Design and Implementation of a Compiler for Cellular Automata Machines

by

Daniel R. Risacher

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 Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 1996

Author____

Department of Electrical Engineering and Computer Science

May 28, 1996

Certified by____________________________________

Norman Margolus

Thesis Supervisor

Accepted by____________________________________

F. R. Morgenthaler

Chairman, Department Committee on Graduate Theses

Design and Implementation of a Compiler for Cellular Automata Machines

by

Daniel R. Risacher

Submitted to the Department of Electrical Engineering and Computer Science

on May 28, 1996, in partial fulfillment of the

requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a compiler for the CAM-8 computer. The CAM-8 is a spatially organized cellular automata machine. The source language of the compiler is similar to C, and the destination language is a set of operations that can be performed by the CAM-8 hardware. The unique architecture of the CAM-8 presents uncommon problems for compiler design and the system implemented explores ways to resolve these challenges. In particular, automated data organization and optimization by combining operations into groups that can be performed simultateously were studied. Test cases were compiled to determine that the compiler was producing correct code, and also to evaluate the optimizations that the compiler performed.

Thesis Supervisor: Norman Margolus

Title: Visiting Scientist

Acknowledgments

I would thank Norman Margolus, my thesis supervisor and friend, without whose support and encouragement this work would not have been possible.

I would also like to thank my wonderful wife, Sheryl Risacher, for being both supportive and motivating, despite having her own thesis to worry about.

Thanks go also to Harris Gilliam, for being a good friend and for introducing me to the CAM-8 and its mysteries.

Raissa D'Souza and the other members of the Information Mechanics group also deserve a world of thanks for listening to my stories and ideas, and providing laughter and feedback.

Financial support was provided by STEP Research, and indirectly by the National Science Foundation, grant no. NSF 9305227-DMS.

Contents

Chapter 1

Introduction

Fifty years hence, computers will look nothing like what they look like today. They may still come in boxes with power cables and fans, but on the inside, where the computation happens, things will be substantially different. While I do not know what they will look like, one thing is certain to remain the same: they will obey the same laws of physics.

If we assume that computers continue to get faster, and computing elements continue to get smaller, the design for computers are likely to be highly constrainted by the laws of physics. We can infer therefore, that the most powerful computers will be spatially organized to extract the maximum computation from the physical laws that govern them. These considerations have motivated the development and use of spatially organized cellular automata machines[11, 9].

1.1 Objective

The objective of this research was to develop a compiler for the CAM-8 computer, and to investigate compiler techniques for cellular automata machines in general. The CAM-8 is a special purpose supercomputer built at MIT's Laboratory for Computer

Science by the Information Mechanics research group[8]. The CAM-8 is a cellular automata machine; it is a piece of computing hardware that was designed and optimized for a specific class of spatially organized computations called cellular automata.

Cellular automata (CA) are uniform arrays of simple digital processing elements, called *cells*, locally interconnected, and updated in parallel. The updating of the cells is taken to be synchronous, unless specified otherwise. The CA model was introduced in the 1940's by Von Neumann and Ulam[4, 12] as a theoretical model of complex self-modifying spatially-extended systems.

From a computer science perspective, a cellular automaton is a discrete, dynamic spatial array calculation. That is, a CA is a lattice of cells where each cell has a discrete state that changes in discrete time steps. The state of a given cell in the lattice at some time step depends only on the state of the cells in some local neighborhood at the previous time step. The rule that governs transitions from one state to the next is therefore both uniform and local. In other words, a cellular automaton is some spatially organized computation, the same laws apply everywhere and events can only affect other nearby events.

1.1.1 Organization

This thesis is presented in five sections. In the introduction, I will set the stage by explaining the nature, motivations and scope of the work that has been done. I will then describe the programming, hardware and compiler models around which the work is based. In the third section, I will discuss the methods and techniques used to implement and test the compiler. In the final two sections, I will summarize the results, and discuss the strengths, weaknesses and future of the work.

1.2 Motivation

1.2.1 Developing the CAM-8

In a real sense, the purpose of the project was primarily to make the CAM-8 easier to program. Currently the CAM-8 is programmed by writing FORTH programs that run on the host computer and generate instruction streams for the hardware. While a large library of FORTH routines have been developed over the years to make this process easier, it still has the character of programming in assembly language. The programmer must understand the capabilities and limitations of the hardware intimately, and be willing to adapt to the machine.

It has often been said that the existence of a good compiler can make or break a computer architecture. The CAM-8 has survived without one only because people were willing to suffer programming it with inferior tools to take advantage of its immense computing power. The combination of the extreme speed of the hardware with decent programming tools will be even more computationally powerful.

Machines need compilers. The purpose of a compiler is to hide the hardware from the programmer. This leads to numerous advantages. First, the human mind programming the machine can focus on the details of the problem he is trying to solve, and not be distracted by the tools he uses. Second, as machine architectures change and new machines are developed, compilers can allow old code to run on new machines. These arguments apply to cellular automata machines as much as to any other computer, perhaps even more so.

1.2.2 Spatially Organized Computing

Suitability for Physical Simulation

Physical simulation is, in some ways, the holy grail of computing. Computers are not an end in themselves; they are tools, and ultimately, they should help us to understand and manipulate our world better. Physical simulation is one of the most fundamental ways that computers can aid us in understanding our world. We use simulations to understand fluid flows, we use it to do volumetric rendering, we use it to calculate molecular dynamics, and many other things besides. For physical simulation it always helps to be able to run a bigger simulation, with finer granularity, in less time. There is no reasonable end to the demand for speed in this part of the computational spectrum.

Since physics is uniform, local, and parallel, a machine which is optimized for these types of computations will excel at physical simulation. This is one of the reasons that cellular automata have been interesting since they were first proposed: they are ideal models of the real world. Figure 1-1 shows two examples that demonstrate the types of models which have been run on the CAM-8[3, 1].

Physical Constraints

Beyond making the CAM-8 easier to use in specific, it was my goal to develop programming methods for spatially organized machines in general. There are various compelling reasons why spatially organized machines, like the CAM-8 will be the way of the future. This type of architecture is honest about the physical constraints on computation, and seeks to exploit the laws of physics in ways that are uniquely suited for simulation of physical systems. Furthermore, this relatively simple design idea is both feasible and computationally powerful.

There are fundamental constraints on computation that arise from the laws of

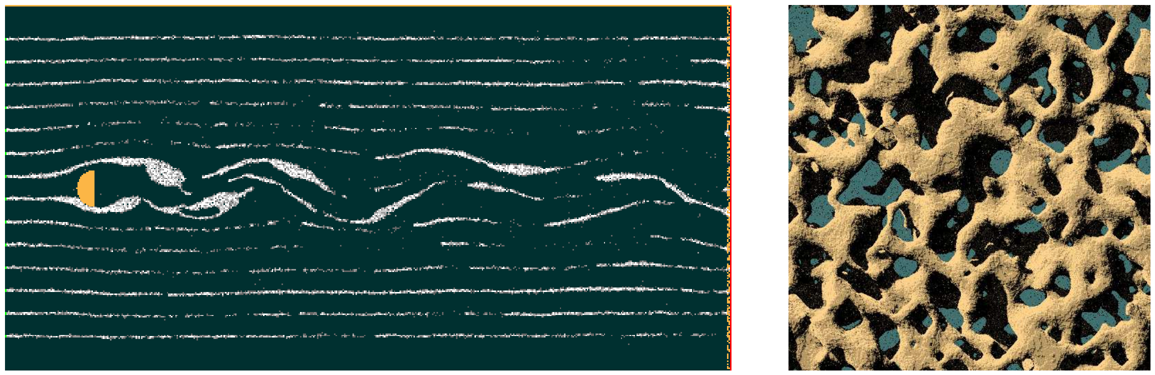

Figure 1-1: Physical simulation using CA algorithms. (a) Flow past a half cylinder using an integer lattice gas, visualized using a tracer fluid flowing with it. (b) A three dimensional Ising simulation, rendered by directly simulating light within the CA system.

physics. For instance, signals cannot propagate through a wire faster than the speed of light. The interactions of physics are uniform, local, and parallel; in other words, the same laws apply everywhere, events only ever depend on their immediate surroundings, and many things happen all at once.

Spatially organized computation has these same properties; the data is laid out in space, and the operations on the data are uniform and local. Physics then lends itself to this type of computation; processors can be laid out in space, connected to their local neighbors, and can operate in a uniform manner, each operating in parallel. Cellular automata machines are the fine-grained limiting case of massively parallel machines.

Feasibility of Uniform Spatial Computation

Since both memory and processing are distributed in a spatial computation, the hardware that implements them can be similarly distributed. More interestingly, however, the idea of uniform computation means that the hardware should also be uniform. This implies that such hardware could consist of an array of repeated cells. The design and fabrication complexity of a uniform array is low. One can easily imagine a uniform 3D array of self-assembling molecules that could be used as a computing medium of unprecedented computing power.

Micro-scale Configurability

Generally, the fastest way to compute something is to program it in silicon. That is, the highest computational speed, is achieved through physical realizations: dedicated circuitry. So why don't we do all our computation on dedicated circuits? Obviously, building dedicated circuits is usually wasteful because they aren't general purpose. When we want to compute something else, we have to build a new circuit each time. Suppose, for a moment, that we could build reconfigurable circuits that could be retasked on the fly, without fabricating new hardware. Such a system would be both powerful and general.

This argument is essentially what has driven the development of field programmable gate arrays (FPGAs). FPGA architectures are only one possible realization of this idea. Cellular automata architectures such as the CAM-8 offer another way to implement this idea. One of the uses of the CAM-8 has been to simulate the operation of circuits in real time. The advantage of this mode of operation is that the circuit is encoded in the cellular automaton merely as an initial condition. Changing the circuit is essentially a software change.

1.3 Scope

There are two main aspects of a cellular automata experiment: the dynamics and the initial configuration. Automated tools can help to generate both of these descriptions, but my compiler focused exclusively on the experiment dynamics. In other words, I focused on ways to translate the high-level specification of a transition rule into operations that can actually be performed by the CAM-8 hardware.

Earlier work that I did as part of the Undergraduate Research Opportunities Program resulted in a program called 'campat' that attempted to tackle the initial configuration problem. The development of a general purpose tool for creating initial configurations for experiments on the CAM-8 is difficult because of the huge number of different features that users of the CAM-8 may want. This aspect of programming CA machines is a separate problem that I was not trying to incorporate into this thesis project.

Moreover, I focused primarily on compiler techniques, rather than on language design. The source language currently used is simple and effective, but has many ways in which it could be improved. Many of the ways in which the language could be improved are independent of the internal workings of the compiler, and could be implemented by minor changes to the front end.

There is currently no well defined "assembly language" for the CAM-8, nor is there an object file format. To this end, I have only implemented the compiler phases necessary to translate from the source language into realizable operations. In other words, my project is limited to the generation of code that is completely realizable on the CAM-8, but does not include the actual run-time utilities needed to execute the code. Until these utilities are developed the compiler can be used in an advisory capacity for programmers using the current FORTH environment.

Chapter 2

Models

2.1 Programming Model

The programming model that I used as a goal for my compiler was a very simple one. A human who is trying to program a cellular automaton should only need to think about the automaton itself; he or she should never need to think about the details of how to implement it with the available computing hardware. Hiding the details of the hardware is the job of the compiler.

The programmer, therefore, should only need to worry about two things: the persistent state of the automaton, and the transition rule. In the language implemented by my compiler, these are the only things that are specified by the source file. The data fields that define a cell are explicitly named and defined, and the transition for a cell from state t to state $t+1$ is described as a series of executable statements in a high-level language with syntax very similar to the C programming language.

The transition rule is defined as if for a single reference cell, but in reality, the same operations occur simultaneously at every cell in the experiment. Furthermore, the programmer does not need to consider the mechanisms for data movement in the CAM-8 memory, but instead can simply refer to the fields of neighboring cells as array offsets from the reference cell. This programming model is high-level and general enough to apply to any hardware that CA might be implemented on.

2.1.1 The language

The language that was implemented is a subset of the C programming language[7]. A source file of the language has two sections, a data field declaration section, and a transition rule definition section. In the data field declaration section, the programmer specifies a set of named data fields that are the state of the CA at each time step. The only data types supported are integers of various sizes. In fact, the programmer can specify a specific range of values, or can specify the size of the variable in bits.

The transition rule section consists of a single block of code, with no loops. All the arithmetic operations of C are available, as are if-conditionals, but while-loops or for-loops are not available. The code in the block specifies a sequence of operations that are performed on the data fields at each time step of the automaton. Unlike the current FORTH system, the semantics of the language are that operations happen in the order they are specified. In the FORTH based system, operations didn't take effect until a "propagate" operation. While these semantics allowed the programmer to specify several things that happen simultaneously, they are extremely confusing to the naive user.

Unlike C, the programmer can use new temporary variables at will without explicitly declaring them or their type. Only variables that comprise the state of the automaton at the step boundaries need be specified. Since no data is saved across the time-step boundary except the declared cell fields, and the range of possible values for the declared fields are explicitly known, the compiler can calculate the range of possible values for any temporary variables.

In order to make data movement from cell to cell completely transparent, the language is defined to have array-like operations which painlessly allow the programmer to reach into some other cell at some offset from the reference cell and look at the data fields of that cell. Part of the compiler's task is to figure out how to move the data into the reference cell, and, if needed, put it back.

2.2 Hardware Model

In order to understand what my compiler does, you need to understand a little bit about the machine it compiles code for, the CAM-8. The CAM-8 is the product of a decade of development by the Information Mechanics group. It is a cellular automata machine; a special purpose computer that is extremely powerful for spatially organized computing. In order to take advantage of the physics of computation, the architecture of the CAM-8 is extremely unusual, compared to other modern computers. In terms of computer architecture buzzwords, it is a perfectly scalable, virtual-processor, mesh-architecture, single-instruction/multiple-data (SIMD) parallel machine.

2.2.1 Memory

The details of the CAM-8 architecture are fascinating (if you like that sort of thing) but complex. The best way to understand the CAM-8 therefore, is in terms of a simple model. For the purposes of my compiler, I modeled the memory of the CAM-8 as an n-dimensional array of 16-bit words. The state of a cell in an automaton is composed of one or more 16-bit words. Each of these words is called a subcell. The set containing a given bit (0-15) from all the subcells is called a layer.

2.2.2 Instructions

For the purposes of my project, there are really only two interesting operations that the CAM-8 can perform. These instructions are broadcast to all the processors simultaneously from a separate "host" workstation, and all processors execute the instructions in unison. The operations are called a "scan" and a "kick", and are illustrated in figures 2-1 and 2-2. In a scan operation, for each cell, a 16-bit word is read from the memory, used as an index into a lookup table, and the result is written back into the memory. Any arbitrary manipulation of the data can be performed in one step, as long as the data can be placed into a single 16-bit word.

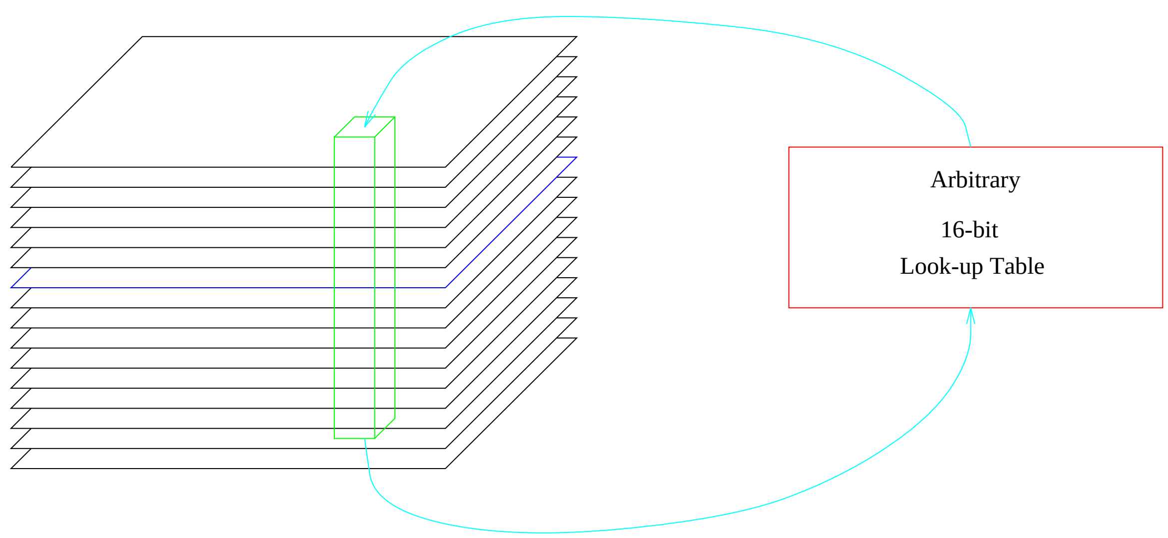

Figure 2-1: Scan operation performs an arbitrary function on each word in the CAM-8 memory. The vertical box highlights one particular 16-bit cell.

The function of the kick operation is more subtle: some specific bit of every 16-bit word (a layer) can be moved relative to the rest of the memory. Effectively, each cell in the lattice can pass part of his personal data to his neighbor who is offset from him by some vector offset D, and at the exact same time, his neighbor at offset

$-D$ will pass him the corresponding data. A kick operation can also exchange the corresponding layers of different subcells within a cell.

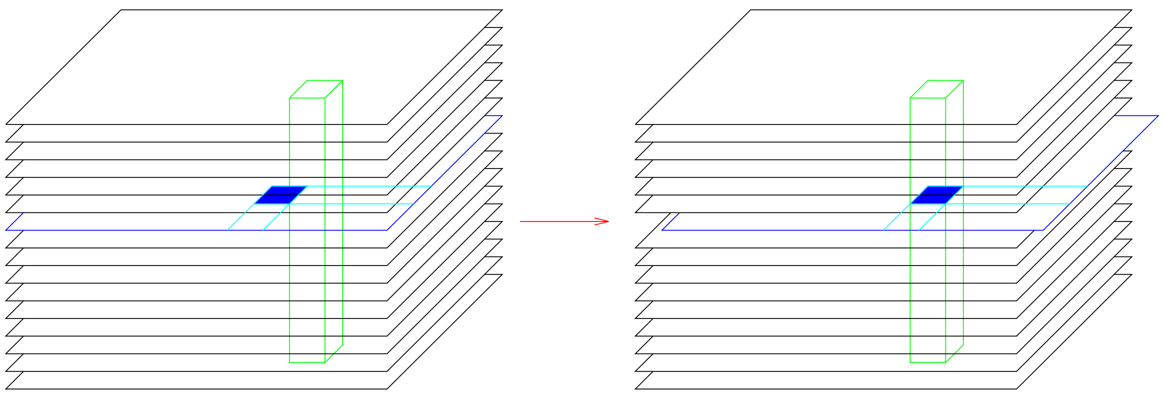

Figure 2-2: Kick operation moves data from each word in the memory to a neighboring word. The vertical box highlights one particular 16-bit cell. The figure illustrates how shifting a layer of bits results in new data moving into the highlighted cell. Data shifted past the edge of the space is wrapped around.

The time it takes to perform a scan operation is linearly proportional to the size of the lattice divided by the amount of computing hardware being used. The kick operation, on the other hand, takes a constant amount of time to perform, and has virtually no cost.

2.3 Implications

What are the roadblocks between the language and the hardware? The hardware model described above has some unusual implications for any programming tool created for such an architecture. The highly unorthodox nature of the machine imposes a programming methodology that is similarly unorthodox. Specifically, programming on the machine is dominated by the lack of control flow, the look-up table, and the strict requirements for data organization. These challenges have confronted the humans programming the CAM-8 to date, but can be handled in an automated fashion, leaving humans the more pure programming model described as a goal above.

2.3.1 Control Flow

The first painful fact-of-life on the CAM-8 is that there is no control flow mechanism; there are no branch instructions. This is a consequence of the SIMD nature of the machine; every virtual processor in the machine executes the same instruction as all the other processors, so the instruction order cannot depend on the data at each cell, but is determined globally. All the conditionality of the machine therefore resides in the look-up table. Unfortunately, human programmers think in terms of if-then statements, not look-up tables. A good compiler can convert one type of condition into the other in a reasonable way.

2.3.2 Table Look-up

The second fact-of-life is the 16-bit table look-up. Table look-up is extremely powerful because it is completely general. Any arbitrary function can be encoded as a look-up table, provided the function is small enough. A problem arises, however, when the function to be implemented is too large. Suppose the programmer specified a 32-bit multiply operation, how then do you encode a 32-bit multiply as a 16-bit operation? To date, users of the CAM-8 have solved this problem by using very small data types. Because of this, the machine has not been used for a large number of the potential applications for which it would outperform traditional computers by several orders of magnitude. Automating the process of function decomposition into smaller operations is both feasible and useful, but has not been attempted to date.

2.3.3 Data Organization

The third consequence that the hardware design has on the programming environment is that the organization of data in the memory is both crucial and tricky. In addition to the common constraints on data organization faced by any compiler, the use of a table look-up for arbitrary functions dictates that the operands of any given instruction must be located in different bits of the data word. Because of this constraint, each scan operation may require some preceding operations to move data around to be able to access its operands. Clever arrangement of the data in the memory makes these steps unnecessary, and can make the computation run significantly faster.

2.4 Compiler Model

Functionally, a compiler is a black box that digests one language and produces another. What happens inside the box? The details, of course, depend on the particulars of the problem that it is trying to solve. The general form of compilers is a multi-stage process. First, the source file is parsed and converted into an equivalent program in an "intermediate language". Once the IL code is generated, it passes through some number of internal phases that transform it in various ways. Finally, a code generation phase transforms the intermediate language into the target language. There may also be other phases, such as another optimization phase that operates on the output of the code-generation phase.

The use of an intermediate language gives the compiler a convenient form of the code to analyze and optimize, and also makes it possible to adapt existing compilers to new target languages or new source languages without causing a re-write of the other parts of the compiler. For example, changes to the source language can be accommodated by minor changes to the grammar and the actions; as long as the revisions still generate correct intermediate code, the optimization and code generation phases will not need any changes.

The compiler I implemented loosely fits this model. Since there is no real assembly code or object file format for the CAM, the code generation phase is somewhat indistinct. Instead, since the capabilities of the hardware are well known, the system manipulates the intermediate language into a form that is trivially realizable as hardware operations, and outputs the code in a human readable form.

