An Optimized CAM-8 Simulator for the SPARC Architecture

by

Milan M. Shah

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 1992

© Milan M. Shah, 1992. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature Redacted

Author ......................................................
Department of Electrical Engineering and Computer Science

Signature Redacted

Certified by ......................................................
Norman Margolus
Research Scientist
Thesis Supervisor

Signature Redacted

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

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available.

Thank you.

Despite pagination irregularities, this is the most complete copy available.

No Page numbered 2
An Optimized CAM-8 Simulator for the SPARC Architecture

by

Milan M. Shah

Submitted to the Department of Electrical Engineering and Computer Science on May 8, 1992, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

Abstract

CAM-8 is a massively parallel cellular automata machine being developed by the Information Mechanics group at MIT. At the heart of the CAM-8 architecture is a pointer-based relative data movement technique that is well suited not only for efficient hardware realization, but also to efficient software emulation of the architecture. Here, a CAM-8 simulator, running on Sun Microsystem's SPARC architecture, is constructed. The simulator accepts CAM-8 instruction streams and compiles them into optimized SPARC instruction streams; it then immediately executes the compiled instructions to simulate the behavior of the actual CAM-8 hardware. This thesis is about the design and construction of such a CAM-8 simulator. The process of deducing, at run time, an optimal emulation algorithm, on-the-fly compiling of CAM-8 instruction streams into a SPARC-instructions realization of the emulation algorithm using a single-pass compilation phase, using incremental compilation for efficiency, and the design of a simple and efficient register allocation algorithm are the subject of this thesis. Several sample experiments representative of the type of experiments that will be targeted to the CAM-8 hardware are run on the CAM-8 simulator. The performance of the simulator on these experiments is analyzed on two fronts — the SPARC code generated by the simulator is analyzed to gain insights into resource usage characteristics of the code, and the run time of these experiments is compared against those of other cellular automata software available for the SPARC. The simulator is found to meet initial performance expectations and is expected to meet its goal as a suitable platform for developing and prototyping experiments targeted to the CAM-8 hardware.

Thesis Supervisor: Norman Margolus
Title: Research Scientist
Acknowledgments

I would like to thank my advisor, Norman Margolus, for his infinite patience, guidance, and encouragement. Working with him, and Tommaso Toffoli, in the Information Mechanics group has been a truly enriching and educational experience. Thank you for providing me with the privilege.

My thanks to Mike Biafore, Mark Smith, and the other members of the Information Mechanics group for their wonderful humor and all the help they provided me to make this thesis a reality. Without their help this thesis would have taken twice as long and would have been no fun.

My deepest gratitude to my aunt and uncle, Anita and Dilip Shah, and my extended family whose unusual generosity and constant encouragement made my graduate studies possible and worthwhile.

Lastly, and certainly not the proverbial least, I would like to thank my parents, Meera and Mahendra Shah, for their incredible love, understanding, and support. I am fortunate to have the two best parents in existence.
## Contents

1 Introduction .................................................. 8

2 Overview of the CAM-8 Architecture ....................... 10
   2.1 Cellular Automata ........................................... 10
   2.2 The CAM-8 Architecture ...................................... 11
      2.2.1 Scanning ...................................................... 13
      2.2.2 Kicking ....................................................... 14
      2.2.3 Dimensions ................................................. 16
      2.2.4 Sources and Destinations .............................. 16
      2.2.5 Permuting the Scan ....................................... 21
      2.2.6 Summary .................................................... 23

3 Motivations for an Optimizing CAM-8 simulator ......... 26
   3.1 Organization of a CAM-8 simulator ...................... 27
   3.2 Scan Optimizations .......................................... 29
   3.3 Reducing the effective number of planes ................ 30
   3.4 Source and Destination optimizations ................... 32
   3.5 Fast Dimensions ............................................. 32

4 Design of the CAM-8 Simulator ............................... 34
   4.1 Basic Design .................................................. 35
   4.2 Detecting the Optimizations .............................. 37
   4.3 Summary ....................................................... 38
# Code Layout

## 5.1 CAMState — The Global CAM-8 State Data Structure

## 5.2 Scan Code Components

---

# Intermediate-Code, Variables, and Symbol Table

## 6.1 Variables

## 6.2 Symbol Table

## 6.3 Intermediate-Code

### 6.3.1 Operand Data Types

### 6.3.2 Arithmetic and Logic Instructions

### 6.3.3 Code Labels and Branch Instructions

### 6.3.4 Load-Store Instructions

### 6.3.5 Procedure Entry, Exit and Call Instructions

### 6.3.6 Basic Block Instructions

## 6.4 Stack Organization and Register Usage Conventions

## 6.5 Incremental Compilation

## 6.6 Summary

---

# Register Allocation and Target Code Generation

## 7.1 Requirements

## 7.2 Interface

## 7.3 Priority-based Allocation

## 7.4 Implementation

---

# Measurements

## 8.1 Objectives

## 8.2 Method

## 8.3 Compile vs Execution Times

## 8.4 Profile of the Compile Phase

## 8.5 Incremental Compilation

## 8.6 Performance of Priority-based Register Allocation Strategy

---
8.7 Performance relative to Generalized Scan Algorithm ........................................... 69
8.8 Performance Comparison with Cellsim ................................................................. 70

9 Conclusions ............................................................................................................... 73

A Interface to CAM-Forth and X Windows ................................................................. 75
   A.1 Host Interface Registers ....................................................................................... 77
   A.2 STEP Instruction ................................................................................................. 79
   A.3 Interfacing to CAM-8 Forth ............................................................................... 80
   A.4 X interface for the CAM-8 Simulator ................................................................. 81

B CAM-8 Simulator Code Modules ................................................................................ 83
   B.1 Main .................................................................................................................... 83
   B.2 Hinterface — The Host Interface Module ......................................................... 84
   B.3 Steplist — The STEPList Execution Module ..................................................... 84
   B.4 Compiler — The Scan-code Compiler ............................................................... 85
   B.5 Icode — The Intermediate-code Module .......................................................... 85
   B.6 Regmgr — The Register Allocator Module ....................................................... 85
   B.7 SPARCAsm — The SPARC Assembler Module ............................................... 86
   B.8 Utils — The Utility Module ............................................................................... 86
   B.9 Heap — The Heap Manager Module .................................................................. 86
   B.10 Xinterface — The X-window Interface Module ............................................... 87
   B.11 Memmgr — The Memory Manager Module ..................................................... 88
   B.12 Errmgr — The Error Manager ......................................................................... 88
   B.13 AS - The Intermediate-code Assembler ......................................................... 89
Chapter 1

Introduction

CAM-8 is a massively parallel cellular automata machine being constructed by the Information Mechanics group at MIT. It offers practically unlimited expandability in the three physical dimensions, a high degree of programmability, interactive operation, and built-in data analysis capabilities. Central to the architecture of CAM-8 is a pointer-based relative data movement technique that is well suited not only to efficient hardware realization but also to efficient software emulation: CAM-8 provides a "virtual machine" model that can be implemented efficiently on a variety of computers.

By providing efficient and faithful simulation of the full hardware functionality, the efforts of scientists and engineers all over the world in developing and documenting a CAM language and an application development and analysis environment will be leveraged: anyone with access to an appropriate workstation (a much more likely event than access to the actual CAM-8 hardware!) will be able to develop and use CA applications in a format suitable for high-performance execution on a CAM machine. CA experiments will become useful to a much wider audience.

In order to be usable as a interactive platform for development of CA experiments, the CAM-8 simulator must use efficient algorithms to emulate the CAM-8 architecture. Since CAM-8 has a very large number of operation modes, it would be next to impossible to code the necessary large set of efficient algorithms for all, or even a large fraction, of the modes of CAM-8 operation.
In this thesis, a CAM-8 simulator is built with a design similar to that of an optimizing CA compiler for the SPARCStation, translating machine-language programs written for CAM-8 into native code for the SPARC chip. Such an approach of incremental compilation and execution is efficient as long as the compile time is much shorter than the execution time. Since CAM experiments typically use the scan-program for many thousands of iterations before changing the scan, compile times are likely to be small compared to execution times.

The subject of this thesis is the design, construction, and performance measurement and design validation of such a CAM-8 simulator. The rest of this thesis is organized as follows:

- Chapter 2 describes in some detail the CAM-8 architecture and what is involved in simulating it on a conventional workstation.

- Chapter 3 introduces the notion of using optimized scan algorithms to emulate the different CAM-8 operating modes.

- Chapters 4, 5, 6 and 7 describe in detail the design of the CAM-8 simulator and how it systematically derives and generates, at run time, the required optimized scan algorithm.

- Chapter 8 tabulates performance data that validates the design of the simulator, and proves its suitability for general purpose use as a platform for developing CA experiments.
Chapter 2

Overview of the CAM-8 Architecture

2.1 Cellular Automata

Cellular Automata, first conceived by the mathematician John von Neumann to model the behavior of complex, extended systems [16], are systems consisting of an extended space of identical finite automata. Each finite automaton normally has a relatively small amount of state (small compared to the number of states a modern day sequential computer can have). Each automaton transitions to its new state at every time step according to a fixed rule depending on its previous state and the states of its neighbors. Many different kinds of neighborhoods are possible—for example, in a two dimensional array of finite automata, the eight automata surrounding a particular automaton could be that automaton’s neighborhood. Alternately, the neighborhood could consist of just the automata to the north and east. Each point in space along with the finite automaton at that point is called a cell or a site. Cellular automata, therefore, are extended spaces of cells.

Cellular automata can be characterized by three features - first, they are inherently parallel, because each finite automaton can update its state in parallel with all the others in the space; secondly, cellular automata are uniform, since each finite automaton follows the same rules while updating its state; and lastly, cellular automata
are *local* as typically only a fixed and small set of neighboring cells can affect the state of a particular finite automaton.

These three issues, namely *parallelism, uniformity, and locality* are central to many computer architectures today. Cellular Automata architectures embody the finest grain realizations of these parameters. Certain applications, like physical modeling, image processing, and even VLSI circuit simulation can be very naturally mapped on to parallel architectures that have a very fine grain of uniformity and locality. Cellular automata machines (CAM) are very well suited for such applications [5].

### 2.2 The CAM-8 Architecture

CAM-8 is an implementation of a *partitioning cellular automaton*. Partitioning cellular automata (PCA) partition the space into individual sites, i.e., a particular site’s state is isolated and is not allowed to influence the new state of any other site. In some sense, a site is not allowed to have any neighborhood at all, and its new state is a function merely of its old state. This kind of arrangement is very convenient from a hardware implementation point of view, since to update a site’s state at every time step, the hardware needs to look at only that site’s current state. Also, site states need not be double buffered, since a particular site’s state information cannot influence any other site’s new state, and therefore can be updated completely independently of all other site updates. In fact, since a site’s state is needed to update only its own state, all the sites can be updated in parallel, or in any desirable order, without altering the final outcome.

This might seem like a very severe restriction on the cellular automata model, and it is. This restriction is eliminated, however, by adding the capability to transfer part of the state of one site to another site, thereby letting one site influence the state of another.

PCA therefore operate in two alternating stages—*Data Update*, during which they partition the entire space into sites, and update the state of all the sites in parallel, and *Data Transport*, when every site’s state is separated into bit-planes, and the
bit-planes are moved with respect to one another.

The HPP Gas model \cite{9} will serve to illustrate the difference between CA and PCA. In this model of fluids, the physical system is modeled as a two dimensional space with particles moving in one of four directions (North, South, East, and West) at unit speed. When particles collide head on, they emerge from the collision travelling in a direction perpendicular to their original direction. For example, after a collision between a particle travelling east and one travelling west, the system will have two particles, one moving north and the other moving south. If there is no collision or if the collision is not head on, the particles continue moving in their original directions.

Implementing such a system using a regular cellular automaton consists of updating every site's state depending on the state of its north, south, east, and west neighbors. Since every site state is used to update the states of four other sites, the state information must be double buffered.

Simulating the HPP gas model on a PCA consists of using 4 bits of state information per site; each bit of state indicates the presence or absence of a particle moving in a particular direction at that site. For example, bit 0 could indicate the presence or absence of a particle moving north, and so on. During the data update stage, each site's 4-bit state gives complete information about particles moving in each direction at that site, so this state information is enough to update the site to its new state. During the data transport stage, the 4 bits of a site's state are separated and moved to the appropriate neighbor. For example, bit 0 of a site's state (which encodes the absence or presence of a particle moving north at that site) is moved to the site's north neighbor, bit 1 is moved to the site's east neighbor, and so on. After this data transport is carried out for all sites, the PCA is ready to carry out another data update operation.

The CAM-8 architecture efficiently implements both the data update and the data transport operations.

Since CAM-8 is a fairly powerful architecture, its architecture is described in steps, starting from its most basic features and progressing to its enhanced capabilities. The description here is given from the point of view of the design of a software simulator.
of CAM-8; important features of CAM-8, like the mechanisms for connecting multiple CAM-8 modules, are not described here since they do not affect the software simulator of CAM-8. A full description of the hardware architecture is given in [15].

The basic building block of the CAM-8 architecture is the CAM Module. Each module is responsible for maintaining and updating the states of a sector of the total cell space. A sector within a module has up to 4 mega-cells, (or 16 mega-cells if the appropriate memory chips are used), with each cell containing a maximum of 16 bits of state information. In the following discussion, assume that each module has 4 mega-cells.

2.2.1 Scanning

The hardware is organized as 16 separate bit-planes (or, more generally, hyperplanes); each plane stores exactly 1 bit of state information of a particular cell, i.e., the 16 bits of state of every cell are distributed across the 16 bit-planes. Controlling each of the 16 planes are 16 identical STEP chips. Together, the 16 STEP chips are responsible for updating the state of all of the 4 mega-cells belonging to that module; thus the updates of the 4 mega-cells within a module occur sequentially. The process of updating the module's sector once is called a scan; every STEP chip maintains a Scan Index Register or SIR to perform a scan. In the most elementary case, a scan is performed by first assembling the 16 bits of a cell's state — each STEP chip contributes one bit of the state from its bit-plane — using this 16 bit value as an index into a lookup table or LUT, extracting a new 16 bit value from the LUT, and then storing the 16 bits of the new state back into the original position.1 After a single site is updated, all the STEP chips increment their SIR. Each STEP chip also has an End-of-Scan Pointer or ESCP; this field has a value from 0 to 23, indicating the bit of the SIR that is to serve as the most significant bit during the scan. When there is a carry overflow from this bit, the end-of-scan condition is said to occur, and the

---

1Because of the partitioning, no double buffering is required to keep the simulation of fine grained parallelism from confusing old and new states. Also, the updating is pipelined to give one updated cell per clock.
scan is completed. By appropriately initializing the SIR and the ESCP, it is possible to update only a part of the sector belonging to the module. This scanning operation is illustrated in figure 2-1.

