Simulation of Massively Parallel SIMD Architectures Using FPGA Support

by

Timothy N. Kutscha

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

Bachelor of Science in Electrical Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author [illegible]
Department of Electrical Engineering and Computer Science
May 16, 1994

Certified by [illegible]
Thomas F. Knight Jr.
Thesis Supervisor

Accepted by [illegible]
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Simulation of Massively Parallel SIMD Architectures Using FPGA Support

by

Timothy N. Kutscha

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 1994, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

People often simulate a computer before its construction to find problems with its architecture and to concurrently develop algorithms for it. When simulating massively parallel computers, simulation time generally increases proportionately with the number of processing nodes. Thus, for large numbers of nodes, a simple optimized C program can effectively simulate all relevant register values while running significantly faster then many high level languages such as Verilog or VHDL. Instead of taking many processor cycles to simulate complex processor logic with software, we can extract the complex logic into a faster external hardware unit, typically implemented as a field programmable gate array (FPGA) for flexibility. We ran simulations of early vision algorithms on an Intel 80960 platform with an assisting FPGA and found that the FPGA only speeded up our simulations by a negligible amount due to the slowness of the external bus. The extra added cost and effort of programming the FPGA was not worth the incremental speed increase.

Thesis Supervisor: Thomas F. Knight Jr.
Title: Principal Research Scientist

Acknowedgements

I would like to thank Tom Knight for all his support and help during my thesis. The basis for this simulator comes from the Abacus project headed by Mike Bolotski. I wouldn't have been able to complete the hardware without the Intel 80960 Platform from the MIT Transit project headed by Andre Dehon. Ian Eslick helped me out immeasurably with Verilog simulation and synthesis tools. Over the years, Tom Simon taught me much VLSI and hardware design and I would like to thank Rajeevan Amirtharajah for all his work and friendly support with the Abacus project. Chris Pezzee and Andy Shultz also spent much time programming the compiler and assembler for the Abacus machine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Overview

The motivation for researching fast simulation implementations for highly parallel SIMD architectures arose from the Abacus project currently developing under Tom Knight at the MIT Artificial Intelligence Lab. The Abacus research group wishes to develop processing algorithms as custom ASIC development continues. In an effort to cut simulation time, we created an efficient simulation system to compute all relevant register values in the Abacus system.

After describing the basics of the Abacus architecture and a brief description of the software simulator, the remainder of the paper will discuss the following topics:

- Details of the Abacus simulator
- Implementation of a few vision algorithms
- Comparison of simulation times on different platforms
- Improvements to the system and future possibilities

1.2 Abacus Architecture Description

The Abacus computer is currently being developed by Mike Bolotski at the MIT Artificial Intelligence Lab.[4] It consists of an array of 256K processing elements arranged

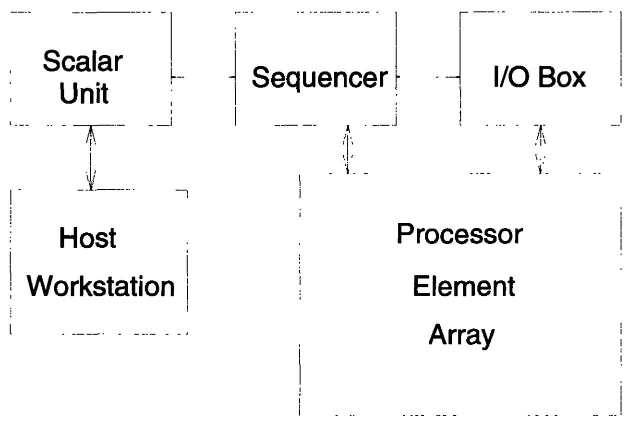


Figure 1-1: Block Diagram of Abacus System

in a two-dimensional toroidal architecture. Each processing element computes one bit and communicates with its nearby neighbors through two internal networks. An external sequencer delivers a 58 bit instruction word to all processors each cycle. It also performs conditional instructions based on a global-OR signal computed from all the processing elements.

1.2.1 Physical Implementation

Custom silicon chips will be used to implement the Abacus architecture. A 32x32 array of processing elements will exist on a single ASIC chip. In turn, a 16x16 array of ASIC chips will be interconnected with PC-boards to complete the toroidal architecture of the machine. Each ASIC chip will have eight megabytes (2 megawords) of local DRAM memory for external storage. A fast microsequencer, possibly made of ECL components, sends instructions and memory controls to all 256 chips each cycle.

1.2.2 Processing Element

Each processing element, shown in figure 1-2, computes a pair of bits and stores the result in two 32-bit register files. During an instruction cycle, the processor reads out two data bits from each register file and performs an operation on a combination of those bits in two different ALUs. The processor then writes the results from each

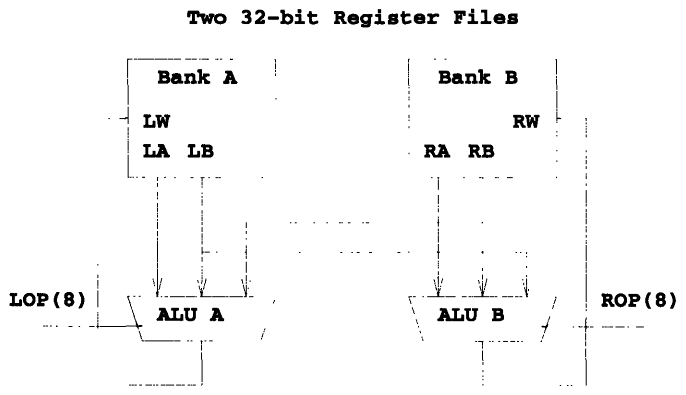


Figure 1-2: Block Diagram of Processing Element

ALU back into its corresponding register file.

1.2.3 Instruction Word

The Abacus is a SIMD machine where all 256K processing elements receive the same instruction each cycle. The Abacus' instruction word is 58 bits wide to allow flexible functionality within the processor. 30 bits in the instruction determine the two read addresses and write address for each of the 32-bit register files. Another 16 bits determine the operation to be performed in each of the processor's ALUs. Since each ALU has three inputs from the register files, 8 bits of operation per ALU provides the flexibility to compute any function of those inputs. The remaining twelve bits control instruction distribution data movement through local communications networks.

1.2.4 NEWS Network

The Abacus allows communication with nearest neighbors as well as the ability to propagate signals beyond neighbors with an internal mesh network called the NEWS (North-East-West-South) network. Each processor owns one node on the NEWS network. After all nodes have been precharged, a processor can pull down its node and let the low logic value propagate to nearby neighbors. Two selection bits in the register file control the propagation direction. Another bit turns off propagation from a given processor. A fourth bit acts as the port for reading and writing values to the network. Writing the port register causes a network precharge and changes the value of the processors' node. Reading the port register returns the value on the node at the adjacent processor selected by the selection bits. Propagation through the NEWS network cannot cross chip boundaries; however, a processor along a boundary may read a value broadcast by the corresponding processor on the other side of the boundary after two clock cycles of latency.

1.2.5 DRAM Plane and IO Plane Networks

To exchange information with the outside world, the Abacus implements two other networks. These networks shift information to and from external DRAM memory or IO structures. We can think of the movement of data through the chip as a plane of information, shifting from one edge of the chip to the other. Parts of the instruction word control the movement of these two information planes. By accessing a particular register in the register file, we can write data to or read data from these planes. The IO and DRAM plane networks are separate due to the different requirements placed on the external pins.

1.2.6 Sequencer and Global OR Line

An external sequencer runs the Abacus machine by distributing a 58 bit instruction word to all the processors each cycle. A host workstation initially loads the sequencer with instructions. The sequencer performs its own loops, subroutines, and conditional branches based on a global-OR signal returning from the ASIC processor arrays. One corner of the NEWS network serves as the global-OR output for each processor. External logic, in turn, combines all the ASIC outputs together and presents them to the sequencer for conditional processing.

The sequencer issues instructions to control the ALU portions of the chip as well as control the movement of the DRAM plane and IO plane networks described earlier. Some bits of the instruction word drive the inputs to the external DRAM to control the memory directly. In order to break up the symmetry of the toroidal mesh of the Abacus on bootup, the instruction word also has provisions for loading local addressing information to the individual processors through the NEWS network.

1.3 The Abacus Software Simulator

To speed up algorithm development, I wrote a C program to simulate the behaviors of the Abacus architecture. The C program, developed on a Sun Sparcstation, was ported over to an Intel 80960 hardware platform under the control of a host Sparcstation. The i960 platform runs at 5 Mhz and provides local access to the processor bus through plug-in slots. I added a plug-in card to the hardware platform which contained an FPGA to speed up parts of the simulation. In addition to speeding up simulations with external logic, access to the local bus also allows us to interface IO hardware directly with the software simulation. I will discuss the implementation of this software simulation system and the issues of implementing the simulator on a stand-alone platform with external FPGA simulation support.

Chapter 2

The Abacus Simulator

The Abacus simulator is called Slocus (SLOw-abaCUS), and consists of two major parts: the simulation software and the assisting hardware. The software is written in GNU C code on a sparcstation for ease of debugging and then ported over to the i960 hardware platform. The hardware platform contains a Field Programmable Gate Array (FPGA) that attempts to replace many of the low-speed software subroutines with high-speed hardware logic.

2.1 Simulation Software

The Slocus software is written in C code because it closely mimics assembler code and can be ported across many different processors. This allows us to program and develop C code on a Sparc workstation and port the code directly over to the i960 platform simply by recompiling it. Through pointer arithmetic and program optimizations, we can speed up run time and hint to the compiler what kind of assembler loops we want. All string processing functions and array manipulations are rewritten and optimized to prevent excess overhead. The actual source code listing is included in Appendix A for reference.

Variable	Description
slocus_instr0	Low 32-bits of instruction word
slocus_instr1	High 26-bits of instruction word
slocus_control	State of the internal control bits
slocus_special	Special instruction flag
slocus_membits	High 10 bits of instruction to control DRAM
slocus_dramareg	Internal DRAM address register

Table 2.1: Global variables

This shows global variables which the machine uses to determine the environment for simulating the current cycle.

2.1.1 Storage of Machine State

The actual state of the Abacus machine is stored in an eight megabyte general global array. As the user changes the size of the machine for development purposes, pointers to different parts of the machine are simply moved within the general array to prevent the overhead of dynamic array sizing. The remainder of the general array is tagged as user memory which the developer can use as storage space to swap out state for debugging purposes or to store images. Simulation procedures only have access to machine state in the forms of the global array and the global variables shown in table 2.1. The section sizes of the Abacus machine state are listed in table 2.2. Modular procedures concentrate on computing the distinct sections of the Abacus architecture, namely the ALU operations, special instructions, the NEWS network communication and the DRAM/IO network communication.

2.1.2 Program Structure

The Slocus program consists mainly of a string parser which dispatches to different procedures based on the input command. The three main types of procedures do the following kinds of operations:

1. View/Edit the state of the machine

2. Single step the machine with a specified instruction

Variable	Size	Description
slocus_regfile	(chips)*32*64	64-bit register file for each PE
slocus_news	(chips)*32	State of the NEWS network
slocus_newsedgein	(chips)*4	NEWS in state at chip edge
slocus_newsedgein2	(chips)*4	Second NEWS in state at chip edge
slocus_newsedgeout	(chips)*4	NEWS out state at chip edge
slocus_drampedgeinhi	(chips)	DRAM-Plane high input at chip edge
slocus_drampedgeinlo	(chips)	DRAM-Plane low input at chip edge
slocus_drampedgeouthi	(chips)	DRAM-Plane high output at chip edge
slocus_drampedgeoutlo	(chips)	DRAM-Plane low output at chip edge
slocus_iopedgein	(chips)	IO-Plane input at chip edge
slocus_iopedgeout	(chips)	IO-Plane output at chip edge
usrmem	remaining	General purpose user memory

Table 2.2: Memory Allocation for Machine State

Space allocated for storing the Abacus state in the 8 Megabyte global integer array. XCHIPS and YCHIPS define the number of physical chips(chips) containing a 32x32 array of processors.

3. Help and Debugging options

The `main` procedure in the stand-alone Slocus program simply accepts strings from the standard input and sends results to the standard output. To make the program more flexible, a user may also `include` the `slocus.c` program within a higher-level C program designed to communicate with the Abacus simulator interactively. When the user includes the Slocus program as part of his own source code, the `main` function is commented out so he can send strings to the parsing routing directly. This allows a programmer to write a high-level interface to observe graphically what the simulator is doing to its registers. Image processing algorithms for the Abacus could easily be written and debugged using a graphical interface.

2.1.3 Simulating Sections of the Abacus Architecture

When the main parsing routine receives a command to single-step the simulated machine, it calls the following procedures:

1. `alu-comp` – parses the instruction and simulates the ALUs

2. `process-special` – handles special instructions

3. `dpm-comp` – shifts or loads the DRAM/IO Plane if dpclk is asserted and alters registers at chip boundaries

4. `news-comp` – propagates signals through the NEWS network and handles NEWS communication at chip boundaries.

The following describes how each procedure operates to simulate the machine and the optimizations done for each procedure.

Simulating the processor ALUs

We simulate the processor ALUs by parsing the instruction word for read addresses, write addresses, operations and special instructions. If the current instruction word is a special instruction, we skip simulating the ALU portion of the machine since we would only be executing a NOP instruction. If the instruction is not special, we continue simulating the ALU portion of each processing element in the array.

As described previously, each processor computes two bits and performs two ALU operations based on cross-coupled inputs from its two register files. Each ALU is essentially an 8-to-1 multiplexor which uses its three inputs to select one of the 8 opcode bits.