Roughly speaking, the process of my compiler is this:

1. Parse the input file, and generate the IL code.
2. Optimize the IL code sequence.
3. Group the IL instructions into scan operations.
4. Decide how to arrange the data in the memory.
5. Ensure realizable operations by inserting data movement operations.

which is shown schematically in figure 2-3. It should be noted that this structure is a design choice, and is only one of many possible structures. In particular, the grouping phase, data organization phase, and data movement phase are closely interrelated, and could be performed in some other order or perhaps combined. This particular structure was chosen because it is simple, understandable, and effective.

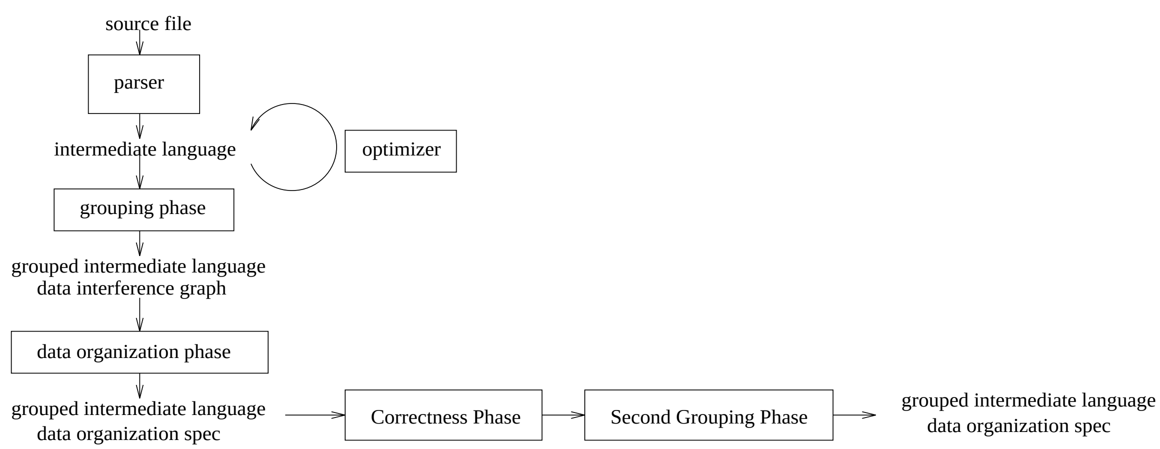

Figure 2-3: Block diagram of the compiler phases.

Chapter 3

Methods

3.1 Syntax-Directed Translation

Like most compilers, this project was based around the concept of syntax-directed translation[2]. The top level of the system is essentially a parser generated from a context-free grammar by the LALR parser-generator "bison". Programs such as bison are standard tools for generating a program that reads an input file. Any context-free language can be processed by such a program, and the program can be easily specified by a generative grammar. The grammar for the language used in this project is listed in section A.20.

As the parser operates, it combines or "reduces" terminal input tokens into non-terminal symbols as specified by the generative grammar, and similarly reduces low-level non-terminals into higher level non-terminals. At each step, the programmer can specify an action to perform on the symbols. When used for syntax-directed translation, the action for a given reduction creates the analogous construct in the target or intermediate language.

After the entire source file is read, and its contents have been reduced to a single "start" symbol, the final action will be to do "everything else". In this context, "everything else" means optimizing and manipulating the intermediate language and ultimately generating the target code.

3.2 Intermediate Language

My compiler, like almost all compilers, first converts the source file into an intermediate language before producing the final target code. While there are many types of intermediate languages, the typical intermediate language consists of some structure of three-address code statements. A three-address statement is of the form:

$$\texttt{x := y } op \texttt{ z}$$

where `x`, `y` and `z` are names, constants, registers or whatever else is appropriate for that particular compiler, and *op* is some arithmetic operator. The intermediate code will usually have other types of statements (such as branches) that do not fit this particular format but are similar. The way that these statements are structured together reflects the control flow paths.

Since there is no provision for branches in the CAM hardware model, the structure for the intermediate language is flat. Since every virtual processor must execute every instruction, there is no way that a given processor can change its instruction sequence conditionally. There is a single block and the statements are executed sequentially.

Despite the lack of control branches in the CA architecture, there is still room for conditionality. All conditionality must be embedded into the table look-up, instead of in control flow. Because of this, the intermediate language that I implemented consists of a sequence of what I have called "four-address code". A four-address code statement is of the form:

$$\text{if } \texttt{w}\text{, then } \texttt{x := y } op \texttt{ z}$$

Any such operation can be encoded as a look-up table (LUT), where the inputs to the table must include `w`, `y`, `z` and possibly `x` (if `w` is not a constant), and the outputs must include at least `x`. Any non-looping conditional structure can be encoded this way, provided that the size of the inputs in bits is smaller than the size of the look-up table.

During parsing, the compiler maintains a stack of the current condition variables. Each intermediate language instruction generated only happens if its condition (`w`) is true, and the condition for a statement is whatever value was on the top of the condition stack when it was created.

When the compiler encounters an "if" statement in the source, it translates that statement into a series of statements in the intermediate language. It first generates the code that computes the conditional expression in the standard way, and then appends a statement that *unconditionally* computes a new temporary variable which is the logical-and of the conditional expression and the top of the stack. It then pushes the temporary variable onto the condition stack and converts the body of the "if" statement. After the body of the "if" statement is complete, it pops the condition from the stack.

In more broad terms, as it goes through the code, the compiler keeps a list of all the things that must be true to get where it is. Each time it computes a new such variable, it takes the logical-and with the previous thing. Thus, if the source program contains an "if" statement within another "if", the corresponding IL code will have the same semantics, despite the inability to branch past the code which is not executed.

3.3 Shadow variables

One of the goals of the language was to have transparent access to the data in neighboring cells. In order to implement access to data in neighboring cells, the layers which that data reside on are "kicked" into position. But what if an operation has two operands that are the same field, but with different relative offsets? The only way to implement such an operation is to copy the field into some other layers, so that each copy can be kicked independently.

In order to handle these situations, each time a field in the cell is referenced, it is considered a "shadow variable". Shadow variables have unique names inside the compiler, and are treated much like any other variable. Thus, the operands of the four-address code statements may have some specified offset from the reference cell. There is also a separate variable which is considered to be the original data. During the data organization phase, each shadow variable will be assigned a location in the memory. If possible, the shadow copy will be located in the same place as the original variable, thus obviating the need to explicitly make copies.

Shadow variables are essentially temporary copies of specified data fields that are used as inputs to scan operations. Each one has a name and a location: the name is the name of the field and the realative offset, and the location is chosen like any other variable. Whenever a shadow variable is used as an rvalue (i.e. read) in the intermediate language, it is kicked by the relative offset. Whenever a shadow variable is used as an lvalue (i.e. assigned), it is *not* kicked. Thus, when the source code contains an operation that reads a cell data field, the corresponding shadow is kicked into position and used. If necessary, the cell field will first be copied into the location of the shadow variable.

The behavior of a source language assignment to a cell data field with offset is undefined.

3.4 Intermediate Language Grouping

The great advantage of table look-up, which is not captured by the intermediate code as described above, is that a look-up table can encode any arbitrary function of the inputs, not just conditional arithmetic operations. Likewise, a look-up table may have multiple outputs that can be computed simultaneously.

In short, multiple four-address statements may be combined into a group, such that the entire group of statements can be performed by a single table look-up. This grouping is done by an annealing process. First, each statement is converted into a group that contains it as the only member of the group. This produces an ordered sequence of groups. For each group, the compiler maintains a list of the variables that are inputs and a list of the variables that are outputs. It also maintains a list of variables that are considered "live", and the variables that are "killed" by that group. A live variable is one whose data will be used later on; the data it holds can't be over-written. To say that a group "kills" a variable means that the group writes over it; any previous data it contained will be lost when the group is executed.

Two groups can be annealed if the following conditions are met:

1. The inputs needed to implement the combined group are smaller than the inputs of a LUT.

2. The outputs needed to implement the combined group are smaller than the outputs of a LUT.

3. No group exists between the two groups that depends on the execution of the first group and affects the execution of the second group.

The actual algorithm for doing the annealing is fairly simple. Starting at the end and working backwards, the compiler considers each pair of groups and anneals them if the constraints are met. When it finds a pair of groups that can be annealed legally it does so and never looks back. This means that annealing phase will sometimes find sub-optimal solutions, but it also means that it finds reasonable solutions quickly.

3.5 Data organization

The way that the data is organized in the memory of the machine is very important on this type of architecture. Every variable that is stored in the machine's memory has a location; it is stored in some specific layers in some specific subcell. There are three constraints on the organization of data:

1. If variable S1 and variable S2 are sources for the same group, then they must be in different layers. Likewise, if variable D1 and variable D2 are destinations of the same group, then they must be in different layers.

2. If variable D is written while some variable S is live, then D and S must either be in different layers or in different subcells, or both.

3. If variable S is read when variable D is written, then D and S must either be in different layers or in the *same* subcell.

In compilers for conventional computers, there is the problem of register allocation. For instance, a procedure has some number of variables associated with it, and the compiler assigns each variable to a register. Since registers are much faster than main memory, but there are only a few of them, it is a big advantage to put all variables into registers but also to use as few registers as possible. This problem is analogous to the data organization problem on a cellular automata machine. There are a limited number of layers in a word of the memory, and a limited amount of memory in the entire machine.

Standard register allocation is essentially the same as graph coloring, which is why it is often called register coloring. Pairs of variables that are live at the same time are said to "interfere" with each other. Such variables must be located in different registers. After all the interferences are detected, the variables and interferences constitute a bidirection graph, where the variables correspond to nodes, and the interferences correspond to edges.

Finding the optimal coloring for a graph is NP-hard, so heuristics are used that provide a good coloring, but not necessarily the best one. A good, fast method is to keep removing the least constained node from the graph and pushing it onto a stack. When all the nodes are removed from the graph, pop them back off the stack in order, insert them back into the graph, and assign them the first legal color[2].

This procedure is essentially how a human would try to color a graph; color the hardest (most constrained) nodes first, then worry about the less constrained nodes. Tackling the hardest problems first usually keeps the heuristic from backing itself into a corner. If the algorithm cannot color the graph in less than the available number of colors, then it "spills" a variable. A spilled variable is stored in main memory, rather than in a register, and is loaded and stored from the memory as needed.

The constraints of the data organization problem are similar to the interferences described above, but there are three different kinds of edges with three different meanings and the assignment is a two dimensional feature: subcell and layers. Nevertheless, the standard heuristic is still a reasonable approach: pick a location for the most constrained variables first, then work on the easier ones.

After constructing the graph from the sequence of groups produced by earlier phases, my system cycles through the nodes in the graph, and evaluates a cost function for each node. As implemented, the cost function is simply the number of edges from that node. Once it has selected the lowest cost node, it removes it and all its edges from the graph. When the last node is removed, it starts putting them back into the graph in the reverse order that it took them out. When it put a node back into the graph it picks a legal location for the variable.

The first time it colors a shadow variable of a given field, or whenever it colors an original variable that has shadows, it marks the original variable as "prefering" that location. When it tries to pick a location for a shadow variable or an original variable that has shadows, it tries the preferred location before trying any other location. This attempts to place shadows and originals in the same location whenever possible.

3.6 Copy steps

As mentioned earlier, when shadow variables are used that are not in the same location as the original variables, the "real" value stored in the original variable must be copied into the shadow before it is used. After the variable locations have been picked, these copy operations are inserted. Such an operation must be in a separate scan from the operation in which the shadow is used as an rvalue, since the implied kick of the shadow's relative offset must happen between the two scan operations. Since the kick is implicit, the grouping phase is likely to want to combine these two operations. To prevent this, the compiler marks the shadow variable output of the copy operation as a "required write". The grouping phase will not anneal two groups if some shadow is a "required write" of the first and an input of the second. This is required for correctness.

Sometimes, there will be no good place to locate a variable. In other words, there is no location that meets all of the constraints listed in section 3.5. This is analogous to "spilling" the variable in the standard register coloring case. The variable is marked as "spilled" and is assigned a location that doesn't overlay anything that is live at the same time, but may not be in layers that work for the scan operations that it is used in.

When this variable is used, it needs to be copied into the correct place. Thus, after the statements are grouped and the variables are colored, if there is some variable that was spilled, then special “copy steps” are inserted before or after each group that uses it, as appropriate. If a spilled variable is used as an input to some look-up table, but the location of the variable overlaps with some other input to the LUT, then a new temporary variable is created that is located in a more convenient location, and a special group is inserted that copies the spilled variable to the temporary location. The group is then modified to use the special copy, instead of the spilled variable. A similar procedure is done for spilled outputs.

3.7 Function calls

Since there are no branches in this type of architecture, it is not really possible to implement true function calls. The only reason that a user might want to use function calls is for stylistic reasons; function calls can make code much clearer, even without recursion.

Should the programmer want to specify function calls, the way to implement them on this type of machine is in-lining; simply insert the code of the function into the main part of the program. This will also allow the grouping phase to ignore the boundaries of the function and group statements across the call or the return, and also to consider the data location needs of the function at the same time as the rest of the program.

This functionality can be completely implemented as a pre-processing phase. Rather than develop a special pre-processor, this system simply uses the familiar C pre-processor, which completely satisfies the requirement.

3.8 Testing

Quite unfortunately, there is currently no run-time environment for the CAM-8 that is appropriate for this compiler. Until the run-time environment is available, code from the compiler must be laboriously checked for correctness by hand.

The CAM-8 is a 16-layer machine, and there are no plans to build machines with either more or less layers. Nevertheless, the number of layers available is specifiable on the command line when the compiler is invoked. This feature was included only for testing purposes.

The problem with hand testing is that any test case that is complicated enough to really exercise the features of the compiler will be complicated enough that it will be hard to understand what the output code is doing. Extremely simple test cases, on the other hand, are trivially realizable on a 16-layer machine. For instance, the update rule for Conway's game of life[5] (inarguably the most well known CA) can be implemented as a single scan on a 16-layer machine. Since there is no real challenge to find the best solution, it doesn't make a particularly good test of the compiler.

However, the same automaton is challenging to implement on a 4-layer machine; it is no longer possible to anneal the entire transition rule into a single scan, but the output code produced is still simple enough that it is easy for a human being to understand.

For these reasons, I have tested my compiler mainly with relatively simple test cases, targeting both a 16-layer machine and a 4-layer machine. Besides the option to set the number of layers available, there are also command line flags to output the code after each major phase, or to skip individual phases altogether. These flags allow for some glass box testing, and were particularly useful while debugging the system.

The two test cases that I used most often were Conway's life and a version of the

HPP lattice gas[6]. I examined the performance of the compiler on two experiments with essentially every possible combination of the flags described above, with various numbers of layers available, from 4 to 16.

I also constructed some special test cases to verify the behavior of the compiler for certain constructs of the source language which were not used by either HPP or Life, such as nested conditionals.

Chapter 4

Results

There are two main performance criteria for a compiler: it must be correct, and it must produce efficient code. It also is desirable to have the compiler itself be efficent, but this is a secondary consideration that I did not really strive for in this project.

4.1 Correctness

To say that the compiler produces correct code means that the set of operations that are produced as output are a faithful implementation of the code as specified in the source file. All the constraints imposed by the hardware must be met: all the inputs and outputs of any scan operation must be arranged in such a way that they can be assembled into a single subcell, variables that are live at the same time must be in non-overlapping locations, et cetera. Moreover, the sequence of computational operations must be equivalent to the high level code that specifies the transition rule.

On the set of test cases that I tried, it produces code that is correct, both for the 16-layer tests and the 4-layer tests. Some representative cases are shown in appendix B. These tests consist of simple, easily understood and well known cellular automata, as well as ad hoc test cases designed to test specific features that were not tested by the other experiments.

4.2 Optimization

There are several ways to measure how good the code is that a compiler produces: program size, memory consumption, execution speed. Of these, the execution speed is the most critical. Because the instruction stream of the CAM-8 is a fixed sequence, the program size (in scans) is directly proportional to the execution speed. That is, the time it takes to execute a given transition is just the time it takes to perform a scan, times the number of scans that the compiler used to implement the transition.

The primary feature of the CAM-8 is its speed for spatial computation, so it is safe to assume that users of CA machines are using it because they are concerned about speed.

The solutions found for Conway's life on both 16 and 4 layer machines were the best possible, considering the types of optimizations that it performs. That is, there are certain types of optimization which are possible, but were not attempted. For instance, the tranistion rule is expected to happen cyclicly, so it is possible to anneal first group after the last group. This optimization will almost always apply since the first group is often a special copy step that could be done at the same time as the last operation in the sequence.

The solutions found for Conway's life on both a 16-layer machine and on 4-layer machine are the best possible solutions without doing something unusually clever, such as packing multiple spatial sites of the automaton into a single subcell. With 16 layers available, the transition is encoded into a single scan with a single copy step, and with 4 layers, the transition takes 6 scans, plus a 1 scan special copy.

The solutions found for the HPP lattice gas experiment are somewhat less than optimal. For the 16-layer machine, it anneals the entire transition into a single scan, but the data organization that it picks requires it to perform a copy step which is unnecessary. Thus, as in the Life case, it prepends a scan to maintain correctness, but, unlike the Life example, the extra step would not have been necessary, given a better choice of data organization.

On a 4-layer machine, it does even worse. The HPP rule is such a simple rule that it could be performed in a single scan even on a 4-layer machine, but the compiled output takes 11 scans. Of these, 5 scans are purely copy steps. This is, in part, due to the way the test case was coded. Interestingly, the solution found for the 16-layer machine only uses 4 layers, and the results for a 5-layer machine are the same as the 16-layer result.

Chapter 5

Discussion

While the software that I have produced is not a turnkey system, it does meet the major project goals. It compiles correct code that specifies the dynamics of a CA experiment in terms of operations that can be performed by the CAM-8. Moreover, for most cases, it produces code that is reasonably optimized.

Unfortunately, the system is not as smart as the humans currently programming the CAM-8, and the best code it produces is not as fast as the best code that a human being could produce. In the near term, this is likely to discourage the current users of the CAM-8 from utilizing it. However, to paraphrase Larry Wall, you can *write faster code* by hand, but you can *write code faster* with a compiler. The long term usefullness of a compiler is not that is produces fast code, but that it allows the human programmer to think about the code at a higher level.

More importantly, perhaps, this system provides proof of concept, and the development of the system has explored some of the design problems unique to compiler design for CA architectures.

5.1 Limitations

As currently implemented, my thesis project has a number of liabilities to offset its abilities. Some of these limitations are due to misguided design decisions, but many of the missing features are lacking because their implementation was not attempted. The purpose of this project was to show that the basic idea was reasonable, and to investigate the design issues surrounding compilers for CA machines. Since the amount of time was limited, I tried to focus on the most challenging problems that I thought I could solve in the time available. In this section, I will try to describe some of the things that can be improved.

5.1.1 Nomenclature

The most visible limitation that the compiler has that should be remedied is the lack of a good name. As a research system, it is fine to say "the compiler" and "the source language" all the time, but before the system is used in a fully operational mode, it needs to have some catchy acronyms.

5.1.2 Operator decomposition

Less easily fixed, however, the current system does not attempt to do operator decomposition. Any arithmetic operation that the user can specify, such as addition, mulitplication, bitwise exclusive-or, et cetera, can be decomposed into a series of small operations. Under the current system, the size of the operands for such operations is limited by the size of the look-up table, and thus data fields bigger than 8 bits are impractical.

Operator decomposition could be encoded as a completely automated process, and the only potential design issues that would be faced are when to do it, and how much. Since the annealing phase will ideally re-group statements into the largest possible chunks, one possible approach would be to always divide all operations into the smallest possible operations.

For example, a 32-bit addition would be decomposed into a series of 1-bit add-with-carry's. During the grouping phase, these operations would be likely to be grouped back together, but they might be combined with some other operation acting on the same data. Thus, the decomposition into smallest possible operations could potentially find optimzations that a human would never even consider, by breaking the entire transition rule into pieces so simple, only a computer could understand it.

As said before, the purpose of a compiler is to hide the details of the hardware from the programmer. Until operator decomposition is implemented, programmers for the CAM-8 will still be faced with the limitations of a 16-bit look-up table. Implementing operator decomposition should be relatively straightforward, but labor intensive, so it was not included in the scope of this project.

5.1.3 Data organization

The heuristic used for data organization does not always find the best solution. This was expected, but it doesn't always find the best solution even for fairly simple cases, which was not expected. The most likely cause of this problem is that the cost function used to determine which variable is "least constrained" is too simplistic. As currently implemented, the cost function is just the number of edges that connect the variable to the rest of the graph. A better cost function might also incorporate the size of the variable in bits or whether the variable is a shadow or not.

In the end, barring a full exponential search, the data organization problem will always have the possibility for sub-optimal solution, but better performance than the current system is possible. "Tweaking" the cost metric as described above should improve the results, but was not seriously investigated.

5.1.4 Code wrapping

As described in the previous chapter, the cyclic nature of cellular automata means that some optimizations can be applied across the boundary from one transition to the next. This optimization is likely to be useful for many experiments, and should be very straightforward to implement.

5.1.5 Diagnostics

As is the case for many compilers, the diagnostic messages produced are sometimes less than enlightening. This is unfortunate; it was originally hoped that the other limitations of the compiler would be offset by good diagnostics, but this has not turned out to be the case.

5.2 Future Work

5.2.1 Run Time Environment

The primary thing that needs to be done for this work to continue to go forward is the development of an appropriate run-time environment for the CAM-8. The current environment is based on an interpreted design principle, and is closely integrated with the FORTH programming environment.

Ideally, some sort of either assembly code or object file format would be developed, along with the tools to load and send programs to the CAM-8 hardware. To this end, Harris Gilliam and I have developed a library of C code for creating properly formatted instruction streams and communucating them to the CAM-8 over the SBus interface.

If this library were integrated into a set of tools capable of loading, saving, and broadcasting CAM-8 instructions, then the compiler tools developed in this project would be much closer to being an operational system.

5.2.2 Source language design

This project focused on the structure of a CA compiler, and deliberately avoided the problems of language design by implementing a subset of the C programming language. There is a lot of room for investigation into what features a language specially developed for CA should have, and what things might be useful.

For instance, the current language does not support the definition of structured data types; would they be useful? It would definitely be useful if the programmer could specify new operators as small look-up tables. What other types of syntactic sugar would be appropriate?

One thing that a CA language should have is a clean way to merge multiple programs. For instance, it should be possible to run Conway's life and HPP at the same time, and to define some way in which they interact. If the two experiments have already been defined in the language, it ought to be simple to specify an interacting synthesis of the two. It is not clear how this should be done, but it is an important direction for future research.