2.2.2 Kicking

To facilitate the movement of bit-planes (hyperplanes) relative to one another so they can interact, CAM-8 provides a powerful data-movement facility whereby the contents of any bit-plane can be moved relative to the contents of the other bit-planes. This data-movement facility is called kicking. For example, suppose one wanted to shift
all the bits in layer 0 one position down so that what was originally bit 0 of cell 5 becomes bit 0 of cell 4, bit 0 of cell 6 becomes bit 0 of cell 5 and so on. Bits at the boundary get wrapped around to the opposite boundary. (i.e., in this case, bit 0 would shift to bit 4 million, assuming there are 4 million bits per plane). This is synonymous to thinking of bits on plane 0 forming one layer, and cells on planes 1-15 forming another; shifting plane 0 data one bit down causes site $x$ to transmit its bit 0 state to site $x - 1$ and to inherit its new bit 0 state from site $x + 1$.

Rather than physically moving the bits around, kicking is done on CAM-8 by maintaining an offset register on every STEP chip. The offset register contains the address of the logical origin of that layer, i.e., when the data for cell 0 is being assembled, all STEP chips provide the bit pointed to by their respective offset registers. So, kicking all bits of bit-plane 0 down by 1 is simply a matter of adding 1 to the
offset register of bit-plane 0. One kicks a bit-plane by writing to a STEP chip’s Kick register, which causes the written value to be subtracted from that STEP chip’s offset register.

Figure 2-2 shows how the offset register affects the process of updating a site.

2.2.3 Dimensions

So far, the “bit-planes” have been assumed to be 1 dimensional. Section 2.2.2 talked about ‘shifting’ all bits of a bit-plane in one direction. In general, however, CAM-8 supports a programmable-dimension space. Every STEP chip has a Dimension Cut Mask Register, or DCMR, that tells it how to interpret the contents of the offset and kick registers as n-dimensional quantities. The DCMR simply contains a 1 in the bit positions that correspond to the MSB’s of each dimension; i.e., the DCMR segments the contents of the offset and kick registers into n-dimensional vector quantities, each component of which is interpreted as a 2’s complement number.

For example, in figure 2-3, we see that the lowest, or X, dimension extends for 5 bits, and the next higher dimension extends for 5 bits; the space is organized into a 32 by 32 2D array of cells. One important consequence of this is the way it affects wrap-around when kicking; for example, with the 32 by 32 configuration, kicking by 1 in the X dimension will cause bit 30 to shift to bit 31, and bit 31 to shift to bit 0 (and not bit 32 as would be the case for a 1D configuration). Since the offset register can be 24 bits wide (for a 16 mega-cell sector), CAM-8 can support up to a 24 dimensional space.

2.2.4 Sources and Destinations

CAM-8 would be limited if all it could do was use the cell state (also referred to as site data) as an index into a LUT to retrieve its new state. While this is uniform, it can be too fine a grain of uniformity. For example, to use a checkerboard type of update rule, the next state of a cell would depend on its location within the space as well as on its previous state. As another example, one might want to initialize the
Figure 2-3: Multidimensional Cell Spaces in CAM-8
cell space to some known initial configuration.

While the most common scanning mode of CAM-8 will have the new cell state retrieved from the LUT using the old state as an index, CAM-8 provides five other sources for the new cell state. Any one of these five sources can be combined with the old cell state to compute the new state. Four of the possible sources are described below; the fifth source is a special case and is described later.

- **Unglued Data** — What has been labeled as "LUT Address" in figures 2-1, 2-2, and 2-3, is actually just the Unglued Data source. LUT Address is just one of the possible destinations that can be a function of this source.

The CAM-8 hardware architecture allows for the connection of multiple CAM-8 modules along any of the three physical dimensions. The entire cell space then consists of logically and physically connected sectors spread out over as many CAM-8 modules. When one module kicks site data along a sector boundary, that data needs to be transmitted to the neighboring module to maintain the abstraction of an extended cell space. (Of course, if there is just one module, that data will simply be transmitted to the opposite edge of the same module.) Kicked data that maintains this sector-to-sector continuity is called Glued Data. If this continuity is broken at the sector boundaries, and data along one edge of a sector boundary is simply wrapped around to the opposite edge of the same sector, that data is called Unglued Data. Since the CAM-8 simulator only simulates one module of the CAM-8 hardware, Glued Data is the same as Unglued Data and they are both names for bit-plane data kicked by the correct amount.

- **Host** — When the host computer wants to download cell states directly, it writes to the Scan IO Register, or SIOR. Writing to the SIOR causes the cell pointed to by the Scan Index Register (SIR) (via, of course, the offset adjustment), to be set to the value written to the SIOR. The SIR then increments, thereby allowing the host to quickly set a large segment of space to some initial value. When data written to the SIOR is used as a source, the selected source is called
Host.

- **Flywheel** — This is a mechanism that allows a data acquisition system to directly place data into cell space. This source is of no relevance to a software emulator of CAM-8.

- **LUT** — The value retrieved by accessing the lookup table can be used as a source. In fact, it is expected to be the most frequently used source.

In addition to updating the cells to their new state, CAM-8 also provides other destinations for data computed from the sources. The cell is actually just one of several destinations that can be the target of a computed value. The other destinations are described below.

- **Lookup Table Address**, or LUT Address — 16 bits contributed by the 16 STEP chips is used as an index into the LUT to produce the LUT source described above. The index is called the LUT Address destination.

  There is one exception to the above description of sources. Using the LUT value as a source to itself would lead to a non-sensical configuration; instead, one of the *Scan Index Register*’s 24 bits is used as the source. The bit selected is determined by the every STEP chip’s 5 bit *Scan Index Bit Select Register* or SIBSR. In addition, the SIBSR might just select the constant 1 or 0 value instead of selecting one of the SIR bits. The SIR is the fifth source mentioned above. It serves as a source only to the LUT Address and the Flywheel Output destinations.

- **Flywheel Output** — This is a mechanism to output data to some external hardware device, and is of no concern to the software emulator.

- **Site Data** — Data written to the cell state (i.e., written to the bit-planes) is said to go to the site data destination.

- **Event Count** — Every STEP chip maintains an event count register which can serve as a destination. Instead of getting written, data gets added to the event...
count register.

- **Display** — This is the mechanism used to transfer cell state data back to the host computer, presumably, but not necessarily, for display purposes. When the host reads the SIOR, the SIOR serves as a destination. The value computed for this destination is transferred to the host. When the SIOR is read, the value is transferred, and the SIR is incremented, so the host computer can read the cell states of a large number of cells by issuing consecutive reads to the SIOR register.

Finally, the selected source is combined with the Glued Data (which, as mentioned above, is the same as unglued data for a single-module CAM-8 and for the software simulator) by a two-input, one-output boolean function. There are 16 such functions, and any one of them can be specified.

Every STEP chip has 5 Destination Select registers, which selects one of the 4 available sources, and 5 corresponding Destination Map registers, which selects one of 16 functions to combine the selected source and the glued data to produce the
new value for the destination. Figure 2-4 illustrates clearly how the sources and destinations are connected.

### 2.2.5 Permuting the Scan

Since the SIR is a simple auto-incrementing counter, it would appear that the scan order is fixed—all the cells forming one line of the lowest dimension get updated in order, and lines forming the next higher dimension get updated in order, and so on. To allow the user to control the order in which the cells are updated, every STEP chip has a Scan Index Permutation Register, or SIPR, which tells it how to permute the SIR to obtain the Site Address of the cell to be updated. It is, in fact, the site address, (and not the scan index register as was shown in Figure 2-1), which gets added to the offset to obtain the DRAM bit address of the bit to be presented as the unglued data.

Scan permutation is illustrated in Figure 2-5.

The complete update mechanism, illustrating all the parts described above, is
Figure 2-6: CAM-8 Architecture
shown in Figure 2-6.

2.2.6 Summary

Tables 2.1 and 2.2 below summarize all the registers described above that control the CAM-8 module. Some registers, not mentioned so far, are also included in the table. Their function is fairly straightforward and is described in the table. The table is the starting point for the design of the CAM-8 simulator, for these registers will directly control the actions taken by the simulator to emulate the CAM-8 module. A more complete description is given in [15].
Since CAM-8 is an extensible architecture, allowing identical CAM-8 modules to be cascaded in any of the three dimensions, the host computer needs a mechanism to select a particular or group of modules to communicate with. This register serves as a selection mechanism. Since it emulates a single module, the CAM-8 simulator merely warns the user if the user tries to communicate with it without first setting the MSR correctly.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Module Select Register (MSR)         | Writing non-zero values to each of the following fields initiates an action.  

- **Scan Sync Mode**: Indicates whether scan is to be performed.  
- **Run Type**: Indicates whether the Display Destination output is to be captured to the external frame buffer.  
- **Event Count Transfer**: when set high, causes the STEP chip's internal Event Counters to be transferred to a register which can then be read by the host; it also clears the Event Counters.  
- **Repeat Previous Kick**: bit causes the contents of the Kick registers to be subtracted from the Offset registers.  
- **Active LUT Toggle**: switches the roles of active and inactive LUT between the two LUTs (look-up tables) that each module is equipped with. |
| Run Mode Register (RMR)              | Writing to this register causes the written value to be subtracted from the Offset register, thus causing a plane to be “kicked”. The sign and kick max fields in the hardware registers are used for multimodule configurations. They are ignored in the software simulator. |
| Kick Register                        | As described in Section 2.2.4, one of the 24 bits of the Scan Index Register can be used as a source for the LUT Address Destination. This field selects which bit. |
| Scan Index Bit Select Register (SIBSR) | These registers determine how to compute the value for each of the 5 possible destinations. A 2-bit Select field selects one of the 4 possible sources. The selected source is then combined with Glued Data using a 2-input 1-output boolean function. A 4 bit Map field selects one of the 16 possible functions. |
| LUT Address Source Select (LASR) Site Data Source Select (SDSR) Event Count Source Select (ECSR) Display Source Select (DSR) | The host reads this register to read the accumulated Event Count. The Event Count must first be transferred to the ECR. (see RMR). |
| Event Count Register (ECR)           |                                                                                                                                                                                                             |

Table 2.1: CAM-8 Registers
<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT Index Register (LIR)</td>
<td>Autoincrementing indexe into the inactive LUT for IO between the host and the inactive LUT.</td>
</tr>
<tr>
<td>LUT Index Permutation Register (LIPR)</td>
<td>Permutes the LIR before it is used as a pointer into the inactive LUT.</td>
</tr>
<tr>
<td>LUT IO Register (LIOR)</td>
<td>Serves as a one-entry window into the inactive LUT. The entry selected is that pointed to by the LIR after being permuted as per LIPR. Reading and writing this register will alter the entry in the LUT.</td>
</tr>
<tr>
<td>Scan Index Register (SIR)</td>
<td>The auto-increment counter that serves as the primary selector of the cell that will be used as the Unglued Data (and Glued Data) source and the Site Data Destination during the scan.</td>
</tr>
<tr>
<td>Scan Index Permutation Register (SIPR)</td>
<td>Permutes the SIR to obtain the Site Address of the cell that is to be used as the Unglued / Glued Data source and the Site Data Destination.</td>
</tr>
<tr>
<td>Scan IO Register (SIOR)</td>
<td>If written to, serves as the Host source. If read from, it serves as the Display destination. Reading or writing the SIOR will cause the SIR to increment, so a large number of cells can be initialized or read by successively writing or reading the SIOR. (It is, in fact, possible that the Host source is selected while the SIOR is read. This is a non-sensical configuration).</td>
</tr>
<tr>
<td>Scan Format Register (SFR)</td>
<td>The End-of-Scan Pointer selects a bit of the SIR; a carry generated out of this bit (not at the bit!) is used as the End-of-scan event and concludes a scan initiated by writing a non-zero value to the RMR's Scan Sync Mode field.</td>
</tr>
<tr>
<td>Offset Register</td>
<td>A per bit-plane (i.e., per STEP chip) register that contains the DRAM Address of the bit that is the origin of that bit-plane.</td>
</tr>
<tr>
<td>Dimension Cut Mask Register (DCMR)</td>
<td>Indicates the locations of each of the MSB's of the n components of the Offset and Kick registers. It divides the Offset and Kick registers into an n-dimensional vector.</td>
</tr>
<tr>
<td>Hardware Environment Register (HER)</td>
<td>The Active LUT Select field of this register indicates which of the two LUTS, LUT 0 or LUT 1, is currently the active one. One writes to this field to force a specific LUT to become the active one. The Active LUT Toggle field of the RMR can be used to switch from one LUT to the other, but not to select a particular one.</td>
</tr>
</tbody>
</table>

Table 2.2: CAM-8 Registers
Chapter 3

Motivations for an Optimizing CAM-8 simulator

The architecture, speed, and scalability of CAM-8 will open up a whole new spectrum of computation to scientists and experimenters. While the whole premise of building the CAM-8 machine is that it will offer price/performance and silicon area/performance ratios far exceeding those that can be extracted from a software emulation running on even the fastest supercomputers, it is still expected that

- A vast majority of the scientists and users who would like to explore CAM-8 (and the new spectrum of computation it offers) will not be able to immediately acquire this supercomputer class machine.

- A group of users who have limited access to a CAM-8 will need an alternate platform for actual development of experiments to run on CAM-8, so that the CAM-8 hardware is used for simulations over large spaces rather than for debugging experiments.

- At least in the near future, general purpose workstations will remain a lot more accessible to scientists than custom CAM-8 hardware.

From the brief overview of the CAM-8 architecture in Chapter 2, it should be clear that the operations of scanning and kicking, the two fundamental aspects of
CAM-8 that make it a partitioning cellular automaton, can easily be emulated on a conventional workstation. Moreover, the pointer-based mechanism used for kicking in CAM-8 is suitable for an efficient emulation of the CAM-8 architecture. If one were to build a CAM-8 simulator that ran on a conventional workstation and achieved a scan rate of a few frames per second, the simulator could be effectively used for prototyping experiments for CAM-8. For the simulator to be a useful prototyping tool, it must have a high performance. Typically, users are expected to run hundreds of scans over cell-spaces that, at the low end, are in the vicinity of a few tens of thousands cells. The simulator must be able to perform at least a few scans of these spaces per second to provide any kind of useful feedback to the user.

