unrouted) {
    printf("Routing failed on following nodes:\n");
    for (it=an->nodes[UNROUTED-1];it;it=it->next) {
      printf("%s\n",((Node)it->key)->name);
    }
    if(dop) print_connections();
    exit(1);
  } else fprintf(outfile,"Routing succeeded.\n");
  fprintf(outfile,"The number of nets is %d\n",an->num_nets);
  /* table and pattern generation */                                                  1590
  init_ss();
  for(i=0;i<num_ss-1;i++) min_ss();
  if (!slut) make_lut();
  make_pat();
  /* test vector generation */
  if(do_sm) write_pif();
  if(dop) print_connections();
  fclose(outfile);
}
                                                                                      1600
print_usage() {
  fprintf(stderr,"cals v1.0\n\n");
  fprintf(stderr,"usage:  cals [-ahclosv] [-fdmnprtxyz<num>] <filename1[.blif or .pof]> [<file
  fprintf(stderr,"-a:  compare the two pof files after this argument\n");
  fprintf(stderr,"-b:  this is a big file, so do only one pass at state min\n");
  fprintf(stderr,"-h:  show this help\n");
  fprintf(stderr,"-c:  use internal or external clocks\n");
  fprintf(stderr,"-l:  use long wires\n");
  fprintf(stderr,"-o:  print info to standard output if no second filename\n");
  fprintf(stderr,"-s:  do not make pif and pof files for fsm\n");                     1610
  fprintf(stderr,"-q:  do not print out the layout\n");
```

```
  fprintf(stderr,"-v:  generate random test vector component in place of don't care\n");
  fprintf(stderr,"-f<num>:  number of iterations for force-direct placement\n");
  fprintf(stderr,"-d<num>:  number of dimensions\n");
  fprintf(stderr,"-m<num>:  state minimization chunk sizes\n");
  fprintf(stderr,"-n<num>:  number of state minimization recursions\n");
  fprintf(stderr,"-p<num>:  number of (parallel) lookup-tables\n");
  fprintf(stderr,"-r<num>:  number of iterations for annealed routing\n");
  fprintf(stderr,"-t<num>:  annealing temperature exponential factor\n");
  fprintf(stderr,"-x<num>:  size of x-dimension\n");                                1620
  fprintf(stderr,"-y<num>:  size of y-dimension\n");
  fprintf(stderr,"-z<num>:  size of z-dimension\n");
  fprintf(stderr,"\n
Note that the dimension sizes must be entered in x,y,z order and should be\n
consistent with the number of dimensions.\n");
  fprintf(stderr,"
If the -a option is used to compare two pof files, then the two filesnames\n
used are the arguments following it.\n");
  fprintf(stderr,"
If the -o option is used without a second filename, the generated info gets\n   1630
dumped to standard output, otherwise the generated info gets dumped to the\n
file specified by the second filename\n");
  fprintf(stderr,"
If a second filename is not specified, the generated info is dumped to\n
filename1[.tab]\n");
  fprintf(stderr,"All other files produced as output are as follows\n");
  fprintf(stderr,"CAM-8 LUT:  filename1[.tab]\n");
  fprintf(stderr,"CAM-8 pattern:    filename1[.pat]\n");
  fprintf(stderr,"CAM-8 probe input file:    filename1[.pif]\n");
  fprintf(stderr,"CAM-8 probe output file (for comparison):    filename1[.pof]\n");  1640
}
/* netlist.c */

#include "structures.h"
extern FILE *outfile;

void finalErr(char *s)
```

```
{
  fputs("*** ",stderr);
  fputs(s,stderr);
  fputs(" ***\n",stderr);
  exit(1);
}

Net add_net(Node parent_node,Node child_node,Net child_net,int coor,int level,int dist,int dim) {
  /* makes a new net based on the parameters and returns it */
  Net net;
  el it;
  for (it=st->tess[coor]->nets;it;it=it->next) {
    net=it->key;
    if (net->level==level) {
      if ((net->parent_node==parent_node)&&(!el_lookup(parent_node->inets,net))) {
            net->child_nodes=add_el(net->child_nodes,child_node);
            if (child_net) {
              net->child_nets=add_el(net->child_nets,child_net);
              child_net->parent_net=net;
            }
            return net;
      }
    }
  }
  net=(Net)malloc(sizeof(struct NetRecord));
  an->num_nets++;
  net->coor=coor;
  net->level=level%st->num_levels;
  st->levels[net->level]=add_el(st->levels[net->level],net);
  net->parent_node=parent_node;
  net->dist=dist;
  net->dim=dim;
  net->snum=0;
  net->child_nodes=net->child_nets=net->parent_net=NULL;
  net->child_nodes=add_el(net->child_nodes,child_node);
  if (child_net) {
```