5.2.3 Other CAM-8 features

For simplicity's sake, my model of the CAM-8 only considered the scan and kick operations. As it happens, the hardware has a number of other features that should also be available to a programmer of the machine who wants to use the compiler. The machine has special event counters, built in display hardware, input/output ports, and a host of other goodies. There should be some way to access these features using a compiler based system, but without making the compiler completely specific to this particular hardware.

5.2.4 Other techniques for grouping and data organization

As with all compilers there is always some room to improve the optimization. One easy way that the output code might be improved is to insert a re-ordering phase, that shuffles the instructions of the intermediate language (within the bounds of correctness) before the grouping is performed. A simple re-ordering of the IL designed to reduce the lifetime of temporary variables would have fixed the problem of the worst test case, HPP on a 4-layer machine.

In the longer view, however, the tasks of data organization and grouping of the intermediate language are not completely orthogonal. There is a synergistic relationship between the two; how the data is arranged determines how many copy steps a given grouping will have, and how the statements are grouped determines the constraints on the data organization. This suggests that an ideal optimization phase might consider both problems simultaneously.

Even without trying to solve both problems at once, the heuristics used for either problem could be improved. Both processes might benefit from the ability to backtrack. The grouping process, in particular, is effectively a hill-climbing search algorithm, and is prone to getting stuck on local maxima.

5.2.5 Intermediate language design

Another topic of investigation might be ways that the intermediate language could be improved. One possible improvement is to make the kick operations explicit. The current system of implicit kicks being specified by the offset of the shadow variable is fine for normal operations, but is highly confusing for special copy operations that must *not* be kicked.

In the current system, each variable and each shadow have a specific location that is constant throughout the execution of the transition rule. The system would be more powerful if it were possible for the compiler to relocate variables through the course of the experiment. In the current implementation, if a field is live, then the compiler protects the original variable from being overwritten, as if it were the only true copy of the variable. If, however, each instance of the variable were considered equally valid and the compiler only insisted that *some* copy of the variable be protected, the output code would be both faster and cleaner. To take advantage of this change, we would want to create a new name for each use of a shadow variable.

These are only two suggestions for ways to improve the IL. There may be others. It was originally thought that this IL was simple enough that no improvements could be made, but this turned out not to be the case.

5.2.6 Other compiler tricks

Many of the optimizing tricks used by traditional compilers, such as constant folding, code hoisting, et cetera, are not applicable to this architecture. Others, such as dead code elimination and common sub-expression elimination very well might apply in some situations. These techniques should be investigated and integrated into the CA compiler where they are appropriate.

There are also optimizations that can be done on the target code, such as the use of partial LUT's, and the use of permuted identity LUTs for permutations and copies of the data. These features should be incorporated into the back-end of the compiler.

5.2.7 Other targets

Finally, as it stands, the compiler is tailored to the CAM-8 architecture, but the overall design should be applicable to other SIMD or vector based architectures. A challenging and potentially useful project might be to attempt to create a new back-end for the system that targets some other piece of hardware. Hopefully, future CA machines will eventually be available, and the system should be largely compatible with these systems as-is.

Another useful thing might be to develop a back end for the compiler that produces code in some traditional language such as C. In other words, the system would produce programs that would simulate cellular automata on non-parallel architectures. The CAM-8 simulator [10] would also be a likely alternate target.

5.3 Conclusion

A compiler for cellular automata machine architectures is both possible and desirable. The main parts of such a system are implemented and work correctly. While there is room for improvements in the output code, this is true for any compiler, and the system works remarkably well, considering that it is the first effort of its kind.

Before the system is fully operational, an appropriate run-time environment needs to be developed, and the two systems need to be integrated. When this is done, users of the CAM-8 machine will have newfound expressive power, and hopefully the improved programming environment will encourage more people to understand and use the power of spatially organized computing.

Afterword

A programmer is a person who designs a computation. He is limited by many things, including the architecture of the machine that he programs on, but that's an artificial constraint. Ultimately, he is constrained by the laws of physics.

As computers get faster and smaller, their designers are becoming increasingly aware of the limitations imposed by physics. If they continue on this trend, they will ultimately reach a place where they realize that the only way to squeeze more computing performance from a machine is to work *with* the laws of physics. When that time comes, they will design machines for spatially organized computation, and will explain to the programmers that we need to start thinking about the physics of computation.

Cellular Automata Machines are such machines. The CAM-8 is ahead of its time. Spatially organized machines openly acknowledge that not all memory locations are equally far from the processor. To really get the most computation for your time, you need to take advantage of facts like that. People who design algorithms should be considering the fundamental constraints on computation, and those constraints should be dictated by physics, not by architecture.

And yet, despite the goal of allowing programmers to think about physics, rather than architecture, the design of the CAM-8 sometimes makes it hard to forget that architecture plays a big role. If we are going to start thinking about the computation in terms of the ultimate constraints, we need a way to forget about the technological details. For this reason, compilers for cellular automata machines are essential.

Appendix A

Compiler Source Code

A.1 Makefile

```
CAM8BASE = /cam8/working

# Program to use for installing files
INSTALL = install −c

# Prefered utilities
CC = gcc
FLEX = flex
YACC = bison −y −p dt
YFLAGS = −d
C8CPP = $(CAM8BASE)/bin/c8cpp

DEBUG = −g
PROF =

INCLUDES = −I.. −I$(CAM8BASE)/ −I/usr/local/ −I/cam8/working/
−I/usr/local/perl−5.002b3/lib/sun4−sunos/5.002/CORE

FFLAGS = −Cfe −B −Pdt
CFLAGS = $(DEBUG) $(PROF) $(INCLUDES)
LDFLAGS = −L/usr/local/lib −L$(CAM8BASE)/lib
LDLIBS = −lperl −lCAM −lm
```

```
LIBRARIES = -L/usr/local/perl-5.002b3/lib/sun4-sunos/5.002/CORE
LIBS = -lperl -lm

DEPEND = makedepend
DEPENDFLAGS = $(INCLUDES)
-I/usr/local/gcc-2.7.0/lib/gcc-lib/sparc-sun-sunos4.1.4/2.7.0/

LIBRARY = libCAM.a
INC_DIR = $(CAM8BASE)//CAM
LIB_DIR = $(CAM8BASE)/lib
BIN_DIR = $(CAM8BASE)/bin


###SRC =    $(XSRC:xc=c)
CSRC =     thesis.c actions.c perlutil.c \
    statement.c objects.c print.c \
    math.c anal.c conflicts.c size.c \
    edge.c kolor.c group.c anneal.c \
    group_conflicts.c copy.c
SRC = lex.c yacc.c $(CSRC)


INC =    $(XINC:xh=h)

LINC =    $(XLINC:xh=h)

OBJ =    $(SRC:c=o)


##########################################################################

.SUFFIXES:    .c .h .flex .y


.y.c:
    $(YACC.y) $<
    replace YY DT y.tab.c y.tab.h
    replace yy dt y.tab.h y.tab.c
    mv y.tab.c $@
```

```
.y.ln:
    $(YACC.y) $<
    $(LINT.c) −o $@ −i y.tab.c
    $(RM) y.tab.c
.y.o:
    $(YACC.y) $<
    $(COMPILE.c) −o $@ y.tab.c


.flex.c:
    $(FLEX) $(FFLAGS) −t $< > $*.c
    replace YY DT $*.c
    replace yy dt $*.c

.c.o:
    $(CC) $(CFLAGS) −c $<

##########################################################################

all: thesis

depend: $(SRC) $(INC) $(LINC)
    $(DEPEND) $(DEPENDFLAGS) $(SRC)

$(LIBRARY): $(OBJ)
    ar cvr $(LIBRARY) $?
    ranlib $(LIBRARY)

##########################################################################

thesis: proto.h $(OBJ)
    $(CC) −o thesis $(OBJ) $(LIBRARIES) $(LIBS)

install: CAMlib.h $(LIBRARY)
    $(INSTALL) −d −m 775 $(INC_DIR)
    $(INSTALL) −m 664 $(INC) $(INC_DIR)
    $(INSTALL) $(LIBRARY) $(LIB_DIR)
    $(INSTALL) c8cpp $(BIN_DIR)
    ranlib $(LIB_DIR)/$(LIBRARY)
```

```
proto.h:  $(CSRC)
    /im/magnus/bin/proto $(CSRC) > proto.h

count::
     wc $(CSRC) kludge.h proto.h yacc.y lex.flex


clean::
    /bin/rm -f *.o *.a core *~

ci::
    ci -l $(SRC)
```

A.2 actions.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"

SV**     fetch_result;
AV*     conditions = NULL;
HV*     var_namespace = NULL;
HV*     var_shadowspace = NULL;
HV*     var_space = NULL;
FILE*   my_stdin = stdin;
FILE*   my_stdout = stdout;
FILE*   my_stderr = stderr;


HV* top(HV* cell, HV* rule) {

  live_var_anal(cell, rule);

  if (print_intermediate) {
    fprintf(stdout,"\ncode sequence: \n");
    fprint_block(stdout, rule);
```

```
}
live_var_anal(cell, rule);
STORE(rule,"grouping",convert_code_to_groups(FETCH(rule,"code")));

if (do_anneal) {
  better_anneal_groups(FETCH(rule,"grouping"));
}
if (do_anneal && print_after_anneal) {
  fprint_grouplist(stdout,FETCH(rule,"grouping"));
}
group_mark_conflicts(rule);

union_hv(var_space, var_shadowspace);
union_hv(var_space, var_namespace);

kolor(var_space);

if (do_copy) {
  simple_copy_operands(FETCH(rule,"grouping"));
}
if (print_after_copy) {
  fprintf(stdout,"%ccode after copy: \n",PAGE_BREAK);
  fprint_grouplist(stdout,FETCH(rule,"grouping"));
}

if (do_second_anneal) {
  better_anneal_groups(FETCH(rule,"grouping"));
}
if (print_final) {
  fprintf(stdout,"%cfinal code after regrouping: \n",PAGE_BREAK);
  fprint_grouplist(stdout,FETCH(rule,"grouping"));
}
if (print_variables) {
  fprintf(stdout,"\nvariables: \n");
  fprint_varspace_verbose(stdout,var_namespace);
  fprintf(stdout,"\nshadows: \n");
  fprint_varspace_verbose(stdout,var_shadowspace);
}

if (print_conflicts) {
```

```
    fprint_varspace_conflicts(stdout,var_namespace,"use_edges");
    fprint_varspace_conflicts(stdout,var_shadowspace,"use_edges");
    fprint_varspace_conflicts(stdout,var_namespace,"live_edges");
    fprint_varspace_conflicts(stdout,var_namespace,"src_dest_edges");
    fprint_varspace_conflicts(stdout,var_shadowspace,"src_dest_edges");
  }

  return rule;
}

void initialize() {
  conditions = newAV();
  printf("dt output\n");
  av_push(conditions,REF(new_constant(1)));
  var_namespace = newHV();
  var_shadowspace = newHV();
  var_space = newHV();
}

HV* add_field(HV* field, HV* cell) {
  SV* name;
  SV* ref;
  if (NULL == field) { dterror("null field"); return cell; }
  name = FETCH(field,"name");
  ref = REF(field);
  STORE(cell,SvPV(name,na),ref);
  STORE(var_namespace,SvPV(name,na),ref);
  return cell;
}

HV* ref_constant(int a) {
  return (HV*)new_expr(new_constant(a));
}

HV* ref_variable(char *a) {
  SV* entry;
  entry = FETCH_Q(var_namespace,a);
  if (!entry) {
    HV* new;
    new = (HV*) new_variable(a);
```

```
    STORE(var_namespace,a,REF(new));
    return (HV*)new_expr((new));
  } else if (has_("field",entry)) {
    return (HV*)new_expr(new_shadow(HASH(entry)));
  } else {
    return (HV*)new_expr(HASH(entry));
  }
}

HV* unary_op(int op, HV* expr) {
  SV* a;

  a = FETCH(HASH(FETCH(expr,"result")),"type");
  if (eq(SvPV(a,na),"constant")) {
    a = FETCH(HASH(FETCH(expr,"result")),"value");
    sv_setiv(a,perform_unary_op(op,SvIV(a)));
    return expr;
  }

  a = (SV*) unary_assign_statement(op,FETCH(expr,"result"));
  append_statement(FETCH(expr,"code"),a);
  STORE(expr,"type",FETCH(HASH(a),"type"));
  STORE(expr,"result",FETCH(HASH(a),"result"));

  return expr;
}

HV* binary_op(HV* expr1, int op, HV* expr2) {
  SV* a;

  a = FETCH(HASH(FETCH(expr1,"result")),"type");
  if (eq(SvPV(a,na),"constant")) {
    a = FETCH(HASH(FETCH(expr2,"result")),"type");
    if (eq(SvPV(a,na),"constant")) {
      SV* b;
      a = FETCH(HASH(FETCH(expr1,"result")),"value");
      b = FETCH(HASH(FETCH(expr2,"result")),"value");
      sv_setiv(a,perform_binary_op(SvIV(a),op,SvIV(b)));
      return expr1;
    }
```

```
  }

  a = (SV*) binary_assign_statement(FETCH(expr1,"result"),
                    op,
                    FETCH(expr2,"result"));
  concat_statements(FETCH(expr1,"code"),FETCH(expr2,"code"));
  append_statement(FETCH(expr1,"code"),a);
  STORE(expr1,"type",FETCH(HASH(a),"type"));
  STORE(expr1,"result",FETCH(HASH(a),"result"));

  return expr1;
}

HV* list_op(HV* expr1, HV* expr2) {
  SV* e1_res;
  SV* list;

  concat_statements(FETCH(expr1,"code"),FETCH(expr2,"code"));
  STORE(expr1,"type",newSVpv("list",0));
  e1_res = FETCH(expr1,"result");
  list = FETCH_Q(expr1,"list");
  if (!list) {
    STORE(expr1,"list",REF(newAV()));
    list = FETCH(expr1,"list");
    av_push((AV*)SvRV(list),e1_res);
  }
  av_push((AV*)SvRV(list),FETCH(expr2,"result"));
  STORE(expr1,"result",FETCH(expr2,"result"));
  return expr1;
}

SV* listize_expr(HV* expr) {
  SV* foo;
  if (foo=FETCH_Q(expr,"list")) {
  } else {
    STORE(expr,"list",REF(newAV()));
    foo = FETCH(expr,"list");
    av_push((AV*)SvRV(foo),FETCH(expr,"result"));
  }
   return foo;
```

```
}

HV* ghost_offset_ref(HV* expr) {
  SV* result;
  SV* foo;
  static SV* zero = NULL;

  if (!zero) { zero = listize_expr(new_expr(new_constant(0))); }
  result = FETCH(expr,"result");

  if (is_("variable",result)) {
    if (FETCH_Q(HASH(result),"field")) {
      STORE((HASH(result)),"offset",zero);
      if (foo = FETCH_Q(var_shadowspace,
            short_term_var_name(FETCH(expr,"result")))) {
    STORE(expr,"result",foo);
      } else {
    STORE(var_shadowspace,
          short_term_var_name(FETCH(expr,"result")),
          FETCH(expr,"result"));
      }
    }
  }
  return expr;
}

HV* offset_ref(HV* expr, HV* list_expr) {
  HV* result;
  SV* foo;
  STORE((HASH(FETCH(expr,"result"))),
    "offset",
    listize_expr(list_expr));
  result = HASH(FETCH(expr,"result"));
  if (FETCH_Q(result,"field")) {

    /* add variable ref to shadow list */
    if (foo = FETCH_Q(var_shadowspace,
            short_term_var_name(FETCH(expr,"result")))) {
      STORE(expr,"result",foo);
    } else {
```

```
      STORE(var_shadowspace,
        short_term_var_name(FETCH(expr,"result")),
        FETCH(expr,"result"));
    }
  } else {
    dterror("offset used on non-field %s", SvPV(FETCH(result,"name"),na));
  }
  return expr;
}

HV* assign_op(HV* lvalue, HV* expr) {
  SV* a;

  a = assign_statement(FETCH(lvalue,"result"),FETCH(expr,"result"));
  append_statement(FETCH(expr,"code"),a);
  STORE(expr,"type",FETCH(lvalue,"type"));
  STORE(expr,"result",FETCH(lvalue,"result"));

  return expr;
}



HV* if_stat(HV* expr, HV* block) {

  /* condition block with new condition */
  concat_statements(FETCH(expr,"code"),FETCH(block,"code"));
  SvREFCNT_inc(FETCH_Q(expr,"type"));
  SvREFCNT_inc(FETCH_Q(expr,"result"));
  STORE(expr,"type",newSVpv("if_statement",0));
  STORE(expr,"result",FETCH(block,"result"));
  STORE(expr,"if_conditional",av_pop(conditions));
  return expr;
}

HV* if_start(HV *expr) {
  SV* a;

  a = binary_assign_statement(FETCH(expr,"result"),
                tAND,
```

```
                  (*av_fetch(conditions,
                     av_len(conditions),
                     0)));
  uncondition(a);
  append_statement(FETCH(expr,"code"),a);
  SvREFCNT_inc(FETCH_Q(expr,"result"));
  STORE(expr,"result",FETCH(HASH(a),"result"));
  av_push(conditions,FETCH(expr,"result"));
  return expr;
}

HV* new_s_list() {
  HV* res;
  res = newHV();
  STORE(res,"type",newSVpv("statement_list",0));
  STORE(res,"code",REF(newAV()));
  return res;
}

HV* add_statement(HV* s_list, HV* s) {
  concat_statements(FETCH(s_list,"code"),FETCH(s,"code"));
  SvREFCNT_inc(FETCH_Q(s_list,"result"));
  STORE(s_list,"result",FETCH(s,"result"));
}

HV* process_block(HV* block) {
  return block;
}
```

A.3 anal.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"
```

```
/* marks the variables which are live after each statement in
   "live_vars" */

live_var_anal(HV* cell, HV* rule) {
  AV* a;
  int i;
  HE* he;
  HV* live_vars;
  live_vars = newHV();

  hv_iterinit(cell);
  while(he = hv_iternext(cell)) {
    STORE(live_vars,hv_iterkey(he,&na),hv_iterval(cell,he));
  }


  a = (AV*)SvRV(FETCH(rule,"code"));

  distinguish_statement(*av_fetch(a,0,0));

  for (i=av_len(a); i>=0; i--) {
    analyze_statement(*av_fetch(a,i,0),live_vars);
  }

}

distinguish_statement(SV* s) {
  HV* h;
  h = HASH(s);

  STORE(h,"start",newSViv(1));
}


analyze_statement(SV* s, HV* live_vars) {
  SV* cond;
  SV* result;
  SV* arg;
  SV* foo;
  HV* h;
```

```
h = HASH(s);

STORE(h,"live_vars",REF(copy_hv(live_vars)));
/* if I do this here, then "live_vars" stores the things
   that are live *after* the instruction, but if I do it
   at the end, it is the things live before */

if (is_("comment",s)) return;

cond = FETCH(h,"condition");
result = FETCH(h,"result");

if (is_("variable",cond)) {
  STORE(live_vars,name_(cond),cond);
  if (!has_("temp",result)) {
    if (foo = FETCH_Q(HASH(result),"fraction")) { result = foo; }
    if (foo = FETCH_Q(HASH(result),"shadow")) { result = foo; }
    STORE(live_vars,name_(result),result);
  } else {
    DELETE(live_vars,name_(result));
  }
} else {
  while (EXISTS(live_vars,name_(result))) {
    DELETE(live_vars,name_(result));
  }
}

if ((arg = FETCH_Q(h,"argument"))) {
  if (is_("variable",arg)) {
    if (foo = FETCH_Q(HASH(arg),"fraction")) { arg = foo; }
    if (foo = FETCH_Q(HASH(arg),"shadow")) { arg = foo; }
    STORE(live_vars,name_(arg),arg);
  }
}
if ((arg = FETCH_Q(h,"argument1"))) {
  if (is_("variable",arg)) {
    if (foo = FETCH_Q(HASH(arg),"fraction")) { arg = foo; }
    if (foo = FETCH_Q(HASH(arg),"shadow")) { arg = foo; }
    STORE(live_vars,name_(arg),arg);
  }
```

```
  }
  if ((arg = FETCH_Q(h,"argument2"))) {
    if (is_("variable",arg)) {
      if (foo = FETCH_Q(HASH(arg),"fraction")) { arg = foo; }
      if (foo = FETCH_Q(HASH(arg),"shadow")) { arg = foo; }
      STORE(live_vars,name_(arg),arg);
    }
  }

}
```

A.4 anneal.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"

SV* convert_code_to_groups(SV* ss) {
  int i;
  SV* foo;
  AV* g_list, *s_list;
  s_list = (AV*)SvRV(ss);
  g_list = newAV();
  for (i =0; i<=av_len(s_list); i++) {
    foo = convert_statement_to_group(*av_fetch(s_list,i,0));
    av_store(g_list,i,foo);
  }
  return REF(g_list);
}


simple_anneal_groups(SV *gs) {
  int i;
  AV* g_list;
  int change = 1;
  g_list = (AV*)SvRV(gs);
```

```
  while (change) {
    change = 0;
    for (i=av_len(g_list); i>0; i--) {
      if (speculate_concat(*av_fetch(g_list,i-1,0),*av_fetch(g_list,i,0))) {
    concat_groups(*av_fetch(g_list,i-1,0),*av_fetch(g_list,i,0));
    array_delete(g_list,i-1);
    change = 1;
      }
    }
  }
}




better_anneal_groups(SV *gs) {
  int i, j;
  AV* g_list;
  SV* depends = NULL;
  SV* current_group;
  SV* other_group;
  int change = 1;
  int acquired;
  SV* foo;
  g_list = (AV*)SvRV(gs);

    for (i=av_len(g_list); i>0; i--) {
      current_group = *av_fetch(g_list,i,0);
      if (depends) hv_undef(HASH(depends));
      depends = REF(newHV());
      for (j = i-1; j ≥ 0; j--) {
    other_group = *av_fetch(g_list,j,0);
    acquired = 0;
    if (speculate_concat(other_group, current_group)) {
      foo = REF(newHV());
      special_union_hv(HASH(foo),HASH(FETCH(HASH(other_group),"outs")));
      special_intersect_hv(HASH(foo),HASH(depends));
      if (0 == varlist_countbits(foo)) {
          concat_groups(other_group,current_group);
```