The major objective of the CAM-8 simulator, apart from a faithful emulation of CAM-8 hardware, is performance.

It would be a simple matter to write a simulator in a language like C; the bit-planes could be simulated by suitably declared arrays, and scans would simply involve coding loops which calculate the site address of the next cell to be updated, assemble all the sources, combine the sources, and update the destinations. While this might suffice, it would only come at a very high cost in efficiency. In this chapter, we examine the principle optimizations that can be performed by a CAM-8 simulator.

3.1 Organization of a CAM-8 simulator

Before looking at optimizations, the basic properties of any CAM-8 simulator are described. These properties will serve as a starting point for thinking about what optimizations are possible.

- The 16 bit planes would be implemented as 16 separate arrays of memory. Since generally to do an update we need all 16 bits of a cell’s state, we might be tempted to store the cell states as 16-bit quantities rather than splitting the bits over 16 different arrays. However, doing this would make kicking, which involves moving a particular bit-plane with respect to all the others, a horrendously difficult task. The benefits of CAM-8’s kick mechanism of simply
maintaining an offset register for every plane would be completely lost.

- Virtually all workstations available today sport a 32-bit architecture. We assume that the platform for the CAM-8 simulator would have a 32-bit word size, 32-bit registers, and support operations on 32-bit quantities.

- Since we will be storing each bit of any particular cell state in a separate memory word, for each bit plane we need two quantities to locate a particular cell’s data bits—a word address pointing to the word containing the cell’s state data, and a bit offset to select one of the 32 bits of the word. The concatenation of the word address with the corresponding bit offset is referred to as the DRAM address. Because each plane has its own offset register, and because plane origins are expected to move with respect to each other, it will almost always be the case that the 16 bit offsets and word addresses of a particular cell’s 16 data bits will be different from one another. However, a particular machine word always contains one bit of data from 32 consecutive cells.

- The disparity between memory speeds and processor speeds is large today and is expected to continue to get worse [11]. Hence it is assumed that trading memory operations for arithmetic and logical operations will continue to result in efficiency and speed gains.

Figure 3-1 illustrates the basic layout in memory of the CAM-8 simulator bitplanes. To update a cell’s state, the simulator first adds the current SIR to each of the 16 offset registers to compute 16 DRAM addresses. Then, it strips out the last 5 bits of the DRAM address to calculate 16 word addresses. The last 5 bits of the DRAM address are the bit offsets into the 32-bit words. The simulator loads each of the 16 words pointed to by the word addresses, and shifts the bits at the bit-offsets to assemble a 16-bit cell state or site data. Once the site data is assembled, it can be combined with the other sources to form the new data for each of the five destinations. The new data can then be stored back. With this general process in mind, we look at some of the possible optimizations.
Figure 3-1: Layout in SPARC Memory Space of a CAM-8 Simulator Bit-Plane

3.2 Scan Optimizations

In general, CAM-8 supports a completely arbitrary scanning order via the Scan Index Permutation Register, SIPR. Most experiments for cellular automata, however, have no particular preference for the scanning order. (This is natural, because in the abstract sense, each cell of a cellular automaton should be getting updated in parallel, and it makes no sense to talk about a scanning order!). Thus, it is expected that on CAM-8 consecutive cells will be updated consecutively most of the time.

Since each memory word holds site data for 32 consecutive sites, and if at least 32 consecutive sites are being updated in order, then we can load the 16 words, update 32 sites, and then store the 16 words back again. Instead of having to do 16 loads and 16 stores per cell update, we need only do 16 loads and 16 stores for every 32 cell updates. We have already noted that memory operations are regarded as expensive. This simple optimization has the potential to speed up scans by as much as a factor of 30.

Another benefit of updating 32 sites in order is that we need not recompute the
DRAM address, word address and bit-offset for every site. The word-address remains the same and the bit-offsets simply increment. For the cost of computing the DRAM address of one cell, we can update 32 cells.

To clarify the optimization, the algorithm in figure 3-2 outlines the steps needed to update a group of 32 consecutive cells. For the sake of simplicity, it is assumed that there are only two bit-planes, so that each cell has only two bits of state; it is also assumed that site data is used as the LUT Address and the LUT entry is used as the new site data. Notice, that in the algorithm, Word 0 and Word 1 are simply shifted in a loop to update consecutive cells. Since Bit Offsets 0 and 1 could be any value from 0 to 31, we would reach the tail end of Word 0 after updating \(31 - \text{Bit Offset}\) cells; at that point we would have to load the word following Word 0. We simply increment the address of Word 0, but we need to make sure that dimension wrap-around occurs at the correct word boundary. So, we increment modulo the length of the lowest, or X, dimension expressed in number of words. The algorithm expects that the length of the lowest dimension is at least 32 sites.

In summary, then, an in-order or sequential scan yields two important benefits: it can sharply reduce the number of loads and stores needed to update a group of cells, and it can reduce the number of next-site-address calculations needed. This optimization alone has the potential to make an optimized simulator run close to 30 times faster than a more straightforward simulator.

### 3.3 Reducing the effective number of planes

Even though CAM-8 supports 16 bit-planes and allows each cell to have 16 bits of state information, many experiments are expected to require fewer than 16 bits of state per cell. It is possible to detect from the information in the 5 Destination Select registers which bits of each destination could possibly suffer a change upon being updated.\(^1\) For example, if the Site Data Select Register for bit 15 selects Site Data

\(^1\)Since the Site Data Destination can also serve as a source, it is necessary to check whether a particular bit of the Site Destination is used as a source for any destination in addition to checking whether that bit can suffer a change during the scan.
DRAM Address 0 = Scan Index Register + Offset 0
DRAM Address 1 = Scan Index Register + Offset 1
Word Address 0 = DRAM Address 0 .SHIFT RIGHT. 5
Word Address 1 = DRAM Address 1 .SHIFT RIGHT. 5
Bit Offset 0 = DRAM Address 0 .AND. .HEX.(1F)
Bit Offset 1 = DRAM Address 1 .AND. .HEX.(1F)

Load Word at Word Address 0
Load Word at Word Address 1
Rotate Word 0 so that Bit Offset 0 is the MSB
Rotate Word 1 so that Bit Offset 1 is the MSB
for (i = 0 to 31 do)
    Shift Word 1 MSB into Hold Register
    Shift Word 0 MSB into Hold Register
    Do lookup
    Shift Bit 1 of lookup value into Word 1 LSB
    Shift Bit 0 of lookup value into Word 0 LSB
    if (((31 - i).EQUALS. Bit Offset 0))
    Store Word 0.
    Load Word at (Word 0 Address + 1 .MOD. X Dimension length)
    endif
    if (((31 - i).EQUALS. Bit Offset 1)
    Store Word 0.
    Load Word at (Word 1 Address + 1 .MOD. X Dimension length)
    endif
end for.

Figure 3-2: Optimized Scan Algorithm
(i.e., Glued Data), as the source, then that bit of Site Data cannot possibly change upon update. There is, in fact, no need to update bit 15 of Site Data at all! Similarly, if the Source Registers for the Event Count destination is set to Constant 0 via the Event Count Map Register, we know that for every cell update, we will simply be adding a constant 0 to the Event Counters, thereby not affecting the Event Counters at all. Again, there would be no need to update the Event Counters.

Detecting how many of the 16 bits of each of the destinations can possibly change can result in the simulator updating only a few of the 16 bits of state per cell. Some very important classes of experiments, like lattice gas rules [14, p. 171-182] use fewer than 16 bits of state per cell. The HPP gas rule only requires 4 bits of state per cell, potentially reducing by 75% the amount of work that a simulator needs to do for every cell update.

3.4 Source and Destination optimizations

It is expected that not all possible sources will be used during every scan; similarly, it is expected that event counts will be monitored only periodically. On those particular scans where the event counts are not needed, the event counters need not be updated at all. Sometimes, especially during Scan IO Reads, where the cell states are being transferred to the host computer, the cell states themselves are not changed. During such scans, we need not compute the value intended for the Site Data destination.

3.5 Fast Dimensions

The contents of the Dimension Cut Mask Register have two implications—first, it means that the Offset and Kick registers need to be manipulated as n-dimensional vector quantities. When subtracting the kick from the offset, we need to break the borrows at the dimension cuts. Though one could certainly code a loop to mask out the bits belonging to a single dimension, do the subtract, and then mask the resulting bits back in, there is a much more efficient way to do this. The algorithm in figure 3-3
Offset Dimension Sign Bits = Offset Register .AND. DCMR
Offset Dimension Values = Offset Register .AND. .BIT-INVERSE.(DCMR)
Kick Dimension Sign Bits = Kick Register .AND. DCMR
Kick Dimension Values = Kick Register .AND. .BIT-INVERSE.(DCMR)

% Now set the MSB's of Offset register of every dimension to 1
% so that there can be no borrow from one dimension to the next
% when we subtract the Kick

Offset Dimension Values = Offset Dimension Values .OR. DCMR
New Offset = Offset Dimension Values - Kick Dimension Values

% Add the old sign bits
Dimension Sign Bits = (Offset Dimension Sign Bits .XNOR.
                        Kick Dimension Sign Bits)
New Offset = New Offset .XOR. Dimension Sign Bits

Figure 3-3: Algorithm to subtract Kick from Offset

subtracts the kick from the offset, breaking the borrows at the dimension cuts.
Chapter 4

Design of the CAM-8 Simulator

Given some of the optimizations described in Chapter 3, the question as to how best to implement them arises. One approach would be to code separate tight loops to do scans for the possible configurations of CAM-8, and conditionally execute the applicable loop at run time. However, there are various drawbacks to this:

- It would be impossible to eliminate a lot of redundant work. For example, referring to the scan algorithm in figure 3-2, we see that after every site update, we need to test against bit offset to see if we have reached the tail end of the word we are updating. This test needs to be performed after every site update, and once for every plane. Since bit offsets change dynamically during run time, it would be impossible for any compiler to unroll the loop and try to eliminate the compares. Yet, since the bit offsets are known just before the loop is executed, and never change during the loop execution, it also becomes known when exactly the conditions are going to evaluate to true. It seems very wasteful to check for a condition inside a loop even though it is fully known when that condition will be true before hand.

- It was mentioned that it will often be the case that not all planes are used. We would like the simulator to update only the relevant planes. To achieve that we would either need 15 otherwise identical loops that update from 1 to 16 planes, or check after every cell update whether a particular plane is relevant.
or not. The first case is not very elegant, and the second case involves a lot of redundant work.

- Once we have loaded all the relevant plane words, we would like for them to be resident (if possible) in machine registers until we have updated the cell at the tail end of those words. It would be difficult to force a compiler to preferentially keep a selected set of variables in machine registers.¹

- To scan, we need to assemble all the sources that are used, need to combine them as specified by the Map registers, and update all the relevant destinations. However, not every source is used in every scan, and not every destination needs to be updated. Again, the most tightly coded loop will have to check, for every cell update, which the relevant sources and destinations are. This checking will amount to a lot of redundant work.

In light of the above limitations, it would be very inefficient to code a group of scan codes at compile time and selectively execute them at run time. This approach, therefore, was discarded in favor of a much more elegant solution.

### 4.1 Basic Design

In order to make use of the mentioned optimizations, CAM-8 simulator is designed like a compiler. It accepts instructions streams targeted to the CAM-8 hardware (in the form of values written to the CAM-8 registers) and compiles them to a SUN SPARC machine-instruction realization of an optimized scan algorithm. It then executes the generated code; executing this code will emulate the CAM-8 hardware in a highly efficient way. Figure 4-1 illustrates the various phases of the CAM-8 simulator.

The benefits of compiling at run time are that loops can be unrolled to get rid of the redundant condition checking, and only the needed sources can be assembled and the relevant destinations can be updated, without having to check after every

¹Register variables in 'C' are merely hints to the compiler; in practice, declaring a large number of register variables fails to keep the variables in registers.
cell update whether particular sources or destinations are relevant. Similarly, the generated code need only contain code to update the relevant planes. The compile phase can control exactly the contents of the registers when the scan code is executed, thereby using its knowledge of the most needed variables to minimize the number of memory operations needed. Lastly, the generated code can be designed to maximize cache hits; such control is difficult to achieve via a general purpose compiler.

Note that the total time taken to execute a user’s instruction to scan a space of cells is composed of two parts – the time taken to compile the user’s instructions into an optimal scan code, and the time taken to execute the scan code. To maximize performance, the compile phase itself has to be efficient and the generated scan code has to be as optimal as possible. Of course, as a larger and larger space is updated, the fraction of time spent compiling becomes smaller, and the benefits of using an optimal scan code becomes greater. In spite of that, the compile phase has to be efficient, or all the performance benefits to be gained by executing an optimized scan code will be lost in generating the optimized scan code.

Figure 4-1: Basic Design of the CAM-8 Simulator
Figure 4-2: CFG Description of CAM-8 operations

4.2 Detecting the Optimizations

The key to systematically build an optimal algorithm for performing a scan of the cell space is to observe that the set of optimal algorithms form a hierarchy which can be described in a form similar to a context free grammar. Figure 4-2 illustrates this idea.

Having described all the possible scan algorithms in this form, it becomes easy to create a Syntax-Directed translation tree for it. Then, a simple $ll(1)$ parsing technique like recursive descent parsing can be employed to traverse down the translation tree and perform the syntax-directed translation [1, 6]. The output of this translation will be an intermediate code representation of the optimal scan algorithm. This representation closely parallels the actual machine-code that will be generated. In fact, the intermediate-code differs from the final-target code mainly in that at the intermediate code level, we have an unlimited number of virtual registers which get resolved to machine registers by a register allocator to produce the final target code.

A representative part of this syntax-directed translation scheme is shown in figure 4-3. The complete translation tree would be needlessly bulky and redundant. To actually traverse the translation tree, the $ll$-parser looks at the appropriate CAM-8 registers.
OPTIMIZED-SCAN → emit(DRAM Addresses = Offsets + SIR)
  emit(Word Addresses = DRAM Addresses ▷ 5 + Plane Base Addresses)
  emit(Bit Offsets = DRAM Addresses & 0x1F)
  emit(# of sweeps = End of Sweep Ptr / Length of X Dimension)
  emit(loop label 1)
OPTIMIZED-SWEEP
  emit(decrement # of sweeps)
  emit(if # of sweeps != 0 goto loop label 1)

Figure 4-3: Syntax-Directed Translation Scheme

For example, to decide how to translate the non-terminal SCAN, it looks at the contents of the SIR and the SIPR to see whether SCAN expands to OPTIMIZED-SCAN or GENERAL-SCAN.