Being a SIMD machine, all processors perform the same ALU operation on each of the selected bits. Similarly, we can effectively simulate 32 ALUs in parallel because our C code handles 32-bit integers. By programming a pseudo-multiplexor using C programming logic, we cycle through all the processor rows of a chip and quickly compute 32 result bits which we write back in to the register file. The example code in figure 2-1 shows the function for computing the left ALU result for 32 processors.

Some registers in the register file are dedicated to the DRAM/IO planes or the NEWS network. The simulation parses these addresses out and retrieves the appropriate data before running the ALU computation. Similarly, Abacus processors have an `IDLE` bit which disables that particular processor from performing any operation, allowing data dependent processing. The simulation software masks the resulting 32-

```
lwd =
  (~sel2 & ~sel1 & ~sel0 &
   (((lop >> 7) & 1) ? 0xFFFFFFFF:0)) |
    (~sel2 & ~sel1 & sel0 &
     (((lop >> 6) & 1)? 0xFFFFFFFF:0)) |
       (~sel2 & sel1 & ~sel0 &
        (((lop >> 5) & 1)? 0xFFFFFFFF:0)) |
          (~sel2 & sel1 &sel0 &
           (((lop >> 4) & 1)? 0xFFFFFFFF:0)) |
             (sel2 & ~sel1 & ~sel0 &
              (((lop >> 3) & 1)? 0xFFFFFFFF:0)) |
                (sel2 & ~sel1 & sel0 &
                 (((lop >> 2) & 1)? 0xFFFFFFFF:0)) |
                   (sel2 & sel1 & ~sel0 &
                    (((lop >> 1) & 1)? 0xFFFFFFFF:0)) |
                      (sel2 & sel1 & sel0 &
                       ((lop & 1) ? 0xFFFFFFFF:0));
```

Figure 2-1: Logic Programming to Compute ALU Result

bit word with the IDLE bits to prevent the state of halted processors from changing. The IDLE can always be written to to prevent the machine from halting.

2.1.4 Handling Special Instructions

Several control signals inside the Abacus ASIC remain constant for hundreds of cycles. Instead of using additional pins to control these signals, the Abacus has special instructions which perform a NOP internally and change internal control registers during that cycle. If the 8-bit opcode for the left ALU is all zeros (i.e. clear a bit) then the Abacus looks at the otherwise irrelevant input addresses to the ALU. If the addresses are zero, a normal clear instruction occurs; otherwise, the internal instruction decoder performs a NOP for that cycle and parses the remainder of the instruction word to alter the internal state of the Abacus.

When the simulation software finds a special instruction command, it simply skips the subroutines which would normally be run during a regular instruction and branches to a separate routine which sets internal program flags for subsequent instructions.

The special instructions include the following:

- Set the DRAM address register
- Alter internal control register values
- Load an immediate constant into the NEWS network
- Load value from the IO port, optionally shifting the IO plane
- Store value to the IO port, optionally shifting the IO plane

Internal control registers control aspects of the external pad ring. They shut down the global-OR line, the DRAM interface, and the NEWS network.

Simulating the DRAM and IO Planes

To communicate with DRAM and the outside world, the Abacus uses two planes of shift registers. We can represent each row of the DRAM and IO plane with a 32-bit integer corresponding to the 32 processors in a given chip row. The data planes shift left with their input/output located at the right edge of the processing array.

The array shifts to the left; thus, we can easily left shift all the 32-bit integers once to simulate a shift in the DRAM or IO plane. We have to make sure to save the most-significant bit and replace it in the least-significant bit location. Simulating this shifting could be more efficient by storing the shifting data in vertical 32-bit integers; however, loading values to and from these networks from the register files (stored horizontally) would take much more time. Since the relative time it takes to simulate this shifting is tiny compared to simulating other parts of the machine, we essentially sacrifice no speed. We can also more easily visualize the storage of the DRAM and IO plane bits in terms of 32 rows of 32-bit integers.

Propagating Data Across the NEWS Network

Each processor can communicate with nearest neighbors or propagate information several processors away using the NEWS network. The distinct electrical path between two adjacent processors defines a segment. After the network is precharged, a single pulled-down node will propagate down a path of pass gates sixteen segments long. The path of the segments is determined by the control bits in each processor's register file.

If the Abacus ASIC logic parses the instruction word and finds that the NEWS network is written to on the following clock cycle, then it precharges the network so that values written to the NEWS network will propagate out to other chips. For simulation purposes, we determine if the code writes to the NEWS network output register by parsing the instruction word. If the NEWS network is being written to, we effectively precharge the network by setting the initial value of the network to the values in the NEWS output registers. The simulation then iterates over the network sixteen times and propagates signals down one segment per iteration. This procedure takes the vast majority of simulation time because it must be run sixteen times per clock cycle to effectively simulate the Abacus architecture.

The simulation uses page swapping for propagating information. The current state of the entire NEWS network resides in one page of memory. The next state of the NEWS network is written onto a second page of memory. At the end of the iteration, the pointers to both pages are swapped and the next iteration starts. In a situation where we must simulate 256 chips taking up multiple megabytes of state memory, the program runs much faster if we perform all sixteen iterations on a single chip before moving on to the next. Keeping the state of a particular chip localized in memory, we more effectively utilize the data cache on our Sparcstation. Since NEWS propagation does not occur across chip boundaries, keeping iterations local to a chip is feasible.

Three main locations in the register file control the propagation of signals through the NEWS network: two direction bits and a break bit. The two direction bits determine which path to a nearest neighbor is used for propagation (north, south, east, or west). The break bit actually determines whether this path is turned on or not. With a bi-directional pass gate connecting nodes, a node may either drive a signal outward to its neighbor or read a value from that neighbor. Thus, after precharging the network and initially pulling down several nodes, an un-discharged node may become discharged over the course of the sixteen iterations if one of following happens:

1. A nearby discharged node is configured (by bits in the corresponding register file) to drive the un-discharged node through a pass gate.

2. The un-discharged node is configured (by its own corresponding register file) to read the value of a nearby neighboring discharged node.

When actually simulating this network in software, I initially tried to treat each node as a 32-bit number. As a discharged node propagated, I could trace the path of propagation for debugging purposes. Unfortunately, this took 1024 words of storage for each simulated chip and simulation time was abysmally slow. Later, I optimized the simulation by computing a row of 32 nodes in parallel, much like I did earlier with simulating the ALU. Fortunately, each ASIC chip only has a 32x32 array of processors on it. Since signals do not propagate outside the chip boundary, using 32-bit integers for these calculations was easily implemented.

Using the page swapping described earlier to take the current state and compute the next state of the network, we can reasonably simulate the NEWS network using simple shifts and logical operations. Figure 2-2 shows a sample of code taken from the simulation program which computes the next state of an internal row of NEWS network nodes.

As we can see in the code, a logic one represents a discharged node. A node becomes discharged because it is already discharged, discharged by its four nearest neighbors, or discharged by looking at one of its four nearest neighbors.

2.1.5 Simulating the environment

Clocking instructions into our processor array is ineffective until we provide information to compute on. The actual physical implementation of the Abacus ASIC has a DRAM interface port on it as well as an IO port that can be attached to external data producing hardware, such as a NTSC/PAL television signal decoder. For the software simulation, several global variables exist that represent the external ports on the chip.

```
/* start with self */
*(snews++) = mynode |
/* check for above node driving down on us*/
(lregtmp[k-1] & rregtmp[k-1] & *(pnews-1) & ~(brktmp[k-1])) |
/* check for seeking upward */
(~lselfile & ~rselfile & brkfile & *(pnews-1)) |
/* check for below node driving up on us */
(~(lregtmp[k+1]) & ~(rregtmp[k+1]) & *(pnews+1) & ~(brktmp[k+1])) |
/* check for seeking downward */
(lselfile & rselfile & brkfile & *(pnews+1)) |
/* check left node driving right on us */
/* AND off the top bit to correct arithmetic shift */
(((~lselfile & rselfile & mynode & brkfile) >> 1) & 0x7FFFFFFF) |
/* check seeking left */
/* AND off the top bit to correct arithmetic shift */
((lselfile & ~rselfile & ((mynode >> 1) & 0x7FFFFFFF) & brkfile)) |
/* check right node driving left on us */
((lselfile & ~rselfile & mynode & brkfile) << 1) |
/* check right seeking */
(~lselfile & rselfile & brkfile & (mynode << 1));
```

Figure 2-2: Logic Programming to Compute NEWS State

Although it may be much easier from a simulation standpoint to write an image to be processed directly into the global register file array, presenting values at the chip boundary and clocking them into the chip creates a much more realistic situation.

In addition to presenting data at the chip boundary to compute, the toroidal structure of the Abacus must have its symmetry broken on bootup time in order to address individual processors. As mentioned before, addressing data for each processor may be directly written into the global register file, or, more realistically, loaded into the chip through special instructions that provide for immediate values.

2.1.6 Possible Future Additions

Because the entire simulator is written in C (with the exception of the external hardware portion), we can easily modify our architecture through reprogramming and compilation. Adding new features and special instructions as well as an entire new communications network is relatively easy. As stated previously, the `slocus.c` program may be included in another C program and used to help a higher level program simulate the Abacus. In the future, we hope to integrate more system functions such as the following:

- Microsequencer simulation and conditional looping based on the global-OR signal.
- Extended support for external DRAM memory storage - 2 gigabyte for the entire system.
- Simulation of data acquisition hardware such as an NTSC/PAL decoder.

2.2 Hardware Implementation of the Simulator

The following sections describe the Intel 80960 platform that the simulator runs on as well as the FPGA that supports the software simulation.

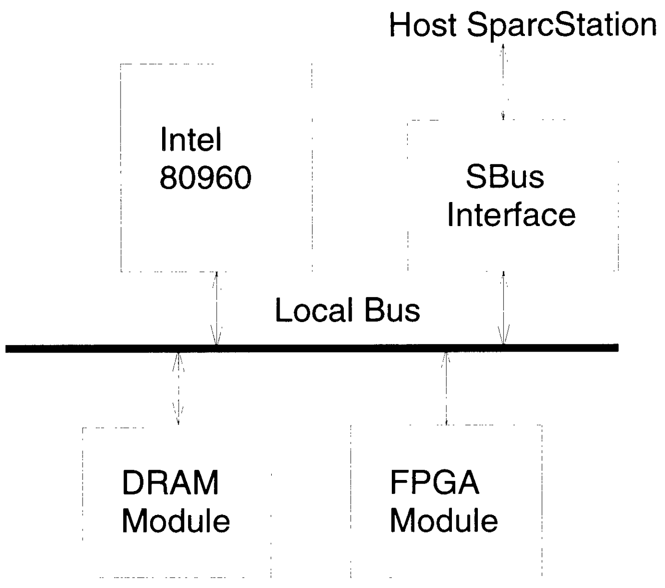


Figure 2-3: Layout of Simulation Hardware

2.2.1 Intel 960 Platform

To allow the easy addition of assisting FPGAs to our simulation, the Slocus simulator FPGA is contained on a daughter card on an already existing Intel i960 platform built by the Transit MIMD architecture group at the MIT Artificial Intelligence Lab under Tom Knight.[9] A diagram for the platform is shown in figure 2-3. The card connector slots on the platform provide full access to all of the i960 processor control signals and busses. The Slocus simulator daughter card contains sixteen megabytes of DRAM for storing the Abacus machine state. An FPGA containing the DRAM controller also has addressable hardware logic programmed into it for simulating part of the Abacus machine. For external communication, the i960 platform has an Sbus interface to a host Sparc workstations so we can download and run code on the platform.[8]

The host Sparc workstation runs a software package called `Nindy` which allows the user to download and run binary files on the i960 platform.[10] Nindy contains standard user interface provisions for accessing keyboard input from the host, printing output to the host screen, and accessing files on the host hard-drive. Code previously developed and compiled to run on a Sparc workstation can be recompiled for the i960 processor with `gcc960` and run on the external i960 platform with host user input/output through Nindy.

One major factor affecting simulation time is the relative clock speeds of the Sparc workstation and the i960 platform. Currently, the workstation runs the C simulation program at a 25Mhz clock speed with overhead for multitasking and window operations. The i960 platform runs at a much slower 8Mhz clock speed, but has the assistance of the locally addressable FPGA.

2.2.2 Utilizations of the FPGA

Field programmable gate arrays find extensive use in this simulation because they can compute complex logic in one cycle that normally takes the simulating processor many cycles. The simulation code consists mainly of loops which scan through the state of each processor and compute the next state. These loops can take unusually long for the full 256K processor implementation.

We can exploit the advantage of the common bus from the 80960 stand-alone platform. By accessing an FPGA as memory from the common bus, we write the current state of a row of nodes as well as the nodes around it into the FPGA. The FPGA implements in one clock cycle all the complex logic that the 80960 implements in several clock cycles. As a general model, the FPGA has eight general purpose input register and produces a function of those inputs.

After researching a few different FPGAs, Actel components were used to implement the external programmable logic. In addition to having most of the tools readily available, Actel components are relatively dense devices and offer faster propagation delays relative to many other general FPGAs. The basic logic block inside Actel components is a 4-to-1 multiplexor with an extra logic gate on one of the multiplexor select lines. Since we require high fan-in for computing our logic (eight 32-bit registers compute one 32-bit output), this logic block nicely suits our purposes.[1] [2]

Ideally, we would have simulated all aspects of the Abacus architecture using the FPGA support; however, the component lacked a sufficient number of gates. Although the ALU portion of the Abacus more elegantly maps to the multiplexing structure of the Actel component, simulating the NEWS network with its sixteen iterations per clock cycle is the most time-critical component in our simulation. Thus we implement the logic function described in the simulation software section. The simulator quickly writes eight values out to the input registers of the FPGA and reads out the result, implementing a single NEWS iteration in about nine bus accesses.