```
          array_delete(g_list,j);
          i--; change = 1; acquired = 1;
          STORE(HASH(*av_fetch(g_list,i-1,0)),"live_after",
            FETCH(HASH(current_group),"live_before"));

      }
      hv_undef(HASH(foo));
    }
    if (!acquired) {
        special_union_hv(HASH(depends),HASH(FETCH(HASH(other_group),"ins")));
    }
      }
    }
}

/* special version of union_hv that originalizes the source */
special_union_hv(HV* target, HV* source) {
  HE* he;
  SV* bar;
  hv_iterinit(source);
  while(he = hv_iternext(source)) {
    bar = var_original(hv_iterval(source,he));
    STORE(target,short_term_var_name(bar),bar);
  }
  return;
}
```

A.5 conflicts.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"


mark_conflicts(HV* rule) {
  AV* a;
```

```
  int i;
  HE* he;
  HV* live_vars;
  live_vars = newHV();

  a = (AV*)SvRV(FETCH(rule,"code"));

  for (i=0; i≤ av_len(a); i++) {
    constrain_statement(*av_fetch(a,i,0));
  }

}

constrain_statement(SV* s) {
  SV* cond;
  SV* result;
  SV* arg;
  HV* live_vars;
  HV* h;
  h = HASH(s);

  if (is_("comment",s)) return;

  live_vars = HASH(FETCH(h,"live_vars"));
  cond = FETCH(h,"condition");
  result = FETCH(h,"result");

  result = var_original(result);

  add_live_edges(result, live_vars);

  add_edge_by_type(cond,result,"src_dest_edges");

  if ((arg = FETCH_Q(h,"argument"))) {
    if (s = FETCH_Q(HASH(arg),"fraction")) { arg = s; }
    add_edge_by_type(arg,cond,"use_edges");
    add_edge_by_type(arg,result,"src_dest_edges");
  }
  if ((arg = FETCH_Q(h,"argument1"))) {
    if (s = FETCH_Q(HASH(arg),"fraction")) { arg = s; }
```

```
    add_edge_by_type(arg,cond,"use_edges");
    add_edge_by_type(arg,result,"src_dest_edges");
  }
  if ((arg = FETCH_Q(h,"argument2"))) {
    if (s = FETCH_Q(HASH(arg),"fraction")) { arg = s; }
    add_edge_by_type(arg,cond,"use_edges");
    add_edge_by_type(arg,FETCH(h,"argument1"),"use_edges");
    add_edge_by_type(arg,result,"src_dest_edges");
  }
}

add_live_edges(SV* s, HV* live_vars) {
  HE* he;
  SV* foo;

  /* assumes s is not a shadow or fraction */

  hv_iterinit(live_vars);
  while (he = hv_iternext(live_vars)) {
    add_edge_by_type(s,hv_iterval(live_vars,he),"live_edges");
  }
}
```

A.6 copy.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"


simple_copy_operands(SV* gs) {
  int i;
  AV* g_list;
  g_list = (AV*)SvRV(gs);

  for (i=0; i<= av_len(g_list); i++) {
    i += group_do_copies(g_list,i);
```

```
  }
}


/* requires that all variables have locations of some kind */
int group_do_copies(AV* gl,int i) {
  SV *foo, *bar;
  SV* gs;
  HV* gh;
  HE* he;
  int inserts = 0;
  SV* special_copy_var;

  gs = *av_fetch(gl,i,0);
  gh = HASH(gs);
  foo = FETCH(gh,"ins");

  /* for each input */
  /* generate a copy step */
  /* if needed */
  /* and prepend it to the group */

  hv_iterinit(HASH(foo));
  while (he = hv_iternext(HASH(foo))) {
    if (has_("shadow",hv_iterval(HASH(foo),he))) {
      inserts += prepend_shadow_copy(hv_iterval(HASH(foo),he), gl, i,
                   HASH(FETCH(gh,"live_before")));
    }
    /* if input is spilled
       copy it over the corresponding output
       and use the copy */
    if (has_("spilled",hv_iterval(HASH(foo),he))) {
      special_copy_var = make_special_copy_var(hv_iterval(HASH(foo),he),
                          FETCH(gh,"outs"));
      inserts += insert_var_copy(special_copy_var,
                 hv_iterval(HASH(foo),he),
                 gl, i,
                 HASH(FETCH(gh,"live_before")));
      DELETE(HASH(foo),short_term_var_name(hv_iterval(HASH(foo),he)));
      STORE(HASH(foo),short_term_var_name(special_copy_var),special_copy_var);
```

```
      /* need to do actually change the statements ! */
      substitute_var_in_code(hv_iterval(HASH(foo),he), special_copy_var,
FETCH(gh,"code"));
    }
  }

  foo = FETCH(gh,"outs");

  /* for each output */
  /* do the same */
  /* but append it to the group */
  /* instead */

  hv_iterinit(HASH(foo));
  while (he = hv_iternext(HASH(foo))) {
    if (has_("shadow",hv_iterval(HASH(foo),he))) {
      inserts += append_shadow_copy(hv_iterval(HASH(foo),he), gl, i+inserts,
                    HASH(FETCH(gh,"live_after")));
    }
    /* if output is spilled
       copy corresponding input over it
       and use the copy */
    if (has_("spilled",hv_iterval(HASH(foo),he))) {
      SV* c_var;
      c_var = find_corresponding_var(hv_iterval(HASH(foo),he),
                          FETCH(gh,"ins"));
      if (c_var) {
    special_copy_var = make_special_copy_var(c_var, FETCH(gh,"ins"));
    inserts += insert_var_copy(special_copy_var, c_var, gl, i,
                  HASH(FETCH(gh,"live_after")));
      DELETE(HASH(FETCH(gh,"ins")),short_term_var_name(c_var));

STORE(HASH(FETCH(gh,"ins")),short_term_var_name(special_copy_var),special_copy_var);
      substitute_var_in_code(c_var, special_copy_var, FETCH(gh,"code"));
      }
    }
  }

  return inserts;
}
```

```
/* find the var in vspace that is in the same layers as  v */
/* NOTE: this is an overly simplistic model, and assumes that
   only a single variable covers v */

SV* find_corresponding_var(SV* v, SV* vspace) {
  HE* he;
  HV* vspace_h;
  vspace_h = HASH(vspace);
  hv_iterinit(vspace_h);
  while (he = hv_iternext(vspace_h)) {
    if (mask_variable(v) & mask_variable(hv_iterval(vspace_h,he))) {
      return hv_iterval(vspace_h,he);
    }
  }
  return NULL;
}

/* make a variable that is in the same layers as orig, but
   each layer is from the same subcell as vspace */
SV* make_special_copy_var(SV* orig, SV* vspace) {
  HE* he;
  HV* vspace_h;
  SV* special_copy_var;
  vspace_h = HASH(vspace);
  special_copy_var = REF(new_temp());
  STORE(HASH(special_copy_var),    "low", FETCH(HASH(orig),"low"));
  STORE(HASH(special_copy_var),    "high", FETCH(HASH(orig),"high"));
  STORE(HASH(special_copy_var),    "layer", FETCH(HASH(orig),"layer"));
  STORE(HASH(special_copy_var),    "subcell", newSViv(-1));

/*   hv_iterinit(vspace_h); */
/*   while( */

  return special_copy_var;

}


int prepend_shadow_copy(SV* var, AV* gl, int i, HV* live_vars) {
```

```
  SV* target;
  SV* source;
  target = var;
  source = var_original(var);
  return insert_var_copy(target, source, gl, i, live_vars);
}

int append_shadow_copy(SV* var, AV* gl, int i, HV* live_vars) {
  SV* target;
  SV* source;
  target = var_original(var);
  source = var;
  return insert_var_copy(target, source, gl, i+1, live_vars);
}

int insert_var_copy(SV* target, SV* source, AV* gl, int i, HV* live_vars) {
  /* if they don't overlap layers */
  if (!(mask_variable(target) & mask_variable(source))) {
    return standard_var_copy(target,source,gl,i,live_vars);
  } else {
    /* they do overlap */
    /* if they are in the same subcell */
    if (SvIV(FETCH(HASH(target),"subcell")) ==
    SvIV(FETCH(HASH(source),"subcell"))) {
      /* noop */
      return 0;
    } else {
      /* else different subcells */
      /* if odd # bits */
      if (1 & size_bits(target)) {
    /* if smaller than half subcell */
    if (size_bits(target)*2 <= num_layers) {
      /* do small odd intersubcell copy */
      return small_odd_var_copy(target,source,gl,i,live_vars);
    } else {
      /* else bigger than half subcell */
      /* do big odd intersubcell copy */
      fprintf(stdout,"WARNING: copy steps for large odd size moves between subcells
weren't implemented\n");
      return 0;
```

```
    }
      } else {
    /* else even # bits */
    /* do even intersubcell copy */
    return even_var_copy(target,source,gl,i,live_vars);
      }
    }
  }
}

int standard_var_copy(SV* target, SV* source, AV* gl, int i, HV* live_vars) {
  SV* group;
  char *ct;
  group = new_group();
  ct = (char *)strsave("copy step");
  append_statement(FETCH(HASH(group),"code"), comment_statement(ct));
  append_statement(FETCH(HASH(group),"code"),assign_statement(target,source));
  /* fix the other attributes of group */
  STORE(HASH(FETCH(HASH(group),"ins")),short_term_var_name(source),source);
  STORE(HASH(FETCH(HASH(group),"outs")),short_term_var_name(target),target);

STORE(HASH(FETCH(HASH(group),"required_writes")),short_term_var_name(target),target);
  union_hv(HASH(FETCH(HASH(group),"live_before")),live_vars);
  union_hv(HASH(FETCH(HASH(group),"live_after")),live_vars);
  array_insert(gl,i,group);
  return 1;
}

int small_odd_var_copy(SV* target, SV* source, AV* gl, int i, HV* live_vars) {
  int count = 0;
  SV* magic_temp;
  magic_temp = REF(new_temp());
  STORE(HASH(magic_temp),"low", FETCH(HASH(source),"low"));
  STORE(HASH(magic_temp),"high", FETCH(HASH(source),"high"));
  add_live_edges(magic_temp, live_vars);
  add_edge_by_type(magic_temp,target,"use_edges");
  add_edge_by_type(magic_temp,source,"use_edges");
  if (!pick_kolor(magic_temp,0)) {
    fprintf(stdout,"I give up!\n");
  }
```

```
  count += insert_var_copy(magic_temp, source, gl, i,live_vars);
  count += insert_var_copy(target, magic_temp, gl, i + count,live_vars);
  return count;
}


int even_var_copy(SV* target, SV* source, AV* gl, int i, HV* live_vars) {
  SV* group;
  int count = 0;
  char *ct;
  SV *ts, *f1, *f2;
  int midpoint = (size_bits(target)>>1)-1;
  group = new_group();

  ct = (char *)strsave("copy step");
  ts = comment_statement(ct);
  append_statement(FETCH(HASH(group),"code"), ts);

  f1 = REF(new_fraction(target,0,midpoint));
  f2 = REF(new_fraction(source,midpoint+1,size_bits(source)-1));
  ts = assign_statement(f1,f2);
  append_statement(FETCH(HASH(group),"code"), ts);
  STORE(HASH(FETCH(HASH(group),"outs")),short_term_var_name(f1),f1);
  STORE(HASH(FETCH(HASH(group),"ins")),short_term_var_name(f2),f2);

STORE(HASH(FETCH(HASH(group),"required_writes")),short_term_var_name(f2),f2);
  union_hv(HASH(FETCH(HASH(group),"live_before")),live_vars);
  union_hv(HASH(FETCH(HASH(group),"live_after")), live_vars);
  array_insert(gl,i,group);

  group = new_group();
  f1 = REF(new_fraction(target,midpoint+1,size_bits(target)-1));
  f2 = REF(new_fraction(source,0,midpoint));
  ts = assign_statement(f1,f2);
  append_statement(FETCH(HASH(group),"code"), ts);

  ct = (char *)strsave("flip semantics of variable foo bar baz");
  sprintf(ct,"flip semantics of variable %s",name_(target));
  ts = comment_statement(ct);
  append_statement(FETCH(HASH(group),"code"), ts);
```

```
  STORE(HASH(FETCH(HASH(group),"outs")),short_term_var_name(f1),f1);
  STORE(HASH(FETCH(HASH(group),"ins")),short_term_var_name(f2),f2);

STORE(HASH(FETCH(HASH(group),"required_writes")),short_term_var_name(f2),f2);
  union_hv(HASH(FETCH(HASH(group),"live_before")),live_vars);
  union_hv(HASH(FETCH(HASH(group),"live_after")), live_vars);
  array_insert(gl,i+1,group);

  return 2;
}

substitute_var_in_code(SV* v_old, SV* v_new, SV* ss) {
  int i;
  SV* stat;
  SV* old;
  AV* s_list;
  s_list = (AV*) SvRV(ss);
  for (i=0;i<=av_len(s_list);i++) {
    stat = *av_fetch(s_list,i,0);
    if (old = FETCH_Q(HASH(stat),"result")) {
      if (SvRV(v_old) == SvRV(old)) {
    STORE(HASH(stat),"result",v_new);
      }
    }
    if (old = FETCH_Q(HASH(stat),"argument")) {
      if (SvRV(v_old) == SvRV(old)) {
    STORE(HASH(stat),"argument",v_new);
      }
    }
    if (old = FETCH_Q(HASH(stat),"argument1")) {
      if (SvRV(v_old) == SvRV(old)) {
    STORE(HASH(stat),"argument1",v_new);
      }
    }
    if (old = FETCH_Q(HASH(stat),"argument2")) {
      if (SvRV(v_old) == SvRV(old)) {
    STORE(HASH(stat),"argument2",v_new);
      }
    }
```

```
    if (old = FETCH_Q(HASH(stat),"condition")) {
      if (SvRV(v_old) == SvRV(old)) {
    STORE(HASH(stat),"condition",v_new);
      }
    }
  }
}
```

A.7 edge.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"

/* adds an edge from one to two */
/* adds an edge from two to one */
/* does nothing on duplicates */

add_edge_by_type(SV* one, SV* two, char *type) {
  HV* el_one;
  HV* el_two;
  SV* foo;

  if (!is_("variable",one)) return;
  if (!is_("variable",two)) return;

  el_one = HASH(FETCH_OR_NEW_HASH(HASH(one),type));
  el_two = HASH(FETCH_OR_NEW_HASH(HASH(two),type));

  if (foo = FETCH_Q(el_one,short_term_var_name(two))) {
    /* frowny face */
  } else {
    STORE(el_one,short_term_var_name(two),two);
    foo = FETCH(HASH(one),"edge_count");
    sv_setiv(foo,SvIV(foo)+1);
  }
```

```
  if (foo = FETCH_Q(el_two,short_term_var_name(one))) {
    /* frowny face */
  } else {
    STORE(el_two,short_term_var_name(one),one);
    foo = FETCH(HASH(two),"edge_count");
    sv_setiv(foo,SvIV(foo)+1);
  }
}

/* removes an edge from two pointing to one */
/* does nothing on duplicates */

remove_edge(SV* one, SV* two, char *type) {
 HV* el_two;
 SV* foo;

 el_two = HASH(FETCH_OR_NEW_HASH(HASH(two),type));

 if (foo = FETCH_Q(el_two,short_term_var_name(one))) {
    DELETE(el_two,short_term_var_name(one));
    foo = FETCH(HASH(two),"edge_count");
    sv_setiv(foo,SvIV(foo)-1);
  } else {
    /* frowny face */
  }

}

/* removes all edges pointing at one */

turn_off_variable(SV* one) {
  turn_off_variable_by_type(one,"use_edges");
  turn_off_variable_by_type(one,"live_edges");
  turn_off_variable_by_type(one,"src_dest_edges");
}

turn_off_variable_by_type(SV* one, char *type) {
  HV* el_one;
  HE* he;
```

```
  el_one = HASH(FETCH_OR_NEW_HASH(HASH(one),type));

  hv_iterinit(el_one);
  while(he = hv_iternext(el_one)) {
    remove_edge(one,hv_iterval(el_one,he),type);
  }

}

/* restores all edges pointing at one */
/* that is, for each edge pointing away from one
   creates an edge pointing back */

turn_on_variable(SV* one) {
  turn_on_variable_by_type(one,"use_edges");
  turn_on_variable_by_type(one,"live_edges");
  turn_on_variable_by_type(one,"src_dest_edges");
}


turn_on_variable_by_type(SV* one, char *type) {
  HV* el_one;
  HE* he;

  el_one = HASH(FETCH(HASH(one),type));

  hv_iterinit(el_one);
  while(he = hv_iternext(el_one)) {
    add_edge_by_type(one,hv_iterval(el_one,he),type);
  }

}
```

A.8 group.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
```

```
#include "y.tab.h"

SV* new_group() {
  HV* res;
  res = newHV();
  STORE(res,"type",newSVpv("group",0));
  STORE(res,"code",REF(newAV()));
  STORE(res,"ins",REF(newHV()));
  STORE(res,"outs",REF(newHV()));
  STORE(res,"kills",REF(newHV()));
  STORE(res,"live_before",REF(newHV()));
  STORE(res,"live_after",REF(newHV()));
  STORE(res,"required_writes",REF(newHV()));
  return REF(res);
}


int speculate_concat(SV* g1, SV* g2) {
  HV *h1, *h2;
  SV* foo;
  HV* foo_h;
  h1 = HASH(g1);
  h2 = HASH(g2);

  foo_h = newHV();
  foo = REF(foo_h);
  /* ins = (ins2 - kills1) + ins1 */
  union_hv(foo_h,HASH(FETCH(h2,"ins")));
  subtract_hv(foo_h,HASH(FETCH(h1,"kills")));
  union_hv(foo_h,HASH(FETCH(h1,"ins")));

  /* count bits in ins */
  if (num_layers < varlist_countbits(foo)) {
    return 0;
  }
  hv_undef(HASH(foo));

  foo_h = newHV();
  foo = REF(foo_h);
  /* outs = (outs1 | outs2) & live_after2 */
```

```
  union_hv(foo_h,HASH(FETCH(h2,"outs")));
  union_hv(foo_h,HASH(FETCH(h1,"outs")));
  special_intersect_hv(foo_h,HASH(FETCH(h2,"live_after")));

  /* count bits in outs */
  if (num_layers < varlist_countbits(foo)) {
    return 0;
  }
  hv_undef(HASH(foo));

  foo_h = newHV();
  union_hv(foo_h, HASH(FETCH(h1,"required_writes")));
  intersect_hv(foo_h, HASH(FETCH(h2,"ins")));
  if (!empty_hash(foo_h)) {
    return 0;
  }
  hv_undef(foo_h);


  return 1;

}

/* concats into g2 */
SV* concat_groups(SV* g1, SV* g2) {
  HV *h1, *h2;
  h1 = HASH(g1);
  h2 = HASH(g2);

  /* ins =  (ins2 - kills1) + ins1 */
  subtract_hv(HASH(FETCH(h2,"ins")),HASH(FETCH(h1,"kills")));
  union_hv(HASH(FETCH(h2,"ins")),HASH(FETCH(h1,"ins")));

  /* outs = (outs1 | outs2) & live_after2 */
  union_hv(HASH(FETCH(h2,"outs")),HASH(FETCH(h1,"outs")));
  special_intersect_hv(HASH(FETCH(h2,"outs")),HASH(FETCH(h2,"live_after")));

  /* kills = kills1 | kills2 */
  union_hv(HASH(FETCH(h2,"kills")),HASH(FETCH(h1,"kills")));
```

```
  concat_statements(FETCH(h1,"code"),FETCH(h2,"code"));
  STORE(h2,"code",FETCH(h1,"code"));

  spe-
cial_intersect_hv(HASH(FETCH(h2,"live_before")),HASH(FETCH(h2,"live_after")));
  special_union_hv(HASH(FETCH(h2,"live_before")),HASH(FETCH(h2,"ins")));
  subtract_hv(HASH(FETCH(h2,"live_before")),HASH(FETCH(h1,"kills")));
  special_union_hv(HASH(FETCH(h2,"live_before")),HASH(FETCH(h1,"ins")));

  return g2;
}

/* this is a special version of intersect_hv that un-shadows and
   un-fractions the elements of the target hash.  Gak. */

special_intersect_hv(HV* target, HV* source) {
  HE* he;
  hv_iterinit(target);
  while(he = hv_iternext(target)) {
    if (!EXISTS(source,short_term_var_name(var_original(hv_iterval(target,he))))) {
      DELETE(target,hv_iterkey(he,&na));
    }
  }
  return;
}


/* requires that statement have live_variables defined */
SV* convert_statement_to_group(SV* s) {
  SV* g;
  HV *hg, *hs;
  SV* foo;
  SV* result;
  AV* code;
  g = new_group();
  hg = HASH(g);
  hs = HASH(s);
  code = (AV*)SvRV(FETCH(hg,"code"));
  av_store(code,0,s);
```

```
  union_hv(HASH(FETCH(hg,"live_after")),HASH(FETCH(hs,"live_vars")));
  union_hv(HASH(FETCH(hg,"live_before")),HASH(FETCH(hs,"live_vars")));

  if (!is_("comment",s)) {
    varlist_add(FETCH(hg,"kills"), FETCH(hs,"result"));
    varlist_delete(FETCH(hg,"live_before"), FETCH(hs,"result"));

    result = FETCH(hs,"result");
    if (foo = FETCH_Q(HASH(result),"fraction")) { result = foo; }
    if (foo = FETCH_Q(HASH(result),"shadow")) { result = foo; }

    if (varlist_fetch(FETCH(hg,"live_after"),result)) {
      varlist_add(FETCH(hg,"outs"), FETCH(hs,"result"));
    }

    if (is_("constant",FETCH(hs,"condition"))) {
      varlist_delete(FETCH(hg,"live_before"), FETCH(hs,"result"));
      varlist_delete(FETCH(hg,"ins"), FETCH(hs,"result"));
    } else {
      varlist_add(FETCH(hg,"live_before"), var_original(FETCH(hs,"condition")));
      varlist_add(FETCH(hg,"live_before"), var_original(FETCH(hs,"result")));
      varlist_add(FETCH(hg,"ins"), FETCH(hs,"condition"));
      varlist_add(FETCH(hg,"ins"), FETCH(hs,"result"));
    }
    if (foo = FETCH_Q(hs,"argument")) {
      varlist_add(FETCH(hg,"live_before"), var_original(foo));
      varlist_add(FETCH(hg,"ins"), foo);
    }
    if (foo = FETCH_Q(hs,"argument1")) {
      varlist_add(FETCH(hg,"live_before"), var_original(foo));
      varlist_add(FETCH(hg,"ins"), foo);
    }
    if (foo = FETCH_Q(hs,"argument2")) {
      varlist_add(FETCH(hg,"live_before"), var_original(foo));
      varlist_add(FETCH(hg,"ins"), foo);
    }
  }
  return g;
}
```