4.3 Summary

The CAM-8 simulator is organized much like a compiler or interpreter. It accepts as input a CAM-8 instruction stream in the form of writes and reads to and from CAM-8 registers. To emulate the CAM-8 hardware in an efficient manner, the simulator compiles the CAM-8 instruction stream to SPARC machine instructions and then executes the compiled code to produce the end results. The possible CAM-8 actions are described in a Context Free Grammar form and a Syntax-Directed Translation Scheme is used to translate the CAM-8 instruction stream to an intermediate-code which closely parallels the target-code and differs mainly in its use of virtual machine registers. A final register allocation phase resolves the virtual registers to machine registers to generate executable target code. This code is then executed to emulate the CAM-8 hardware.
Chapter 5

Code Layout

Chapter 4 showed that the various optimal algorithms to perform a scan of the cell space can be organized as a context free hierarchy. The CAM-8 simulator includes a syntax-directed translator that traverses the parse tree and emits, as output of the translation, the optimal code sequence required to perform the required scan. This code is referred to as the scan code.

In this chapter, two concepts central to the operation of the syntax-directed translator are introduced.

5.1 CAM8State — The Global CAM-8 State Data Structure

To traverse the translation tree, the syntax-directed translator needs to know the complete state of the CAM-8 hardware registers. For example, to decide if Event Count is a relevant destination, the translator needs to know the values in the Event Count Source Registers.

The state of the CAM-8 registers can collectively be interpreted as a program to be executed by CAM-8 (or, in this case, by the CAM-8 simulator).

The state of all the CAM-8 registers is maintained in a large global data structure called CAM8State. The state of this data structure is used to as a ‘program’
specified by the user and is used by the syntax-directed translator to translate this user-specification into an optimal scan code that will execute the user's program. Appendix A describes how exactly the user communicates her program to the CAM-8 simulator. The global state data structure is shown in Figure 5-1.

Most fields have a one-to-one correspondence with CAM-8 registers, and they simply store the most recent value written to the corresponding register by the user. Some fields of this data structure serve different purposes than they do on the CAM-8 hardware. For example, the SIOR field of this data structure actually holds a pointer to a data buffer; this data buffer contains or receives a list of values that were going to be written or read from the CAM-8 hardware SIOR. Similarly, the pnLIOR field contains a pointer to the the inactive LUT. Lastly, there are some fields in this data structure which have no equivalent in the actual hardware. For example, the bReadMode field is true if a Scan IO read is being performed and is false otherwise.

This data structure, along with the contents of the 16 bit-planes, completely stores the entire state of CAM-8 hardware as maintained by the CAM-8 simulator. Controlling and programming the CAM-8 simulator is performed exclusively by changing the state as stored in this data structure.

5.2 Scan Code Components

It follows from Figure 4-2 that the scan code can be logically broken up into component parts rather than a monolithic piece. For example, the block that does the OPTIMIZED-SCAN can be viewed as a procedure which calls another procedure, named OPTIMIZED-SWEEP, which in turn calls the UPDATE-CELL procedure many times. While doing the translation, it is possible, of course, to simply insert in-line all the logical procedures and create one monolithic piece of code that is the complete scan code itself. There are, however, advantages to actually keeping the logical blocks as separate procedures.

• On an architecture like the SPARC, which provides a register window mechanism, if the code is organized as separate procedures, each of which uses its own
/* Definition of struct CAM8State, the CAN module state structure. */

struct tag_Source {
    BYTE bFiller : 2;
    BYTE bMap : 4;
    BYTE bSelect : 2;
};

struct CAM8State {
    BYTE bNSR;
    struct RNR {
        BYTE bFiller : 2;
        BYTE ActiveLUTToggle : 1;
        BYTE RepeatPrevKick : 1;
        BYTE EventCountifer : 1;
        BYTE RunType : 1;
        BYTE ScanSync~ode : 2;
    } RNR;
    LONG32 arlKick [CHIPS.PER.MOD];
    BYTE arsSABSR [CHIPS.PER.MOD];
    struct Source sASLAR [CHIPS.PER.MOD];
    struct Source sFUSR [CHIPS.PER.MOD];
    struct Source sSDSR [CHIPS.PER.MOD];
    struct Source sEDSR [CHIPS.PER.MOD];
    struct Source sBSR [CHIPS.PER.MOD];
    BYTE bShowScan;
    LONG32 lEventCount[CHIPS.PER.MOD];
    INT16 nLUTIndex;
    BYTE arsLIPR [CHIPS.PER.MOD];
    INT16 *pnLIOG;
    LONG32 ISIR;
    LONG32 lSiteAddress;
    BYTE arsSIPR [SCAN_INDEX_LENGTH] [CHIPS.PER.MOD];
    INT16 *pnSIOR;
    BOOL bByteMode;
    BOOL bImmediate;
    BOOL bReadMode;
    struct SFR {
        BYTE bScanMode;
        BYTE bESCP;
        BYTE bESWP;
        BYTE bESTP;
        BYTE bECL;
        BYTE bEBRC;
        BYTE bECL;
    } SFR;
    LONG32 arlroffsets [CHIPS.PER.MOD];
    LONG32 lDCNR;
    INT16 nHER;
};

/* Kick register per chip */
/* LUT address bit select */
/* Flywheel output Source */
/* Site Data Source */
/* Event Count Source */
/* Display Source */
/* Show Scan Register */
/* Event Counts */
/* LUT Index Permutation */
/* Scan Index Registers */
/* Site Address - not always valid! */
/* Scan Index Permutation */
/* Scan Index Permutation */
/* LUT 1/0 */
/* Scan 1/0 - points to TBUS buffer */
/* if byte mode transfer */
/* true if pnSIOR is the data itself */
/* true if Scan 10 read */
/* Scan Format Register */
/* End-of-Scan Pointer */
/* End-of-Repeat Pointer */
/* End-of-Repeat Pointer */
/* Refresh Cycle Length */
/* Sweeps between refresh */
/* Event Counter Length */
/* Offset Register */
/* Dimension Cut Mask */
/* Hardware Environment register */

Figure 5-1: CAM-8 state Global Data structure
register window, one effectively increases the number of machine registers that are available. Many of the optimal scan codes are optimal precisely because they can reuse variables already loaded into registers, thereby saving loads and stores from and to memory. Having more machine registers, which can all be preloaded with needed variables and then used in a tightly coded scan loop, can be a tremendous advantage.

- It makes *incremental compilation* possible. Note that as far as the CAM-8 simulator is concerned, the time it takes to compile a scan code is a component part of the time it takes to perform the scan. If the compiler could detect that a certain code block need not be recompiled, then it could avoid regenerating that block, thereby reducing the amount of work it needs to do.

On an architecture like the SPARC which provides a register window mechanism, one can further differentiate normal procedures (which use their own register window) from *leaf procedures* which use the caller’s register window. A leaf procedure is one which does not itself call other procedures. An advantage of a leaf procedure is that it’s caller can preload the registers with variables that the leaf procedure will need and then call the leaf procedure. The leaf procedure does not have to load its operating set of variables into registers. This is a big advantage if the leaf procedure is to be called many times (as, for example, the Update Cell code is likely to be) from within a loop body. In such a situation, many loads and stores can be saved.

Figure 5-2 illustrates all the different code blocks that the translator can generate, organized as a call tree. Of course, to perform any one given scan not all blocks need to be generated. The grey blocks are leaf procedures that use their caller’s register window.
Figure 5-2: Code layout of Scan Code
Chapter 6

Intermediate-Code, Variables, and Symbol Table

The immediate output of the Syntax-directed translator described in Chapter 4 is a three-address type intermediate code, similar to those found in production compilers. The intermediate code level serves the following functions:

- It allows the translator to work in terms of “variables” instead of concerning itself with the manipulation of actual machine registers. This allows the actual code generation to be separated from code dealing with loading, spilling and storing machine registers.

- It allows the translator to work in terms of meta-instructions; procedure entry instructions, for example, are encapsulated into a single intermediate code instruction. This makes the purpose of the code being generated very clear.

- It creates an interface that is very convenient for porting the CAM-8 simulator to other platforms; only the intermediate code to target code converter needs to be machine-specific; the rest of the CAM-8 simulator can simply be recompiled.
6.1 Variables

The main purpose of the code generated by the compile phase is to update the cell states stored in the bit-planes (and the other destinations) of the CAM-8 simulator. To do the updates, the generated code needs many temporary memory variables to keep track of its own current state. For example, it needs to keep track of the addresses of the 16 bit-plane memory words that it is directly manipulating; it needs loop counters so that it can keep track of how many cells it has already updated; it needs to know the bit-offsets within each plane memory word of the particular cell it is updating; and so on. The register allocator and intermediate-code to target-code translator also needs fixed locations into which it can spill registers if it runs out of machine registers. All these temporary variables have been defined and a complete list provided in Table 6.1. Since this set of variables is static and the size of the set remains the same from one scan to the next (even though not all variables are accessed during every scan), they are stored in an appropriately sized memory block which is created once when the CAM-8 simulator program is initialized. Once this memory block is created, the absolute addresses of all the variables becomes known; the compiled code can use these addresses directly.

Another advantage of storing these variables in this heap-like structure is that access becomes faster. The reason for this is twofold:

- Since the heap size is only a couple of hundred words, chances are very good that the heap will fit into even a small data cache that the workstation may have. It is also likely that accessing any one variable in this heap will bring a large part of the heap into cache, so that accessing subsequent variables is likely to result in a cache-hit, thus reducing the variable access overhead. This would not be the case if the variables were not stored physically close to each other.

- Most current workstations, including the target SPARC architecture, are load-store architectures; they also typically have a fixed instruction-word length of 32 bits. This means that accessing an arbitrary memory location actually takes two instructions - one to load the full 32-bit address into a register, and another to
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HighBitMask</td>
<td>Similar to CAM-8 DCMR, except only has a 1 in the bit position corresponding to the highest dimension cut. This is used after every sweep as a mask to compute the word address of the beginning of the next sweep.</td>
</tr>
<tr>
<td>XHighBitMask</td>
<td>Similar to HighBitMask, but for the lowest dimension. It is used inside a sweep to compute the address of the next word of the sweep.</td>
</tr>
<tr>
<td>XByteLength</td>
<td>Length, in number of bytes, of the length of the smallest dimension. 8 times this many sites will be updated during every sweep.</td>
</tr>
<tr>
<td>XCtr</td>
<td>Loop counter used during sweep. It is initialized to the number of sites that will be updated during the sweep, and then decremented after every update.</td>
</tr>
<tr>
<td>SweepCtr</td>
<td>Loop counter for counting number of sweeps.</td>
</tr>
<tr>
<td>Offsets</td>
<td>A copy of the 16 offset register values. Used for next-site-address calculations.</td>
</tr>
<tr>
<td>BitOffsets</td>
<td>Storage for the bit-offsets into the 32-bit word of the 16 data bits of a site. Bit offsets are counted from MSB of the word; i.e., offset 0 means bit 31, offset 1 means bit 30, and offset 31 means bit 0 of the word.</td>
</tr>
<tr>
<td>Dcmr</td>
<td>Copy of CAM-8 DCMR.</td>
</tr>
<tr>
<td>ScanIndexReg</td>
<td>Copy of CAM-8 SIR. Auto-increments.</td>
</tr>
<tr>
<td>LutAddr</td>
<td>Pointer to start of active Look-up Table.</td>
</tr>
<tr>
<td>BitPlaneAddr</td>
<td>16 pointers to bit-plane bases.</td>
</tr>
<tr>
<td>NextWordAddr</td>
<td>16 Word Addresses into bit-plane arrays.</td>
</tr>
<tr>
<td>PlaneWord</td>
<td>16 locations for spilling bit-plane memory words.</td>
</tr>
<tr>
<td>EventCount</td>
<td>16 Event Counters.</td>
</tr>
<tr>
<td>GlueData</td>
<td>Spill location for assembled (i.e., 16-bit) Glue Data.</td>
</tr>
<tr>
<td>SiteData</td>
<td>Spill location for assembled Site Destination.</td>
</tr>
<tr>
<td>TbusData</td>
<td>Spill location for Host Data.</td>
</tr>
<tr>
<td>LutData</td>
<td>Spill location for value retrieved from LUT.</td>
</tr>
<tr>
<td>TbusAddr</td>
<td>Pointer to buffer containing the Host Data.</td>
</tr>
<tr>
<td>Frame</td>
<td>Pointer to leftmost corner of X-window into which the site data destination is to be written for display purposes.</td>
</tr>
<tr>
<td>ColorMap</td>
<td>Pointer to base of a color map array that converts a 16-bit Site Data value to the 8-bit pixel value that is to be written into the frame buffer.</td>
</tr>
<tr>
<td>UpdateSiteProc</td>
<td>Used to store a pointer to the UPDATE-CELL procedure.</td>
</tr>
<tr>
<td>Tempx</td>
<td>Volatile variables used as scratch values. Their location in heap is never used. However, they are still variables to avoid having special cases for temporary registers.</td>
</tr>
</tbody>
</table>

Table 6.1: Variables used by Scan Code
access the variable stored at that address. (This is necessarily the case because a full 32-bit address cannot be part of a 32-bit instruction!) By keeping all the variables in this heap like structure, we can dedicate a register to always point at the base of this heap; we can then access any variable by specifying an offset (which is only a few bits wide) from this heap base, thus reducing access to 1 machine instruction.

6.2 Symbol Table

To keep track of the variables, the compile phase uses a Symbol Table. The Symbol Table is a very simply data structure — it is simply an array, with one entry per variable. Note that certain variables, like the Next Word Address, are logically 16 element arrays themselves; these variables have 16 entries in the symbol table. Each entry in the symbol table has the following information:

- **Heap Offset**: This always has the offset into the heap at which this particular variable can be found.

- **Register Location**: This is used by the register allocator to identify if this variable is currently in a machine register. A value of 0 indicates that this variable is not in any machine register.

- **Priority**: This field is used by the register allocator as the basis of its allocation strategy. Variable priorities are described in Chapter 7.

- **Name**: This field is how variables are identified at the intermediate code level. Variable names are described more fully below.

In any compiler, the critical aspect of a symbol table is access time. Our current organization makes this easy - each variable is identified by a Name, which is actually just an index into the symbol table pointing to that variable’s entry in the symbol table. Accessing a variable’s symbol table entry can be done in O(1) time.
At the intermediate-code level, a variable is known only by the index of its symbol table entry.

The intermediate-code interface is discussed next.