Actually applying the equations into the the Actel required writing out the equations in long-hand and piping them through a logic optimizer called espresso. I uses existing logic optimization tools previously developed for a different project at the lab under Andre De'Hon.[6] After running espresso to simplify the logic components, we ran the synopsis design compiler to map the logic to Actel gates. The NEWS network is a mesh in itself and mapped easily to the mesh of Actel logic blocks within the chip.

2.2.3 Future Possibilities

Access to the local bus of the i960 processor platform allows us much flexibility if we decide to expand the simulation system in the future. The Sbus interface with the platform allows the program to access files on the host Sparcstation easily. We might implement the following in the future:

- Denser and additional FPGA support to speed up simulation further
- Vector testing of a single or small array of actual Abacus ASICs
- Comparison between simulation results against the actual Abacus ASIC
- Hardware interface to an NTSC/PAL decoder

Chapter 3

Software and Algorithms

With the completed hardware simulator, we can run compiled Abacus code on it to debug algorithms for the final custom silicon. The simulator will also provide us important clues for altering the architecture to more efficiently handle common instructions. In turn, the simulation software will be altered to reflect the new changes in architecture, showing the advantages and disadvantages of such changes. We can also develop software concurrently with the hardware, thus cutting down total development time. In this chapter we will discuss the different languages that run on the Abacus, tools used to implement those languages, and a few early vision algorithms run on the simulation system that use the languages.

3.1 Language Hierarchy for the Abacus

Before we discuss the actual algorithms implemented on the Abacus using the Slocus simulator, we should discuss the different levels of software used to code the algorithms. The simulation system has four levels of code, each of which compiles to the next lower level.

1. Ascheme - A high level parallel version of Scheme

2. Assembler - Simple logical commands with macro support

3. Nanocode - The actual bits of the instruction word

4. Sim Commands - commands used to run the simulation system

Higher levels of code are generally abstract and small, making them easier to program and understand; the lower levels run more efficiently by utilizing more processor power per clock cycle. By breaking up languages into this hierarchy, we can concentrate on optimizing each level to the next lower level, thereby making compilers easier. The Abacus group intends to get all levels of code running reliably before trying to implement optimizations. When a compiler for the Abacus matures, we can flatten these levels of code out so that the compiler may convert parallel Ascheme directly into highly efficient nanocode, thus bypassing some of the inefficiencies of hierarchy overhead.

3.1.1 Ascheme

Ascheme is a version of Scheme specifically designed for the Abacus because of its allowances for parallel variables. Currently the Ascheme compiler is ineffective for coding most applications because it currently cannot exploit the advantages of a microsequencer. All loops are unrolled and compiled down to long lists of assembly code consisting mainly of optimized macros. An example piece of Ascheme code which computes optical flow for images is shown in figure 3-1.

Just from comparing the size of this higher-level program to the corresponding program in hand-coded assembler, we can see how much easier it would be to program extensive algorithms in Ascheme.

3.1.2 Assembler

Currently, most programming for the Abacus is done in assembler code. This allows us to access the low-level functions of the machine directly so we can optimize algorithms to more fully utilize every clock cycle. Assembler code is much easier to debug and understand than nanocode and lets us explicitly specify the two instruction opcodes for each processing element ALU pair. In the future, the assembler code compiler will read labels and perform conditional jumps based on the global by accessing the

```
(progn
 (set max 0)
 (set bestx 0)
 (set besty 0)
 (set i1 val)
 (set i2 (image_shift val 1 2))
 (loop i 1 3
       (loop j 1 3
             (progn (set score 0)
                    (set i3 (image_shift i1 i j))
                    (set i4 (match i2 i3))
                    (regionop i4 boo 3
                              (set score (+ score boo)))
                    (if (> score max)
                        (progn (set max score)
                               (set bestx i)
                               (set besty j)))
                    )))
 max
 bestx
 besty
 )
```

Figure 3-1: Ascheme Code for Optical Flow

functions of the microsequencer.

The main advantage of using assembler code is that we can call abstract procedures which simplify many clock cycles of operation down to one line of code. Each procedure call typically runs a piece of pre-optimized assembler code which handles functions such as addition, multiplication, shifting, and other instruction sequences which would be too tedious to hand code in the main program. The Ascheme compiler, still in development, effectively uses these optimized procedures to compile operators such as addition and shifting without diving into the complexity of the actual hardware. The next section describes many of these optimized procedures and the data types used in those procedures.

3.1.3 Nanocode

When first performing simulations of the Abacus, we did not have an assembler, and thus hand coded everything in nanocode. Nanocode for the Abacus consists of the raw ones and zeros that actually make up each 58-bit instruction word, shown in table 3.1. The most-significant bit is an instruction polarity bit which reduces the number of flipping instruction lines and saves power by optionally changing their polarity each cycle. Currently, all simulation loops are unrolled into nanocode and compiled down to simulator commands. With this implementation, we currently do not have provisions for conditional branches using the global-OR line. In the future, we will simulate the microsequencer and implement conditional loops. The microsequencer (running at half speed), will take in two 58-bit instructions (116 bits) as well as some additional bits to control program flow.

3.1.4 Sim Commands

After we actually compute several 58 bit instruction words, we want to simulate them. The simulation language consists of commands, shown in table 3.2, which alter the state of the machine as well as execute instructions. This allows the user to observe variables as the machine is running or force variables to desired values

Bits	**Description**
0-7	8-bit operand for left bank ALU
8-12	First left bank read address
13-17	Second left bank read address
18-22	Left bank write address
23-30	8-bit operand for right bank ALU
31-35	First right bank read address
36-40	Second right bank read address
41-45	Right bank write address
46	Inter-chip NEWS propagation direction
47	increment the DRAM address counter
48	DRAM-Plane clock: advance DRAM plane
49	Row/Column address select to DRAM
50	Output enable to DRAM
51	Write enable to DRAM
52	Refresh signal to DRAM
53	Write/Read signal to DRAM
54	Column address latch to DRAM
55	Row address enable to DRAM
56	Upper/Lower word latch select
57	Instruction word polarity bit

Table 3.1: Breakdown of 58 bit Instruction Word

This shows the various sections of the instruction word as interpreted by the Abacus. The instruction word polarity bit saves power by conserving the number of switching instruction bits per cycle.

Command	Description
c	reset the machine by clearing all bits to a known state
i	single step the machine with a specified instruction word
l	single step the machine with the last instruction run
s	single step the machine with a NOP instruction
rs	reads a single number from a section of the Abacus
rb	read a sequential block of numbers from the Abacus
ws	writes a single number to a specified section
wb	writes a sequential block of numbers to a section
m	moves a block from one section of the Abacus to another
p	prints a debug string to the user on the standard output
q	quits the simulation program

Table 3.2: Brief Description of Abacus Simulator Commands

during simulation. In addition to the simulator, I wrote a program (included in the appendix) to load images into the simulator, run algorithms on the images, save the resulting images, and log the time it took to compute that algorithm.

3.2 Optimized Assembler Macros Operations

Since each processor represents a single bit of an image, we need to group sets of processors together in "clusters" to perform operations on multi-bit words. To make the programmer's job easier and the compiler's job easier, I wrote several assembly code procedures to allow running of abstract operations on arbitrarily long numbers. The following sections describe the configuration of the machine to operate on certain data types and optimization of procedures to operate on those data types.

3.2.1 Common Data Types

Word boundaries and least-significant to most-significant bit paths are coded in the configuration bits loaded at bootup. The Abacus primarily performs operations on 16-bit words distributed among the processors in the following "snake" format with the numbers denoting bit position:

0	1	2	3
7	6	5	4
8	9	10	11
15	14	13	12

Four configuration bits denote the relative bit position of each processor within the word, while two configuration bits flag the most-significant and least-significant bit of the word. This square array allows us to represent a 128x128 array of pixels in our machine with a reasonable level of precision.

3.2.2 Machine Configuration

At bootup time, we need to produce all the configuration bits for the Abacus machine. Instead of producing the constants externally and piping them in using the IO or DRAM plane, we can use a special instruction to load an immediate value into the NEWS network directly from the instruction stream. The immediate value from the instruction word presents itself at the east edge of the NEWS network on each chip. We can use this to break the symmetry of our toroidal architecture and send configuration data deterministically to each processing element. Because configuration information is identical to every chip, we can configure all chips simultaneously. Any data which is chip dependent must be loaded uniquely into each chip through the DRAM plane or IO plane.

Configuration data may have either horizontal or vertical dependencies on processor location. Because our NEWS network can propagate information 16 nodes at a time, we can present data at the east edge of the NEWS network and load it two cycles later into the register files, provided that the data is identical for all the processors in the horizontal direction. Loading configuration information with horizontal dependencies takes longer because we must specify a column value with an immediate instruction and shift the NEWS network left for all 32 columns.

As described earlier, we organize our data into 16-bit snaked clusters which divides the configuration for these clusters up into eight similar columns across the chip. We can exploit this regularity by configuring the right four columns with the immediate value instruction. We can then shift the data from these four columns to the left and copy the values into each of the other seven columns of snaked clusters.

In total, we have ten configuration bits to set. Four of these we can load in about five cycles because they have no horizontal dependency. Of the ten bits, four bits flag the 16-bit snakes on the edges of the chip so we can handle latency across chip boundaries. Another four bits configure the bit order of the 16-bit snakes themselves. The remaining two bits flag the most and least significant bits of each snake so we can handle number boundaries. These last two bits may be computed from the four bits which determines the bit order within the snakes. With these configuration bits, we can perform arithmetic operations on 16-bit clusters and effectively shift 16-bit clusters in any direction across chip boundaries.

3.2.3 Shifting and Chip Boundaries

Many mesh algorithms require the values of nearby clusters for their result. A processor on one side of an inter-chip boundary may only read the value from the processor on the other side of the chip boundary after two clock cycles of latency. With clusters configured in a 4x4 array, we run into a problem when we try to shift our 128x128 mesh of data across chip boundaries. To compensate for boundary latencies, we use four configuration bits to flag clusters on the four edges of each chip. If we want to shift our array of clustered numbers north using the NEWS network, we take four cycles to shift on-chip data north and tell the horizontal row on the south edge of each chip to shift north for two additional clock cycles. This sequence corrects for the two cycle inter-chip latency. Additionally, inter-chip NEWS network pads are bi-directional but can only pass information in one direction at a time. One bit in the instruction word determines the direction of propagation for a given clock cycle, allowing rapidly configurable pads.

If we keep shifting operations within clusters, chip-boundaries become irrelevant. Many operations, such as multiply and divide, make extensive use of logical and arithmetic shifts on numbers for computation. These operations are fairly simple, as we can orient the NEWS network to point towards or away from the most-significant bit of our snake clusters. Once oriented, a single cycle can move bits one position up or down the word chain. The only major concern when doing shifts within clusters is to observe the word boundaries at the least-significant and most-significant bits. If we do not configure our NEWS network properly, one word might "spill" over into the next.

3.2.4 Addition and Subtraction

Performing additions and subtractions in bit-parallel clusters is optimized in the Abacus architecture through the NEWS network. We utilize the propagation of the NEWS network along multiple segments between processors to implement a Manchester Carry Chain. Using the configuration bits in our snaked clusters, we can orient the NEWS network to let information propagate towards the most-significant bit in our word. We then perform bitwise logical functions on our two input operands to either drive the NEWS network at our bit or let information from a lower bit propagate through to higher bits. Bitwise logical functions on the NEWS result produce the sum of the two input operands. As listed in the source code in the appendices, special precautions must be taken to prevent propagating carry bits from "spilling over" into other adjacent clusters.

Subtraction is similar to addition in that we invert the subtracted operand and activate the carry at the least-significant bit of the NEWS network. This creates the twos-complement negative of the subtracted operand as part of the addition.

3.2.5 Accumulation and Decummulation

Bit-parallel additions may be fast compared to bit-serial ones; however, adding up large numbers of variables is best done using accumulations with a carry-save-adder implementation. Instead of using the NEWS network to implement a Manchester Carry Chain, the accumulate procedure computes two numbers which we add together after all variables have been accumulated. When we implement this accumulation in assembly code, we have a procedure to start the accumulation which takes four clock cycles. Each subsequent accumulation takes two cycles and the final addition takes five more cycles. This makes accumulation more efficient than simple adding if we accumulate more than eight numbers, such as in multiplication.

Decummulation, the opposite of accumulation, is where we wish to subtract a number from our running total instead of add a number to it. This operation also only takes two cycles and helps the efficiency of algorithms which require many additions and subtractions to achieve a final result. The code for both accumulation and decummulation exists in the appendices.

3.2.6 Multiplication, Division and Remainder

Although some algorithms require multiplication by variables, most of them multiply registers by constants. The multiplication algorithm, listed in the appendices, performs sixteen accumulations after shifting and testing bits in the source operands to achieve its result. This algorithm was programmed mainly for use by an abstract compiler. Mulitplications by a constant become much more efficient when the user tailors the assembly program to optimize for the constant. For example, multiplying by five would entail logically shifting a register left by two bit places and adding it to itself.

Currently, procedures for computing a quotient and remainder have not been written. As mentioned before, most algorithms require dividing by a constant, such as the surface approximation algorithm described later. It is generally better to optimize division by constant using additions and shifts. Division by five is as simple as multiplying by three and adding the result shifted to the right by four, eight, and twelve bit places. In essence, we are multiplying by `1100110011` and shifting right twelve places, thus multiplying by `.001100110011`. See the surface reconstruction algorithm listed in the appendices for an example of dividing by five.