```
    net->child_nets=add_el(net->child_nets,child_net);
    child_net->parent_net=net;
  }
  st->tess[coor]->nets=add_el(st->tess[coor]->nets,net);
  return net;
}
                                                                          1690
Node make_node(Name name,ntype type) {
  /* makes a new node */
  Node node=(Node) malloc(sizeof(struct NodeRecord));
  node->name=strdup(name);
  node->type=type;
  node->placed=node->latch=node->clatch=0;
  node->cubes=node->inputs=node->outputs=node->net=node->inets=NULL;
  node->coor=node->level=0;
  an->nodes[type]=add_el(an->nodes[type],node);
  an->num_nodes++;                                                        1700
  an->num_unrouted++;
  an->nodes[UNROUTED-1]=add_el(an->nodes[UNROUTED-1],node);
  return node;
}

void add_segment(Node node_in,Node node_out) {
  /* makes a connection between two nodes */
  node_in->num_out++;
  node_out->num_in++;
  node_in->outputs=add_el(node_in->outputs,node_out);                     1710
  node_out->inputs=add_el(node_out->inputs,node_in);
}

static int lcnt=0;

void make_latch(Node d,Node latch,String type,Node clock) {
  /* makes a new latch */
  el it;
  Node node;
```

```
Name lclk=(Name)malloc(sizeof(char));                                    1720
Node clatch=NULL;
an->nodes[LATCH-1]=add_el(an->nodes[LATCH-1],latch);
add_segment(d,latch);
add_segment(latch,latch);
add_segment(clock,latch);
if (!strcmp(type,"re")) {
  for (it=clock->outputs;it;it=it->next) {
    node=(Node)it->key;
    if ((node->clatch)&&(!strstr(type,node->name))) {
         clatch=node;                                                    1730
         break;
    }
  }
  if (!clatch) {
    sprintf(lclk,"%s%d",type,lcnt++);
    add_segment(clock,clatch=make_node(lclk,INT_NODE));
    clatch->latch=1;
    clatch->clatch=1;
    clatch->cubes=add_el(clatch->cubes,strdup("1"));
    an->nodes[LATCH-1]=add_el(an->nodes[LATCH-1],clatch);               1740
  }
  add_segment(clatch,latch);
  latch->cubes=add_el(latch->cubes,strdup("01-1"));
  latch->cubes=add_el(latch->cubes,strdup("-01-"));
  latch->cubes=add_el(latch->cubes,strdup("111-"));
} else if (!strcmp(type,"fe")) {
  for (it=clock->outputs;it;it=it->next) {
    node=(Node)it->key;
    if ((node->clatch)&&(!strstr(type,node->name))) {
         clatch=node;                                                    1750
         break;
    }
  }
  if (!clatch) {
    sprintf(lclk,"%s%d",type,lcnt++);
```

```
      add_segment(clock,clatch=make_node(lclk,INT_NODE));
      clatch->latch=1;
      clatch->clatch=1;
      clatch->cubes=add_el(clatch->cubes,strdup("1"));
      an->nodes[LATCH-1]=add_el(an->nodes[LATCH-1],clatch);                    1760
    }
    add_segment(clatch,latch);
    latch->cubes=add_el(latch->cubes,strdup("10-1"));
    latch->cubes=add_el(latch->cubes,strdup("-11-"));
    latch->cubes=add_el(latch->cubes,strdup("001-"));
  } else if (!strcmp(type,"ah")) {
    latch->cubes=add_el(latch->cubes,strdup("1-1"));
    latch->cubes=add_el(latch->cubes,strdup("-10"));
  } else {
    latch->cubes=add_el(latch->cubes,strdup("1-0"));                           1770
    latch->cubes=add_el(latch->cubes,strdup("-11"));
  }
}