```
int varlist_countbits(SV* l) {
  HE *he;
  HV* h;
  int bits = 0;
  h = HASH(l);

  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    bits += size_bits(hv_iterval(h,he));
  }

  return bits;
}

SV* varlist_fetch(SV* l, SV* v){
  HV* h;
  if (is_("variable",v)) {
    h = HASH(l);
    return FETCH_Q(h,short_term_var_name(v));
  }
  return (SV*)NULL;
}

varlist_add(SV* l, SV* v){
  HV* h;
  if (is_("variable",v)) {
    h = HASH(l);
    STORE(h,short_term_var_name(v),v);
  }
}

varlist_delete(SV* l, SV* v) {
  HV* h;
  if (is_("variable",v)) {
    h = HASH(l);
    DELETE(h,short_term_var_name(v));
  }
}
```

A.9 group_conflicts.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"


group_mark_conflicts(HV* rule) {
  AV* a;
  int i;
  HE* he;
  HV* live_vars;
  live_vars = newHV();

  a = (AV*)SvRV(FETCH(rule,"grouping"));

  for (i=0; i<= av_len(a); i++) {
    constrain_group(*av_fetch(a,i,0));
  }

}

constrain_group(SV* s) {
  SV* cond;
  SV* result;
  SV* arg;
  HV* live_vars;
  HV* h;
  h = HASH(s);

  live_vars = HASH(FETCH(h,"live_after"));

  varlists_add_edges(HASH(FETCH(h,"outs")),1,
          HASH(FETCH(h,"live_after")),0,
          "live_edges");

  varlists_add_edges(HASH(FETCH(h,"outs")),0,
          HASH(FETCH(h,"ins")),0,
          "src_dest_edges");
```

```
  varlist_add_edges(HASH(FETCH(h,"ins")),
          "use_edges");

  varlist_add_edges(HASH(FETCH(h,"outs")),
          "use_edges");

  varlist_add_edges(HASH(FETCH(h,"live_after")),
          "live_edges");
}

varlist_add_edges(HV* h, char *type) {
  AV* l;
  int i;
  HE* he;
  l = newAV();
  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    for (i=0; i<= av_len(l); i++) {
      add_edge_by_type(*av_fetch(l,i,0),hv_iterval(h,he),type);
    }
    av_push(l,hv_iterval(h,he));
  }
  av_undef(l);
}

varlists_add_edges(HV* h1, int o1, HV* h2, int o2, char *type) {
  HE* he1;
  HE* he2;
  SV *v1, *v2;

  if (h1 == h2) {
    fprintf(stderr,"oopsie, something bad happened. check the source code.\n");
  }
  hv_iterinit(h1);
  while (he1 = hv_iternext(h1)) {
    hv_iterinit(h2);
    while (he2 = hv_iternext(h2)) {
      if (hv_iterval(h1,he1) != hv_iterval(h2,he2)) {
    v1 = o1?var_original(hv_iterval(h1,he1)):hv_iterval(h1,he1);
```

```
    v2 = o2?var_original(hv_iterval(h2,he2)):hv_iterval(h2,he2);
    add_edge_by_type(v1,v2,type);
      }
    }
  }
}
```

A.10 kludge.h

```
#include <sys/types.h>

#define A_REALLY_LARGE_NUMBER 9999999
#define PAGE_BREAK 32 /* 12 */
#define MALLOC malloc

extern SV**      fetch_result;
extern AV*      conditions;
extern HV*      var_namespace;
extern int      num_layers;

extern int      print_intermediate;
extern int      print_conflicts;
extern int      print_after_copy;
extern int     print_variables;
extern int     do_copy;
extern int      do_anneal;
extern int      do_second_anneal;
extern int     print_after_anneal;
extern int      print_final;

#define FETCH(hash,key)    ((fetch_result=hv_fetch(hash,key,strlen(key),0))? \
              *fetch_result: \
              (SV*)dterror("failed to find " #key " in " #hash \
                  " at " __FILE__ ":%d", __LINE__))

#define FETCH_Q(hash,key) ((fetch_result=hv_fetch(hash,key,strlen(key),0))? \
              ((SV*)*fetch_result):((SV*)NULL))
```

```
#ifdef RISKY
#define FETCH_OR_NEW_HASH(hash,key) \
    ((fetch_result=hv_fetch(hash,key,strlen(key),0))? \
     ((SV*)*fetch_result): \
     ((SV*)*hv_store(hash,key,strlen(key),newRV((SV*)newHV()),0)))
#else
#define FETCH_OR_NEW_HASH(hhh,key) fetch_or_new_hash(hhh,key)
#endif


#define EXISTS(hash,key) (hv_exists(hash,key,strlen(key)))

#ifdef 0
#define STORE(hash,key,s) do { SvREFCNT_inc(hv_fetch(hash,key,strlen(key),0)); \
                       hv_store(hash,key,strlen(key),s,0); } while(0)
#else
#define STORE(hash,key,s) do { SV* vvv = s; SvREFCNT_inc(vvv);\
                       hv_store(hash,key,strlen(key),vvv,0); } while(0)
#endif

#define DELETE(hash,key) do { hv_delete(hash,key,strlen(key),0); } while(0)

#define HASH(ref)    (HV*)SvRV(ref)
#define REF(t)       newRV((SV*)t)

#include "proto.h"
```

A.11 kolor.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"

static top_subcell = 0;

kolor(HV* varspace) {
  AV* stack;
```

```
  SV* current;
  int done = 0;

  stack = newAV();
  while (!done) {
    char* best_name;
    SV* best_node;
    int best_score = A_REALLY_LARGE_NUMBER;
    HE* he;

    done = 1;
    hv_iterinit(varspace);        /* find the easiest var to color */
    while (he = hv_iternext(varspace)) {
      if (best_score > var_score(hv_iterval(varspace,he))) {
    best_score = var_score(hv_iterval(varspace,he));
    best_node = hv_iterval(varspace,he);
    best_name = hv_iterkey(he,&na);
    done = 0;
      }
    }
    if (!done) {
      turn_off_variable(best_node);
      av_push(stack,best_node);
      DELETE(varspace,best_name);
    }
  }

  while (av_len(stack) >= 0) {
    current = av_pop(stack);
    if (!pick_kolor(current,0)) {
    } else {
/*      fprintf(stdout,"\ncolored "); */
/*      fprint_variable(stdout, current); */
    }
    turn_on_variable(current);
  }
}

/* this could be more sophisticated */
int var_score(SV* var) {
```

```
  return SvIV(FETCH(HASH(var),"edge_count"));
}

/* picks a location for var
   returns 1 if it found a good place
   returns 0 otherwise
   becomes desperate when it can't find a good place
   if desperate, looks for a bad place */

int pick_kolor(SV* var, int desperate) {
  int size;
  int guess_layer = 0;
  int guess_subcell = 0;
  int done = 0;
  int prefer_guess = 0;
  HE* he;
  HV* edges;
  u_int guess_mask = 0;
  u_int edge_mask = 0;
  size = size_bits(var);

  if (has_("field",var)) {
    if (has_("preferred_layer",var_original(var))) {
      guess_layer = SvIV(FETCH(HASH(var_original(var)),"preferred_layer"));
      guess_subcell = SvIV(FETCH(HASH(var_original(var)),"preferred_subcell"));
      prefer_guess = 1;
    }
  }

  while (1) {
    /* test our subcell guess */

    edge_mask = 0;

    if (!desperate) {
      edges = HASH(FETCH_OR_NEW_HASH(HASH(var),"use_edges"));
      hv_iterinit(edges);
      while (he = hv_iternext(edges)) {
    edge_mask |= mask_variable(hv_iterval(edges,he));
      }
```

```
}

edges = HASH(FETCH_OR_NEW_HASH(HASH(var),"live_edges"));
hv_iterinit(edges);
while (he = hv_iternext(edges)) {
  if (guess_subcell == subcell(hv_iterval(edges,he))) {
edge_mask |= mask_variable(hv_iterval(edges,he));
  }
}


if (!desperate) {
  edges = HASH(FETCH_OR_NEW_HASH(HASH(var),"src_dest_edges"));
  hv_iterinit(edges);
  while (he = hv_iternext(edges)) {
if (guess_subcell ≠ subcell(hv_iterval(edges,he)))
  edge_mask |= mask_variable(hv_iterval(edges,he));
  }
}

done = 0;
while (!done) {

  /* test each possible layer position */
  guess_mask = ((1≪size)−1)≪guess_layer;

  if (!(guess_mask & edge_mask)) {
/* SUCCEED */
STORE(HASH(var),"layer",newSViv(guess_layer));
STORE(HASH(var),"subcell",newSViv(guess_subcell));
top_subcell = MAX(top_subcell, guess_subcell);
if (has_("field",var)) {
  if ((!has_("preferred_layer",var_original(var))) ||
      (!has_("shadow",var))) {
    STORE(HASH(var_original(var)),"preferred_layer",newSViv(guess_layer));
    STORE(HASH(var_original(var)),"preferred_subcell",newSViv(guess_subcell));
  }
}
return desperate?0:1;
  }
```

```
  /* failed for that layer position */

  /* if we didn't get the preferred spot,
 go back to the begining and start over */
  if (prefer_guess) {
guess_layer = 0;
guess_subcell = -1;
prefer_guess = 0;
done = 1;
  } else {
guess_layer++;
  }
  if ((guess_layer + size) > num_layers) {
guess_layer = 0;
done = 1;
  }
}

/* failed for that subcell */
if (guess_subcell > top_subcell) {
  /* failed for a completely new subcell */
  /* time to get desparate */
  if (!has_("spilled",var)) {
STORE(HASH(var),"spilled",newSViv(1));
fprintf(stdout,
    "Warning: Couldn't find a good location for %s\n",
    short_term_var_name(var));
return pick_kolor(var,1);
  } else {
fprintf(stdout,
    "Failure: Couldn't even find a bad location for %s !\n",
    short_term_var_name(var));
return 0;
  }
}

/* guess a new subcell */
guess_subcell++;
edge_mask = 0;
```

```
  }
}

/* return a mask of the bits that var occupies in subcell */
u_int mask_variable(SV* var) {
  u_int m;
  int l;
  l = layer(var);
  if (l == -1) {
    return 0;
  } else {
    m = (1 << size_bits(var)) - 1;
    m <<= l;
  }
  return (m & ((1<<num_layers)-1));
}
```

A.12 lex.flex

```
/* lex.flex                                        */
/* Copyright (C) 1995, 1996 Daniel Risacher        */
/* All rights reserved.                            */
/*                                                 */
/* This software may be freely copied, modified, and redistributed,   */
/* provided that this copyright notice is preserved on all copies.    */
/*                                                 */
/* You may not distribute this software, in whole or in part, as part of  */
/* any commercial product without the express consent of the authors.     */
/*                                                 */
/* There is no warranty or other guarantee of fitness of this software    */
/* for any purpose.  It is provided solely "as is".  */
/*                                                 */
/* $Id: lex.flex,v 1.4 1996/05/02 01:16:18 magnus Exp magnus $ */

%{
#ifdef I_STRING
#include <string.h>
#else
```

```
#include <strings.h>
#endif
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"
#include "gc.h"
extern char *strsave();
extern int dtlineno;
%}
alpha    [a-zA-Z]
special    [\_-]
digit    [0-9]
exp    [Ee][-+]?{digit}+
string    {alpha}({alpha}|{digit}|{special})*
filename "/"?"/"?(("."|".."|{string})"/")*{string}
%p 9400
%e 1500
%n 600
%%
[ \t]        ;
"\n"        dtlineno++;
^#        handlehash();
"/*"        skipcomments();
if            return tIF;
then            return tTHEN;
else            return tELSE;
"&&"        return tAND;
"||"        return tOR;
"=="            return tEQ;
"!="            return tNE;
"<="            return tLE;
">="            return tGE;
"<<"            return tLS;
">>"            return tRS;
"bits"        return tBITS;
".."        return tDD;
{digit}+        {dtlval.i = atoi(dttext); return tINT;}
{string}        {dtlval.c = strsave(dttext); return tSTRING;}
.        return dttext[0];
```

```
%%
dtwrap() {return 1;}
/*
 * Skip over comments.
 */
skipcomments()
{
  char c;

  while (1) {
    while (input() ≠ '*')
      ;
    if ((c = input()) == '/')
      return;
    unput(c);
  }
}

char *
strsave(s)
char *s;
{
        char *tmp;

        if (s == (char *)NULL)
                return (char *)NULL;

        tmp = (char *)MALLOC((unsigned)strlen(s) + 1);
        (void)strcpy(tmp, s);
        return tmp;
}


/*
 * Deal with ccp-produced lines of the form:
 * # n "filename"
 * and
 * # n
 * Where filename is the name of the file being processed, and n is
```

```
 * the current line number in that file.
 */
handlehash()
{
    char buf[BUFSIZ];
    int i;
    extern int dtlineno;
    extern char *dtfilename;

    /*
     * Read the entire line into buf.
     */
    for (i = 0; (buf[i] = input()) ≠ '\n'; i++)
            ;
    unput(buf[i]);        /* To make sure consecutive # lines work. */
    buf[i] = (char)NULL;    /* Replace newline with NULL. */

    /*
     * Complain if the line was not of the form #n "filename"
     */
    if ((i = sscanf(buf,"%d \"%[^\"]s\"", &dtlineno, dtfilename)) == 0 &&
        (i = sscanf(buf," line %d \"%[^\"]s\"",&dtlineno,dtfilename))==0) {
        fprintf(stderr, "Unknown '#' control (%s).\n",buf);
        exit(1);
    }
    if (i == 1 && (−1 ≠ (char *)NULL)) {
        /*
         * Filename was given as ""
         */
        (void)strcpy(dtfilename, "stdin");
    }
    dtlineno−−;  /* The newline we unput will increment dtlineno */
}
```

A.13 math.c

```
#include <EXTERN.h>
#include <perl.h>
```

```
#include "kludge.h"
#include "y.tab.h"

int perform_unary_op(int op,int val)
{
  switch (op)
    {
    case '!': return !val;
    case '+': return val;
    case '-': return −val;
    case '~': return ∼val;
    }
  return val;
}

int perform_binary_op(int a, int op, int b)
{
  switch (op)
    {
    case tOR: return (a || b);
    case tAND: return (a && b);
    case '|': return (a | b);
    case '^': return (a ˆ b);
    case tNE: return (a ≠ b);
    case tEQ: return (a == b);
    case '<': return (a < b);
    case '>': return (a > b);
    case tLE: return (a ≤ b);
    case tGE: return (a ≥ b);
    case tLS: return (a ≪ b);
    case tRS: return (a ≫ b);
    case '+': return (a + b);
    case '-': return (a − b);
    case '*': return (a ∗ b);
    case '/': return (a / b);
    case '%': return (a % b);
    }
}

int max_binary_op(int al, int ah, int op, int bl, int bh) {
```

```
  switch (op)
    {
    case tOR: return (ah || bh || al || bl);
    case tAND: return ((ah || al) && (bh || bl));;
    case '|': return (ah | al | bh | bl);  /* this is wrong */
    case '^': return (ah | al | bh | bl);  /* but I'm tired */
    case tNE: return !((al==ah)&&(ah==bh)&&(bh==bl));
    case tEQ: return (((al<=bh)&&(ah>=bl))||((bl<=ah)&&(bh>=al)));
    case '<': return (al < bh);
    case '>': return (ah > bl);
    case tLE: return (al <= bh);
    case tGE: return (ah >= bl);
    case tLS: return (ah << bh);
    case tRS: return (ah >> bl);
    case '+': return (ah + bh);
    case '-': return (ah - bl);
    case '*': return (MAX(ah*bh,al*bl));
    case '/': return (MAX(ah/bl,al/bh));
    case '%': return (bh);
    }
}

int min_binary_op(int al, int ah, int op, int bl, int bh) {
  switch (op)
    {
    case tOR: return (0);
    case tAND: return (0);
    case '|': return (0);
    case '^': return (0);
    case tNE: return ((ah<bl)||(bh<al));
    case tEQ: return ((al==ah)&&(ah==bh)&&(bh==bl));
    case '<': return (ah < bl);
    case '>': return (al > bh);
    case tLE: return (ah <= bl);
    case tGE: return (al >= bh);
    case tLS: return (al << bl);
    case tRS: return (al >> bh);
    case '+': return (al + bl);
    case '-': return (al - bh);
    case '*': return (MIN(al*bl,MIN(ah*bl,al*bh)));
```

```
    case '/': return (MIN(ah/bl,al/bh));
    case '%': return (0);
    }
}

int max_unary_op(int op,int vl, int vh)
{
  switch (op)
    {
    case '!': return (vl<=0&&vh>=0);
    case '+': return vh;
    case '-': return -vl;
    case '~': return MAX(vh,1-vl);
    }
  return vh;
}

int min_unary_op(int op,int vl, int vh)
{
  switch (op)
    {
    case '!': return !(vl<0||vh>0);
    case '+': return vl;
    case '-': return -vh;
    case '~': return 0;
    }
  return vl;
}
```

A.14 objects.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"

HV* new_cell() {
  HV* res;
```

```
  res = newHV();
  return res;
}

HV* new_field(char *name, int low, int high) {
  HV* res;
  res = newHV();
  STORE(res,"type",newSVpv("variable",0));
  STORE(res,"field",newSVpv("true",0));
  STORE(res,"location",newSVpv("undecided",0));
  STORE(res,"name",newSVpv(name,0));
  STORE(res,"low",newSViv(low));
  STORE(res,"high",newSViv(high));
  STORE(res,"edge_count",newSViv(0));
  return res;
}

HV* new_fraction(SV* s, int lo, int hi) {
  HV* res;

  res = (HV*)copy_hv(HASH(s));
  STORE(res,"fraction",s);
  STORE(res,"lo",newSViv(lo));
  STORE(res,"hi",newSViv(hi));
  return res;
}

HV* new_shadow(HV* h) {
  HV* res;
  res = (HV*)copy_hv(h);
  STORE(res,"shadow",REF(h));
  STORE(res,"edge_count",newSViv(0));
  return res;
}

SV* var_original(SV* v) {
  SV* foo;
  if (foo = FETCH_Q(HASH(v),"fraction")) { v = foo; }
  if (foo = FETCH_Q(HASH(v),"shadow")) { v = foo; }
  return v;
```

```
}

HV* new_expr(HV* result) {
  HV* res;
  res = newHV();
  STORE(res,"type",newSVpv("expr",0));
  STORE(res,"code",REF(newAV()));
  STORE(res,"result",REF(result));
  return res;
}

HV* new_constant(int a) {
  HV* res;
  res = newHV();
  STORE(res,"type",newSVpv("constant",0));
  STORE(res,"value",newSViv(a));
  STORE(res,"code",REF(newAV()));
  return (res);
}

HV* new_variable(char *a) {
  HV* res;
  res = newHV();
  STORE(res,"type",newSVpv("variable",0));
  STORE(res,"name",newSVpv(a,strlen(a)));
  STORE(res,"location",newSVpv("undecided",0));
  STORE(res,"edge_count",newSViv(0));
  return res;
}

HV* new_temp() {
  static char* tname = NULL;
  static int tnum = 1;
  HV* res;
  if (!tname) tname = (char*) MALLOC(50);
  sprintf(tname,"t%d",tnum++);
  res = new_variable(tname);
  STORE(res,"temp",newSVpv("true",0));
  STORE(var_namespace,tname,REF(res));
```

```
  return res;
}

/* takes a ref to a hash and returns if the type is eq to thing */
int is_(char* thing,SV* s) {
  HV* h;
  h = HASH(s);
  return (eq(thing,SvPV(FETCH(h,"type"),na)));
}

int has_(char* thing,SV* s) {
  HV* h;
  h = HASH(s);
  return (NULL != FETCH_Q(h,thing));
}

char* name_(SV* s) {
  HV* h;
  h = HASH(s);
  return SvPV(FETCH(h,"name"),na);
}

int bitsize(int n) {
  int m,i;

  i=m=0;
  if (0>n) {
    i++;
    n=-n;
  }

  while (m<n) {
    m = 1+(m<<1);
    i++;
  }
  return i;
}

/* might need to rework this later */
int size_bits(SV* s) {
```

```
  HV* h;
  int low,high;
  h = HASH(s);
  if (is_("variable",s)) {
    if (has_("fraction",s)) {
      return (1 + SvIV(FETCH(h,"hi")) - SvIV(FETCH(h,"lo")));
    } else {
      low = SvIV(FETCH(h,"low"));
      high = SvIV(FETCH(h,"high"));
      return bitsize(high-low);
    }
  } else if (is_("constant",s)) {
    return 0;
  } else {
    printf("waka waka\n");
    print_sv(s);
  }
}


int layer(SV* s) {
  HV* h;
  SV* foo;
  h = HASH(s);
  foo = FETCH_Q(h,"layer");

  if (foo) return SvIV(foo);
  else return -1;
}

int subcell(SV* s) {
  HV* h;
  SV* foo;
  h = HASH(s);
  foo = FETCH_Q(h,"subcell");

  if (foo) return SvIV(foo);
  else return -1;
}
```

A.15 perlutil.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"

#define MAX_INDENT 20
int indent = 0;


int perl_eval(char *string) {
  char *argv[2];
  argv[0] = string;
  argv[1] = NULL;
  perl_call_argv("_eval_", 0, argv);
}

print_hv_keys(HV* h) {
  HE* he;
  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    printf("%s ",he->hent_key);
/*      printf("%s %d %d + ",he->hent_key,he->hent_hash,he->hent_klen); */
  }
}

HV* copy_hv(HV *h) {
  HV* c;
  HE* he;
  c = newHV();