3.2.7 Comparisons

Often times, an algorithms will conditionally perform a function based on the comparative values of two numbers, such as loading the maximum of two variables or testing them for equality. When comparing word clusters to determine which is greater, we compare corresponding bits starting with the most-significant bit and cycling down to the least-significant bit. As soon as we find a mismatch, we know the numerically greater cluster. We must then broadcast the difference in that single bit location to the rest of the bits within the cluster so all cluster processors can perform their conditional operation.

The NEWS network provides an ideal method for quickly comparing two numbers in this fashion because it propagation through sixteen bits in a single cycle. By using a method similar to the Manchester Carry Chain described in the section on additions, processors can drive their nodes if the first variable bit is greater than the second. Processors with same corresponding bits simply let values propagate from the LSBit to the MSBit. A value of one will propagate to the MSBit if the first variable is greater and zero will propagate if the second variable is greater or equal to the first. We can then clear the NEWS network and broadcast the result captured at the MSBit back to the remaining fifteen processors. See the `compare.asm` file in the appendices for the implementation of this algorithm. Checking variables for the same value is easier since we can simply let bit differences drive the NEWS network and propagate to all processors.

After all processors in a cluster have received a conditional bit based on the comparison, they can set their `IDLE` bits to perform conditional statements. For example, if we wanted to increment a counter if R1 was more than R2, we would compute a greater-than function on R1 and R2, "turn off" all the processors that contained a smaller R1, and then increment the counter. After the frozen processors fail to increment their counter, we can "turn on" everyone again by disabling the `IDLE` bit.

3.2.8 Saving state to DRAM

For extensive algorithms which require amounts of memory beyond the two 32-bit register files, the user may use the DRAM plane network to save registers to an external DRAM memory. Currently, no algorithms have needed this resource, so no procedure has been written to save the state of a specified register. A future version of the simulator with associated assembly procedures will support this function.

3.2.9 Cycle Counts for Simple Procedures

Table 3.3 lists the cycles counts necessary to implement the listed functions. All procedures were written to operate on all registers in either register file. Thus, some extra clock cycles within each procedure are needed to load input values into temporary registers located in the proper bank.

3.3 Vision Algorithm Implementation

Due to its architecture, most algorithms for the Abacus will concentrate on fast image processing. Applications will include object recognition through bit plane convolution and high-definition television frame processing. The IO plane port allows the user to quickly shift in images to each processor for computation and shift them back out. During simulation, loading images through the IO plane may take a few thousand cycles to propagate the entire image to all the processors. Final simulations should take this image loading overhead into account; however, for simple algorithm purposes, we will directly write images into the register file to cut simulation time.

The following algorithms are currently written for the Abacus and simulated on the Slocus simulator:

- Gaussian convolution algorithm
- Edge detection algorithm
- Surface approximation algorithm

Procedure	**Cycles**	**Description**
init-config	146	Initial machine configuration
add	5	Perform addition
acc-start	4	Start a sequence of accumulations
accumulate	2	Accumulate two registers
minus	8	Subtract two numbers
unacc-start	4	Start a sequence of decummulations
unaccumulate	2	Decummulate the accumulator
greater-than	9	Compare and distribute the result
less-than	9	Compare and distribute the result
equal	4	Produce all ones if all bits equal
orient-high	3	Data flows to MSB
orient-low	3	Data flows to LSB
orient-north	3	Data shifts up the toroid
orient-south	3	Data shifts down the toroid
orient-west	3	Data shifts left on the toroid
orient-east	3	Data shifts right on the toroid
global-or	9	Computes and broadcasts global-OR
sign	4	Broadcasts the sign bit to all other word bits
mult16	131	Multiply two 16-bit numbers
asr	4	Arithmetic shift right
asr2	5	Arithmetic shift right by two
asr4	7	Arithmetic shift right by four
lsr	2	Logical shift right
lsr2	3	Logical shift right by two
lsr4	5	Logical shift right by four
asl	2	Arithmetic(Logical) shift left
asl2	3	Arithmetic(Logical) shift left by two
asl4	5	Arithmetic(Logical) shift left by four
numgen-zero	2	Shift left and add zero
numgen-one	2	Shift left and add one
shift-cluster-north	8	Move clusters up on toroid
shift-cluster-south	8	Move clusters down on toroid
shift-cluster-east	8	Move clusters right on toroid
shift-cluster-west	8	Move clusters left on toroid

Table 3.3: Cycle Counts for Various Optimized Procedures

- Object tracking algorithm

Assembler code for all of these algorithms reside in the appendices and are based on algorithms implemented on the massively parallel Silt chip created by Bolotski and Barman.[3]

3.3.1 Gaussian Algorithm

We can convolve our image with a Gaussian by passing it through two triangular filters, one which is horizontal and the other vertical. Thus, we can achieve our Gaussian transform by performing a two-dimensional convolution with the coefficients $\frac{1}{4}, \frac{1}{2}, \frac{1}{4}$ on both axes. The following edge detection algorithm shows the results of the Gaussian algorithm.[13]

3.3.2 Edge Detection Algorithm

We can find edges within a picture using the Marr-Hildreth edge detection algorithm by passing the image through a Gaussian filter, performing a Laplacian transform and finding zero-crossings.[12][15] We can use the Gaussian filter described in the previous section for this. For the Laplacian filter, we convolve the image with the discrete array shown here:

1	4	1
4	-20	4
1	4	1

The Laplacian filter produces both positive and negative numbers in our image. We can use a sign function to determine the polarity of each pixel and then `XOR` the result with a shifted version of itself. These `XOR`s find the horizontally and vertically dependent zero crossings, after which we can `OR` these functions together to get our edges.

Figure 3-2 shows the results of our simulations. The images show that as we apply more Gaussian transforms, the image smears and we pick out larger and larger objects. Additional Gaussians may cut down on high-frequency noise, but they remove finer detail as well.

3.3.3 Surface Calculation Algorithm

Many times for vision applications, we need to reconstruct a surface from noisy input data. We can use the following equations that exploit the local communication of the Abacus:[11]

$$u_{i,j}^{k+1} = \frac{1}{4}\left[(u^k i - 1, j + p_{i-1,j}^k) + (u_{i,j-1}^k + q_{i,j-1}^k) + (u_{i,j+1}^k - q_{i,j}^k) + (u_{i+1,j}^k - p_{i,j}^k)\right]$$

$$p_{i,j}^{k+1} = \frac{1}{5}\left[(u_{i+1,j}^{k+1} - u_{i,j}^{k+1}) + p_{i,j-1}^k + p_{i,j+1}^k + p_{i-1,j}^k + p_{i+1,j}^k\right]$$

$$q_{i,j}^{k+1} = \frac{1}{5}\left[(u_{i,j+1}^{k+1} - u_{i,j}^{k+1}) + q_{i,j-1}^k + q_{i,j+1}^k + q_{i-1,j}^k + q_{i+1,j}^k\right]$$

We see from the equations that the computed depth is a function of current depth of surrounding pixels and surrounding slopes. Computed slopes are an average of surrounding slopes and difference in depths. Figure 3-3 shows the result of passing a grayscale image through the algorithm with initial slopes of zero.

3.3.4 Optical Flow Algorithm

Often times, we use high speed vision machines for tracking moving objects or to determine whether an object has moved or not. We can use optical flow techniques to do this. To see if an object has moved, we take a picture at a given point in time and compare it with another picture taken at some given time later. We shift the second picture to locally by a square of pixels and compare it to the first. The comparison consists of adding up a window of pixels in which corresponding pixels from both images are the same. The window with the highest score (i.e. the most similar pixels) wins and thus determines the highest probability displacement for the moved object. This algorithm works best when the two compared pictures are binary bitmaps. We can use the edge detection algorithm described earlier to convert a

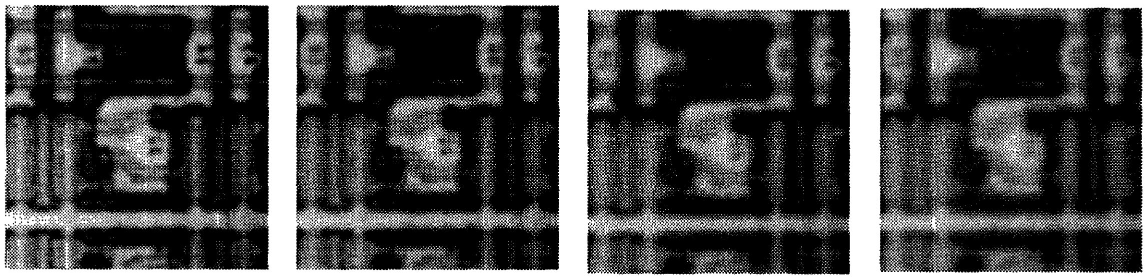
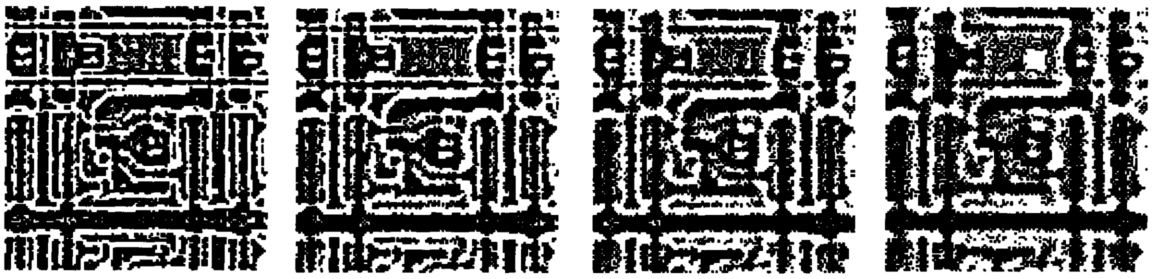
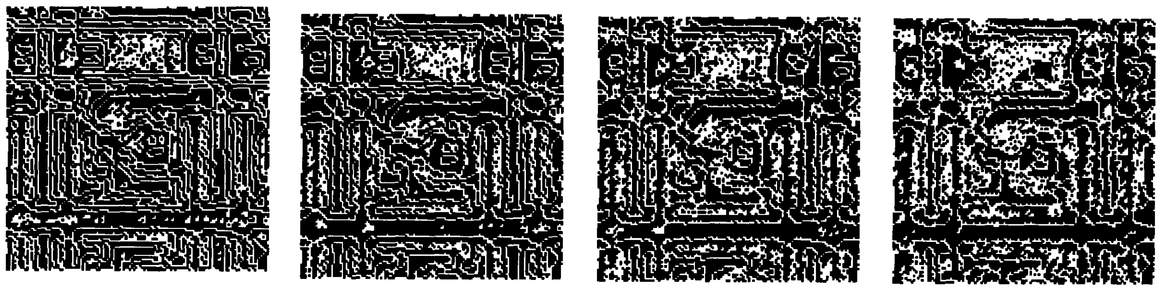


Image Passed Through Gaussian Filters



Zero Crossings of Image After Laplacian



Edges as Computed from Zero Crossings

Figure 3-2: Results of Edge Detection Algorithm
This figure shows the various stages that an image goes through during edge detection. The first row shows the image resulting from subsequent double Gaussian filters. The second row shows the zero crossings resulting from the Laplacian filter. The third row shows the edges resulting from the horizontal and vertical zero crossings.

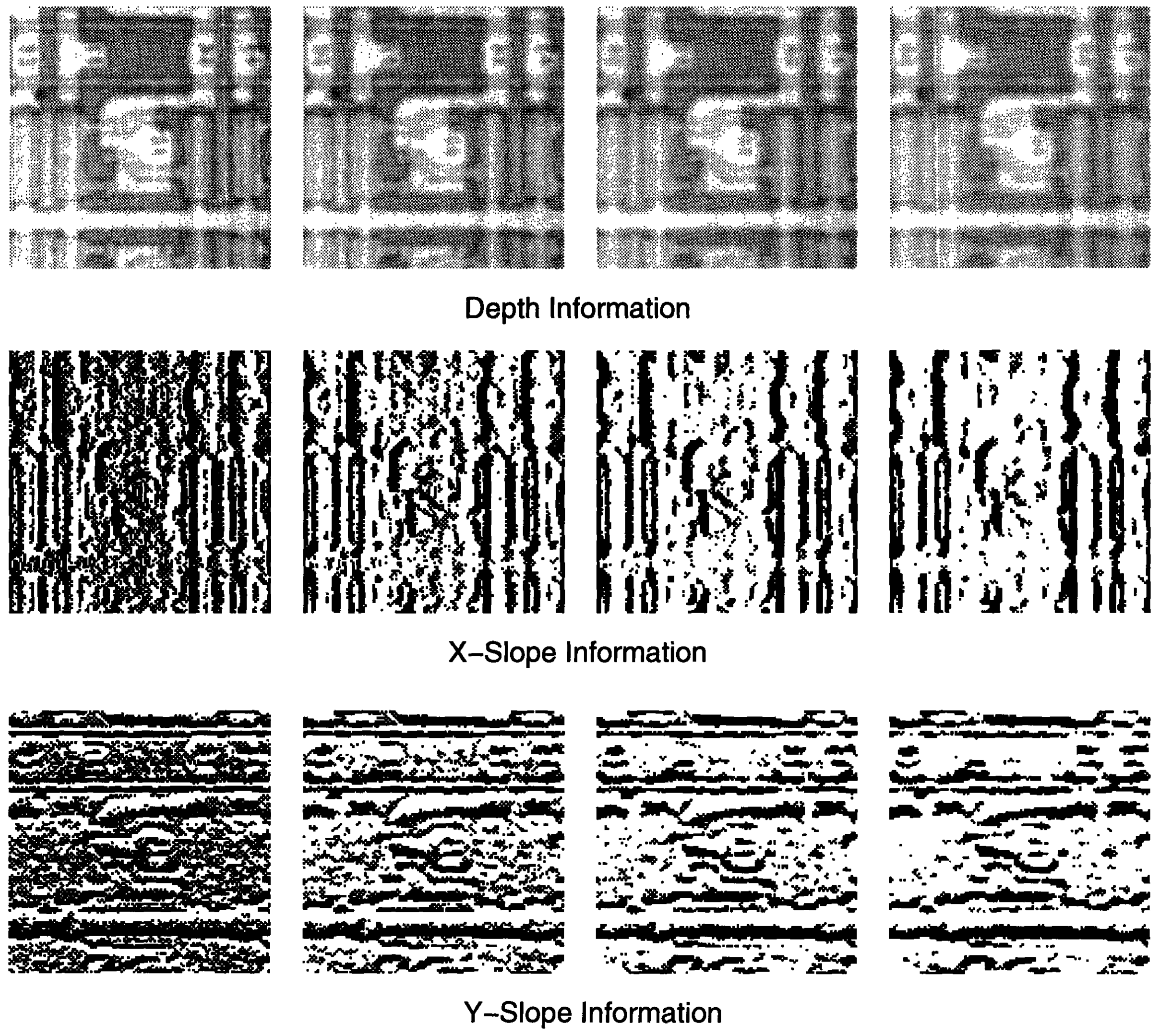


Figure 3-3: Results of Surface Reconstruction Algorithm

This figure shows results of the surface reconstruction algorithm. We initially specified slopes of zero. These four columns show resulting depths and slopes after subsequent iterations. Because of twos-complement encoding, negative slopes are shown as bright pixels.