Node node_lookup(Name name) {
  /* finds a node with a certain name and returns it */
  el it;
  int i;
  for (i=INPUT;i<NUM_TYPES;i++) {
    for (it=an->nodes[i];it;it=it->next) {                                     1780
      if (!strcmp(((Node)it->key)->name,name)) {
          return it->key;
      }
    }
  }
  return NULL;
}

void print_net(Net net) {
  /* prints out information about a net */                                     1790
  el it;
```

```
  printcor(net->coor);
  fprintf(outfile,"%d [%d]",net->level,net->snum);
  for (it=net->child_nets;it;it=it->next) {
    fprintf(outfile,"->");
    print_net(it->key);
    if (it->next) fprintf(outfile,"\n");
  }
}
                                                                    1800
void print_connections() {
  /* prints out information about the netlist which has been laid out */
  el it,it2;
  int i;
  Node node;
  fprintf(outfile,"Connections:  \n");
  for (i=INPUT;i<NUM_TYPES;i++) {
    fprintf(outfile,"Type=%d\n",i);
    for (it=an->nodes[i];it;it=it->next) {
      node=it->key;                                                 1810
      fprintf(outfile,"<%s> \n",node->name);
      printcor(node->coor);
      fprintf(outfile,"\nLevel=%d\n",node->level);
      fprintf(outfile,"%d Inputs:  \n",node->num_in);
      for (it2=node->inputs;it2;it2=it2->next) {
          fprintf(outfile,"%s ",((Node)it2->key)->name);
      }
      fprintf(outfile,"\n%d Outputs:  \n",node->num_out);
      for (it2=node->outputs;it2;it2=it2->next) {
          fprintf(outfile,"%s ",((Node)it2->key)->name);            1820
      }
      fprintf(outfile,"\nCubes:  \n");
      for (it2=node->cubes;it2;it2=it2->next) {
          fprintf(outfile,"%s ",it2->key);
      }
      fprintf(outfile,"\nNet:  \n");
      if (node->net) {
```

```
            print_net(node->net);
            net_feasible(node->net);
        }                                                                    1830
        fprintf(outfile,"\n\n");
      }
    }
  }

  Node add_node(Name name) {
    /* adds a node with a certain name */
    Node node;
    ntype i;
    if (!(node=node_lookup(name))) {                                         1840
      node=make_node(name,INT_NODE);
    }
    return node;
  }

  void sweep() {
    /* gets rid of all nodes with no inputs or outputs */
    Node node;
    el it,next;
    int i;                                                                   1850
    for (i=INPUT;i<NUM_TYPES;i++) {
      for (it=an->nodes[i];it;it=next) {
        node=it->key;
        next=it->next;
        if (!node->num_in&&!node->num_out) {
              an->nodes[i]=del_el(an->nodes[i],it);
              an->num_nodes--;
              free(node);
        }
      }                                                                      1860
    }
  }
```

```
void traverse(Node node,int stage) {
  /* dfs of circuit */
  /* easy enough to add the capability for returning the linked list of
     critical elements if needed */
  el it;
  if (node->latch) {
    if (stage>an->max_stage) an->max_stage=stage;                          1870
  } else {
    for (it=node->outputs;it;it=it->next) {
      traverse((Node)it->key,stage+1);
    }
    if ((node->type==OUTPUT)&&(stage>an->max_stage)) an->max_stage=stage;
  }
}

void depth() {
  /* finds the depth of the circuit */                                     1880
  el it,it2;
  for (it=an->nodes[INPUT];it;it=it->next) {
    for (it2=((Node)it->key)->outputs;it2;it2=it2->next) {
      traverse(it2->key,2);
    }
  }
  for (it=an->nodes[LATCH-1];it;it=it->next) {
    for (it2=((Node)it->key)->outputs;it2;it2=it2->next) {
      traverse(it2->key,1);
    }                                                                      1890
  }
}