  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    STORE(c,hv_iterkey(he,&na),hv_iterval(h,he));
  }
  return c;
}

union_hv(HV* target, HV* source) {
  HE* he;
```

```
  hv_iterinit(source);
  while(he = hv_iternext(source)) {
    STORE(target,hv_iterkey(he,&na),hv_iterval(source,he));
  }
  return;
}

subtract_hv(HV* target, HV* source) {
  HE* he;

  hv_iterinit(source);
  while(he = hv_iternext(source)) {
    DELETE(target,hv_iterkey(he,&na));
  }
  return;
}

intersect_hv(HV* target, HV* source) {
  HE* he;

  hv_iterinit(target);
  while(he = hv_iternext(target)) {
    if (!EXISTS(source,hv_iterkey(he,&na))) {
      DELETE(target,hv_iterkey(he,&na));
    }
  }
  return;
}



int eq(char* a, char* b) {
  return !strcmp(a,b);
}

int print_indent() {
  int i;
  printf("\n");
```

```
  for (i=0;i<indent; i++) {
    printf("  ");
  }
  return indent;
}

print_sv(SV* s) {
  if (!s) {
    printf("(null)");
  } else if (!SvOK(s)) {
    printf("(undef 0x%x)",s);
  } else if (SvIOKp(s)) {
    printf("(IV %d) ",SvIV(s));
  } else if (SvNOKp(s)) {
    printf("(NV %f) ",SvNV(s));
  } else if (SvPOKp(s)) {
    printf("(PV \"%s\") ",
       SvPV(s,na));
  } else if (SvROK(s)) {
    printf("(ref ");
    switch (SvTYPE(SvRV(s))) {
    case SVt_IV:
    case SVt_NV:
    case SVt_PV:
      return print_sv(SvRV(s));
    case SVt_PVAV:
      return print_av((AV*)SvRV(s));
    case SVt_PVHV:
      return print_hv(HASH(s));
    default:
      printf("other");
    }
    printf(")");
  } else {
    printf("(unusual SV)");
  }
  fflush(stdout);
}
```

```
print_hv(HV* h) {
  HE* he;
  hv_iterinit(h);
  printf("(hash ");
  indent++;
  if (indent < MAX_INDENT) {
    while(he = hv_iternext(h)) {
      print_indent();
      printf("\"%s\" => ",hv_iterkey(he,&na));
      print_sv(hv_iterval(h, he));
    }
  } else {
    printf("....");
  }
  indent--;
  printf(")");
  fflush(stdout);
}

print_av(AV* a) {
  int i;
  SV** s;
  printf("(array ");
  indent++;
  if (indent < MAX_INDENT) {
    for (i=0; i<=av_len(a); i++) {
      s = av_fetch(a,i,0);
      if (s) {
    print_indent();
    print_sv(*s);
      } else {
    print_indent();
    printf("(null entry)");
      }
    }
  } else {
    printf("....");
  }

  indent--;
```

```
  printf(")");
  fflush(stdout);
}


array_insert(AV* a_list, int ns, SV* s) {
 SV** temp;
 SV* temp2 = NULL;
 int original_length;
 original_length = 1 + av_len(a_list);

 for (; ns <= original_length; ns++) {
   temp = av_fetch(a_list, ns, 0);
   temp2 = ((temp)?(*temp):NULL);
   SvREFCNT_inc(temp2);
   av_store(a_list, ns, s);
   s = temp2;
 }
}


array_delete(AV* a_list, int ns) {
 SV** temp;
 SV* temp2 = NULL;
 int original_length;
 original_length = av_len(a_list);

 for (; ns < original_length; ns++) {
   temp = av_fetch(a_list, ns+1, 0);
   temp2 = ((temp)?(*temp):(&sv_undef));
   SvREFCNT_inc(temp2);
   av_store(a_list, ns, temp2);
 }
 av_pop(a_list);
}

SV* fetch_or_new_hash(HV* hash, char* key) {
  SV* ret;
  ret = FETCH_Q(hash,key);
  if (!ret) {
```

```
    ret = REF(newHV());
    STORE(hash,key,ret);
  }
  return ret;

}


int empty_hash(HV* h) {
  HE* he;
  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    return 0;
  }
  return 1;
}
```

A.16 print.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"

fprint_grouplist(FILE* f, SV* gl) {
  SV** s;
  AV* l;
  int i;

  l = (AV*)SvRV(gl);
  for (i=0; i<=av_len(l); i++) {
    s = av_fetch(l,i,0);
    if (s) {
      fprint_group(f, *s);
    }
  }
}
```

```
fprint_group(FILE *f, SV* g) {
  fprintf(f,"------------------------group-\n");
  fprintf(f,"ins: ");
  fprint_varspace(f,HASH(FETCH(HASH(g),"ins")));
  fprintf(f,"\n");
  fprintf(f,"live_before: ");
  fprint_varspace(f,HASH(FETCH(HASH(g),"live_before")));
  fprintf(f,"\n");
  fprint_block(f,HASH(g));
  fprintf(f,"outs: ");
  fprint_varspace(f,HASH(FETCH(HASH(g),"outs")));
  fprintf(f,"\n");
  fprintf(f,"live_after: ");
  fprint_varspace(f,HASH(FETCH(HASH(g),"live_after")));
  fprintf(f,"\n");
}

fprint_block(FILE* f, HV* block) {
  SV** s;
  AV* l;
  int i;

  l = (AV*)SvRV(FETCH(block,"code"));
  for (i=0; i<=av_len(l); i++) {
    s = av_fetch(l,i,0);
    if (s) {
      fprint_statement(f, *s);
    }
  }
}

fprint_statement(FILE *f, SV* s) {
  HV* h;
  SV* t;
  SV* ct;
  h = HASH(s);
  t = FETCH(h,"type");
  if (eq("comment",SvPV(t,na))) {
    fprintf(f," \tcomment: \t");
    ct = FETCH_Q(h,"comment");
```

```
  if (ct) {
    fprintf(f, "%s\n",SvPV(ct,na));
  } else {
    fprintf(f, "comment text lost\n");
  }
  return;
} else if (eq("binary_assign",SvPV(t,na))) {
  if (ct = FETCH_Q(h,"input_bits")) fprintf(f,"bits in: %d ",SvIV(ct));
  fprintf(f, " \tif: ");
  if (EXISTS(h,"condition")) {
    fprint_val(f, FETCH(h,"condition"));
    fprintf(f, "\t ");
  }
  fprint_val(f, FETCH(h,"result"));
  fprintf(f, " = ");
  fprint_val(f, FETCH(h,"argument1"));
  fprint_op(f,SvIV(FETCH(h,"operator")),h);
  fprint_val(f, FETCH(h,"argument2"));
} else if (eq("unary_assign",SvPV(t,na))) {
  if (ct = FETCH_Q(h,"input_bits")) fprintf(f,"bits in: %d ",SvIV(ct));
  fprintf(f, " \tif: ");
  if (EXISTS(h,"condition")) {
    fprint_val(f, FETCH(h,"condition"));
    fprintf(f, "\t ");
  }
  fprint_val(f, FETCH(h,"result"));
  fprintf(f, " = ");
  fprint_op(f,SvIV(FETCH(h,"operator")),h);
  fprint_val(f, FETCH(h,"argument1"));
} else if (eq("assign",SvPV(t,na))) {
  if (ct = FETCH_Q(h,"input_bits")) fprintf(f,"bits in: %d ",SvIV(ct));
  fprintf(f, " \tif: ");
  if (EXISTS(h,"condition")) {
    fprint_val(f, FETCH(h,"condition"));
    fprintf(f, "\t ");
  }
  fprint_val(f, FETCH(h,"result"));
  fprintf(f, " = ");
  fprint_val(f, FETCH(h,"argument"));
} else {
```

```
    return old_fprint_statement(f,s);
  }

  if ((ct=FETCH_Q(h,"live_bits"))) {
    fprintf(f,"\t live_bits: %d", SvIV(ct));
  }

  fprintf(f,"\n");
}

old_fprint_statement(FILE *f, SV* s) {
  HV *h;
  HE* he;
  h = HASH(s);
  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    fprintf(f, "%s ",hv_iterkey(he,&na));
    if (SvPOK(hv_iterval(h,he))) {
      fprintf(f,"%s\n",SvPV(hv_iterval(h,he),na));
    } else if (SvIOK(hv_iterval(h,he))) {
      fprintf(f,"0x%x\n",SvIV(hv_iterval(h,he)));
    } else {
      fprintf(f,"0x%x\n",hv_iterval(h,he));
    }
  }
}

fprint_val(FILE *f, SV *s) {
  HV* h;
  SV* sl;

  if (!SvROK(s)) {
    print_sv(s);
    return;
  }
  h = HASH(s);
  if (eq("variable",SvPV(FETCH(h,"type"),na))) {
    fprint_variable(f,s);
  }
  if (eq("constant",SvPV(FETCH(h,"type"),na))) {
```

```
    fprintf(f,"%d",SvIV(FETCH(h,"value")));
  }
}

sprint_val(char *f, SV *s) {
  HV* h;
  SV* sl;

  if (!SvROK(s)) {
    print_sv(s);
    return;
  }
  h = HASH(s);
  if (eq("variable",SvPV(FETCH(h,"type"),na))) {
    sprintf(f,"%s",SvPV(FETCH(h,"name"),na));
    if (sl = FETCH_Q(h,"offset")) {
      int i;
      AV* l;
      l = (AV*)SvRV(sl);
      sprintf(f,"[");
      for (i=0;i<av_len(l); i++) {
    sprint_val(f,*av_fetch(l,i,0));
    sprintf(f,",");
      }
      sprint_val(f,*av_fetch(l,i,0));
      sprintf(f,"]");
    }
  }
  if (eq("constant",SvPV(FETCH(h,"type"),na))) {
    sprintf(f,"%d",SvIV(FETCH(h,"value")));
  }
}


fprint_op(FILE* f, int op, HV* h)
{
  switch (op)
    {
    case tOR: return fprintf(f," || ");
    case tAND: return fprintf(f," && ");
```

```
    case '|': return fprintf(f," | ");
    case '^': return fprintf(f," ^ ");
    case tNE: return fprintf(f," != ");
    case tEQ: return fprintf(f," == ");
    case '<': return fprintf(f," < ");
    case '>': return fprintf(f," > ");
    case tLE: return fprintf(f," <= ");
    case tGE: return fprintf(f," >= ");
    case tLS: return fprintf(f," << ");
    case tRS: return fprintf(f," >> ");
    case '+': return fprintf(f," + ");
    case '-': return fprintf(f," - ");
    case '*': return fprintf(f," * ");
    case '/': return fprintf(f," / ");
    case '%': return fprintf(f," \% ");
    case '!': return fprintf(f," ! ");
    case '~': return fprintf(f," ~ ");
    }
  fprintf(f," (op %d) ",op);
  print_hv(h);
}

fprint_varspace(FILE* f,HV* h) {
  HE* he;
  SV* sv;
  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    fprint_variable(f,hv_iterval(h,he));
    fprintf(f," ");
  }
}

fprint_varspace_verbose(FILE* f,HV* h) {
  HE* he;
  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    fprint_variable_verbose(f,hv_iterval(h,he));
  }
}
```

```
fprint_variable(FILE* f,SV* v) {
  HV* h;
  SV* foo;
  h = HASH(v);
  fprintf(f,"%s",SvPV(FETCH(h,"name"),na));
  if (foo = FETCH_Q(h,"offset")) {
    int i;
    AV* l;
    l = (AV*)SvRV(foo);
    fprintf(f,"[");
    for (i=0;i<av_len(l); i++) {
      fprint_val(f,*av_fetch(l,i,0));
      fprintf(f,",");
    }
    fprint_val(f,*av_fetch(l,i,0));
    fprintf(f,"]");
  }
  if (foo = FETCH_Q(h,"fraction")) {
    fprintf(f,"<%d-%d>",SvIV(FETCH(h,"lo")),SvIV(FETCH(h,"hi")));
  }
}

fprint_variable_verbose(FILE* f,SV* v) {
  HV* h;
  SV* foo;
  h = HASH(v);

  fprint_variable(f,v);

  ((foo = FETCH_Q(h,"low"))?fprintf(f," \t%d .. ",SvIV(foo)):fprintf(f,"\t? .. "));
  ((foo = FETCH_Q(h,"high"))?fprintf(f,"%d\t",SvIV(foo)):fprintf(f,"?\t"));
  if (foo = FETCH_Q(h,"layer")) {
    fprintf(f,"%d:%d-",layer(v),layer(v)+size_bits(v)-1);
    (-1==subcell(v)?fprintf(f,"special\t"):fprintf(f,"%d\t",subcell(v)));
  }

  ((foo = FETCH_Q(h,"spilled"))?fprintf(f,"spilled"):fprintf(f,""));
  ((foo = FETCH_Q(h,"field"))?fprintf(f,"field\n"):fprintf(f,"\n"));

}
```

```
fprint_variable_conflicts(FILE *f, SV* v, char *constraint) {
  HV* h;
  SV* foo;
  h = HASH(v);
  if (foo = FETCH_Q(h,constraint)) {
    if (!empty_hash(HASH(foo))) {
      fprint_variable(f,v);
      fprintf(f,"\t %s with \t",constraint);
      fprint_varspace(f,(HASH(foo)));
      fprintf(f,"\n");
    }
  }
}

fprint_varspace_conflicts(FILE* f,HV* h, char *constraint) {
  HE* he;
  hv_iterinit(h);
  while(he = hv_iternext(h)) {
    fprint_variable_conflicts(f,hv_iterval(h,he),constraint);
  }
}

char *copy_var_name(SV* s) {
  return (char*)strsave(short_term_var_name(s));
}

char *short_term_var_name(SV* s) {
  static char* buf = NULL;
  char *dummy;
  HV* h;
  SV* sl;
  if (!buf) { buf = (char*)MALLOC(50); }
  h = HASH(s);

  sprintf(buf,"%s\0",SvPV(FETCH(h,"name"),na));
  if (sl = FETCH_Q(h,"offset")) {
    int i;
    AV* l;
    l = (AV*)SvRV(sl);
```

```
    dummy = strchr(buf,0);
    sprintf(dummy,"[");
    for (i=0;i<av_len(l); i++) {
      dummy = strchr(buf,0);
      sprint_val(dummy,*av_fetch(l,i,0));
      dummy = strchr(buf,0);
      sprintf(dummy,",");
    }
    dummy = strchr(buf,0);
    sprint_val(dummy,*av_fetch(l,i,0));
    dummy = strchr(buf,0);
    sprintf(dummy,"]");
  }
  return (char*)(buf);
}
```

A.17 proto.h

```
int main ( int , char **argv, char **env ) ;
int pre_process ( int , char **argv, FILE *,char * ) ;
HV* top ( HV* , HV* ) ;
void initialize ( ) ;
HV* add_field ( HV* , HV* ) ;
HV* ref_constant ( int ) ;
HV* ref_variable ( char * ) ;
HV* unary_op ( int , HV* ) ;
HV* binary_op ( HV* , int , HV* ) ;
HV* list_op ( HV* , HV* ) ;
SV* listize_expr ( HV* ) ;
HV* ghost_offset_ref ( HV* ) ;
HV* offset_ref ( HV* , HV* ) ;
HV* assign_op ( HV* , HV* ) ;
HV* if_stat ( HV* , HV* ) ;
```

```
HV* if_start ( HV * ) ;
HV* new_s_list ( ) ;
HV* add_statement ( HV* , HV* ) ;
HV* process_block ( HV* ) ;
int perl_eval ( char * ) ;
int print_hv_keys ( HV* ) ;
HV* copy_hv ( HV * ) ;
int union_hv ( HV* , HV* ) ;
int subtract_hv ( HV* , HV* ) ;
int intersect_hv ( HV* , HV* ) ;
int eq ( char* , char* ) ;
int print_indent ( ) ;
int print_sv ( SV* ) ;
int print_hv ( HV* ) ;
int print_av ( AV* ) ;
int array_insert ( AV* , int , SV* ) ;
int array_delete ( AV* , int ) ;
SV* fetch_or_new_hash ( HV* , char* ) ;
int empty_hash ( HV* ) ;
SV* append_statement ( SV* ,SV* ) ;
int add_condition ( HV* ) ;
int uncondition ( SV* ) ;
AV* concat_statements ( SV* , SV* ) ;
SV* comment_statement ( char* ) ;
SV* assign_statement ( SV* , SV* ) ;
SV* unary_assign_statement ( int , SV* ) ;
SV* binary_assign_statement ( SV* , int , SV* ) ;
int binary_add_limits ( HV* , int , SV* , SV* ) ;
int unary_add_limits ( HV* , int , SV* ) ;
int copy_limits ( HV* , SV* ) ;
HV* new_cell ( ) ;
HV* new_field ( char *, int , int ) ;
HV* new_fraction ( SV* , int , int ) ;
HV* new_shadow ( HV* ) ;
SV* var_original ( SV* ) ;
HV* new_expr ( HV* ) ;
HV* new_constant ( int ) ;
HV* new_variable ( char * ) ;
HV* new_temp ( ) ;
int is_ ( char* ,SV* ) ;
```

```
int  has_ ( char* ,SV*  ) ;
char*  name_ ( SV*  ) ;
int  bitsize ( int  ) ;
int  size_bits ( SV*  ) ;
int  layer ( SV*  ) ;
int  subcell ( SV*  ) ;
int fprint_grouplist ( FILE* , SV*  ) ;
int fprint_group ( FILE *, SV*  ) ;
int fprint_block ( FILE* , HV*  ) ;
int fprint_statement ( FILE *, SV*  ) ;
int old_fprint_statement ( FILE *, SV*  ) ;
int fprint_val ( FILE *, SV * ) ;
int sprint_val ( char *, SV * ) ;
int fprint_varspace ( FILE* ,HV*  ) ;
int fprint_varspace_verbose ( FILE* ,HV*  ) ;
int fprint_variable ( FILE* ,SV*  ) ;
int fprint_variable_verbose ( FILE* ,SV*  ) ;
int fprint_variable_conflicts ( FILE *, SV* , char * ) ;
int fprint_varspace_conflicts ( FILE* ,HV* , char * ) ;
char  *copy_var_name ( SV*  ) ;
char  *short_term_var_name ( SV*  ) ;
int  max_binary_op ( int , int , int , int , int  ) ;
int  min_binary_op ( int , int , int , int , int  ) ;
int live_var_anal ( HV* , HV*  ) ;
int distinguish_statement ( SV*  ) ;
int analyze_statement ( SV* , HV*  ) ;
int mark_conflicts ( HV*  ) ;
int constrain_statement ( SV*  ) ;
int add_live_edges ( SV* , HV*  ) ;
int  check_sizes ( HV*  ) ;
int  size_check_statement ( SV* , int , AV*  ) ;
int  kill_arguments ( SV* , HV* , int , AV*  ) ;
int add_edge_by_type ( SV* , SV* , char * ) ;
int remove_edge ( SV* , SV* , char * ) ;
int turn_off_variable ( SV*  ) ;
int turn_off_variable_by_type ( SV* , char * ) ;
int turn_on_variable ( SV*  ) ;
int turn_on_variable_by_type ( SV* , char * ) ;
int kolor ( HV*  ) ;
int  var_score ( SV*  ) ;
```

```
int  pick_kolor ( SV* , int  ) ;
u_int  mask_variable ( SV*  ) ;
SV*  new_group (  ) ;
int  speculate_concat ( SV* , SV*  ) ;
SV*  concat_groups ( SV* , SV*  ) ;
int special_intersect_hv ( HV* , HV*  ) ;
SV*  convert_statement_to_group ( SV*  ) ;
int  varlist_countbits ( SV*  ) ;
SV*  varlist_fetch ( SV* , SV*  ) ;
int varlist_add ( SV* , SV*  ) ;
int varlist_delete ( SV* , SV*  ) ;
SV*  convert_code_to_groups ( SV*  ) ;
int simple_anneal_groups ( SV * ) ;
int better_anneal_groups ( SV * ) ;
int special_union_hv ( HV* , HV*  ) ;
int group_mark_conflicts ( HV*  ) ;
int constrain_group ( SV*  ) ;
int varlist_add_edges ( HV* , char * ) ;
int varlists_add_edges ( HV* , int , HV* , int , char * ) ;
int simple_copy_operands ( SV*  ) ;
int  group_do_copies ( AV* ,int  ) ;
SV*  find_corresponding_var ( SV* , SV*  ) ;
SV*  make_special_copy_var ( SV* , SV*  ) ;
int  prepend_shadow_copy ( SV* , AV* , int , HV*  ) ;
int  append_shadow_copy ( SV* , AV* , int , HV*  ) ;
int  insert_var_copy ( SV* , SV* , AV* , int , HV*  ) ;
int  standard_var_copy ( SV* , SV* , AV* , int , HV*  ) ;
int  small_odd_var_copy ( SV* , SV* , AV* , int , HV*  ) ;
int  even_var_copy ( SV* , SV* , AV* , int , HV*  ) ;
int substitute_var_in_code ( SV* , SV* , SV*  ) ;
```

A.18 size.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"
```

```
int check_sizes(HV* rule) {
  AV* a;
  int i;
  HE* he;
  int modified = 0;
  HV* live_vars;
  live_vars = newHV();

  a = (AV*)SvRV(FETCH(rule,"code"));

  for (i=0; i≤ av_len(a); i++) {
    modified += size_check_statement(*av_fetch(a,i,0), i, a);
  }
  return modified;

}

int size_check_statement(SV* s, int ns, AV* a_list) {
  SV* cond;
  SV* arg;
  HV* live_vars;
  HV* h;
  int input_size = 0;
  int output_size = 0;
  int live_size = 0;
  h = HASH(s);

  if (is_("comment", s)) return 0;

  live_vars = HASH(FETCH(h,"live_vars"));
  cond = FETCH(h,"condition");

  live_size = output_size += (((arg = FETCH_Q(h,"result")))?size_bits(arg):0);

  input_size += (((arg = FETCH_Q(h,"condition")))?size_bits(arg):0);
  if (is_("variable",arg)) {
    live_size += (FETCH_Q(live_vars,name_(arg)))?size_bits(arg):0;
    input_size += (((arg = FETCH_Q(h,"result")))?size_bits(arg):0);
  }
```

```
  /******/
  if (arg = FETCH_Q(h,"argument")) {
    input_size += size_bits(arg);
    if (is_("variable",arg))
      live_size += (FETCH_Q(live_vars,name_(arg)))?size_bits(arg):0;
  }
  if (arg = FETCH_Q(h,"argument1")) {
    input_size += size_bits(arg);
    if (is_("variable",arg))
      live_size += (FETCH_Q(live_vars,name_(arg)))?size_bits(arg):0;
  }
  if (arg = FETCH_Q(h,"argument2")) {
    input_size += size_bits(arg);
    if (is_("variable",arg))
      if (!eq(name_(arg),name_(FETCH_Q(h,"argument1"))))
    live_size += (FETCH_Q(live_vars,name_(arg)))?size_bits(arg):0;
  }