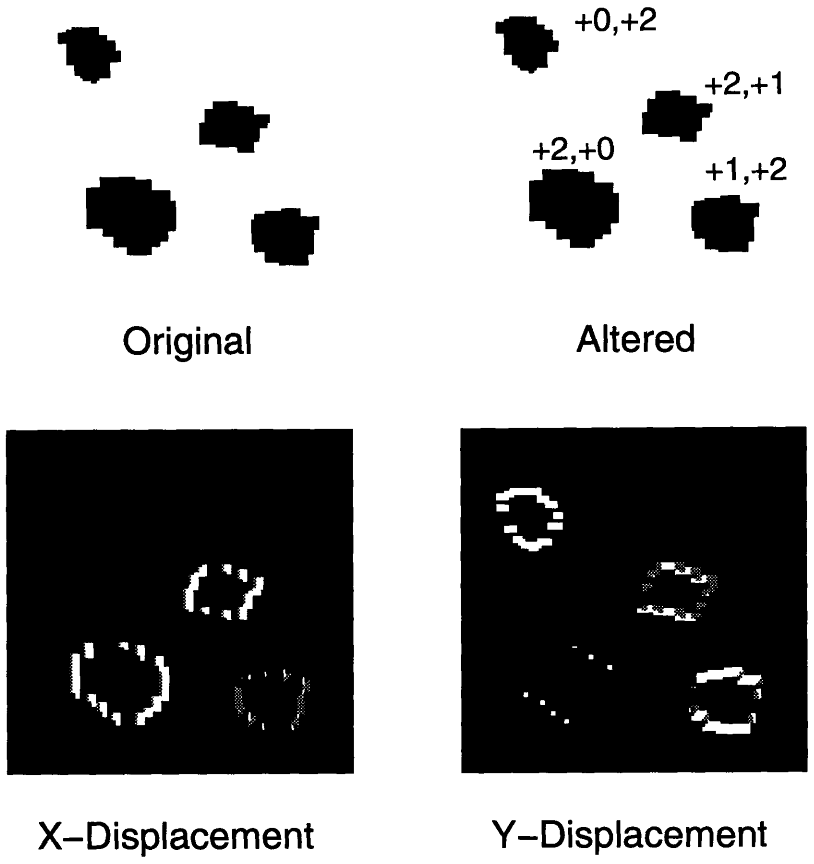


Figure 3-4: Results of Optical Flow Algorithm
This figure shows the optical flow result for some shifted objects. The altered picture lists the corresponding displacements. The lower two pictures show the X and Y displacements respectively with brighter pixels being greater displacement. This program only summed up a 3 x 3 region and allowed a maximum positive displacement of two on both axes.

gray-scaled image into areas of ones and zeros.[12][14] The algorithm was hand coded in assembly language so we only allowed a maximum positive shifting on both axes of two. We also only used a summing window of three pixels on a side, which led to further glitches in the output. Given a compiler and microsequencer we could more effectively implement this algorithm with smaller code and greater accuracy.

3.3.5 Algorithm Execution Cycles

Table 3.4 lists the number of clock cycles required to perform a single iteration of each algorithm. Code for each of these algorithms was hand coded in assembler using

Algorithm	**Cycles**
Configuration	146
Gaussian	119
Edge Detection after two Gaussians	403
Surface Reconstruction	388
Optical Flow max 2 shift 3x3 sum	1811

Table 3.4: Cycle Count for Different Algorithms

This table shows the number of clock cycles required to perform one iteration of each listed Algorithm. Cycle counts reflect hand coding assembly language with pre-optimized macros.

predefined macro procedures. The Abacus Ascheme compiler was not reliable enough at the time to code the various algorithms, although it would have been much easier to do so. Our purpose in generating these algorithms is to determine how effective the different simulation platforms handle the algorithms. Also, by simulating the algorithms on the full 256K processor Abacus machine, we will more fully grasp the abilities and limitations of the architecture and make changes to the final ASIC implementation.

Chapter 4

Observations

4.1 Simulation Times

Now that we have coded our simulation and written some algorithms, let us take a look at some of the times for our simulations. We should also discuss the differences between platforms and how the simulation times depend on these differences. The platforms we ran the algorithms discussed in the previous chapter on include:

- An unloaded Sparcstation 1
- An unloaded Sparcstation 10
- The Transit Intel 80960 Platform
- The Transit Intel 80960 Platform with assisting FPGA

As we can see from tables 4.1 and 4.2, simulation times on the Sparc platforms were much faster than simulation times on the i960 platform. Sparc platform results were obtained on an unloaded computer at night; thus, a system running additional simulations or an active window manager would take longer.

Algorithm	**Sparc 1**	**Sparc 10**	**i960**	**i960/FPGA**	**Abacus**
Gaussian	79s	22s	990s	989s	0.952us
Edge Detection	265s	74s	3353s	3349s	3.224us
Surface Approx.	254.5s	71.5s	3186s	3175s	3.104us
Optical Flow	1186s	332s	14760s	14726s	14.49us

Table 4.1: Times for Various Algorithms

This shows the times (in seconds) required by different machines to compute the listed algorithms. The last column shows the time it would actually take the final hardware to compute the algorithm, assuming it runs at 125 Mhz.

Machine	**Cycle Time**	**Variation**
Sparc 1	0.658s	±0.89%
Sparc 10	0.184s	±0.50%
i960	8.2s	±52.3%*
i960/FPGA	8.18s	±52.4%*
Abacus	8ns	–

Table 4.2: Cycle Times for Different Machines and Percentage Variation

*This table shows the average cycle times for simulating the Abacus computer as well as the percentage of variation in times due to additional loading on the machine. The i960 columns reflect times gathered over long simulation times *(see text). The final custom silicon will run eight orders of magnitude faster than our Sparc 10 simulation, which runs about one order of magnitude faster than the i960 platform.*

4.2 Abacus Code Dependencies

If we concern ourselves only with overall algorithm simulation time, neglecting cycle simulation times, we should make sure our algorithms are coded in as few cycles as possible. If we simulated algorithms produced with an unoptimized Ascheme compiler, our simulations would take much longer due to the abstraction levels. The simulation times listed above reflect the efficiency of hand-assembled code. Various algorithms also use the NEWS network more more than others, which takes more time within the NEWS network simulation routines.

4.3 Platform Dependencies

The simulation times for the Sparc platform greatly differed from the simulation times on the external Intel 960 platform due to the hardware. I've observed the following points to have a significant impact on the simulation speed:

- Architectural differences in the Sparc processor and the i960 make compiling for both of them different.
- The optimizing compilers used for each might have created significantly different code efficiencies.
- The Sparc 1 clock (25 Mhz) is 5 times faster than the i960 clock (5 Mhz) while the Sparc 10 clock (50 Mhz) is even faster.

Major cause for the variation in simulation times on the i960 platform came from changes in the I/O bandwidth over time. The I/O interface program(Nindy) running on the host Sparc station loses execution priority over time and runs more slowly. Our benchmark test ran four iterations of each vision algorithm in sequence, taking over 25 hours of actual computation time. Simulated cycle times at the beginning of the benchmark only took 3.9 seconds. This time exponentially slowed to 8.2 seconds after about two hours of simulation time, thus showing that I/O consumed fifty percent of simulation time. Additionally, loading and saving images to simulated registers on the i960 took roughly five minutes for each register transferred, adding another 3.5 hours to the simulation. We actually save an order of magnitude in time initially configuring the simulated Abacus by using simulated special instruction cycles instead of using the I/O system to directly load registers. I/O time on the Sparc consumed only one percent of simulation time, including transferring images to and from simulated registers. Although we might increase the clock on the i960 platform, the fraction of time spent on I/O operations would increase until time spent on actual simulation would be insignificant. This I/O bandwidth problem effectively limits the ability to interactively monitor the simulation on the i960 platform.

In addition to faster simulation times, differences in the platforms also affected ease of simulation and cost of the system. Some of these points include the following:

- Reprogramming the logic of the FPGA for the i960 platform takes much more time than simply changing the C-code. C-code also has debugging facilities that the FPGA does not.

- The Sparc platform has a much better user interface to the simulation. We could program code to interactively use simulator results for screen graphics.

- The i960 is an external piece of hardware with a custom FPGA attachment. This costs us additional money and debugging time.

- We can run multiple separate cases of simulations on a multitasking Sparc platform. We can also run separate simulations on different Sparc platforms to distribute the load. We would need additional i960 platforms for each separate simulation.

- Sparc platforms are more readily available and probably have better C optimizing compilers from the long-established market.

- The i960 platform has the advantage of access to its local bus. giving us the ability to create a faster I/O bus or hardware interface to realtime data such as an NTSC coded signal.

Chapter 5

Conclusion

In an effort to cut down simulation times, we've demonstrated the use of an optimized C program to simulate the Abacus, a massively parallel SIMD architecture. The C program was ported to an auxiliary hardware platform and run with the assistance of an FPGA hardware unit. We've implemented algorithms with assembler code and simulated those algorithms to test the effectiveness of the simulator running on different platforms. Looking back at the simulation results, the following section will discuss possible improvements to the system and compare optimized C simulation with other levels of simulation. We will also ponder the validity of using optimized C with external FPGA support as a viable solution to simulating SIMD architectures.

5.1 Improving the Simulation

We used the Intel 80960 processor for our auxiliary platform processor since it was debugged and readily available to us from the Transit Project being developed in the same lab.[9] We have seen from raw numbers that our Sparc simulations were easier to implement and faster; however, we have not considered how speeds might change if we altered our i960 platform to be more efficient. We should consider the following options for improving the simulation system:

5.1.1 Additional Hardware

We have seen that our simulation runs faster on a performance Sparc workstation than a costly auxiliary hardware platform with FPGA support. If we abstract away the problem of I/O briefly, we could consider altering our auxiliary hardware platform to make it significantly faster than the host Sparc workstation.

IO optimization

Many poorly designed high speed computers today are I/O bandwidth limited. The major factor which reduced our simulation time for the external i960 platform consisted of I/O operations to the host workstation. Our Sbus interface hardware could be dramatically improved with faster parts and a more efficient design. By downloading the entire set of test vectors to the external platform before simulation, we could avoid I/O delays during simulation; however, this would prevent us from accessing the simulation interactively.

Using Larger FPGAs

One factor limiting our current design is the size of our FPGA. The Actel component we use cannot contain all of the logic that would have liked to implement. Our design was not pin limited as we only needed to interface with the i960 local bus. By using a larger FPGA we could have simulated the NEWS network of the Abacus more effectively and also included simulating the ALU portion of the Abacus as well. Adding additional smaller FPGAs might also help solve our problem of lacking enough register space. However, partitioning up the NEWS network logic into many FPGAs would be difficult and waste clock cycles on inter-chip communication. Reprogramming the FPGAs for each simulated architecture would also be less flexible and consume more development time. Moreover, adding more FPGAs or larger ones with faster cycle times would cost significantly more and might not justify the incremental speed increase in simulation.

Using a Large Array of FPGAs

If cost was not an issue with our simulation system, we could possibly implement a large array of large FPGAs to simulate our architecture. Some problems arise with this. FPGAs typically do not have large amounts of RAM on-board; thus, we need to add external SRAMs to each FPGA for storage of machine state. Each FPGA would probably need its own controller to cycle though the machine state and compute the next state. An alteration in the programming would mean reburning an EPROM to serially shift in a new configuration. Debugging would be extensive and difficult due to the large number of hardware pieces. Connections among FPGAs and message passing protocols would depend on the simulated architecture. Such an interconnection network would tend to be pin limited and possibly difficult to route. Additionally, a host processor running the array would have difficulty monitoring it and accessing a piece of information from a particular FPGA. Again, the simulation would tend to be I/O limited because we would have a tough time getting information in and out of the array at high speed. This setup of many FPGAs pushes closer to actually building the SIMD architecture itself and presents itself as inflexible to architectural changes.