6.3 Intermediate-Code

The intermediate-code is a three-address code. The central facility that the intermediate-code interface offers is the Emit() function - this function translates an intermediate-code instruction directly to target code, resolving all the operands of the intermediate-code instruction to machine registers or constants. The intermediate-code generated by the syntax directed translator of Chapter 4 is never explicitly stored anywhere. The entire compile phase can therefore run in a single pass, where the syntax directed translator emits intermediate-code instructions which are immediately translated to a set of target machine instructions. A separate pass over an intermediate-code representation is not made to emit target code.

6.3.1 Operand Data Types

The intermediate-code interface recognizes operands of five different data types:

- **Variables** : Variables (represented, as discussed above, by their symbol table indexes), can be used as operands or destinations.

- **Constants** : Any 32 bit signed or unsigned constant can be used as the second operands of appropriate intermediate-code instructions. One need not concern one's self with splitting basic instructions, like ADD, into two instructions if the constant being used is too large to fit in the target machine instruction word.

- **Registers** : intermediate-code recognizes several registers - this includes the Stack Pointer register, the Frame Pointer register, the Return Address register, and the Heap Base register. By far the most frequently accessed one is the Heap Base register, which, as the name implies, is guaranteed to always point to the
base of the heap from where all the variables can be accessed using their offset into the heap.

- **Addresses**: This is an unsigned 32 bit quantity, meant as an operand type for call-subroutine instructions. It would have been possible to use constants as address operands, but having an extra type costs nothing and clarifies intent.

- **Labels**: Labels are obtained by calling the NewLabel() call. Labels are used as markers to points in code; labels are also used as operands to branch instructions.

### 6.3.2 Arithmetic and Logic Instructions

The intermediate-code interface provides a full repertoire of arithmetic and logical instructions that specify two operands and a destination. If one of the operands is a constant, it must be the second operand. The destination operand may not be a constant. Apart from these restrictions, the instructions are completely orthogonal.

### 6.3.3 Code Labels and Branch Instructions

The interface also provides a full set of branch instructions and an ADD_LABEL instruction. Labels are obtained exclusively via calls to NewLabel(). They are inserted just before the first instruction that is to be marked by Emit-ing the ADD_LABEL pseudo instruction. They may be used as operands to branch instructions before they are defined as marks via ADD_LABEL. Thus, forward and backward jumps can be easily implemented. An important limitation of the interface is that it does not automatically fill any delay slots after branch instructions that the target architecture (like SPARC) may allow. It is up to the user of the intermediate-code interface to properly order instructions. Thus, complete machine independence does not exist above the intermediate-code level. Any changes, while porting to architectures that do not have delay slots, however, are expected to be minimal.
6.3.4 Load-Store Instructions

The interface provides different flavors of Load and Store instructions. One can issue `LD_MEM` and `ST_MEM` instructions to bring variables from heap into machine registers; this is convenient to move loads and stores outside of loop bodies. For example, consider:

```
LD_MEM var1
DO WHILE (...
...
ADD var1, 1, var2
...
END WHILE
```

The body of the first loop will execute faster than the second because the second loop body might translate the add instruction to a load of var1 followed by an add, whereas the first code sequence forces the load of var1 out of the loop body.

6.3.5 Procedure Entry, Exit and Call Instructions

The most important pseudo instructions that the intermediate-code level provides are those that serve as checkpoints for register state and facilitate the generation of separate blocks of code. These are now described.

`NewBlock` causes the intermediate-code module to adjust internal pointers that determine where in memory the target instructions being emitted are to be stored. `NewBlock` takes as its single argument a Handle (or pointer to a pointer) to an allocated block of memory; target instructions will be stored into this block of memory in sequential order. `NewBlocks()` can be nested, i.e., one could be building a particular block of instructions, issue another `NewBlock()` and build a different block of instructions, and then return to building the first block by issuing an `EndBlock()` call. `EndBlock()` takes no arguments. Every `NewBlock()` must have a corresponding `EndBlock()`. `NewBlock()` and `EndBlock()` do not affect the state of the machine registers, they simply instruct the intermediate-code module to start emitting instructions.
to a different memory location. The syntax directed translator of Chapter 4 creates separate blocks of code for GENERAL-SCAN, OPTIMIZED-SCAN, OPTIMIZED-SWEEP, UPDATE-CELL, etc. as described in Chapter 5.

The ENTER.PROC pseudo instruction translates to a set of instructions to be that create a new stack frame and do all the necessary work to enter a procedure. On the SPARC architecture, this involves the saving of a register window and the use of a new one; thus, some variables which were in machine registers no longer can be accessed via machine registers, while others can now be accessed via different registers. This work is encapsulated into the ENTER.PROC pseudo instructions. The matching RETURN.PROC instruction reverses the actions of ENTER.PROC. An important invariant is that the state of the machine registers is exactly the same after a RETURN.PROC is issued as it was just before the ENTER.PROC was issued.

The ENTER.LEAF pseudo instruction is used to start a new leaf procedure that uses the caller’s stack frame and, in the case of the SPARC, the caller’s register window as well. This is an important instruction for optimization. If the caller is going to call a leaf routine from inside a tight loop, it can preload the variables the leaf routine is going to need and then call the leaf routine in a loop. Because the leaf routine uses the caller’s register window, it will find its variables already loaded, and will not do any unnecessary loads and stores. The RETURN.LEAF pseudo instruction is used to return from a leaf procedure.

As in the case of ENTER.PROC and RETURN.PROC, the state of the machine registers is guaranteed to be the same after a RETURN.LEAF is issued as it was before the ENTER.LEAF was issued.

6.3.6 Basic Block Instructions

Sometimes, it is necessary to mark the state of the registers at some point inside a block of code and return the state of the machine registers to the previously marked state at some later point. A basic block is a block of code that has exactly one entry point and one exit point; i.e., there are no control-transfer instructions inside the
basic block. While compiling a block of code, the state of the registers needs to be the same after exiting from a basic block as before entering that basic block [1][6, p.555]. For these cases, the intermediate-code level provides the ENTER_BASICBLK and RETURN_BASICBLK instructions. RETURN_BASICBLK is a slight misnomer, because it doesn’t cause any control-transfer instruction to be emitted automatically, as is the case with RETURN_PROC and RETURN_LEAF; it is expected that the instruction issued immediately after a RETURN_BASICBLK will typically be a branch type instruction.

6.4 Stack Organization and Register Usage Conventions

As described in the sections above, the intermediate-code interface encapsulates the entire procedure entry and procedure return code into the two pseudo instructions ENTER_PROC and RETURN_PROC. On the SPARC architecture, the procedure entry code consists of setting up a new stack frame and starting up a new register window.

Figure 6-1 shows the layout of the stack frame; this conforms to the format used
by Sun OS [13]. Note the normally unused space that must be allocated in case the Sun OS needs to spill the register windows.

Figure 6-2 shows the register usage convention. The register usage conventions must also conform to that specified by the Sun OS. It is not possible, for example, to save the Frame Pointer in a fixed location and reclaim use of the Frame Pointer register for other uses, even though the hardware would clearly permit it. The reason for not being able to do this is that the Sun OS expects the marked registers to be always valid. If the Frame Pointer register were being used for other purposes, it is possible that the Sun OS might handle an interrupt at that time and try to access the (invalid) Frame Pointer.

6.5 Incremental Compilation

The explicit mechanisms provided by the intermediate-code interface for checkpointing machine register states enables the CAM-8 simulator’s compile phase to do incremental compiles. For example, it is very frequently the case that the user will set up one type of experiment and then scan the space many times; the only state variables
that change from one scan to the other are the values of the offset registers (which, of course, cause the starting DRAM addresses, word addresses, and bit offsets to change). Certain blocks of code, for example the UPDATE-CELL code, are independent of these state-variables, and need to be recompiled only when the contents of the Destination Source or Destination Map registers changes. The compile phase can determine, at run time, whether UPDATE-CELL needs to be recompiled, and if not, can simply return. This will not affect any subsequent instructions that are emitted, because UPDATE-CELL is designed to be a leaf procedure, and the above mechanism guarantees that the state of the machine registers is exactly the same before and after UPDATE-CELL is compiled; thus, the code generated within UPDATE-CELL does not affect the outcome of subsequent code emission.

6.6 Summary

The intermediate-code level provides a convenient encapsulation of target machine instructions. It provides a logical set of operand types that can be manipulated by users of the interface. It provides intermediate-code to be specified in terms of an unlimited supply of virtual registers. It abstracts away underlying architecture specific features like register-windos. By providing an explicit mechanism for returning machine registers to previously marked states, this interface facilitates incremental compilations. Mechanism has also been provided whereby the intermediate-code generator can have great control of the code sequence and thereby generate as optimal a code as possible, in spite of the fact that it is dealing with an intermediate-code representation.
Chapter 7

Register Allocation and Target Code Generation

The intermediate code interface's Emit() function takes an intermediate code instruction and immediately translates it into target machine instructions. First, it needs to map a pseudo instruction into a correct sequence of machine instructions. For example, a pseudo instruction adding a large constant to a variable and putting the result into another variable might translate into instructions to load the first variable from heap into a register, put the large constant into a machine register, add the two and put the result into a third register, and finally, store the result into the destination variable in heap. While selecting the set of instructions is done by Emit(), the resolution of the operands to registers is done by the Register Allocation module. The design of the register allocation strategy is critical for various reasons:

- In general, register allocation can be viewed as a graph coloring problem [3]. In the extreme, then, the problem is NP-complete. However, in our case we need an extremely fast register allocation strategy, because compile time is a component part of the time it takes to complete a scan as specified by the user, and performance is a key goal of the CAM-8 simulator.

- Due to the single-pass nature of the compile phase, the register allocator does not have the benefit of going over an intermediate code representation to do
liveness analysis, or to build interference graphs etc. Thus, generally well known register allocation techniques, like those described in [4], can not be directly used.

- The register allocator cannot simply play it safe and do a load registers – use registers – flush registers for every intermediate code level instruction (even though, of course, the generated code would be correct). This is because many of the most important optimizations of scan routines are based primarily on the observation that the cost of loads and stores of memory words can be amortized over many, not just one, cell updates. If the register allocator were to naively load and store memory variables, the fundamental reason to generate optimized scan routines would be completely defeated.

- The register allocator needs to provide, in an efficient manner, interfaces to store and restore the existing state of machine registers. On architectures like the SPARC which have register windows, this can involve a fair amount of work. The register allocator must be able to accomplish this efficiently.

- The register allocator must try and achieve as optimal an allocation as possible without having the benefit of having any kind of global perspective.

7.1 Requirements

The basic requirement for any register allocation strategy, of course, is that when presented with a request for a machine register, it must be able to identify one that can be used safely. By safely we mean that by allocating a register for use, the allocator guarantees that the state of a variable already resident in a machine register is not destroyed. If it finds that all the machine registers are currently occupied by a variable, it is responsible for spilling an appropriate one back to heap.

The other requirements for the strategy are speed — the register allocator must not use a significant fraction of compile time; simplicity — the register allocator cannot be so complex that saving and restoring its own state becomes an expensive
operation; and *optimal allocation behavior* — a decision must be made based on *local* information that results in a register allocation that is near optimal from a *global* perspective.

### 7.2 Interface

The interface to the Register Allocator module consists of the GetReg() function, which takes as argument a Variable Name (again, this simply means the index of the variable’s entry in the Symbol Table) and returns the machine register that contains the variable. If the variable is not already present in a register, the allocator loads the register with the variable’s value from the heap. (Except under certain conditions, as clarified in Section 7.3).

The interface also provides a SaveRegisterState() and RestoreRegisterState() interface. SaveRegisterState() pushes a snapshot of the state of all the machine registers on an internal stack, and RestoreRegisterState() pops a snapshot from the internal stack and emits any necessary code to return all machine registers to their state as saved in the snapshot. These two functions are critical for compiling basic blocks.

### 7.3 Priority-based Allocation

The register allocation strategy is derived from the Priority-based algorithm described in [4]. There is an important difference — since the set of variables which can be assigned to registers is fixed and known, the priorities are assigned statically. Every variable in table 6.1 has been assigned a priority which does not change during execution of the CAM-8 simulator; the register allocator does not need to compute the priorities of the variables at run time.

Four different priorities have been defined to permit the register allocator to return as optimal an allocation as possible. They are:

- **Volatile**: The intermediate code generated needs a set of scratch registers to store results of intermediate computations in the process of doing a more in-
olved computation. Frequently, the intermediate result is computed and im-
mediately used, after which it is no longer needed. The register allocator makes
no attempt to refresh the variable’s location in heap with the value in the reg-
ister. Variables marked with Volatile priority are never stored back to heap.
Thus, the value of a volatile variable from one use to the next is not necessarily
related.

Another important invariant is that once a register is allocated to a volatile
variable, the variable is never spilled to reclaim use of that register. This does
not mean that we could end up in a situation where all the registers are being
occupied by volatile variables—this particular situation is avoided by requiring
that the total number of volatile variables available is less than the total number
of available registers. Secondly, when a RETURN_PROC, RETURN_LEAF, or
RETURN_BASICBLK pseudo instructions are issued, they cause the register
states to revert back to the one that existed before the matching ENTER.xxx
instruction. Any volatile variable which was allocated a register between the
ENTER.xxx and the RETURN.xxx instructions is once again marked as not
being in any register, and the register allocated it is recovered.

- **Temp**: Variables with Temp priority are those that retain their value from one
  use to the next. That is, if one basic block assigns a Temp variable a value
  and a subsequent basic block uses that variable, the value of the variable in the
  second basic block is that assigned to it by the first basic block. If all available
  machine registers currently hold a variable, a temp variable is the first candidate
  for spilling. Variables like Next Word Addresses, Bit Offsets, etc., which are
  not very frequently accessed are assigned the Temp priority.

- **Loop Ctr**: Variables used as loop counters are assigned the Loop Ctr priority. It
  is expected that these variables will be accessed at least once for every iteration
  of a loop. Loop Ctr’s are spilled after all Temp variables have been spilled.

- **Global**: Variables marked with Global priority have the highest priority. They
  are spilled only after all Temp and Loop Ctr variables have been spilled. This
priority level is meant for those variables which will be accessed many times during a loop body. For example, during an optimized scan, the words containing the cell data are accessed 32 times for every iteration of OPTIMIZED-SWEEP. These variables are assigned global priorities so that it is very unlikely that they will be spilled to heap.