  STORE(h,"input_bits",newSViv(input_size));
  STORE(h,"live_bits",newSViv(live_size));



  if (input_size > num_layers || output_size > num_layers) {
    fprintf(stderr,"\nfailure on pure size constraint with num_layers = %d\n",
num_layers);
    fprintf(stderr,"statement had %d bits input\n",input_size);
    fprintf(stderr,"statement had %d bits output\n",output_size);
    fprint_statement(stderr,s);
    return 0;
  }

  if (live_size > num_layers || live_size > num_layers) {
    fprintf(stderr,"\nfailure on live size constraint with num_layers = %d\n",
num_layers);
    fprintf(stderr,"statement had %d bits input\n",input_size);
    fprintf(stderr,"statement had %d bits output\n",output_size);
    fprint_statement(stderr,s);
    return kill_arguments(s, h, ns,a_list);
  }
```

```
}

int kill_arguments(SV* s, HV* h, int ns, AV* a_list) {
  SV* temp;
  SV* arg, *ts;
  int modified =0 ;
  if (arg = FETCH_Q(h,"argument1")) {
    if (is_("variable",arg)) {
      temp = REF(new_temp());
      ts = assign_statement(temp,arg);
      STORE(HASH(ts),"live_vars", FETCH(h,"live_vars"));
      array_insert(a_list, ns, ts);
      STORE(h,"argument1",temp);
      size_check_statement(ts,ns,a_list);
      modified++;
    }
  }
  if (arg = FETCH_Q(h,"argument2")) {
    if (is_("variable",arg)) {
      temp = REF(new_temp());
      ts = assign_statement(temp,arg);
      STORE(HASH(ts),"live_vars", FETCH(h,"live_vars"));
      array_insert(a_list, ns, ts);
      STORE(h,"argument2",temp);
      size_check_statement(ts,ns,a_list);
      modified++;
    }
  }
  if (arg = FETCH_Q(h,"argument")) {
    SV* res;
    res = FETCH(h,"result");

    if (1 & size_bits(res)) {
      fprintf(stderr,"woozle wozzle\n");
    } else {
      char *ct;
      int midpoint = (size_bits(res)>>1)-1;
      ts = assign_statement(REF(new_fraction(res,0,midpoint)),
               REF(new_fraction(arg,midpoint+1,size_bits(arg)-1)));
```

```
      STORE(HASH(ts),"live_vars", FETCH(h,"live_vars"));
      STORE(HASH(ts),"condition", FETCH(h,"condition"));
      array_insert(a_list, ns, ts);
      size_check_statement(ts,ns,a_list);
      STORE(h,"result",REF(new_fraction(res,midpoint+1,size_bits(res)-1)));
      STORE(h,"argument",REF(new_fraction(arg,0,midpoint)));

      ct = strsave("flip semantics of variable foo bar baz");
      sprintf(ct,"flip semantics of variable %s",name_(res));
      ts = comment_statement(ct);
      array_insert(a_list, ns+2, ts);
      modified++;
    }

  }

  return modified;
}
```

A.19 statement.c

```
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"


SV* append_statement(SV* s_list,SV* s) {
  av_push((AV*)SvRV(s_list),s);
  return s_list;
}


add_condition(HV* s) {
  if (av_len(conditions) != -1) {
    STORE(s,"condition",*av_fetch(conditions,av_len(conditions),0));
  } else {
    printf("ugalugga! %d \n", av_len(conditions));
```

```
  }
}

uncondition(SV* ref) {
  HV* s = HASH(ref);
  STORE(s,"condition",REF(new_constant(1)));
}

/* concates code into s_one, and returns it */
AV* concat_statements(SV* s_one, SV* s_two) {
  SV* s;
  AV *one, *two;

  one = (AV*)SvRV(s_one);
  two = (AV*)SvRV(s_two);
  while (&sv_undef != (s = av_shift(two))) {
    av_push(one,s);
  }
  return one;
}

SV* comment_statement(char* c) {
  HV* res;
  res = newHV();
  STORE(res,"type",newSVpv("comment",0));
  STORE(res,"comment",newSVpv(c,0));
  return REF(res);
}

SV* assign_statement(SV* lvalue, SV* arg) {
  HV* res;
  res = newHV();
  STORE(res,"type",newSVpv("assign",0));
  STORE(res,"argument",arg);
  STORE(res,"result",lvalue);
  copy_limits(HASH(lvalue),arg);
  add_condition(res);
  return REF(res);
}
```

```
SV* unary_assign_statement(int op, SV* arg) {

  HV* res;
  HV* result_v;
  res = newHV();
  STORE(res,"type",newSVpv("unary_assign",0));
  STORE(res,"operator",newSViv(op));
  STORE(res,"argument1",arg);
  result_v = (HV*)new_temp();
  STORE(res,"result",REF(result_v));
  unary_add_limits(result_v,op,arg);
  add_condition(res);
  return REF(res);
}

SV* binary_assign_statement(SV* arg1, int op, SV* arg2) {
  HV* res;
  HV* result_v;
  res = newHV();
  STORE(res,"type",newSVpv("binary_assign",0));
  STORE(res,"operator",newSViv(op));
  STORE(res,"argument1",arg1);
  STORE(res,"argument2",arg2);
  result_v = (HV*)new_temp();
  STORE(res,"result",REF(result_v));
  binary_add_limits(result_v, op, arg1, arg2);
  add_condition(res);
  return REF(res);
}

binary_add_limits(HV* r, int op, SV* arg1, SV* arg2) {
  HV *a1, *a2;
  SV* foo;
  int i,a1l,a1h,a2l,a2h;
  i=1;
  a1 = HASH(arg1);
  a2 = HASH(arg2);
  if (eq("variable",SvPV(FETCH(a1,"type"),na))) {
    (foo = FETCH_Q(a1,"low"))?(a1l=SvIV(foo)):(i=0);
    (foo = FETCH_Q(a1,"high"))?(a1h=SvIV(foo)):(i=0);
```

```
  } else if (eq("constant",SvPV(FETCH(a1,"type"),na))) {
    a1h = a1l = SvIV(FETCH(a1,"value"));
  } else {
    printf("beware! his flashing eyes!\n"); i=0;
  }
  if (eq("variable",SvPV(FETCH(a2,"type"),na))) {
    (foo = FETCH_Q(a2,"low"))?(a2l=SvIV(foo)):(i=0);
    (foo = FETCH_Q(a2,"high"))?(a2h=SvIV(foo)):(i=0);
  } else if (eq("constant",SvPV(FETCH(a2,"type"),na))) {
    a2h = a2l = SvIV(FETCH(a2,"value"));
  } else {
    printf("beware! his floating hair!\n"); i=0;
  }
  if (i) {
    i = max_binary_op(a1l,a1h,op,a2l,a2h);
    if ((foo=FETCH_Q(r,"high"))?(i>SvIV(foo)):1)
      STORE(r,"high",newSViv(i));
    i = min_binary_op(a1l,a1h,op,a2l,a2h);
    if ((foo=FETCH_Q(r,"low"))?(i<SvIV(foo)):1)
      STORE(r,"low",newSViv(i));
  } else {
    printf("\nfailed binary_add_limits\n");
    print_hv(a1);
    print_hv(a2);
  }
}

unary_add_limits(HV* r, int op, SV* arg1) {
  HV *a1;
  SV* foo;
  int i,a1l,a1h;
  i=1;
  a1 = HASH(arg1);
  (foo = FETCH_Q(a1,"low"))?(a1l=SvIV(foo)):(i=0);
  (foo = FETCH_Q(a1,"high"))?(a1h=SvIV(foo)):(i=0);
  if (i) {
    i = max_unary_op(op,a1l,a1h);
    if ((foo=FETCH_Q(r,"high"))?(i>SvIV(foo)):1)
      STORE(r,"high",newSViv(i));
    i = min_unary_op(op,a1l,a1h);
```

```
    if ((foo=FETCH_Q(r,"low"))?(i<SvIV(foo)):1)
      STORE(r,"low",newSViv(i));
  } else {
    printf("\nfailed unary_add_limits\n");
    print_hv(a1);
  }
}

copy_limits(HV* r, SV* arg1) {
  HV *a1;
  SV* foo;
  int i,a1l,a1h;
  i=1;
  a1 = HASH(arg1);
  if (eq("variable",SvPV(FETCH(a1,"type"),na))) {
    (foo = FETCH_Q(a1,"low"))?(a1l=SvIV(foo)):(i=0);
    (foo = FETCH_Q(a1,"high"))?(a1h=SvIV(foo)):(i=0);
  } else if (eq("constant",SvPV(FETCH(a1,"type"),na))) {
    a1h = a1l = SvIV(FETCH(a1,"value"));
  } else {
    printf("beware!\n"); i=0;
  }

  if (!FETCH_Q(r,"field")) {
    if (i) {
      if ((foo=FETCH_Q(r,"high"))?(a1h>SvIV(foo)):1)
    STORE(r,"high",newSViv(a1h));
      if ((foo=FETCH_Q(r,"low"))?(a1l<SvIV(foo)):1)
    STORE(r,"low",newSViv(a1l));
    } else {
      printf("\nfailed to copy limits\n");
      print_hv(a1);
    }
  }
}
```

A.20 thesis.c

```
#include <stdio.h>
#include <stdlib.h>
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "y.tab.h"

int    num_layers;
int     print_intermediate = 0;
int     print_conflicts = 0;
int     print_after_copy = 0;
int    print_variables = 1;
int    do_copy = 1;
int     do_anneal = 1;
int     do_second_anneal = 1;
int    print_after_anneal = 0;
int    print_final = 1;


extern int dtdebug;
extern FILE *dtin;
extern char *dtfilename;
extern int dtlineno;

char *dtprogramname;


main(int argc, char **argv, char **env) {
  int i;
  FILE *f;
  num_layers = 16;

  for (i=1; i<argc; i++)
    {

      if (eq(argv[i],"-usage") || argc < 1) {
    system("/usr/local/bin/less /im/magnus/proj/thesis/usage");
      }
```

```
  if (eq(argv[i],"-l"))
{
  num_layers = atoi(argv[++i]);
  if (num_layers < 2)
    {
      fprintf(stderr, "num_layers set to %d\n", num_layers);
    }
}
  if (eq(argv[i],"-pi")) { print_intermediate = 1; }
  if (eq(argv[i],"-nan")) { do_anneal = 0; }
  if (eq(argv[i],"-npv")) { print_variables = 0; }
  if (eq(argv[i],"-pc")) { print_conflicts = 1; }
  if (eq(argv[i],"-pac")) { print_after_copy = 1; }
  if (eq(argv[i],"-n2an")) { do_second_anneal = 0; }
  if (eq(argv[i],"-pan")) { print_after_anneal = 1; }
  if (eq(argv[i],"-npf")) { print_final = 0; }
  if (eq(argv[i],"-stdin_np"))
{
  f = fopen(argv[++i],"r");
  if (f)
    {
      dtin = stdin;
      dtfilename = "stdin";
      dtlineno = 0;
      dtparse();
    }
  else
    {
      fprintf(stderr, "%s is not a readable input file\n", argv[i]);
    }
}
  if (eq(argv[i],"-f"))
{
  f = fopen(argv[++i],"r");
  if (f)
    {
      dtin = f;
      dtfilename = argv[i];
      pre_process(argc, argv, f, dtfilename);
      dtparse();
```

```
        fclose(f);
      }
    else
      {
        fprintf(stderr, "%s is not a readable input file\n", argv[i]);
      }
  }
    if (eq(argv[i],"-fnp"))
  {
    f = fopen(argv[++i],"r");
    if (f)
      {
        dtin = f;
        dtfilename = argv[i];
        dtparse();
        fclose(f);
      }
    else
      {
        fprintf(stderr, "%s is not a readable input file\n", argv[i]);
      }
  }
    if (eq(argv[i], "-d"))
  {
    dtdebug = 1;
    fprintf(stderr, "debugging on\n");
  }
    if (eq(argv[i], "-"))
  {
    dtin = stdin;
    dtfilename = "stdin";
    dtlineno = 0;
    pre_process(argc, argv, f, dtfilename);
    dtparse();
  }
  }
 return 0;
}
```

```
pre_process(int argc, char **argv, FILE *f,char *filename) {
  int frk;
  int to_par[2];  /* for pipes */

  if (-1 == pipe(to_par))
      fprintf(stderr,"pipe failed with error %d\n", errno);
  if (0 == (frk = fork()))
    {       /* Parent Process */
      char *ppargv[20];
      int argn;
      int count = 0;
      ppargv[count++] = "/lib/cpp";
      ppargv[count++] = "-B";

      for (argn = 0; argn < argc; argn++)
    {
      if (argv[argn][0] == '-' && argv[argn][1] == 'D' && count < 19)
        {
          ppargv[count++] = argv[argn];
        }
    }
      ppargv[count] = (char *)0;

      if (f == NULL)
    f = fopen(filename, "r");

      if (-1 == dup2(fileno(f),fileno(stdin)))
    fprintf(stderr,"dup2 #1 failed with error %d\n", errno);
      if (-1 == dup2(to_par[1],fileno(stdout)))
    fprintf(stderr,"dup2 #2 failed with error %d\n", errno);
      close(to_par[1]); close(to_par[0]);
      execv("/lib/cpp",ppargv);
    }
  else if (frk == -1)
    {
      fprintf(stderr, "Fork failed with error %d\n",errno);
    }
  else
    {        /* Child Process */
```

```
  if (-1 == dup2(to_par[0],fileno(f)))
 fprintf(stderr,"dup2 #3 failed with error %d\n", errno);
  close(to_par[1]); close(to_par[0]);
  return(0);
 }
}
```

A.21 yacc.y

```
/* yacc.y                                              */
/*                                                     */
/* Copyright (C) 1996  Daniel Risacher                 */
/* All rights reserved.                                */
/*                                                     */
/* This software may be freely copied, modified, and redistributed,  */
/* provided that this copyright notice is preserved on all copies.   */
/*                                                     */
/* You may not distribute this software, in whole or in part, as part of */
/* any commercial product without the express consent of the authors.  */
/*                                                     */
/* There is no warranty or other guarantee of fitness of this software */
/* for any purpose.  It is provided solely "as is".    */
/* $Id: yacc.y,v 1.5 1996/05/02 01:16:18 magnus Exp magnus $ */

%{
#include <stdio.h>
#include <EXTERN.h>
#include <perl.h>
#include "kludge.h"
#include "gc.h"
#define DTDEBUG 1



char *dtfilename;           /* Input filename */
extern FILE *dtin;          /* input file pointer */
int dtlineno = 1;           /* Current line # in file */
%}
```

```
%union {
  char *c;
  int i;
  HV* hv;
}

%token <i> tINT
%token <c> tSTRING
%token tASSIGN tELSE tBITS tDD
%token tEQ tNE tLS tRS tLE tGE tIF tOR tAND tTHEN

%type <i> Unary_operator
%type <hv> All Cell_Def Fields Field Rule Block Statements Statement
%type <hv> If_start If_stat Expr Assign_expr Condition_expr Logic_or_expr
%type <hv> Logic_and_expr Incl_or_expr Excl_or_expr And_expr Equal_expr
%type <hv> Relation_expr Shift_expr Add_expr Multiply_expr
%type <hv> Cast_expr Unary_expr Postfix_expr Primary_expr

%%
All        : Begin Cell_Def Rule   { $$ = (HV*)top($2,$3); };
Begin        : epsilon        { initialize(); }
Cell_Def    : '{' Fields '}'      { $$ = $2; } ;
Fields        : epsilon        { $$ = (HV*) new_cell(); }
        | Field Fields         { $$ = (HV*) add_field($1, $2); } ;
Field        : tSTRING tINT tBITS ';'    {$$ = (HV*) new_field($1,0,1<<$2-1);}
        | tSTRING tINT tDD tINT ';'     {$$ = (HV*) new_field($1,$2,$4); } ;
Rule        : Block         { $$ = (HV*) process_block($1); } ;
Block        : '{' Statements '}'    { $$ = (HV*) $2; } ;
Statements    : epsilon         { $$ = (HV*) new_s_list(); }
        | Statements Statement    { $$ = (HV*) add_statement($1,$2); };
Statement       : Expr ';'
        | If_stat;
If_stat        : If_start Block     {$$=(HV*)if_stat($1,$2);} ;
If_start    : tIF '(' Expr ')'     {$$=(HV*)if_start($3);} ;
Expr        : Assign_expr
        | Expr ',' Assign_expr     {$$=(HV*)list_op($1,$3);}
        ;
Assign_expr    : Condition_expr
        | Postfix_expr '=' Assign_expr {$$=(HV*)assign_op($1, $3);}
        ;
```

```
Condition_expr    : Logic_or_expr;
Logic_or_expr    : Logic_and_expr
         | Logic_or_expr tOR Logic_and_expr {$$=(HV*)binary_op($1,tOR,$3);}
        ;
Logic_and_expr    : Incl_or_expr
        | Logic_and_expr tAND Incl_or_expr {$$=(HV*)binary_op($1,tAND,$3);}
        ;
Incl_or_expr    : Excl_or_expr
        | Incl_or_expr '|' Excl_or_expr {$$=(HV*)binary_op($1,'|',$3);}
        ;
Excl_or_expr    : And_expr
        | Excl_or_expr '^' And_expr {$$=(HV*)binary_op($1,'^',$3);}
        ;
And_expr    : Equal_expr
        | And_expr '&' Equal_expr {$$=(HV*)binary_op($1,'&',$3);}
        ;
Equal_expr    : Relation_expr
        | Equal_expr tEQ Relation_expr {$$=(HV*)binary_op($1,tEQ,$3);}
        | Equal_expr tNE Relation_expr {$$=(HV*)binary_op($1,tNE,$3);}
        ;
Relation_expr    : Shift_expr
        | Relation_expr '<' Shift_expr {$$=(HV*)binary_op($1,'<',$3);}
        | Relation_expr '>' Shift_expr {$$=(HV*)binary_op($1,'>',$3);}
        | Relation_expr tLE Shift_expr {$$=(HV*)binary_op($1,'-',$3);}
        | Relation_expr tGE Shift_expr {$$=(HV*)binary_op($1,tGE,$3);}
        ;
Shift_expr    : Add_expr
        | Shift_expr tLS Add_expr {$$=(HV*)binary_op($1,tLS,$3);}
        | Shift_expr tRS Add_expr {$$=(HV*)binary_op($1,tRS,$3);}
        ;
Add_expr    : Multiply_expr
        | Add_expr '+' Multiply_expr {$$=(HV*)binary_op($1,'+',$3);}
        | Add_expr '-' Multiply_expr {$$=(HV*)binary_op($1,'-',$3);}
        ;
Multiply_expr    : Cast_expr
        | Multiply_expr '*' Cast_expr {$$=(HV*)binary_op($1,'*',$3);}
        | Multiply_expr '/' Cast_expr {$$=(HV*)binary_op($1,'/',$3);}
        | Multiply_expr '%' Cast_expr {$$=(HV*)binary_op($1,'%',$3);}
        ;
Cast_expr    : Unary_expr;
```

```
Unary_expr     : Postfix_expr
        | Unary_operator Cast_expr  { $$ = (HV*) unary_op($1, $2); };
Postfix_expr    : Primary_expr        { $$ = (HV*) ghost_offset_ref($1); }
        | Primary_expr '[' ']'  { $$ = (HV*) ghost_offset_ref($1); }
        | Primary_expr '[' Expr ']' { $$ = (HV*) offset_ref($1,$3); };
Primary_expr    : tSTRING    { $$ = (HV*) ref_variable($1); }
        | tINT        { $$ = (HV*) ref_constant($1); }
        | '(' Expr ')'     { $$ = $2; } ;


Unary_operator
    : '!'         { $$ = (int) '!' }
    | '+'         { $$ = (int) '+' }
    | '-'         { $$ = (int) '-' }
    | '~'         { $$ = (int) '~' }
    ;

epsilon        : ;


%%
/*
 * Issue error message containing filename and line number, and exit.
 */
/*VARARGS1*/
/* dterror(s, pat1, pat2) */
/* char *s, *pat1, *pat2; */
/* { */
/*   if (!dtfilename) */
/*     dtfilename = "none"; */
/*   if (!strlen(dtfilename)) */
/*     strcpy(dtfilename,"stdin"); */
/*   if (!s) {s = "";} */
/*   if (!pat1) {pat1 = "";} */
/*   if (!pat2) {pat2 = "";} */

/*   fprintf(stderr,"%s: Error: %s: line %d: ", "parser", */
/*       dtfilename, dtlineno); */
/*   fprintf(stderr, s, pat1, pat2); */
/*   if (*s && s[strlen(s) -1] != '\n') */
```

```
    /* YACC doesn't put newlines on error messages. */
/*      fprintf(stderr,"\n");      */
/*    fflush(stderr); */
/*    return 0; */
    /*      exit(1); */
/* } */


dterror(char *format, ...)
{
  va_list args;
  if (!dtfilename)
    dtfilename = "stdin";
  if (!strlen(dtfilename))
    strcpy(dtfilename,"stdin");

  va_start(args,format);
  fprintf(stderr,"%s: Error: %s: line %d: ", "parser",
      dtfilename, dtlineno);
  /* print out remainder of message */
  (void) vfprintf(stderr, format, args);
  va_end(args);
  fprintf(stderr,"\n");
  fflush(stderr);
  return 0;
}
```

Appendix B

Test Cases

B.1 life.t

This is the source code for Conway's game of Life.

```
/* Conway's game of Life.

The rule is that we look at each 3x3 neighborhood, and count the
number of ones (not including the center).  If this count is 2, the
center remains unchanged.  If this count is 3, the center becomes a 1.
Otherwise, the center becomes a 0.

*/

{
c 1 bits;
}

{
  sum = c[-1,-1] + c[-1,0] + c[-1,1] + c[0,-1]
    + c[0,1] + c[1,-1] + c[1,0] + c[1,1];

  if (sum == 3) {
    c[0,0] = 1;
  }

  if (sum < 2 || sum > 3) {
    c[0,0] = 0;
  }
}
```

B.1.1 thesis -f life.t

This is the compiled output for Conway's game of Life, targeted at a 16-layer machine.