Using the Efficiency of a DSP Chip

Simulating the Abacus architecture consists mainly of doing bitwise and simple arithmetic operations on very large arrays of machine state. Many high performance DSP chips exist today which perform these array operations extremely well. A typical DSP chip, such as the TMS320C31 from Texas Instruments, implements parallel instructions and contains instructions to efficiently perform tight loops with little overhead. Because of their simpler architectures, DSP chips often run at higher clock speeds than more general microprocessors. With on-board memory, parallel instructions, high clock frequency, and quick loops, we could very efficiently utilize a DSP chip to simulate our SIMD architecture.[16]

External Interfaces

If we want our simulation to interact with a real-time data producer, we can transfer data in two ways. We can collect data from the real-time source into a file, process the file with the simulator, and send the file back out to the real-time device. However, this method takes up much I/O time and lags due to the latency of the disk storage. If we develop a fast simulation on an external platform with a local bus, we can directly interface the real-time data device to the local bus and communicate with the simulation interactively. We could process NTSC signals from a television camera this way, or other image producing devices.

An Economical, Flexible Compromise

Combining the structures of large FPGAs to compute our complex logic, a DSP chip to quickly address our machine state array, and a general purpose processor to control the system, we could implement a fast general purpose simulation system. We make a tradeoff between speed and complexity as we add more hardware and increase programming time. We would need programming for the i960, the DSP and the FPGA to all act together, thus making debugging difficult. We could download sections of code to the i960 processor from a host workstation and have the i960 program the FPGA as well as set up the boot code for the DSP before resetting the system. The DSP chip and i960 could both interact with a large section of DRAM memory through DMA channels as well. An fast IO controller (possibly ethernet) or external signal interface (such as NTSC) could use DMA as well. With everything on a common bus, we have the flexibility of adding more FPGAs as we need to simulate larger sections of logic. Multiple DSP chips could even do processing on their own internal memory in parallel. This setup provides a flexible platform which we can upgrade and alter for many different kinds of SIMD architectures without spending excessive money.

5.1.2 Software

Outside of optimizing hardware, many optimizations can be programmed in software to speed up simulation. For our customized C program, most of these optimizations can be handled with the optimizing compiler but we do additional optimizations which take advantage of the hardware available to us.

When using the i960 platform, much of our simulation time was taken up by I/O transfers to and from the host Sparc workstation. We can dig down into the machine code for these transfers and optimize it to speed up the Sbus transfers. Similarly, we could write software that downloads the state of the machine as well as all the instructions to be executed, storing them in local memory to avoid I/O latencies. Also, by simulating a microsequencer for our SIMD machine, we can compress our code into loops instead of unrolling everything into a long list of instructions.

By writing programs that effectively utilize the instruction and data cache on the simulation processor, we can cut our simulation time significantly avoiding cache misses. Even running at full clock speed, the i960 platform may still be slower than a performance Sparc workstation with a large local cache and fast I/O. If this is the case, we either need to run the simulation directly on the Sparc and save on hardware costs, or attach additional hardware to the i960 processing platform to make using the platform faster. As stated before, the faster speed of an auxiliary platform might be unimportant if we do not pay close attention to the I/O bandwidth to the host Sparc workstation. We can save money by more effectively using our limited hardware than we can using specialized hardware at the problem. Ideally, a combination of both optimized software and fast, flexible hardware would work best.

5.1.3 The Future

What does the future of computer system simulation have in store for us? We have already discussed the use of simulating our SIMD architecture with a huge array of FPGAs. Future FPGAs might be fast enough and have enough density to handle complex architectures. This system would be fast but inflexible to architectural changes and difficult to communicate with. We could also have a controlling processor download code to a set of DSP chips and FPGAs on a common bus. This system would be far more flexible, but bus transfers would slow the system down and the inherent serial simulation would be significantly slower than the parallel arrays of FPGAs.

MIMD Simulating SIMD

We could use a massively parallel MIMD computer with message passing and synchronization capability such as the Thinking Machines CM5. Such architectures are good at general purpose massive parallelism; however, message passing and synchronization often takes many hundreds of cycles depending on the routing network. Future MIMD machines might have faster networks and better synchronizers. For the case of the Abacus where we want to simulate an array of 256 chips, we could simulate each separate chip instead of each separate 1-bit processor on a node of the CM5 and only message pass at the end of each cycle. This would cut down on message passing and synchronization.

Computing with DPGA Processors

In our current simulation system on the i960 platform, the superscalar processor with fast instruction cache closely matched the speed of the external FPGA. Future processors might avoid the inter-chip communication delays by including a Dynamically Programmable Gate Array (DPGA) on the same substrate as the processor. The processor core could have direct access to the programmable logic as well as the instruction and data caches. We might even use a MIMD array of these DPGA processors with fast message passing and synchronization to more quickly simulate SIMD architectures.[5][7]

5.2 Tradeoffs with Other Kinds of Simulations

We could have simulated the Abacus SIMD architecture using Verilog, Hspice or even a high-level behavioral model instead of the current optimized C program. Different kinds of simulations trade information for speed. Thus, we need to determine which kind of simulation will run the fastest without sacrificing the critical information we need to debug algorithms.

5.2.1 Hspice

Hspice simulations offer invaluable analog electrical information about the transistors switching within a VLSI circuit. Unfortunately, simulations take up much memory and run very slowly. Thus they are great for optimizing a small specialized circuit within the computer that might get replicated man thousands of times throughout the architecture. In the case of the Abacus, simulation of two processing elements for forty nanoseconds (5 clock cycles) took about twenty minutes of computation on a Sparc 10.

5.2.2 Verilog

Verilog simulations go one step higher than Hspice in the ladder of abstraction as they more quickly result in signal timings, but do not produce analog waveforms. We have traded off information for speed in this case. Unfortunately, Verilog simulations take up large amounts of memory as well, which causes huge lags in simulation time if the processor swaps memory space out to disk. We can effectively simulate small clusters of Abacus processing elements with this to determine the interactions among processors, but for our 256K node machine, this simulation would tend to be disk I/O limited.

5.2.3 Optimized C

Our current optimized C implementation simulates nanocode and is able to determine the exact state of the registers and communication networks in the Abacus on any given clock cycle. On a Sparc 10 we can simulate about 3000 clock cycles for the entire 256K node machine in about ten minutes. While we don't get exact timing information, we can still debug large portions of code for the whole machine in a reasonable amount of time and get feedback from the instruction stream to detect possible flaws and improvements in the hardware.

5.2.4 Abstract Behavioral Model

We could run a behavioral model of our system in Lisp to interpret the algorithms written for the Abacus in Ascheme code. This might be great for future programmers who want to write extensive algorithms taking millions of clock cycles. The simulation would be extremely fast; however, we could only debug Ascheme code and would not see what the actual registers in the Abacus were doing.

5.2.5 Optimal Choice

Our optimal choice for which simulation to use is clearly dependent upon what we try to create. In the case of developing the actual Abacus hardware, we need to run analog simulations using Hspice to engineer special circuits throughout the machine. We can propagate those timing numbers up to Verilog for use in larger scale simulations which assume the specified timings. The optimized C implementation is ideal for developing nanocode and compilers. The abstract behavioral model would be good for programmers who wish to write quick pieces of code they can compile down to nanocode without worrying about all the levels of abstraction. Thus, for creating the hardware, Hspice and Verilog is best while developing compilers and low-level code requires simulation of many clock cycles with an Optimized C simulator.

5.3 Accomplishments

We programmed C and built hardware, forming the Slocus simulator to quickly simulate code for the non-extant Abacus. This allows us to develop code concurrently with the actual hardware as well as find architectural flaws and improvements in the hardware through code analysis. By utilizing an auxiliary processing platform with FPGA support we hoped to decrease our simulation time beyond that of a performance workstation. Unfortunately, the hardware platform ran an order of magnitude slower than the workstation. Even though we might be able to clock the the external platform at its full rate in the future, the computation power of the external device was swamped by the extensive latency needed for input/output operations.

5.4 Was it Worth it?

I found the Abacus simulator to be a very effective tool in debugging algorithms requiring hundreds of clock cycles to execute. The optimized C code without the auxiliary hardware platform ran smoothly on all the Sparc stations in the lab. Additionally, the code was easily ported from one platform to another with little hassle which makes compiling for faster workstations simple.

Not only did I get feedback from the algorithms, but I found bugs in the assembly code compiler as well. After the Abacus simulator was debugged (primarily problems with NEWS network propagation), it worked flawlessly and even pointed out very subtle errors with the assembly language compiler as well as my hand coded algorithms.

Unfortunately, I found the i960 hardware platform lacking in its simulation ability. As mentioned before, I/O was extremely slow and its 5 Mhz clock cycle made it an order of magnitude slower than corresponding Sparc 10 simulations. Engineering and debugging the hardware to simulate the Abacus took much time and programming the assisting FPGA was tedious. The additional monetary cost for the i960 platform, Sbus interface and assisting FPGA daughter card was also undesirable. Given the funds, I would have rather purchased a general simulation system than designed and built my own custom one. A general simulation system could simulate many different architectures and could be used by many people. Ideally we should be able to specify our logic and registers in an abstract code and compile it directly to an array of FPGAs or similarly configurable logic.

Engineers often develop extensive software for a machine only after they build the running hardware, which causes them to miss flaws in the architecture and to cause their customers to wait months before being able to use the new hardware. Programming an optimized C program to quickly simulate the hardware would be valuable for finding bugs in the architecture and for developing extensive software so that it can be released with the machine and not after.

Bibliography

[1] Actel Corporation, 955 East Arques Avenue, Sunnyvale, CA 94086. *ACT 2 Field Programmable Gate Array*, 1990.

[2] Actel Corporation, 955 East Arques Avenue, Sunnyvale, CA 94086. *ACT Family Field Programmable Gate Array DATABOOK*, April 1990.

[3] M. Bolotski, R. Barman, J. J. Little, and D. Camporese. Silt: A distributed bit-parallel architecture for early vision. *International Journal of Computer Vision*, IJCV-11:63–74, 1993.

[4] Michael Bolotski. Abacus: A high-performance software bit-parallel architecture. AI memo, in preparation, MIT, 1993.

[5] Michael Bolotski, André DeHon, and Thomas F. Knight, Jr. Unifying fpgas and simd arrays. In *FPGA Workshop*, 1994.

[6] André DeHon. Working with actel designs. Transit Note 72, MIT Artificial Intelligence Laboratory, July 1992.

[7] André DeHon. Dpga-coupled microprocessors: Commodity ics for the early 21st century. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[8] André DeHon and Samuel Peretz. Transit sbus interface. Transit Note 67, MIT Artificial Intelligence Laboratory, June 1992.

[9] André DeHon, Thomas Simon, Nick Carter, and Henry Minsky. Charles: Second revision mbta node. Transit Note 63, MIT Artificial Intelligence Laboratory, January 1992.

[10] Ian Eslick and Henry Minsky. Gdb/nindy - debugging and run-time software support. Transit Note 77, MIT Artificial Intelligence Laboratory, November 1992.

[11] John G. Harris. The coupled depth/slope approach to surface reconstruction. Master's thesis, MIT, May 1986.

[12] Berthold Klaus Paul Horn. *Robot Vision.* MIT Electrical Engineering and Computer Science Series. The MIT Press, Cambridge, Massachusetts, 1986.

[13] James J. Little, Guy E. Blelloch, and Todd A. Cass. Algorithmic techniques for computer vision on a fine-grained parallel machine. *IEEE Trans. Pattern Analysis and Machine Intellegence*, 11(3):244–257, 1989.

[14] James J. Little and Heinrich H. Bülthoff. Parallel optical flow using local voting. AI Memo 929, MIT, July 1988.

[15] David Marr. *Vision.* Freeman, 1982.

[16] Texas Instruments Incorporated, P.O. Box 1443, Houston, TX 77251-9879. *TMS320C3x User's Guide*, 1992.

Appendix A

Code Listings

A.1 Abacus Simulation System

The following pages list code for the following parts of the Abacus Simulation system:

1. `slocus.c` - The main core of the simulation which computes the behavior of the Abacus architecture.

2. `sloconv.c` - A program to convert nanocode produced by the assembler into simulation commands that `slocus.c` recognizes.

3. `slorun.c` - A program that loads `.pgm` images into specified registers and runs algorithms on those images while logging simulation times and other statistics.