This priority scheme was arrived upon after analyzing the forms of typical scan algorithms. It was observed that the code being generated to do a scan was most conveniently split up into about a dozen basic blocks. Some of the basic blocks almost exclusively did pre-loop-body work like initializing word addresses and bit offsets, for which they needed a lot of scratch and access-once variables. Other basic blocks were part of loop bodies and typically accessed the same set of variables many times in quick succession. The selected priority scheme automatically gives priority to variables with Global priority, making it very unlikely that these frequently accessed variables will be spilled once they are brought into a register. At the same time, this priority scheme allows for the definition of scratch variables, and no unnecessary loading and storing of these scratch, or volatile, variables is generated.

In practice, this simple, and fast, register allocation scheme is found to work quite well. Profiles of the machine register contents at different points in the code are given in Chapter 8.

7.4 Implementation

The implementation of the register allocation strategy is centered around three main data structures:

- **Symbol Table**: The symbol table was described in Section 6.2. For every variable, it maintains an entry indicating its name, priority, offset within heap, and a register location if the variable is resident in a machine register.

- **Current Window Table**: This is a table that has an entry for every machine register. Each entry stores information on which, if any, variable is currently in
that register, what its priority is, what its name is (the name can be used as an index to the variable’s symbol table entry) and what the offset within heap of that variable is.

- **Free and Spill Queues**: These are a set of 4 FIFO queues. The Free Queue records all the registers that are currently unassigned. The Spill Queues, one each for Temp, Loop Ctr, and Global priority variables, record the variables that are currently in some machine register. If a request for a machine register is made, the variable is not already in a register, and the Free Queue is empty, then the variable at the head of the Temp Queue is spilled back to heap and its register reclaimed. If the Temp Queue is empty, the Loop Ctr Queue is examined, followed by the Global Queue. Since Volatile variables are never spilled, there is no queue for them.

An important consideration is that the state of the register allocator is maintained not only in the symbol table and the current window table, but also in the contents of the spill queues. Thus, when storing and restoring register states, it is important to ensure that the Free and Spill queues are also exactly restored (i.e., not only must they contain the same entries upon restore, but they must also maintain the order of the entries).
Chapter 8

Measurements

8.1 Objectives

The primary goal of building the CAM-8 simulator was performance - the compile-on-the-fly design approach was selected primarily to get higher performance, in terms of cells updated per second. The individual parts of the simulator were themselves designed to be as efficient as possible; for example, the priority-based register allocation scheme was specifically designed to make the target code and the register allocator itself be as efficient as possible. In this chapter, performance of the simulator is measured.

To validate the overall design approach of dynamically compiling the scan algorithm, the performance of the simulator is measured against another CAM-8 simulator, Kensim [12]. Kensim is a straight-forward 'C' emulation of CAM-8. It makes no effort to amortize memory operations over multiple cell updates, nor does it try to deduce the next site address from the previous one without having to perform the entire offset addition for every cell. Kensim does, however, update only those planes which are relevant, but leaves it up to the user to tell it how many planes are relevant. Lastly, it is not a full emulation of CAM-8; it simply uses glue data as an index into the look-up table, and uses the look-up table entry to update the site data. In other words, it emulates the simplified CAM-8 behavior illustrated in Figure 2-1. This simulator was designed as an aid for verification during the actual hardware design
process, and as such, performance was not its main goal. However, comparison with Kensim amply illustrates the performance benefits to be gained by generating and using an optimal scan algorithm instead of using a single, precompiled, generalized scan algorithm.

Compile-time is measured against execution time to see how further work should be directed. Also, the compilation phase is profiled to verify that the performance profile of the component stages of the compiler is flat, and no single stage dominates.

The benefits of doing incremental compilations as described in Section 6.5 versus full compilation is studied to see if it yields any appreciable gains in reducing compile times.

The efficiency of the register allocation strategy is measured by timing CA experiments that require an incrementally increasing number of bit-planes; examining the spill code generated should give a fair estimate of how well the register allocation strategy works.

Lastly, the CAM-8 simulator's performance is measured against other completely dissimilar cellular automata programs to determine the adequacy of the CAM-8 simulator as a general-purpose platform for prototyping cellular automata experiments.

8.2 Method

Tests involving the CAM-8 Simulator and other programs that run on the SPARC-Station were all performed on a Sun SPARCStation 1 with 16 Megabyte of memory under Sun OS 4.1.1. The environment was operating under "typical" load conditions, with the standard daemons and a clock program running in the background, and an emacs windows and two xterm windows present but not vying for any IO resources (ie, ps would show them as idle processes). The SPARC 1 workstation has a SPARC CPU running at 20 MHz for a clock cycle time of 50 ns.

Four cellular automata experiments were selected as the test base. They are thought to be typical of the type of experiments that one might want to run on CAM-8. The four selected experiments exercise most of the enhanced features of CAM-8;
if the simulator performs well under all these experiments, it is expected that it will perform well under most experiments that can be written to run on CAM-8, and can therefore be used as a suitable platform for prototyping experiments for CAM-8. The four chosen experiments are described below.

- **HPP Gas**: The CAM-8 architecture, because of its partitioning nature and its facility for moving bit-planes relative to each other, is ideally suited for HPP Gas and other lattice gas models, which, of course, lie at the heart of all fluid dynamics cellular automata experiments. Clearly, one of the most important uses for CAM-8 will be fluid dynamics experiments, and it is imperative that the CAM-8 simulator perform well under this experiment. The HPP rule itself is quite simple. The entire experiment needs only 4 bit-planes, and uses the site data as the LUT Address and the LUT entry as the new site data.

- **Life**: The Game of Life, introduced by the mathematician John Conway [7], is perhaps the most well known cellular automata rule. It uses a fairly large neighborhood (8 neighbors to update 1 cell), and was selected primarily because it is expected to be well known to a large audience. Also, it uses 9 bit-planes (for the most direct implementation), and from the point of view of the simulator, it requires a fair amount of work.

- **Rotate**: This simple experiment can rotate a space consisting of a graphic bit-map by ninety-degrees in a logarithmic amount of time. The update rule depends partly on the spatial location of the cell, so this experiment uses a LUT Address destination that is a combination of various sources. This experiment can be considered as typical of one that uses some of the enhanced capabilities of CAM-8.

- **Patch-Count**: One of the unique features of CAM-8 is that it has facilities for data consolidation in the form of on-chip event counters. In this experiment, counts are taken over patches of the entire experiment space. This involves not only updating the event count destinations, but also scanning in out-of-order
fashion to form the patches. A series of small scans is performed to cover the entire experiment space.

The experiment space size is $256 \times 256$, and typically data is taken by doing 10 runs of 50 scans each and averaging the 10 samples. This average, divided by 50, is used as the time per scan. Because of the nature of the Rotate experiment, however, 10 runs of 48 scans are performed. Similarly, for the Patch-Count experiment, data is taken over 10 runs of 64 scans each.

For timing information, the UNIX `gettimeofday()` call was made before and after performing the requested scans, and the difference was used as the time taken to perform the scans. Note that this is absolute time, and would include time that UNIX spent in executing other tasks, process switches, etc. From a user's point of view, this is the time it would take the user to perform the scans. Even though UNIX provides system calls to extract time used up by just the process under test, this would be unrepresentative of the performance that the user will actually see.

### 8.3 Compile vs Execution Times

In this section, data is presented to compare time it takes to compile the scan algorithm with the time spent executing the generated scan algorithm. Table 8.1 shows the times for each of the four experiments.

For the $256 \times 256$ spaces, compile time is typically 2.8% - 3.5% of the total time. Even for experiment spaces of as little as $32 \times 32$, a reduction factor of 64, compile

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Compile Time (µs)</th>
<th>Execute Time (µs)</th>
<th>Compile %</th>
<th>Execute %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Life</td>
<td>450,547</td>
<td>14,263,315</td>
<td>3.06</td>
<td>96.94</td>
</tr>
<tr>
<td>HPP</td>
<td>302,002</td>
<td>9,425,820</td>
<td>3.10</td>
<td>96.90</td>
</tr>
<tr>
<td>Rotate</td>
<td>338,408</td>
<td>12,003,172</td>
<td>2.74</td>
<td>97.26</td>
</tr>
<tr>
<td>Patch-Count</td>
<td>482,035</td>
<td>1,103,964</td>
<td>31.39</td>
<td>69.61</td>
</tr>
</tbody>
</table>

Table 8.1: Compile Times vs Execute Times
times would still be reasonable compared to execute times. In fact, in the Patch-Count experiment, only a $16 \times 16$ space (or patch) is scanned before a recompile is needed. Table 8.1 shows that even for this case, compile time accounts for merely 31\% of the time. The cost of generating an optimal scan algorithm at run time is minimal, and the benefits to be gained by using an optimized scan algorithm appear to be quite large compared to the cost of realizing the optimizations. The basic design approach of dynamic compilation stands amply validated.

8.4 Profile of the Compile Phase

Profiling the compile phase yields an even more surprising result — that the compile phase can be made even more efficient. A profile of the compilation process was taken using the UNIX gprof [8] utility. Note that it is not possible to use gprof to profile the optimized scan algorithm itself, since the algorithm is generated at run time. As described in [8], gprof does not construct a complete call graph and is unable to profile programs that dynamically change themselves, as is the case with the simulator. For this reason, the simulator was profiled for the compilation phase only; during profiling, the compiled scan code was not executed.

The results of profiling are tabulated in Table 8.2. The data is taken from the flat profile generated by gprof. This profile shows the percentage of the run time each of the program's routines accounts for. In the table, the "rank" column indicates the rank by amount of time spent executing that routine. A rank of "1" indicates that the program spent the largest fraction of time executing code in that routine. In the flat profile, time spent executing a called procedure is not attributed to the calling procedure; thus, the percentage time is the actual time spent directly executing the code of that routine.

It is clear that the compile phase is dominated by the register allocator algorithm; both the RestoreRegState and AddQueElement are part of the register allocator strategy. These two functions are used for maintaining the state of the register manager, and are not directly involved in the priority-based register allocation scheme! Since the
register allocator does support multiple register windows on the SPARC by allocating separate register windows for separate procedure blocks, it is more complex than a straight-forward implementation that uses a single register window. The complexity of RestoreRegState and the frequency of calls to AddQueElement are direct consequences of the decision to use separate register windows for different blocks of the scan algorithm. This decision was made in the hope of acquiring the use of as many machine registers as possible, which in turn would reduce the number of memory operations required by the scan algorithm.

A direct implementation that uses a single register window for all procedure blocks will probably also be more portable to other RISC architectures, most of which do not have multiple register windows. Thus, the register allocator is a prime candidate for a redesign effort. At the time of this writing, a plan for the redesign is being considered.

Note, however, that GetReg, which is primarily responsible for implementing the priority-based allocation scheme, is quite cost-efficient, thus validating the central scheme of the register allocation design. In fact, GetReg accounts for less time than library calls to malloc and free, which, presumably, are comparatively very optimized pieces of code.

In any case, even with the more complex register allocator, the fraction of total time spent in the compilation phase is small enough to justify almost any optimization that can be performed to generate a scan algorithm.
<table>
<thead>
<tr>
<th>Experiment</th>
<th>Time with Incremental Compiles (µs)</th>
<th>Time w/o Incremental Compiles (µs)</th>
<th>% Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Life</td>
<td>450,547</td>
<td>504,969</td>
<td>12.08</td>
</tr>
<tr>
<td>HPP</td>
<td>302,002</td>
<td>363,563</td>
<td>20.38</td>
</tr>
<tr>
<td>Rotate</td>
<td>338,408</td>
<td>343,335</td>
<td>1.46</td>
</tr>
<tr>
<td>Patch-Count</td>
<td>482,035</td>
<td>482,742</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Table 8.3: Performance Effects of Incremental Compilation

8.5 Incremental Compilation

Table 8.3 shows the run times for compilation with and without incremental compilation. Recall from section 6.5 that certain blocks of the scan algorithm need not be recompiled at the beginning of every scan. The cost of detecting when this situation arises is simply the small cost of maintaining flags that are turned on when certain CAM-8 registers are written to. These flags indicate that blocks dependent on these registers need to be recompiled. The other cost of maintaining these flags is added software complexity.

Table 8.3 illustrates that the benefit of incremental compiles justifies the added cost in complexity. However, since incremental compilation does not contribute to the optimality of the generated scan algorithm, and since compile time is such a small fraction of the total time that further optimizing the compile phase itself is likely to yield little to no improvements in overall performance.

8.6 Performance of Priority-based Register Allocation Strategy

One of the central scan algorithm optimizations is that due to the 32-bit word size of the SPARC architecture, it is often possible to amortize the cost of word loads and stores over many, rather than one, cell updates. The priority-based register allocation strategy described in Chapter 7 was designed to minimize load store operations in the target code. In this section, we examine how successful the register allocation
strategy is.

The performance of the register allocation strategy is measured via a modified version of the HPP experiment. In the modified version of HPP, the number of significant planes is incremented from 4 to the maximum of 16 planes by setting the content of the Site Source register; instead of a 16 entry look-up table (which is all that's required for HPP), a 64 K look-up table is used which, in essence, is the basic 16 entry look-up table repeated 1024 times, and bits 4 to 16 of each entry set to identity.

Figure 8-1 shows the time taken for performing a $256 \times 256$ scan vs the number of planes updated during the scan. The register allocator was able to keep all needed variables in registers up to 14 planes. (i.e., for up to 14 planes, the register allocation strategy produces a mapping that requires no spill code to be inserted). For the case with 15 and 16 planes, 2 and 4 registers respectively are spilled; however, they are spilled outside the innermost loop. The innermost loop, in this case, updates 256 cells (and this loop is repeated 256 times to perform the complete scan). The spill code, therefore, is executed only 256 times. As seen in Figure 8-1, it does not make
Table 8.4: Performance Comparison with Generalized Scan Algorithm

<table>
<thead>
<tr>
<th>Space Size</th>
<th># of Cells</th>
<th>Time (µs)</th>
<th>( m = \frac{\Delta x}{\Delta y} ) (µs/cell)</th>
<th>( b = y - mx ) (µs)</th>
<th>Time (µs)</th>
<th>( m = \frac{\Delta x}{\Delta y} ) (µs/cell)</th>
<th>( b = y - mx ) (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 x 64</td>
<td>4096</td>
<td>138,716</td>
<td>33.04</td>
<td>3,379</td>
<td>19,514</td>
<td>57,312</td>
<td>3.07</td>
</tr>
<tr>
<td>128 x 128</td>
<td>16,384</td>
<td>544,727</td>
<td>32.95</td>
<td>4,882</td>
<td>198,219</td>
<td>2.87</td>
<td>10,343</td>
</tr>
<tr>
<td>256 x 256</td>
<td>65,536</td>
<td>2,164,294</td>
<td>32.95</td>
<td>4,882</td>
<td>198,219</td>
<td>2.87</td>
<td>10,343</td>
</tr>
</tbody>
</table>

a measurable difference in the amount of time it takes to perform the scan.