```
dt output
 final code after regrouping:
-------------------------group-
ins: c
live_before: c
  comment:  copy step
  if: 1  c[1,1] = c
  comment:  copy step
  if: 1  c[0,1] = c
  comment:  copy step
  if: 1  c[1,0] = c
  comment:  copy step
  if: 1  c[0,0] = c
  comment:  copy step
  if: 1  c[1,-1] = c
  comment:  copy step
  if: 1  c[0,-1] = c
  comment:  copy step
  if: 1  c[-1,0] = c
  comment:  copy step
  if: 1  c[-1,-1] = c
outs: c[-1,-1] c[0,-1] c[-1,0] c[1,-1] c[0,0] c[1,0] c[0,1] c[1,1]
live_after: c
-------------------------group-
ins: c[-1,-1] c[0,-1] c[-1,0] c[1,-1] c[-1,1] c[0,0] c[1,0] c[0,1] c[1,1]
live_before: c
  if: 1  t1 = c[-1,-1] + c[-1,0]
  if: 1  t2 = t1 + c[-1,1]
  if: 1  t3 = t2 + c[0,-1]
  if: 1  t4 = t3 + c[0,1]
  if: 1  t5 = t4 + c[1,-1]
  if: 1  t6 = t5 + c[1,0]
  if: 1  t7 = t6 + c[1,1]
  if: 1  sum = t7
  if: 1  t8 = sum == 3
  if: 1  t9 = t8 && 1
  if: t9  c[0,0] = 1
```

```
  if: 1  t10 = sum < 2
  if: 1  t11 = sum > 3
  if: 1  t12 = t10 || t11
  if: 1  t13 = t12 && 1
  if: t13  c[0,0] = 0
  comment:  copy step
  if: 1  c = c[0,0]
outs: c c[0,0]
live_after: c

variables:
c   0 .. 1 0:0-0 field
t10   0 .. 1 0:0-0
t1   0 .. 2 0:1-0
sum   0 .. 8 0:3-0
t11   0 .. 1 0:0-0
t2   0 .. 3 0:1-0
t12   0 .. 1 0:0-0
t3   0 .. 4 0:2-0
t13   0 .. 1 0:0-0
t4   0 .. 5 0:2-0
t5   0 .. 6 0:2-0
t6   0 .. 7 0:2-0
t7   0 .. 8 0:3-0
t8   0 .. 1 0:0-0
t9   0 .. 1 0:0-0

shadows:
c[-1,-1]   0 .. 1 8:8-0 field
c[0,-1]   0 .. 1 4:4-0 field
c[-1,0]   0 .. 1 3:3-0 field
c[1,-1]   0 .. 1 2:2-0 field
c[-1,1]   0 .. 1 0:0-0 field
c[0,0]   0 .. 1 1:1-0 field
c[0,1]   0 .. 1 7:7-0 field
c[1,0]   0 .. 1 6:6-0 field
c[1,1]   0 .. 1 5:5-0 field
```

B.1.2 thesis -l 4 -f life.t

This is the compiled output for Conway's game of Life, targeted at a 4-layer machine.

```
dt output
 final code after regrouping:
-------------------------group-
ins: c
live_before: c
  comment:  copy step
  if: 1  c[-1,0] = c
  comment:  copy step
  if: 1  c[-1,-1] = c
outs: c[-1,-1] c[-1,0]
live_after: c
-------------------------group-
ins: c[-1,-1] c c[-1,0] c[-1,1]
live_before: c
  if: 1  t1 = c[-1,-1] + c[-1,0]
  if: 1  t2 = t1 + c[-1,1]
  comment:  copy step
  if: 1  c[0,1] = c
outs: c[0,1] t2
live_after: c t2
-------------------------group-
ins: c[0,1] c[0,-1] t2
live_before: c t2
  if: 1  t3 = t2 + c[0,-1]
  if: 1  t4 = t3 + c[0,1]
outs: t4
live_after: t4 c
-------------------------group-
ins: t4 c[1,-1]
live_before: t4 c
  if: 1  t5 = t4 + c[1,-1]
outs: t5
live_after: t5 c
-------------------------group-
ins: c[1,0] t5
live_before: t5 c
  if: 1  t6 = t5 + c[1,0]
outs: t6
live_after: t6 c
-------------------------group-
ins: c[1,1] t6
live_before: t6 c
```

```
  if: 1  t7 = t6 + c[1,1]
  if: 1  sum = t7
  if: 1  t8 = sum == 3
  if: 1  t10 = sum < 2
  if: 1  t11 = sum > 3
outs: t8 t10 t11
live_after: c t8 t10 t11
-------------------------group-
ins: t8 t10 t11 c[0,0]
live_before: c t8 t10 t11
  if: 1  t9 = t8 && 1
  if: t9  c[0,0] = 1
  if: 1  t12 = t10 || t11
  if: 1  t13 = t12 && 1
  if: t13  c[0,0] = 0
outs: c[0,0]
live_after: c

variables:
c   0 .. 1 3:3-0 field
t10   0 .. 1 1:1-0
t1   0 .. 2 0:1-0
sum   0 .. 8 0:3-0
t11   0 .. 1 0:0-0
t2   0 .. 3 0:1-0
t12   0 .. 1 0:0-0
t3   0 .. 4 0:2-0
t13   0 .. 1 0:0-0
t4   0 .. 5 0:2-0
t5   0 .. 6 0:2-0
t6   0 .. 7 0:2-0
t7   0 .. 8 0:3-0
t8   0 .. 1 2:2-0
t9   0 .. 1 0:0-0

shadows:
c[-1,-1]   0 .. 1 1:1-0 field
c[0,-1]   0 .. 1 3:3-0 field
c[-1,0]   0 .. 1 0:0-0 field
c[1,-1]   0 .. 1 3:3-0 field
c[-1,1]   0 .. 1 3:3-0 field
c[0,0]   0 .. 1 3:3-0 field
```

```
c[0,1]   0 .. 1 2:2-0 field
c[1,0]   0 .. 1 3:3-0 field
c[1,1]   0 .. 1 3:3-0 field
```

B.1.3 thesis -nan -l 4 -f life.t

This is the compiled output for Conway's game of Life, targeted at a 4-layer machine, and with the first grouping phase disabled.

```
dt output
Warning: Couldn't find a good location for t10
Warning: Couldn't find a good location for t8
 final code after regrouping:
-------------------------group-
ins: c
live_before: c
  comment:  copy step
  if: 1  c[-1,0] = c
  comment:  copy step
  if: 1  t14 = c
outs: t14 c[-1,0]
live_after: t14 c
-------------------------group-
ins: t14 c
live_before: t14 c
  comment:  copy step
  if: 1  c[-1,-1] = t14
  comment:  copy step
  if: 1  c[-1,1] = c
outs: c[-1,-1] c[-1,1]
live_after: c
-------------------------group-
ins: c[-1,-1] c c[-1,0] c[-1,1]
live_before: c
  if: 1  t1 = c[-1,-1] + c[-1,0]
  if: 1  t2 = t1 + c[-1,1]
  comment:  copy step
  if: 1  c[0,-1] = c
  comment:  copy step
  if: 1  c[0,1] = c
outs: c[0,1] c[0,-1] t2
live_after: c t2
```

```
------------------------group-
ins: c[0,1] c[0,-1] t2
live_before: c t2
  if: 1  t3 = t2 + c[0,-1]
  if: 1  t4 = t3 + c[0,1]
outs: t4
live_after: t4 c
------------------------group-
ins: c
live_before: t4 c
  comment:  copy step
  if: 1  c[1,-1] = c
outs: c[1,-1]
live_after: t4 c
------------------------group-
ins: t4 c[1,-1]
live_before: t4 c
  if: 1  t5 = t4 + c[1,-1]
outs: t5
live_after: t5 c
------------------------group-
ins: c
live_before: t5 c
  comment:  copy step
  if: 1  c[1,0] = c
outs: c[1,0]
live_after: t5 c
------------------------group-
ins: c[1,0] t5
live_before: t5 c
  if: 1  t6 = t5 + c[1,0]
outs: t6
live_after: t6 c
------------------------group-
ins: c
live_before: t6 c
  comment:  copy step
  if: 1  c[1,1] = c
outs: c[1,1]
live_after: t6 c t8
------------------------group-
ins: c[1,1] t6
```

```
live_before: t6 c t8
  if: 1  t7 = t6 + c[1,1]
  if: 1  sum = t7
  comment:  copy step
  if: 1  t15<0-1> = sum<2-3>
outs: sum
live_after: c t8 sum
-------------------------group-
ins: sum<0-1>
live_before: c t8 sum
  if: 1  t15<2-3> = sum<0-1>
  comment:  flip semantics of variable t15
outs: t15<2-3>
live_after: c t8 sum
-------------------------group-
ins: t15
live_before: c sum
  if: 1  t8 = t15 == 3
outs: t8
live_after: c t8 sum
-------------------------group-
ins: sum<2-3>
live_before: c t10 sum
  comment:  copy step
  if: 1  t18<0-1> = sum<2-3>
outs: t18<0-1>
live_after: c t8 t10 sum
-------------------------group-
ins: t8 sum<0-1>
live_before: c t8 t10 sum
  comment:  copy step
  if: 1  t17 = t8
  if: 1  t18<2-3> = sum<0-1>
  comment:  flip semantics of variable t18
outs:
live_after: c t10 sum
-------------------------group-
ins: sum
live_before: c sum t10
  if: 1  t11 = sum > 3
outs: t11
live_after: c t18 t11
```

```
-------------------------group-
ins: t18
live_before: c t18 t11
  if: 1  t10 = t18 < 2
  comment:  copy step
  if: 1  t20 = t10
outs: t20
live_after: c t17 t20 t11
-------------------------group-
ins: t17 t20
live_before: c t17 t20 t11
  comment:  copy step
  if: 1  t16 = t17
  comment:  copy step
  if: 1  t19 = t20
outs: t16 t19
live_after: t16 c t19 t11
-------------------------group-
ins: t16 t11 t19 c[0,0]
live_before: t16 c t19 t11
  if: 1  t9 = t16 && 1
  if: t9  c[0,0] = 1
  if: 1  t12 = t19 || t11
  if: 1  t13 = t12 && 1
  if: t13  c[0,0] = 0
outs: c[0,0]
live_after: c

variables:
c   0 .. 1 1:1-1 field
t1   0 .. 2 0:1-0
t2   0 .. 3 0:1-0
t3   0 .. 4 0:2-0
t4   0 .. 5 0:2-0
t5   0 .. 6 0:2-0
t6   0 .. 7 0:2-0
t7   0 .. 8 0:3-0
t8   0 .. 1 0:0-1 spilled
t9   0 .. 1 0:0-1
sum   0 .. 8 0:3-0
t10   0 .. 1 0:0-1 spilled
t20   0 .. 1 2:2-0
```

```
t11   0 .. 1 1:1-0
t12   0 .. 1 1:1-0
t13   0 .. 1 0:0-0
t14   0 .. 1 0:0-0
t15   0 .. 8 0:3-special
t16   0 .. 1 0:0-special
t17   0 .. 1 2:2-1
t18   0 .. 8 0:3-special
t19   0 .. 1 0:0-special

shadows:
c[-1,-1]   0 .. 1 1:1-0 field
c[0,-1]   0 .. 1 2:2-0 field
c[-1,0]   0 .. 1 0:0-0 field
c[1,-1]   0 .. 1 3:3-0 field
c[-1,1]   0 .. 1 2:2-0 field
c[0,0]   0 .. 1 1:1-1 field
c[0,1]   0 .. 1 3:3-0 field
c[1,0]   0 .. 1 3:3-0 field
c[1,1]   0 .. 1 3:3-0 field
```

B.2 hpp.t

This is the source code for the HPP lattice-gas.

```
/* The HPP lattice gas automata

In the HPP gas, particles travel with constant speed in one of four
directions (north, east, south, west).  If two particles have a
head-on collision, they each make a 90 degree turn.  In any other
collision, all participating particles continue unchanged.

This lattice gas conserves particles, energy and momentum.

 */


{
n 1 bits;
e 1 bits;
s 1 bits;
```

```
w 1 bits;
}

{
  n = n[0,-1];
  e = e[-1,0];
  s = s[0,1];
  w = w[1,0];

  swap = 0;

  if ((n && s) && !(e || w)) {
    swap = 1;
  }
  if ((e && w) && !(n || s)) {
    swap = 1;
  }
  if (swap) {
    n = e;
    e = s;
    s = n;
    w = e;
  }
}
```

B.2.1 thesis -f hpp.t

This is the compiled output for the HPP lattice-gas, targeted at a 16-layer machine.

```
dt output
 final code after regrouping:
-------------------------group-
ins: e n
live_before: s e n w
  comment:  copy step
  if: 1  e[-1,0] = e
  comment:  copy step
  if: 1  n[0,-1] = n
outs: n[0,-1] e[-1,0]
live_after: s e n w
-------------------------group-
ins: n[0,-1] s[0,1] w[1,0] e[-1,0]
```

```
live_before: s e n w
  if: 1  n[0] = n[0,-1]
  if: 1  e[0] = e[-1,0]
  if: 1  s[0] = s[0,1]
  if: 1  w[0] = w[1,0]
  if: 1  swap = 0
  if: 1  t1 = n[0] && s[0]
  if: 1  t2 = e[0] || w[0]
  if: 1  t3 =  ! t2
  if: 1  t4 = t1 && t3
  if: 1  t5 = t4 && 1
  if: t5  swap = 1
  if: 1  t6 = e[0] && w[0]
  if: 1  t7 = n[0] || s[0]
  if: 1  t8 =  ! t7
  if: 1  t9 = t6 && t8
  if: 1  t10 = t9 && 1
  if: t10  swap = 1
  if: 1  t11 = swap && 1
  if: t11  n[0] = e[0]
  if: t11  e[0] = s[0]
  if: t11  s[0] = n[0]
  if: t11  w[0] = e[0]
outs: s[0] e[0] n[0] w[0]
live_after: s e n w

variables:
s   0 .. 1 1:1-0 field
t10   0 .. 1 0:0-0
t1   0 .. 1 0:0-0
e   0 .. 1 3:3-0 field
t11   0 .. 1 0:0-0
t2   0 .. 1 0:0-0
t3   0 .. 1 0:0-0
w   0 .. 1 0:0-0 field
t4   0 .. 1 0:0-0
t5   0 .. 1 0:0-0
t6   0 .. 1 0:0-0
t7   0 .. 1 0:0-0
swap   0 .. 1 0:0-0
t8   0 .. 1 0:0-0
t9   0 .. 1 0:0-0
```

```
n   0 .. 1 2:2-0 field

shadows:
n[0,-1]   0 .. 1 3:3-0 field
n[0]   0 .. 1 2:2-0 field
e[-1,0]   0 .. 1 2:2-0 field
s[0,1]   0 .. 1 1:1-0 field
s[0]   0 .. 1 1:1-0 field
w[1,0]   0 .. 1 0:0-0 field
e[0]   0 .. 1 3:3-0 field
w[0]   0 .. 1 0:0-0 field
```

B.2.2 thesis -l 4 -f hpp.t

This is the compiled output for the HPP lattice-gas, targeted at a 4-layer machine.

```
dt output
 final code after regrouping:
-------------------------group-
ins: n[0,-1]
live_before: s e n w
  if: 1  n[0] = n[0,-1]
  comment:  copy step
  if: 1  n = n[0]
outs: n[0] n
live_after: s e n w
-------------------------group-
ins: e n w
live_before: s e n w
  comment:  copy step
  if: 1  e[-1,0] = e
  comment:  copy step
  if: 1  n[0] = n
  comment:  copy step
  if: 1  w[1,0] = w
outs: w[1,0] n[0] e[-1,0]
live_after: s e n w
-------------------------group-
ins: s[0,1] w[1,0] n[0] e[-1,0]
live_before: s e n w
  if: 1  e[0] = e[-1,0]
  if: 1  s[0] = s[0,1]
```

```
  if: 1  w[0] = w[1,0]
  if: 1  t1 = n[0] && s[0]
  if: 1  t2 = e[0] || w[0]
  if: 1  t3 =  ! t2
  if: 1  t4 = t1 && t3
outs: t4 s[0] e[0] w[0]
live_after: t4 s e n w
-------------------------group-
ins: e[0]
live_before: t4 s e n w
  comment:  copy step
  if: 1  e = e[0]
outs: e
live_after: t4 s e n w
-------------------------group-
ins: e
live_before: t4 s e n w
  comment:  copy step
  if: 1  e[0] = e
outs: e[0]
live_after: t4 s e n w
-------------------------group-
ins: t4 e[0] n w[0]
live_before: t4 s e n w
  if: 1  swap = 0
  if: 1  t5 = t4 && 1
  if: t5  swap = 1
  if: 1  t6 = e[0] && w[0]
  comment:  copy step
  if: 1  n[0] = n
outs: t6 swap n[0]
live_after: t6 s swap e n w
-------------------------group-
ins: t6 s[0] swap n[0]
live_before: t6 swap s e n w
  if: 1  t7 = n[0] || s[0]
  if: 1  t8 =  ! t7
  if: 1  t9 = t6 && t8
  if: 1  t10 = t9 && 1
  if: t10  swap = 1
outs: swap
live_after: s swap e n w
```

```
-------------------------group-
ins: e n
live_before: s swap e n w
  comment:  copy step
  if: 1  n[0] = n
  comment:  copy step
  if: 1  e[0] = e
outs: e[0] n[0]
live_after: s swap e n w
-------------------------group-
ins: swap s[0] e[0] n[0]
live_before: swap s e n w
  if: 1  t11 = swap && 1
  if: t11  n[0] = e[0]
  if: t11  s[0] = n[0]
outs: s[0] t11 n[0]
live_after: s e n t11 w
-------------------------group-
ins: e n[0]
live_before: s e n t11 w
  comment:  copy step
  if: 1  n = n[0]
  comment:  copy step
  if: 1  e[0] = e
outs: e[0] n
live_after: s e t11 n w
-------------------------group-
ins: s[0] e[0] t11 w[0]
live_before: s e t11 n w
  if: t11  e[0] = s[0]
  if: t11  w[0] = e[0]
  comment:  copy step
  if: 1  e = e[0]
outs: e[0] e w[0]
live_after: s e n w

variables:
s   0 .. 1 1:1-0 field
t10   0 .. 1 0:0-0
t1   0 .. 1 0:0-0
e   0 .. 1 1:1-1 field
t11   0 .. 1 0:0-0
```

```
t2   0 .. 1 0:0-0
t3   0 .. 1 0:0-0
w   0 .. 1 2:2-0 field
t4   0 .. 1 0:0-0
t5   0 .. 1 0:0-0
t6   0 .. 1 3:3-0
t7   0 .. 1 0:0-0
swap   0 .. 1 0:0-0
t8   0 .. 1 0:0-0
t9   0 .. 1 0:0-0
n   0 .. 1 0:0-1 field

shadows:
n[0,-1]   0 .. 1 0:0-1 field
n[0]   0 .. 1 2:2-0 field
e[-1,0]   0 .. 1 3:3-0 field
s[0,1]   0 .. 1 1:1-0 field
s[0]   0 .. 1 1:1-0 field
w[1,0]   0 .. 1 0:0-0 field
e[0]   0 .. 1 3:3-0 field
w[0]   0 .. 1 2:2-0 field
```

B.3 if_test.t

This is a test case written to show the output of nested conditionals.

```
/* test case to show behavior of nested condtionals */

{
c 0..3;
}

{
  sum = c[1] + c[2] ;

  if (sum < 1+1) {
    if (sum > 3) {
      if (c > 1) {
c[0] = 0;
      }
    }
```

```
    c[0] = 2;
  }
  c = c + 1;
}
```

B.3.1 thesis -f if_test.t

This is the compiled output of a test case with nested conditionals.

```
dt output
 final code after regrouping:
-------------------------group-
ins: c
live_before: t4 t6 c
  comment:  copy step
  if: 1  c[2] = c
  comment:  copy step
  if: 1  c[1] = c
outs: c[1] c[2]
live_after: t4 t6 c
-------------------------group-
ins: t4 t6 c[0] c[1] c[2]
live_before: t4 t6 c
  if: 1  t1 = c[1] + c[2]
  if: 1  sum = t1
  if: 1  t2 = sum < 2
  if: 1  t3 = t2 && 1
  if: t3  t4 = sum > 3
  if: 1  t5 = t4 && t3
  if: t5  t6 = c[0] > 1
  if: 1  t7 = t6 && t5
  if: t7  c[0] = 0
  if: t3  c[0] = 2
  if: 1  t8 = c[0] + 1
  if: 1  c[0] = t8
outs: c[0]
live_after: t4 t6 c

variables:
c   0 .. 3 0:1-0 field
t1   0 .. 6 0:2-0
sum   0 .. 6 0:2-0
```

```
t2   0 .. 1 0:0-0
t3   0 .. 1 0:0-0
t4   0 .. 1 3:3-0
t5   0 .. 1 0:0-0
t6   0 .. 1 2:2-0
t7   0 .. 1 0:0-0
t8   1 .. 4 0:1-0

shadows:
c[0]   0 .. 3 0:1-0 field
c[1]   0 .. 3 6:7-0 field
c[2]   0 .. 3 4:5-0 field
```

Bibliography

[1] R. Agin, M. Biafore, R. D'Souza, S. Ganguli, H. Gilliam, N. Margolus, P. Pierini, J. Quick, D. Risacher, and M. Smith. An Early Sampler of CAM-8 Applications. Technical Report LCS-TM-513, MIT, Lab. for Comp. Sci., October 1994.

[2] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Priciples, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1988.

[3] Bruce Boghosian, Jeffrey Yepez, Francis J. Alexander, and Norman Margolus. Integer Lattice Gases. *Phys. Rev. E.*, to be printed in 1996.

[4] Arthur Burks, editor. *Essays on Cellular Automata.* Univ. Ill. Press, 1970.

[5] D. Farmer, T. Toffoli, and S. Wolfram, editors. *Cellular Automata.* North-Holland, 1984.

[6] J. Hardy, O. de Pazzis, and Y. Pomeau. Molecular dynamics of a classical lattice gas: transport properties and time correlation functions. *Phys. Rev.*, A13:1949–1960, 1976.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[8] Norman Margolus. CAM-8: a computer architecture based on cellular automata. In A. Lawniczak and R. Kapral, editors, *Proc. of the 1993 Workshop on Pattern Formation and Lattice-Gas Automata.* American Mathematical Society, 1996.

[9] Norman H. Margolus. *Physics and Computation.* PhD thesis, Massachusetts Institute of Technology, May 1987.

[10] Milan Shah. An optimized CAM-8 simulator for the SPARC architecture. M.S. thesis, MIT, EECS, 1992.

[11] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling.* MIT Press, Cambridge, MA, 1987.

[12] John von Neumann. *Theory of Self-Reproducing Automata.* Univ. Ill. Press, 1966.