A.1.1 Slocus Simulator

```
/* C code for the Slocus Abacus machine simulator */
/* Written by Tim Kutscha for the Abacus Project */
/* This rev 11-10-93 */

/* suggestions: */
/* calculate own array address to save processor time */
/* how do we do global OR ? */
/* take stuff off of upper left corner of news network */
/* adder implemented as manchester carry chain */

#include <stdio.h>

/* The number of processors per chip */
/* is fixed into a 32 x 32 array which */
/* is easily handled by long 32 bit integers */
/* define these if not already predefined */

#ifndef XCHIPS
#define XCHIPS 16
#endif
#ifndef YCHIPS
#define YCHIPS 16
#endif

/* define verbosity level for simulations if not already defined*/
#ifndef VERBOSELEVEL
#define VERBOSELEVEL 1
#endif

/* define addresses in register files */
/* make sure we can define these in either side!! */
/* left register file is 0-31 and right is 32-63 */

/* these have been changed!! */
/* #define IDLE_ADDR 0
  #define SEL_ADDR0 1
  #define DPM_ADDR 2
  #define IOP_ADDR 3
  #define BRK_ADDR 32
  #define SEL_ADDR1 33
  #define NEWS_ADDR 34 */
```

```
/* old values */
#define NEWS_ADDR 0
#define SEL_ADDR0 1
#define SEL_ADDR1 2
#define DPM_ADDR 32
#define BRK_ADDR 33
#define IOP_ADDR 34
#define IDLE_ADDR 35

/* define number of iterations per cycle */
/* for NEWS network */
/* ITERATIONS must be an even number due to swapping !! */
#define ITERATIONS 16

/* set variables for i960 hardware */
#define DRAM_START (long int *)0xE0000000
#define ALU_LDRIVE 0x0C000000
#define ALU_LNODE  0x0C000004
#define ALU_RDRIVE 0x0C000008
#define ALU_RNODE  0x0C00000C
#define ALU_SEL1   0x0C000010
#define ALU_SEL0   0x0C000014
#define ALU_MYNODE 0x0C000018
#define ALU_BREAK  0x0C00001C
#define ALU_RESULT 0x0C000000

/* define this for FPGA support */
/* #define FPGAHARDWARE 1 */

/* define the bit masks for the control register bits */
#define SLOCUS_PADPWR_MASK 0x01
#define SLOCUS_GLOBORPWR_MASK 0x02
#define SLOCUS_DRAMPWR_MASK 0x04
#define SLOCUS_NDIR_MASK 0x08

/* define the bit masks for the dram path bits */
#define SLOCUS_WORDSEL_MASK 0x0200
#define SLOCUS_DPCLK_MASK 0x02
#define SLOCUS_ADDRINC_MASK 0x01

/* set up all global variables for state */
/* general array of 8 megs for everything */
/* long int genreg[XCHIPS*YCHIPS*(32*64+32*2+4+4+32+2)]; */
/* 8 megs of memory and comio flags */
#ifdef sparc
```

```
long int slocus_general[0x200000];
#else
long int *slocus_general=DRAM_START;
#endif

long int slocus_verbose=VERBOSELEVEL, slocus_breaksig=0;

/* left and right register files */
long int *slocus_regfile;

/* NEWS network and chip boundaries */
/* news network is twice as large to handle swapping */
/* newsedgein2 covers the two cycle latency */
long int *slocus_news;
long int *slocus_newsedgeout;
long int *slocus_newsedgein;
long int *slocus_newsedgein2;

/* DRAM plane network and edge registers */
long int *slocus_drampedgeinlo;
long int *slocus_drampedgeinhi;
long int *slocus_drampedgeoutlo;
long int *slocus_drampedgeouthi;

/* IO plane network and edge registers */
long int *slocus_iopedgein;
long int *slocus_iopedgeout;

/* user memory for swapping state and general storage (DPM,etc) */
long int *slocus_usrmem;

/* global instruction word and control bits */
long int slocus_instr0, slocus_instr1, slocus_control, slocus_membits;

/* special instruction flag and dram address register */
long int slocus_special, slocus_dramareg;

/* results from routines if called externally */
long int slocus_result;

/* declare routines to prevent errors */
char *clear_space(char *);
char *next_space(char *);
long int alu_comp();
long int dpm_comp();
long int news_comp();
long int or_comp();
long int slocus(char *);
long int process_special();

/* Main program: */
/* You may include slocus.c in your file but be sure to define */
/* the SLOCUSSLAVE variable and set VERBOSELEVEL=0 */
#ifndef SLOCUSSLAVE
main()
{
  char input[60];
  if(slocus_verbose==1)
    puts("Welcome to the Slocus Abacus simulator.");
  reset_slocus();
  for(;;)
    {
      if(slocus_verbose==1) puts("Slocus Command (h for help):");
      gets(input);
      slocus(input);
    }
}
#endif

/* slocus dispatcher takes a pointer to a command string and */
/* processes it, results are returned in the variable slocus_result */
/* which is a long integer */
long int slocus(char *cmdptr)
{
    char *parse;
    parse=cmdptr;
    parse=clear_space(parse);
/* dispatch on first character */
    if(*parse=='h') print_help();
    else if(*parse=='q') exit(1);
    else if(*parse=='s') step_zero();
    else if(*parse=='i') run_cmd(parse);
    else if(*parse=='r') read_routine(parse);
    else if(*parse=='w') write_routine(parse);
    else if(*parse=='c') reset_slocus();
    else if(*parse=='l') single_step();
    else if(*parse=='m') move_mem(parse);
    else if(*parse=='p') debug_string(parse);
    else puts("Invalid Command - h for help");
    return slocus_result;
  }


/* string parsing routines */
/* clear space takes a character string and returns the pointer */
/* to the first non-space character or NULL if none exists */
char *clear_space(char *parse)
{
  char *ptr=parse;
  for(;;) {
    if (*ptr == 0) return ptr; /* end of string */
    if (*ptr != 32) return ptr; /* found non space */
    ptr++;
  }
}

/* finds next space in character string to get next token */
/* returns NULL if no next space */
char *next_space(char *parse)
{
  char *ptr=parse;
  for(;;){
    if (*ptr == 0) return ptr;  /* end of string */
    if (*ptr == 32) return ptr; /* found space */
    ptr++;
  }
}

/* this routine returns a long integer with the value contained */
/* in the the string scanptr */
long int scanhex(char *scanptr)
{
  long int i,now;
  char *ptr;
  i=0;
  ptr=scanptr;
  for(;;){
    now=(long)*(ptr++);
    if(now>96) now -=32;
    if((now>64) && (now<71)) now -=55;
    else if((now>47) && (now<58)) now -=48;
    else break;
    i=(i<<4)|now;
  }
  return i;
```

```
}

/* this routine converts a long word into hex text scanptr*/
void makehex(long num,char *scanptr)
{
  char *ptr;
  long int k,i,now;
  k=num;
  ptr=scanptr;
  for(i=7;i>=0;i--) {
    now=(long)(k & 15);
    k >>=4;
    if(now>9) *(ptr+i)=55+now;
    else *(ptr+i)=48+now;
  }
/* terminate string */
  *(ptr+8)=0;
}

/* this routine put the Abacus simulated machine in a known */
/* state by setting all register files to zero, clearing the */
/* NEWS network and the DPM netork */
reset_slocus()
{
  long int *nein,*nein2,*neout,*nn,*reg,cnt,i,j;
  long int *dpinhi,*dpinlo,*dpouthi,*dpoutlo,*iopin,*iopout;

/* initialize all pointers to parts of the general array */

  slocus_regfile=slocus_general;

  slocus_news=slocus_regfile+(XCHIPS*YCHIPS*32*64);
  slocus_newsedgeout=slocus_news+(XCHIPS*YCHIPS*32);
  slocus_newsedgein=slocus_newsedgeout+(XCHIPS*YCHIPS*4);
  slocus_newsedgein2=slocus_newsedgein+(XCHIPS*YCHIPS*4);
  slocus_drampedgeinhi=slocus_newsedgein2+(XCHIPS*YCHIPS*4);
  slocus_drampedgeinlo=slocus_drampedgeinhi+(XCHIPS*YCHIPS);
  slocus_drampedgeouthi=slocus_drampedgeinlo+(XCHIPS*YCHIPS);
  slocus_drampedgeoutlo=slocus_drampedgeouthi+(XCHIPS*YCHIPS);
  slocus_iopedgein=slocus_drampedgeoutlo+(XCHIPS*YCHIPS);
  slocus_iopedgeout=slocus_iopedgein+(XCHIPS*YCHIPS);
  slocus_usrmem=slocus_iopedgeout+(XCHIPS*YCHIPS);

/* set instruction to NOP (write 63 and 31 to themselves */
  slocus_instr1=0x3fff;slocus_instr0=0x87ffff0f;
/* clear result and control register */
  slocus_result=0; slocus_control=0;

/* set everything to zero */
  if(slocus_verbose==1) puts("Initializing...");
  nein=slocus_newsedgein; neout=slocus_newsedgeout;
  nein2=slocus_newsedgein2;
  nn=slocus_news;  reg=slocus_regfile;
  dpinhi=slocus_drampedgeinhi; dpouthi=slocus_drampedgeouthi;
  dpinlo=slocus_drampedgeinlo; dpoutlo=slocus_drampedgeoutlo;
  iopin=slocus_iopedgein; iopout=slocus_iopedgeout;

/* cycle through chips */
  cnt=XCHIPS*YCHIPS;
  while(cnt-->0){
    *(nein++)=0;
    *(nein++)=0;
    *(nein++)=0;
    *(nein++)=0;
    *(nein2++)=0;
    *(nein2++)=0;
    *(nein2++)=0;
    *(nein2++)=0;
    *(neout++)=0;
    *(neout++)=0;
    *(neout++)=0;
    *(neout++)=0;
    *(dpinhi++)=0;
    *(dpouthi++)=0;
    *(dpinlo++)=0;
    *(dpoutlo++)=0;
    *(iopin++)=0;
    *(iopout++)=0;
    for(i=0;i<32;i++){
      *(nn++)=0;
      for(j=0;j<32;j++){
/* clear left and right register files */
        *(reg++)=0;
        *(reg++)=0;
      }
    }
  }
  if (slocus_verbose==1) puts("Initialized.");
}
```

```
/* print a help listing */
print_help()
{
  puts("Commands are all lowercase with optional arguments.");
  puts("h - prints this help listing");
  puts("c - clear machine (reset)");
  puts("s - single step the machine with a zero instruction");
  puts("i <hi-instr> <lo-instr> run a given hex instruction");
  puts("l - repeat last instruction command ");
  puts("p - print a string to the screen (for debugging) ");
  puts("q - exits back to the operating system");
  puts("rs <sect> <addr> - read a single 32-bit word");
  puts("rb <sect> <addr> <length> - read multiple words");
  puts("ws <sect> <addr> <value> - write a single value");
  puts("wb <sect> <addr> <length> - write multiple values");
  puts("m <sect> <addr> <sect> <addr> <length> - move memory");
  puts("all numerical values are in hexadecimal");
  puts("<sect> is one of the following:");
  puts("r - registers             u - user memory");
  puts("n - NEWS network          g - machine state");
  puts("i - DPM in register       o - DPM out register");
  puts("e - NEWS edge in register  t - NEWS edge out register");
  puts(" ");
}

/* debugging routine to force strings to print to stdout */
debug_string(char *parse)
{
  char *ptr=parse;
  long int vflag;
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  vflag=slocus_verbose;slocus_verbose=1;
  puts(ptr);
  slocus_verbose=vflag;
}

/* read the contents of a particular array in relative order */
read_routine(char *parse)
{
  long int cnt,*start_addr,offset,max,i,j;
  long int block,flagb,tmp,flagj,skip,vflag;
  char outstr[80],*ptr=parse;
  skip=1;
  ptr++;
  if(*ptr=='s') block=0;
  else if(*ptr=='b') block=1;
  else {
    puts("Incorrect read instruction - type h");
    return;
  }
  if(*(ptr+1)=='s') { flagj=1; ptr++; } else flagj=0;
  if(*(ptr+1)=='1') flagb=1; else flagb=0;
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0) {
    puts("No arguments were specified");
    return;
  }
/* print status of global OR */
  if(*ptr=='g') {
    i=or_comp();
    vflag=slocus_verbose; slocus_verbose=1;
    if(i==1) {
#ifndef SLOCUSSLAVE
      puts("1");
#endif
      slocus_result=1; }
    else {
#ifndef SLOCUSSLAVE
      puts("0");
#endif
      slocus_result=0; }
    slocus_verbose=vflag;
    return;
  }
  else if(*ptr=='n') {start_addr=slocus_news;
                      max=32*XCHIPS*YCHIPS; skip=32; }
  else if(*ptr=='r') {start_addr=slocus_regfile;
                      max=32*64*XCHIPS*YCHIPS; skip=64; }
  else if(*ptr=='i') {start_addr=slocus_drampedgeinlo;
                      max=XCHIPS*YCHIPS; }
  else if(*ptr=='o') {start_addr=slocus_drampedgeoutlo;
                      max=XCHIPS*YCHIPS; }
  else if(*ptr=='e') {start_addr=slocus_newsedgein;
                      max=4*XCHIPS*YCHIPS; skip=4;}
  else if(*ptr=='t') {start_addr=slocus_newsedgeout;
                      max=4*XCHIPS*YCHIPS; skip=4;}
  else if(*ptr=='u') {start_addr=slocus_usrmem; max=0x200000-
```

```
                              XCHIPS*YCHIPS*(32*64+32*2+4+4+32+1+1);}
  else {
    puts("Incorrect section type specified - see h");
    return;
  }
/* if jumping flag not set, set skip back to one */
  if(flagj==0) skip=1;
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0) {
    puts("no address specified");
    return;
  }
  offset=scanhex(ptr);
  if((offset<0) || (offset>=max)) {
    puts("Address must be hexadecimal and in range");