#define MBIG 1000000000
#define MSEED 161803398
#define MZ 0
```

```
#define FAC (1.0/MBIG)                                                    1900

float ran3(long *idum)
{
        static int inext,inextp;
        static long ma[56];
        static int iff=0;
        long mj,mk;
        int i,ii,k;

        if (*idum < 0 || iff == 0) {                                      1910
                iff=1;
                mj=MSEED-(*idum < 0 ? -*idum : *idum);
                mj %= MBIG;
                ma[55]=mj;
                mk=1;
                for (i=1;i<=54;i++) {
                        ii=(21*i) % 55;
                        ma[ii]=mk;
                        mk=mj-mk;
                        if (mk < MZ) mk += MBIG;                          1920
                        mj=ma[ii];
                }
                for (k=1;k<=4;k++)
                        for (i=1;i<=55;i++) {
                                ma[i] -= ma[1+(i+30) % 55];
                                if (ma[i] < MZ) ma[i] += MBIG;
                        }
                inext=0;
                inextp=31;
                *idum=1;                                                  1930
        }
        if (++inext == 56) inext=1;
        if (++inextp == 56) inextp=1;
        mj=ma[inext]-ma[inextp];
        if (mj < MZ) mj += MBIG;
```

```
        ma[inext]=mj;
        return mj*FAC;
}
#undef MBIG
#undef MSEED                                                    1940
#undef MZ
#undef FAC
/* structures.c */

#include "structures.h"

el add_el(el elist,char *key) {
  /* adds a new structure, key, to the netlist pointed to by elist */
  el new=(el) malloc(sizeof(struct element));
  new->prev=NULL;                                               1950
  new->next=elist;
  new->key=key;
  if ((elist!=NULL)&&(elist!=0x1000)) elist->prev=new;
  return new;
}

void repl_el(el elist,char *key1,char *key2) {
  /* replaces structure key1 with key2 in linked list elist */
  el it;
  it=el_lookup(elist,key1);                                     1960
  it->key=key2;
}

el el_lookup(el elist,char *key) {
  /* looks up structure key in elist */
  el it;
  for (it=elist;it;it=it->next) { if (it->key==key) return it; }
  return NULL;
}
                                                                1970
int el_num(el elist,char *key) {
```

```
  /* returns the number from the top of key in elist */
  el it;
  int i=0;
  for (it=elist;it;it=it->next,i++) if (it->key==key) return i;
  return -1;
}

el del_el(el elist,el elem) {
  /* deletes element elem from elist */                                  1980
  if (DEBUG) printf("Here!");
  if (elem) {
    if (elem->prev) {
      elem->prev->next=elem->next;
    }
    if (elem->next) {
      elem->next->prev=elem->prev;
    }
    if (elem==elist) { elist=elist->next; }
  }                                                                      1990
  free(elem);
  return elist;
}

/* fsm.h */

/* a structure which holds the state transition table */

struct stt {
  int num_latches,num_in,num_out,num_terms,num_states;                   2000
  String reset_state;
  String **terms;
  String **states;
  el latch_order;
};

/* global variables and functions */
```

```
extern struct stt *fsm;
void add_stt(String []);
void encode(String []);                                              2010
void write_pif();
void pofcomp(String,String);
/* layout.h */

#define NUM_LAYERS 16
typedef struct CellRecord *Cell;

struct spacetime {
  /* spacetime canvas for circuit layout */
  int num_dim,num_nbhd,num_levels;                                   2020
  int *num_cor;
  int lwires;
  int num_pluts;
  int *nbhd;
  el *levels;
  Cell *tess;
};

extern struct spacetime *st;
                                                                     2030
struct statespace {
  /* structure which holds cell configuration information */
  int num_cfgs;
  el *cfgs;
  el *temp;
  el *inputs;
  el *outputs;
  int *nump;
};
                                                                     2040
struct CellRecord {
  /* a cell which has two parts: nodes and nets */
  el nodes;
```

```
  el nets;
};

/* global procedures and variables */
void convert(int,int *);
int revert(int *);
int round(double);                                        2050
int min(int,int);
int sdist(int,int);
int get_num_levels();
void remove_level(Node);
void fix_latches();
void fix_levels(Node);
int compatible(Node,int,int);
void place_node(Node,int,int);
int schedule(Node,int);
void rand_place(Node);                                    2060
void rand_place_all();
void stitch_place_all();
int force_direct(Node);
int force_place_all();
void place_all(int);
void init_layout();
int node_feasible(Node);
void net_feasible(Net);
int all_feasible();
double avg_dist();                                        2070
int routeAll();
float ran3(long *idum);
extern long idum;
extern int debug;
/* lut.h */