Since HPP represents a very important class of experiments for the CAM-8 architecture, the simple register allocation strategy is expected to work quite well for the most common modes of CAM-8 operation.

8.7 Performance relative to Generalized Scan Algorithm

Section 8.3 shows that the cost of generating an optimized scan algorithm is extremely modest, even if the scan algorithm is used to update just a small number of cells. A related CAM-8 software simulator, referred to as Kensim that uses the same scan algorithm regardless of the type of scan being performed was built for hardware verification purposes. This simulator actually has a performance advantage because it does not simulate the different sources and destinations provided by CAM-8; it simply uses the glued (or unglued, which is the same for a single module) data as an index into a look-up table and uses the look-up table value to update the site data. Moreover, it does not update all 16 planes, and relies on the user to tell it how many planes to update. Since it also runs on the SPARC architecture, the main difference from the simulator of this thesis (apart from the limited capability) lies in the fact that Kensim does not optimize the scan algorithm for the scan to be performed; it simply uses a predefined, generalized scan algorithm.

Table 8.4 tabulates the scan times taken by Kensim and by the CAM-8 simulator for running the HPP rule on three different space sizes. The times shown for Kensim
are those taken only to perform the scan; the time it takes to retrieve the scan instruction, or the time it takes to display the cell space in its X window is *not* included. The time recorded is for doing only the scan. The times shown for the CAM-8 simulator, however, *do* include the time taken for display, since the display update is part of the optimized scan algorithm. In spite of this extra work, Table 8.4 shows clearly that the benefits to be gained by using an optimized scan algorithm are immense.

In Table 8.4 the slopes measure the time it takes to update a single cell. The CAM-8 simulator, using an optimized scan algorithm, is about an order of magnitude better than Kensim. Of course, the CAM-8 Simulator must first compile this optimized scan algorithm, and so has a higher startup cost. This is reflected in the greater y-intercept values for CAM-8 simulator performance. Solving for the intersection of the two performance lines yields a space size of 350 cells (or about an $18 \times 18$ space size). For space sizes smaller than this, the CAM-8 simulator is slower than a simulator using a pre-packaged scan algorithm. For space sizes bigger than $18 \times 18$, the CAM-8 simulator quickly becomes faster, achieving an order of magnitude speedup for a space size as small as $128 \times 128$.

For any type of serious cellular automata experiment, $18 \times 18$ is a trivially small space size, and the CAM-8 simulator approach of compiling an optimized scan algorithm is expected to almost always have a great performance benefit for the user.

### 8.8 Performance Comparison with Cellsim

*Cellsim* is a cellular automata program written by Chris Langton and Dave Hiebeler [10] that runs on the SPARC. A performance comparison between the CAM-8 simulator and Cellsim is interesting because both have performance as a central issue in their design. However, Cellsim is not tied to simulating any particular hardware architecture; Cellsim also does not have the notion of having multiple sources and destinations, (though modifying the program to implement these is certainly feasible).

An interesting feature of the Cellsim implementation is that it does not slice up site
Table 8.5: Performance Comparison with Cellsim

data across multiple bit-planes. Instead, it keeps site data as a byte sized quantity. Because of this, it would be very difficult to adapt Cellsim to cellular automata experiments that require particles moving at different speeds (Cellsim supports a 2 × 2 Margolus neighborhood which can only be used to implement particles moving at unit speed).

One consequence of this organization is that since the site data is stored as a discreetly addressable quantity (a byte), updating a site can be very efficient — a site’s value can be retrieved, and the returned value is stored back, in single instructions. Another consequence is that performance is independent of how many states a site can have in a particular experiment. Since a site’s state is always stored in a byte of memory, that entire byte must be retrieved and updated, even if the site can only have two states (as, for example, in Conway’s life experiment). This is clear from the data in Table 8.5.

With respect to Cellsim, one might expect that the CAM-8 simulator would have a substantial performance handicap, since for every site, the CAM-8 simulator must extract as many as 16 bits of data from 16 different memory words, do the lookup, and then scatter each of the 16 bits of the new data back to 16 memory words. As explained in Chapter 3, however, much of this data retrieval and storing work can be amortized over a large number of site updates, as long as an optimized scan algorithm is generated and used.

Table 8.5 shows performance data for Cellsim and the CAM-8 simulator for Con-
way's Life and HPP Gas experiments for three different space sizes. It is clear that
the CAM-8 simulator is able to optimize the scan algorithm well enough to match the
performance of a simulator that is not tied to simulating any hardware architecture
or has the scope and capability of the CAM-8 simulator. Even in the case of Life,
where the CAM-8 architecture has a distinct performance disadvantage, the CAM-8
simulator performs almost identically as well as does Cellsim. In the case of HPP,
where the CAM-8 simulator only needs to update 4 bits/site whereas cellsim must
update an entire byte, CAM-8 is actually faster than Cellsim.
Chapter 9

Conclusions

In this thesis, a software simulator of the CAM-8 architecture was successfully designed and constructed. To efficiently emulate the hardware architecture, the simulator was designed like a conventional compiler — the simulator first compiles the instruction stream to CAM-8 to an optimized scan algorithm and then executes the scan algorithm to actually emulate the CAM-8 hardware. The performance of the simulator has been measured and compared with the performance of other cellular automata software available for the Sun SPARC architecture. The performance figures validate the basic design of compiling an optimized scan algorithm at run time instead of using a predefined, general, scan algorithm. Moreover, it has been shown that the scan algorithm can be optimized enough so that emulating the CAM-8 hardware architecture in software is not a performance handicap for the simulator relative to a cellular automata program without that constraint. The CAM-8 architecture (and therefore the simulator) provides for a much wider range of cellular automata experiments than is possible with other cellular automata architectures. The optimizing CAM-8 simulator thus provides a platform for new CAM experiments without incurring any performance degradation over other optimized softwares that have much more limited capabilities. Because of the respectable performance of the simulator, it can be profitably used to develop programs and experiments for the CAM-8 hardware without requiring that the user have the actual hardware. In addition, the simulator, because of its faithful emulation of the hardware architecture, has already proved to
be of value in debugging and validating the hardware design.

It is hoped that because the simulator runs on a very popular RISC architecture, does not require any additional resources other than the workstation itself, and provides a very respectable performance compared with other cellular automata software, it will make the field of cellular automaton computationally accessible to a much wider scientific and engineering audience than has been possible to date. The author hopes that the simulator will facilitate the exchange and development of cellular automata experiments by scientists everywhere; certainly, the author hopes that no one will be denied this opportunity simply because she cannot procure the resources to purchase the CAM-8 hardware. The CAM-8 simulator has been proven to be an inexpensive and viable alternative platform for developing cellular automata experiments.

It is also hoped that the simulator itself will be a test bed for experimentation. The simulator itself provides a way to experiment with different CAM-8 architectures. For example, it would be very easy to increase the number of bit-planes available in the simulator, where such a change in hardware would require a considerable effort. Similarly, the lookup table could be replaced by an arbitrary user defined computation for experimentation purposes - while the hardware implementation does have facilities to do exactly this at the hardware level also, it imposes restrictions which are not present in the simulator. Lastly, the family of optimized scan algorithms that the current implementation of the simulator generates can easily be expanded to cover frequently occurring scan configurations.

The current implementation itself could be improved upon on at least two fronts—a better register allocator implementation and the scheduling of instructions within a scan algorithm optimized for super-scalar SPARC RISC chips which are just now becoming available. The second improvement could easily yield a performance increase of 200% to 300% on CPU's with 4 execution pipelines.

In conclusion then, an efficient and optimizing simulator of the CAM-8 architecture was designed and constructed, and its performance was measured to be easily adequate for general purpose use as a platform for developing CAM-8 software.
Appendix A

Interface to CAM-Forth and X Windows

The CAM-8 hardware module is normally interfaced to a host workstation, which, at the time of this writing, is a SUN SPARCStation. The host workstation is equipped with a custom-designed host interface card that interfaces the workstation to the CAM-8 hardware.

CAM-8 Forth, a version of Forth that has been extended to make programming CAM-8 very convenient, is the prototype environment for programming CAM-8. CAM-8 Forth runs on the host SPARCStation and talks via a device driver to the host interface card. It communicates with the host interface card by writing and reading 4 registers on the host interface card. The CAM-8 hardware is "programmed" simply by writing to its various registers (see Tables 2.1 and 2.2 for a synopsis of the available registers). CAM-8 Forth writes to the CAM-8 registers by creating what are called STEP lists, or a linked list of STEP instructions. STEP instructions are simply instructions which are translated by the host interface card into hardware signals that do the actual reads and writes from and to CAM-8's hardware registers.

Figure A-1 illustrates a typical setup. The user interacts with CAM-8 Forth and develops an experiment in Forth. Forth translates the experiments into a sequence of STEP instructions, which, as mentioned above, are mostly requests to write or read the various CAM-8 hardware registers. This STEP list is constructed in a block of
memory that is shared between CAM-8 Forth and the interface card device driver. Once CAM-8 Forth has built the STEP list, it writes the address of the head of the STEP list to the interface card’s command register (via, of course, a call to the device driver). This causes the interface card to translate the STEP list instructions into appropriate hardware signals to write and read CAM-8 registers.

To properly emulate the CAM-8 hardware, the CAM-8 simulator needs to interface to CAM-8 Forth so that it can run any and all experiments written for the CAM-8 hardware. The CAM-8 simulator also has facilities to create an X window to serve as a window into a portion of the cell-space. This provides a very attractive visual feedback to any user writing programs for CAM-8.

The following sections describe in some detail the Host Interface Registers, the format of STEP instructions, how the CAM-8 simulator interfaces to CAM-8 Forth, and the optional X window interface.
A.1 Host Interface Registers

In the interests of brevity, only the fields of the Host Interface Registers that are relevant to the CAM-8 simulator are described here. Many of the fields are related to hardware error conditions and interrupts, which cannot occur within the CAM-8 simulator; these fields have been omitted from the description below. The complete description can be found in [15].

- **R0 - Host Next Pointer** — The R0 register of the host interface card is a read-write register. The address of the beginning of the STEP list that CAM-8 Forth wants to schedule for execution on CAM-8 is written to R0. Reading from this register will return the address of the beginning of the next STEP list that the interface card will translate and transmit to the CAM-8 hardware.

Because STEP lists are required to start on a 16-byte boundary, the LSB 4 bits of the address written/read to/from R0 are always going to be 0. To avoid waste, when R0 is read, the LSB 4 bits are used to return status information.

Bit 0, *Host Next Pointer Written*, indicates that the last write to R0 completed. This is necessary, because CAM-8 Forth could schedule a second STEP list before the previous one started execution; if CAM-8 Forth simply wrote R0 again, the pointer to the previous STEP list would be lost and that STEP list would never get executed. Instead, after writing R0, CAM-8 Forth is required to wait until this bit is set before it is free to go ahead write R0 again.

Bit 1, *Exception Status*, indicates to CAM-8 Forth that CAM-8 reported an error condition.

Bit 3, *Interface Halted*, indicates that the interface is stopped and is not processing STEP lists.

- **R1 - Command Word** — Writing to R1 serves two purposes — writing some bits (as tabulated in Table A.1) enables/disables the conditions under which the host interface card will generate an interrupt; writing to other fields controls the interface card itself. Similarly, reading some bits of R1 (see Table A.2) shows
Table A.1: R1 Bit Fields Definition for Writes

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description for writes to R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Interface Reset</td>
</tr>
<tr>
<td>15</td>
<td>Halt Interface</td>
</tr>
<tr>
<td>12</td>
<td>Enable New List Interrupt (i.e., interrupt when the host starts executing the next STEP list pointed to by R0). The interrupt, when enabled, occurs exactly when Bit 0 of R0 is set.</td>
</tr>
<tr>
<td>9</td>
<td>Enable CAM interrupt</td>
</tr>
<tr>
<td>4</td>
<td>Disable New List Interrupt</td>
</tr>
<tr>
<td>1</td>
<td>Disable CAM interrupt</td>
</tr>
</tbody>
</table>

Table A.2: R1 Bit Fields Definition for Reads

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description for reads from R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>New List Interrupt is enabled</td>
</tr>
<tr>
<td>9</td>
<td>CAM Interrupts are enabled</td>
</tr>
<tr>
<td>4</td>
<td>New List Interrupt</td>
</tr>
<tr>
<td>1</td>
<td>CAM Interrupt</td>
</tr>
</tbody>
</table>

under which conditions the host will interrupt; reading other bits shows which condition caused the interrupt.

- **R2 - Status Word** — This is a read-only register. Reading this register returns the value of the *STEP instruction* (not the STEP list) currently being executed. Since each STEP instruction is aligned on a 16-byte boundary, the LSB 4 bits of this register are used for status information. They are hardware related.

- **R3 - Status Word** — Another read-only register that returns the value of the previous STEP instruction that was executed. Again, LSB 4 bits return status information which is not simulated by the CAM-8 simulator. R3 is intended for debugging, especially in cases where there might be loops in the STEP list, and knowing the *previous* instruction that was executed can tell one whether the current instruction is being executed during the first or subsequent iterations of the loop.
### Table A.3: STEP Instruction Control Word

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reset CAM — CAM Read and Immediate Data must also be set</td>
</tr>
<tr>
<td>30</td>
<td>1 = Read from selected register, 0 = write</td>
</tr>
<tr>
<td>29</td>
<td>Immediate Data — Word 2 of STEP Instruction is the data</td>
</tr>
<tr>
<td>28</td>
<td>Flag-8 — Indicates 8 bit (default is 16 bit) transfers</td>
</tr>
<tr>
<td>15</td>
<td>Host Jump — Use the value in interface register R0 as pointer to the next STEP Instruction if R0 has been written. Else, use STEP Instruction Word 3 as the pointer to the next instruction.</td>
</tr>
</tbody>
</table>

#### A.2 STEP Instruction

A STEP Instruction is a sequence of four 32-bit words - a single STEP instruction is 16 bytes long.

- **Word 0**: Control Word — The first word of the four-word tuple selects one of the CAM-8 hardware registers as a target. The least significant 6 bits of the first word are used to select the CAM-8 register. The rest of the word is organized as shown in Table A.3.