/* global functions */
void init_ss();
void min_ss();
```

```
void make_exp();                                                         2080
/* netlist.h */

typedef struct element *el;
typedef char *String;
typedef enum NodeType ntype;
typedef struct NodeRecord *Node;
typedef char *Name;
typedef struct NetRecord *Net;
typedef char *Cube;
                                                                         2090
/* different node types in a BLIF netlist */
enum NodeType { INPUT, INT_NODE, OUTPUT, NUM_TYPES, LATCH, UNROUTED };

struct AllNodes {
  /* a structure which holds the netlist */
  int max_stage;
  int num_nodes;
  int num_nets;
  int num_unrouted;
  el nodes[NUM_TYPES+2];                                                 2100
};

extern struct AllNodes *an;

struct NodeRecord {
  /* a node */
  String name;
  int num_in;
  int num_out;
  int latch;                                                             2110
  int clatch;
  char init;
  int coor;
  int level;
  ntype type; /* input,latch,etc. */
```

```
  el cubes;
  el inputs;  /* a list of input nodes */
  el outputs; /* a list of output nodes */
  Net net; /* the output net of a node */
  el inets; /* the input nets to the node */                    2120
  int placed;
};

struct NetRecord {
  /* a net */
  int coor;
  int level;
  int snum;
  Node parent_node;
  int dim,dist;                                                 2130
  Net parent_net;
  el child_nodes;
  el child_nets;
};

/* global functions and variables */
void finalErr(char *);
Net add_net(Node,Node,Net,int,int,int,int);
Node make_node(Name,ntype);
void add_segment(Node,Node);                                    2140
void make_latch(Node,Node,String,Node);
Node node_lookup(Name);
void print_connections();
Node add_node(Name);
void sweep();
void traverse(Node,int);
void depth();
extern int def_clk;

                                                                2150
/* structures.h */
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "netlist.h"
#include "layout.h"
#include "fsm.h"                                                    2160

struct element {
  /* an element in a doubly linked list */
  el prev;
  el next;
  /* arbitrary pointer */
  char *key;
};