  When both the Read flag and the Immediate flag are set, the instruction is interpreted to be a NOP. This is why, to Reset CAM-8 hardware, one needs to set the Reset CAM bit and issue a NOP (i.e., set the Read and Immediate flags).

- **Word 1**: Data Pointer — The second word of the STEP Instruction gives a pointer to the data that needs to written to the selected register. If the control word’s Immediate flag is set, however, this word is interpreted as the data itself, rather than a pointer to the data.

- **Word 2**: Length — This field indicates the number of 16-bit words that will be transmitted to the CAM-8 hardware. The CAM-8 module has 16 STEP chips; the length field actually specifies how many bits / chip will be written (or read). If the Immediate flag of the control word is cleared, the data is organized in
groups of 16-bit words where the first word is the LSB of the values to be written to the 16 chips, the next 16-bit word is the next significant bit and so on.

- **Word 3**: Next Instruction Pointer — This field gives the address of the next STEP instruction.

### A.3 Interfacing to CAM-8 Forth

To simulate CAM-8 hardware, the CAM-8 simulator interfaces with CAM-8 Forth and executes STEP lists generated by it. This section describes in some detail the mechanics of the interface.

We see from figure A-1 that Forth creates a STEP list in a block of memory shared with the device driver; the host interface card then executes this STEP list directly from this shared memory. The CAM-8 simulator essentially replaces the host interface card and the CAM-8 hardware, providing their functionality by software emulation. It is expected (though not required) that a user will use the CAM-8 simulator only when her workstation does not have the actual CAM-8 hardware. In this case, she uses the "sim" Forth word to fork off a copy of the CAM-8 simulator. Forth forks off the simulator and establishes a region of shared memory with the simulator; Forth constructs the STEP lists in this block of memory, and the simulator executes them out of this block.

To simulate the host interface card registers, Forth also establishes two pipes to the CAM-8 simulator; one pipe serves as the Forth-Simulator communication channel and the other one serves as the Simulator-Forth communication channel. On the simulator end, these pipes are connected to stdin and stdout.

*The interface between CAM-8 Forth and the CAM-8 simulator consists entirely of a region of shared memory and two pipes.*

Forth simulates writing to the host interface registers by writing to the Forth-Simulator pipe. It sends across a 3-word record. The first word is a 0 for a write request, 1 for reads. The second word selects one of the 4 interface registers. The third word is the value to be written (it is unused if it is a read request).
Immediately after “writing” an interface register, Forth reads on the Simulator-Forth pipe (and thereby goes to sleep, allowing the CAM-8 simulator process to run and process Forth’s write Request). When the simulator is finished processing Forth’s request, it writes essentially a garbage value to the Simulator-Forth pipe, thus causing Forth to awake. Concurrently, the simulator reads on the Forth-Simulator pipe awaiting another request from Forth, and goes to sleep until such event occurs, and the whole cycle repeats.

Some subtle issues arise because one is replacing two systems that run concurrently with one system that time-slices between two processes. For example, in the case of a setup with CAM-8 hardware, the interface card actually processes STEP lists concurrently with Forth creating other STEP lists. It is precisely for this reason that there is a “Host Next Pointer Written” bit field in the interface R0 status word, because Forth needs to know when the interface card has actually started executing the previously scheduled STEP list, so it can safely schedule another one. In a setup with Forth and the CAM-8 simulator, this, of course, cannot happen, because they are both user processes running on the same processor. To achieve synchronization with the Simulator, Forth instead waits by reading on the Simulator->Forth pipe.

A.4 X interface for the CAM-8 Simulator

Cellular Automata experiments and applications generally tend to be visually fascinating, if not always visually informative. The CAM-8 Simulator would be lacking if it did not provide a way for the cell space being scanned to be visually displayed in some manner.

At the time of this writing, the CAM-8 Simulator has a simple X window interface. It simply creates an 256x256 X window on startup. Then, when the cell space is scanned and the cells are updated, the new cell states, (i.e., the value computed as the Site Data Destination) is displayed in the window.

In the interest of speed, the current implementation retrieves a pointer to the actual frame buffer and writes the pixels directly to frame buffer rather than using
the X window system's Image data structure and image manipulation routines. A better X window implementation, one that will let the user choose between X Image manipulation and faster frame buffer display techniques is currently under design. This interface will also allow the user to have better control over what gets displayed.
Appendix B

CAM-8 Simulator Code Modules

The entire code for the CAM-8 Simulator currently stands at approximately 7500 lines of C code, and is too bulky to be included in its entirety here. In this Appendix, the code is described in terms of the modules comprising the code.

B.1 Main

Once CAM-8 Forth forks off the CAM-8 Simulator, control is transferred to CAM-8 simulator’s main() function in the module main.c (as, of course, is true of any program written in ‘C’). The main module first calls initialization routines in the other modules; it then parses the command line for certain parameters. For example, if the simulator is invoked with the -f parameter, main sets an internal flag which causes the simulator to write site data directly to the workstation’s framebuffer. Similarly, the -g option causes the simulator to discard all optimizations while generating the scan code. After parsing the command line, main transfers control to the hinterface module. This module reads stdin, which is initialized by CAM-8 Forth to be the Forth-Simulator pipe. As described in Section A.3, Forth writes to this pipe to give commands to the CAM-8 Simulator.
B.2 Hinterface — The Host Interface Module

This module essentially simulates the host interface card; it serves as the connection point between CAM-8 Forth and the CAM-8 simulator, just like the host interface card connects CAM-8 Forth to the CAM-8 hardware. This module accepts commands from Forth and initiates all actions required to execute the command. Most commands can be performed by the hinterface module itself; for example, when Forth writes to the host interface register R1, the only action necessary is to toggle certain switches maintained as global state. The most important command, of course, is the execute STEP list request, which Forth issues by writing the address of the first instruction in the list to the interface register R0. When this happens, this module calls the steplist module to carry out the Forth command. This module also writes what is actually just a placeholder value to stdout when Forth’s command is completed. Stdout is initialized to the Simulator-Forth pipe by Forth. After issuing a request, Forth sleeps awaiting a read on this pipe. Reading a value from this pipe signals Forth that its request has been completed. The host interface module is therefore responsible for synchronizing with CAM-8 Forth.

B.3 Steplist — The STEPList Execution Module

STEP instructions, as described in Section A.2, are requests to read and write CAM-8 registers. The state of most registers is maintained in a global data structure as described in Chapter 5. The steplist module decodes the step instructions and carries out the instructions by transferring data between the buffers pointed to by the STEP instruction Data Pointer word and this global CAM-8 state data structure. Writes to certain registers, for example the Run Mode Register and the Scan IO register initiate a scan. The Steplist module first calls the compiler module to compile the scan code. It then executes the scan code to actually scan the space as dictated by the current state of the CAM-8 simulator, which, of course, is recorded in the global data structure.
STEP instructions which are read instructions are actually executed by the steplistr module. Logically, the steplist and steplistr module comprise a single module; it has been split to keep the code size of each component module reasonable.

B.4 Compiler — The Scan-code Compiler

The compiler module, along with the CompileNoPerm and the CompileGen module, implement the syntax directed translator described in Chapter 4. These modules carry out the syntax directed translation according to the current state of the CAM-8 simulator as recorded in the global CAM-8 state data structure. The translation consists of calling the intermediate code interface’s Emit() function to output the intermediate code as dictated by the syntax directed translation scheme described in Chapter 4. When the compiler module finishes, it would have generated the intermediate code for the entire scan code needed to emulate the scan being requested. Of course, because the current implementation is single-pass, every call to Emit() actually results in target machine code being generated, and the intermediate code form is never stored in its entirety.

B.5 Icode — The Intermediate-code Module

This module implements the intermediate-code to target-code translator as described in Chapter 6. It relies on the register allocator module to resolve variable names to machine registers and for other service.

B.6 Regmgr — The Register Allocator Module

This module implements the variable name to machine register resolution scheme as described in Chapter 7.
B.7 SPARCAsm — The SPARC Assembler Module

This module, implemented entirely as a header file, provides convenient macros to output the actual bit-patterns for SPARC instructions. A set of macros for every SPARC instruction is provided. The set generally includes a macro for every addressing mode allowed for that instruction. For example, the different forms of the ADD instruction can be emitted by using the ADD_REG_IMM, ADD_REG_REG, ADDcc_REG_IMM, ADDcc_REG_REG, etc. If the simulator needs to be ported to another architecture which is a 32-bit Load Store Architecture, it would be possible to do the port simply by replacing the SPARC assembler module with the appropriate target specific module. (Of course, due to the use of register windows on the SPARC architecture, it would also be necessary to rewrite the RegMgr module for an architecture that did not have register windows).

For debugging and measurement purposes, a reverse assembler is also provided. The module t_printasm provides functions to dump a segment of memory as SPARC instructions. One could use this module to look at, say, the code generated for the UPDATE-CELL procedure.

B.8 Utils — The Utility Module

Do carry out the syntax directed translation, the compiler module often needs to interpret the CAM-8 state as stored in the global state data structure. For example, it needs to know how many of the 16 planes will actually be relevant during the scan being compiled. This information can only be deduced by examining several CAM-8 registers, mainly the Destination Source and the Destination Map registers. Routines to do this deduction are provided by the Utils module.

B.9 Heap — The Heap Manager Module

The scan code needs a set of variables to perform the scan. These variables, as described in Chapter 6, reside in a heap like structure. The heap module is responsible
for initially allocating the block of memory that is to serve as the heap. The heap manager also allocates the memory for the 16 bit-planes and the 2 Lookup Tables. Lastly, the heap manager allocates and initializes the Symbol Table, which is used by the register allocator module to resolve variable names to machine registers. The Symbol Table needs to be properly initialized with the heap offsets, priorities, and names of all the variables.

The code for this deserves a little more explanation. The file heap.h defines a set of constants of the form iXxx where Xxx is some variable’s name, for example, iNextWordAddr. As described in Chapter 6, a variable is exclusively identified by its index in the Symbol Table. The constant iXxx is the index and is therefore the Name of the Variable.

For every variable, there is a defined constant of the form offXxx that gives the offset of the variable into the heap.

Lastly, some of the variables need to be initialized before the compiled scan code is executed. To access the variable’s location in the heap, macros have been defined that evaluate to the variable’s absolute memory address. If the variable’s name is iXxxYyy, the corresponding macro is named XXX.YYY.

To summarize, consider the variable NextWordAddr. iNextWordAddr is the variable’s index into the symbol table, and is therefore its name. offNextWordAddr is the variable’s offset into the heap - this is where the value of the variable is stored. NEXT_WORD_ADDR is the ‘C’ macro that evaluates to the absolute address of the variable’s value (i.e., the absolute address where the variable’s value is stored).

B.10 Xinterface — The X-window Interface Module

This module creates the X window and provides a window in which site data can be displayed. At the time of this writing, the interface is very elementary. A more sophisticated module is under consideration.
B.11 Memmgr – The Memory Manager Module

One of the banes of 'C' programmer's is that they never check for error values returned by system calls like malloc() etc. The reason for this often is that checking for error return values would cause code indentation to get out of hand and the non-error condition code, presumably the only interesting code, gets overwhelmed by the amount of error-condition code. To somewhat alleviate this problem, some of the more critical system calls have been encapsulated into this memory manager module, which does the error checking and aborts with a simple message if there is an error. For example, the memory manager provides a MALLOC() function which returns whatever malloc() returns, except when malloc() returns a NULL, signifying an error. At this point, MALLOC() prints a message and aborts by calling exit().

B.12 Errmgr – The Error Manager

The memory manager, and the bulk of the simulator code, could sometimes run into error conditions that either prevents the simulator from proceeding any further or at least deserves a warning to the user. To this end, a centralized facility for reporting errors has been provided. At the beginning of every module's .c file, a 'C' preprocessor variable MODULE_NAME is defined to be the string representation of the module's name; for example, #define MODULE_NAME "heap". Then, if any function in that module wants to flag an error or issue a warning, it calls the errmgr's CRITICAL_ERROR or WARNING macros. These two macros take in a message and the function name, and print an error message of the form warning-message:function-name:module-name. The module name is extracted from the MODULE_NAME variable. CRITICAL_ERROR calls exit(), thus aborting the CAM-8 simulator. WARNING() simply prints the message.

An interesting experiment was performed for error management. In [2], an intriguing method of using setjmp() and longjmp(), which are 'C' interfaces to saving and returning to continuations, to transfer control to error handling code is described.
This method was used in an early version of the CAM-8 simulator. While the method described is elegant, simple performance analysis revealed that as much as 28% of the execution time was being spent in just storing continuations! This is a very steep price to pay just for coding convenience. This method was discarded in favor of the much simpler, two-level error handling mechanism described above. In practice, the error management facility has proved to be very adequate - a case has not yet been encountered where some error condition is not detected and the CAM-8 simulator enters a random state. When errors are detected, the simulator aborts with a simple message which generally indicates the function and module which issued the error. This has proved to be very useful for pinpointing errors in the code; anyone remotely familiar with the code will have no trouble at least locating the point at which the error occurred.

B.13 AS - The Intermediate-code Assembler

Much of the work done by the syntax directed translator involves calling the intermediate code's Emit() function to build the scan code. Emit() takes seven parameters — an opcode, three operands, and the types of the three operands. Modules that use the intermediate code interface become extremely cluttered by the syntax of the Emit() function. For example, a typical call to Emit() might look like:

\[\text{Emit(ADD, VAR.OP, iNextWordAddr, CONST.OP, 4, VAR.OP, iNextWordAddr);}\]
\[\text{Emit(CALL, ADDR.OP, UpdateSite, NO.OP, 0, NO.OP, 0);}\]

It is not clear at all that the code that was emitted is an add instruction to add 4 to the variable iNextWordAddr. A better syntax is required.

With this in mind, modules that use the intermediate code interface (i.e., call the Emit() function) are processed by an awk script called as.gawk before being compiled by the 'C' compiler. Calls to Emit() are actually coded as shown in the example below:
The awk script translates code of this form into the correct call to Emit(). It looks for lines where the first non-blank character is a period and expects them to be of the above form. It automatically detects the types of the operands and inserts the correct operand type parameters for Emit(). The default type is Variable. A ‘#’ symbol before an operand implies that the operand is a constant. Enclosing the operand in square brackets indicates that the operand is an Address. A ‘:’ precedes a Label type operand, and a ‘%’ precedes an operand of type Register. This format does much to reduce the clutter and enhance the readability of the source code.

The CAM-8 simulator’s make file has been set up to automatically preprocess the .c files by running them through the awk script.
Bibliography