/* global functions */                                              2170
el add_el(el,char *);
void repl_el(el,char *,char *);
el el_lookup(el,char *);
int el_num(el,char *);
el del_el(el,el);
```

Bibliography

[1] Ruben Agin. Spacetime circuitry for virtual logic emulation on cam-8. MIT IM-Group Preprint.

[2] Neil W. Ashcroft and N. David Mermin. *Solid State Physics.* Harcourt Brace College Publishers, New York, 1976.

[3] Jonathan Babb, Russell Tessier, and Anant Agarwal. Virtual wires: overcoming pin limitations in fpga-base logic emulators. In *Proceedings of the IEEE Workshop on FPGA's for Custom Computing Machines*, pages 142–151. IEEE Computer Society Press, 1993.

[4] Jonathan W. Babb. Virtual wires: Overcoming pin limitations in fpga-based logic emulation. SM thesis, MIT, EECS Department, 1994.

[5] Michael Biafore. *CAM-8 Forth User's Manual.* MIT IM-Group Preprint.

[6] Michael Biafore. Cellular automata for nanometer scale computation. *Physica D*, 1993.

[7] Michael Biafore. *Few Body Cellular Automata.* PhD dissertation, MIT, Department of Physics, 1993.

[8] Robert K. Brayton et al. *Logic Minimization Algorithms for VLSI synthesis.* Kluwer Academic Publishers, 1984.

[9] A. W. Burks, editor. *Essays on Cellular Automata.* University of Illinois Press, 1970.

[10] E.F. Codd. *Cellular Automata.* Academic Press, Inc., New York and London, 1968.

[11] Andre DeHon. Dpga-coupled microprocessors: Commodity ic's for the early 21st century. Transit Note 100, MIT, Transit Project, 1994.

[12] Andre DeHon. *Reconfigurable Architectures for General Purpose Computing.* PhD dissertation, MIT, EECS Department, 1996.

[13] M. Denneau. The yorktown simulation engine. In *19th Design Automation Conference*, pages 55–59. IEEE, 1982.

[14] A.K. Dewdnew. *The Turing Omnibus.* Computer Science Press, Rockville, MD, 1989.

[15] Gary Frazier. An ideology for nanoelectronics. In Stuart K. Tewksbury et al., editors, *Concurrent Computations: Algorithms, Architectures, and Technology.* Plenum Press, 1988.

[16] David Harris. Introduction to vlsi for freshmen and sophomores. MEng thesis, MIT, EECS Department, 1994.

[17] Andrew Harter. *Three-dimensional integrated circuit layout.* Cambridge University Press, New York, 1991.

[18] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[19] Mahlon G. Kelly and Nicholas Spies. *Forth: A Text and Reference.* Prentice-Hall, Edgewood Cliffs, NJ, 1986.

[20] Lionel C. Kimerling and Peter T. Panousis. Reaching the limits in silicon processing. *AT&T Technical Journal*, pages 16–31, November/December 1990.

[21] Eugene F. Krause. *Taxicab Geometry.* Dover Publications, Inc., New York, 1986.

[22] C. Lee. An algorithm for path connections and its applications. In *IRE Transactions on Electronic Computers*, volume EC-10, pages 346–365, September 1961.

[23] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann Publishers, Inc., 2929 Campus Drive, Suite 260, San Mateo, CA 94403, 1992.

[24] F.T. Leighton. Theory of parallel and vlsi computation. Notes for MIT Course 6.848, 1992.

[25] P.K. Wolff M. Hanan and B.J. Agule. Some experimental results on placement techniques. In *Proceedings of the 12th Design Automation conference*, pages 214–224, 1979.

[26] F. P. Manning. *Automatic Test, Configuration and Repair of Cellular Arrays.* PhD dissertation, MIT, EECS Department, June 1975.

[27] Norman Margolus. Physics-like models of computation. *Physica D*, pages 81–95, 1984.

[28] Norman Margolus. Quantum computation. In *New Techniques and Ideas in Quantum Measurement Theory*, pages 487–497, 1986.

[29] Norman Margolus. *Physics and Computations.* PhD dissertation, MIT, Department of Physics, 1987.

[30] Norman Margolus. CAM-8: a computer architecture based on cellular automata. In *Pattern Formation and Lattice-Gas Automata.* American Mathematical Society, 1996.

[31] Norman Margolus. An fpga architecture for dram-based systolic computations. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines.* IEEE Comp. Soc. Press, 1997.

[32] Norman Margolus and Tommaso Toffoli. STEP: a Space Time Event Processor. Technical Report 592, MIT Laboratory for Computer Science, 1993.

[33] Gerd Meister. A survey on parallel logic simulation. Technical Report 14-1993,SFB 124, University of Saarland, Department of Computer Science, 1993.

[34] Rajeev Murgai et al. Logic synthesis for programmable gate arrays. In *27th ACM/IEEE Design Automation Conference*, pages 620–625, 1990.

[35] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[36] Douglas L. Perry. *VHDL.* McGraw-Hill, New York, 1994.

[37] Andrew L. Ressler. *Practical Circuits Using Conservative Reversible Logic.* SB thesis, MIT, EECS Department, 1979.

[38] Ellen M. Sentovich et al. Sis: A system for sequential circuit synthesis. Memorandum UCB/ERL M92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, May 4 1992.

[39] R. Shankar. *Principles of Quantum Mechanics.* Plenum Press, New York and London, second edition, 1994.

[40] Mark Andrew Smith. *Cellular Automata Methods in Mathematical Physics.* PhD dissertation, MIT, Department of Physics, 1994.

[41] Eliezer Sternheim. *Digital Design with Verilog HDL.* Automata Publishing Co., 1990.

[42] Tommaso Toffoli. Cellular automata as an alternative to (rather than an approximation of) differential equations in modelling physics. *Physica D*, pages 117–127, 1984.

[43] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines–A New Environment for Modeling.* MIT Press, 1987.

[44] P. Douglas Tougaw and Craig S. Lent. Logical devices implemented using quantum dot cellular automata. *J. Appl. Phys.*, 3(75), February 1 1994.

[45] John von Neumann. Theory of automata: Construction, reproduction,homogeneity. In A. W. Burks, editor, *The Theory of Self-Reproducing Automata*, volume II. University of Illinois Press, Urbana, IL, 1966.

[46] Steven A. Ward and Robert H. Halstead Jr. *Computation Structures.* MIT Press, 1986.