    return;
  }
  if(block==0) {
    if(flagb==0) makehex(*(start_addr+offset),outstr);
    else {
      tmp=*(start_addr+offset);
      for(j=0;j<32;j++){
        if(((tmp << j) & 0x80000000) == 0x80000000) *(outstr+j)='X';
        else *(outstr+j)='.';
      }
      *(outstr+32)=0;
    }
    slocus_result=*(start_addr+offset);
/* always print */
#ifndef SLOCUSSLAVE
    vflag=slocus_verbose; slocus_verbose=1;
    puts(outstr);
    slocus_verbose=vflag;
#endif
    return;
  }
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0) {
    puts("No length specified");
    return; }
  cnt=scanhex(ptr);
  if((cnt<1) || ((cnt-1)*skip+offset>max-1)){
    puts("Offset plus length of block are out of range or not hex");
    return;}
  for(i=offset;i<offset+cnt*skip;i+=skip){
    if(flagb==0) makehex(*(start_addr+i),outstr);
    else {
      for(j=0;j<32;j++){
        if(((*(start_addr+i)<< j) & 0x80000000) == 0x80000000)
          *(outstr+j)='X'; else *(outstr+j)='.';
      }
      *(outstr+32)=0; /* terminate string with slash zero */
    }
    slocus_result=*(start_addr+i);
#ifndef SLOCUSSLAVE
    vflag=slocus_verbose; slocus_verbose=1;
    puts(outstr);
    slocus_verbose=vflag;
#endif
  }
}

/* write values to a particular array in relative order */
write_routine(char *parse)
{
  long int block,cnt,*start_addr,offset,max,i,flagj,skip;
  char input[20],*ptr=parse;
  skip=1;
  ptr++;
  if(*ptr=='s') block=0;
  else if(*ptr=='b') block=1;
  else {
    puts("Incorrect write instruction - type h");
    return;
  }
  if(*(ptr+1)=='s') { flagj=1; ptr++; } else flagj=0;
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0) {
    puts("No arguments were specified");
    return;
  }
  if(*ptr=='n') {start_addr=slocus_news;
                 max=32*XCHIPS*YCHIPS; skip=32;}
  else if(*ptr=='r') {start_addr=slocus_regfile;
                      max=32*64*XCHIPS*YCHIPS; skip=64;}
  else if(*ptr=='i') {start_addr=slocus_drampedgeinlo;
                      max=XCHIPS*YCHIPS;}
```

```
else if(*ptr=='o') {start_addr=slocus_drampedgeoutlo;
                    max=XCHIPS*YCHIPS;}
else if(*ptr=='e') {start_addr=slocus_newsedgein;
                    max=4*XCHIPS*YCHIPS; skip=4;}
else if(*ptr=='t') {start_addr=slocus_newsedgeout;
                    max=4*XCHIPS*YCHIPS; skip=4;}
else if(*ptr=='u') {start_addr=slocus_usrmem;
                    max=0x200000-
                      XCHIPS*YCHIPS*(32*64+32*2+4+4+32+1+1);}
else {
  puts("Incorrect section type specified - see h");
  return;
}
if(flagj==0) skip=1;
ptr=next_space(ptr);
ptr=clear_space(ptr);
if(*ptr==0) {
  puts("no address specified");
  return;
}
offset=scanhex(ptr);
if((offset<0) || (offset>=max)) {
  puts("Address must be hexadecimal and in range");
  return;
}
if(block==0) {
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0){
    puts("no data specified");
    return; }
  *(start_addr+offset)=scanhex(ptr);
  return;
}
ptr=next_space(ptr);
ptr=clear_space(ptr);
if(*ptr==0) {
  puts("No length specified");
  return; }
cnt=scanhex(ptr);
if((cnt<1) || ((cnt-1)*skip+offset>max-1)){
  puts("Offset plus length of block are out of range or not hex");
  return;}
for(i=offset;i<offset+cnt*skip;i+=skip) {
  gets(input);
```

```
    *(start_addr+i)=scanhex(input);
  }
}

/* move a block of memory from one place to another */
move_mem(char *parse)
{
  long int cnt,*src_addr,*dest_addr,offset1,offset2,max1,max2,i;
  long int flagj,skip1,skip2;
  char *ptr=parse;
  skip1=1; skip2=1;
  if(*(ptr+1)=='s') { flagj=1; ptr++; } else flagj=0;
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0) {
    puts("No arguments were specified");
    return;
  }
  if(*ptr=='n') {src_addr=slocus_news;
                 max1=32*XCHIPS*YCHIPS; skip1=32;}
  else if(*ptr=='r') {src_addr=slocus_regfile;
                      max1=32*64*XCHIPS*YCHIPS; skip1=64;}
  else if(*ptr=='i') {src_addr=slocus_drampedgeinlo;
                      max1=XCHIPS*YCHIPS;}
  else if(*ptr=='o') {src_addr=slocus_drampedgeoutlo;
                      max1=XCHIPS*YCHIPS;}
  else if(*ptr=='e') {src_addr=slocus_newsedgein;
                      max1=4*XCHIPS*YCHIPS; skip1=4;}
  else if(*ptr=='t') {src_addr=slocus_newsedgeout;
                      max1=4*XCHIPS*YCHIPS; skip1=4;}
  else if(*ptr=='u') {src_addr=slocus_usrmem;
                      max1=0x200000-
                        XCHIPS*YCHIPS*(32*64+32*2+4+4+32+1+1);}
  else {
    puts("Section invalid for source - see h");
    return;
  }
 /* don't skip blocks if flag not set */
  if(flagj==0) skip1=1;
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0) {
    puts("no source address specified");
    return;
  }
```

```
offset1=scanhex(ptr);
if((offset1<0) || (offset1>=max1)) {
  puts("Source address out of range or not hex");
  return;
}
ptr=next_space(ptr);
ptr=clear_space(ptr);
if(*ptr==0) {
  puts("No destination specified");
  return;
}
if(*ptr=='n') {dest_addr=slocus_news;
               max2=32*XCHIPS*YCHIPS; skip2=32;}
else if(*ptr=='r') {dest_addr=slocus_regfile;
                    max2=32*64*XCHIPS*YCHIPS; skip2=64;}
else if(*ptr=='i') {dest_addr=slocus_drampedgeinlo;
                    max2=XCHIPS*YCHIPS;}
else if(*ptr=='o') {dest_addr=slocus_drampedgeoutlo;
                    max2=XCHIPS*YCHIPS;}
else if(*ptr=='e') {dest_addr=slocus_newsedgein;
                    max2=4*XCHIPS*YCHIPS; skip2=4;}
else if(*ptr=='t') {dest_addr=slocus_newsedgeout;
                    max2=4*XCHIPS*YCHIPS; skip2=4;}
else if(*ptr=='u') {dest_addr=slocus_usrmem;
                    max2=0x200000-
                      XCHIPS*YCHIPS*(32*64+32*2+4+4+32+1+1);}
else {
  puts("Section invalid for destination - see h");
  return;
}
if(flagj==0) skip2=1;
ptr=next_space(ptr);
ptr=clear_space(ptr);
if(*ptr==0) {
  puts("no destination address specified");
  return;
}
offset2=scanhex(ptr);
if((offset2<0) || (offset2>=max2)) {
  puts("Desination address out of range or not hex");
  return;
}
ptr=next_space(ptr);
ptr=clear_space(ptr);
if(*ptr==0) {
    puts("No block length specified");
    return; }
  cnt=scanhex(ptr);
  if((cnt<1) ||
     ((cnt-1)*skip1+offset1>max1-1) ||
     ((cnt-1)*skip2+offset2>max2-1)){
    puts("Block larger than the range of source or destination.");
    return;}
  if(slocus_verbose==1) puts("Copying Block");
  for(i=offset1;i<offset1+cnt*skip1;i+=skip1) {
    *(dest_addr+offset2)=*(src_addr+i);
    offset2+=skip2; }
}


/* run a user specified command string */
run_cmd(char *parse)
{
  char *ptr=parse;
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0) {
    puts("No instruction specified");
    return;
  }
  slocus_instr1=scanhex(ptr);
  ptr=next_space(ptr);
  ptr=clear_space(ptr);
  if(*ptr==0) {
    puts("Need two instruction arguments");
    return;
  }
  slocus_instr0=scanhex(ptr);
  single_step();
}

step_zero()
{
/* set instruction to NOP (write 31 and 63 back on itself) */
  slocus_instr0=0x87ffff0f; slocus_instr1=0x3fff;
  single_step();
}

single_step()
{
```

```
/*   get_instr(); */
  if(alu_comp()==1) return;
  if(dpm_comp()==1) return;
  if(news_comp()==1) return;
  or_comp();
}

/* This function takes two words instr[ipc][0] and */
/* instr[ipc][1] and parses their bits into appropriate */
/* addresses for use by the alu */
/* the bits are as follows: */
/* 0-4 right write address */
/* 5-9 right b read address */
/* 10-14 right a read address */
/* 15-22 right 8 bit operand */
/* 23-27 left write address */
/* 28-32 left b read address */
/* 33-37 left a read address */
/* 38-45 left 8 bit operand */
/* 46    DPM clock bit (clock if one) */
/* 47    DPM selection bit (1=load or 0=shift) */
/* 48    direction of NEWS shifting (0=SE) (1=NW)*/
/*       accross chip boundaries */

/* this alu_comp routine takes the ra,rb... variables */
/* looks up their values, computes a result based on rop */
/* and lop and puts them back into rw and lw addresses */
/* special cases are handled for idle bits and DPM/NEWS */
/* network operations */
long int alu_comp()
{
  long int la,lb,lw,lop,ra,rb,rw,rop,cmd1,cmd2;
  long int lad,lbd,rad,rbd,lwd,rwd,idlbits,ltmp,rtmp;
  long int cnt,lselfile,rselfile,sel0,sel1,sel2,i;
  long int *nein2,*nn,*reg;
/*  parse instruction word */
  cmd1=slocus_instr0; cmd2=slocus_instr1;
/* parse instruction word */
  lop=(cmd1 & 0x000000FF);  cmd1 >>=8;
  la=(cmd1 & 0x0000001F);  cmd1 >>=5;
  lb=(cmd1 & 0x0000001F);  cmd1 >>=5;
  lw=(cmd1 & 0x0000001F); cmd1 >>=5;
/* add 32 to right register file instructions */
  rop=(cmd1 & 0x000000FF);  cmd1 >>=8;
  ra=(cmd1 & 0x00000001)+
    ((cmd2 & 0x0000000F) << 1)+32; cmd2 >>=4;
  rb=(cmd2 & 0x0000001F)+32;  cmd2 >>=5;
  rw=(cmd2 & 0x0000001F)+32; cmd2 >>=5;

/* get timing information bits */
  slocus_membits=(cmd2 & 0x003FF); cmd2 >>=10;

/* check for special instruction */
  if(lop==0 && la!=0) slocus_special=1;
  else slocus_special=0;
  if(slocus_special == 1) {
    process_special();
    if(slocus_verbose==1) puts("special alu done");
    return; }
  reg=slocus_regfile;
  nein2=slocus_newsedgein2;
  nn=slocus_news;
/* cycle through all chips */
  cnt=XCHIPS*YCHIPS;
  while(cnt-->0){
    if(slocus_breaksig==1){
      slocus_breaksig=0;
      return 1;
    }
/* debug */
/*    printf("la %d lb %d lw %d lop %x ra %d rb %d rw %d rop %x \n",
           la,lb,lw,lop,ra,rb,rw,rop);  */
/* step through the rows */
    for(i=0;i<32;i++){
/* special case for NEWS address access */
      if((la==NEWS_ADDR) ||
         (lb==NEWS_ADDR) ||
         (ra==NEWS_ADDR) ||
         (rb==NEWS_ADDR))
        {
        lselfile=*(reg+SEL_ADDR1);
        rselfile=*(reg+SEL_ADDR0);
        ltmp=
/* seek left */
          (lselfile &
           ~rselfile &
/* make sure to get rid of top bit ! */
           ((((*nn) >> 1) & 0x7FFFFFFF) |
           ((*(nein2+3) << i) & 0x80000000))) |
/* seek right */
```

```
              (~lselfile &
               rselfile &
               (((*nn) << 1) |
               ((*(nein2+1) >>(31-i)) & 1)));
/* seek upwards */
        if(i==0)  ltmp |= ~lselfile & ~rselfile & *nein2;
        else      ltmp |= ~lselfile & ~rselfile & *(nn-1);
/* seek downwards */
        if(i==31) ltmp |= lselfile & rselfile & *(nein2+2);
        else      ltmp |= lselfile & rselfile & *(nn+1);
/*      printf("NEWS in value is %x \n",ltmp); */
      }

      if(la==NEWS_ADDR) lad=ltmp;
      else lad=*(reg+la);

      if(lb==NEWS_ADDR) lbd=ltmp;
      else lbd=*(reg+lb);

      if(ra==NEWS_ADDR) rad=ltmp;
      else rad=*(reg+ra);

      if(rb==NEWS_ADDR) rbd=ltmp;
      else rbd=*(reg+rb);

/* try implementing combinational logic here !! */
      sel2=lad;
      sel1=lbd;
      sel0=rbd;
/* compute value for left register file */
      lwd =
        (~sel2 & ~sel1 & ~sel0 &
         (((lop >> 7) & 1) ? 0xFFFFFFFF:0)) |
          (~sel2 & ~sel1 & sel0 &
           (((lop >> 6) & 1)? 0xFFFFFFFF:0)) |
             (~sel2 & sel1 & ~sel0 &
              (((lop >> 5) & 1)? 0xFFFFFFFF:0)) |
                (~sel2 & sel1 &sel0 &
                 (((lop >> 4) & 1)? 0xFFFFFFFF:0)) |
                   (sel2 & ~sel1 & ~sel0 &
                    (((lop >> 3) & 1)? 0xFFFFFFFF:0)) |
                      (sel2 & ~sel1 & sel0 &
                       (((lop >> 2) & 1)? 0xFFFFFFFF:0)) |
                         (sel2 & sel1 & ~sel0 &
                          (((lop >> 1) & 1)? 0xFFFFFFFF:0)) |
                              (sel2 & sel1 & sel0 &
                               ((lop & 1) ? 0xFFFFFFFF:0));
/* compute value for right register file */
      sel2=rad;
      sel1=rbd;
      sel0=lbd;
      rwd =
        (~sel2 & ~sel1 & ~sel0 &
         (((rop >> 7) & 1) ? 0xFFFFFFFF:0)) |
          (~sel2 & ~sel1 & sel0 &
           (((rop >> 6) & 1)? 0xFFFFFFFF:0)) |
             (~sel2 & sel1 & ~sel0 &
              (((rop >> 5) & 1)? 0xFFFFFFFF:0)) |
                (~sel2 & sel1 &sel0 &
                 (((rop >> 4) & 1)? 0xFFFFFFFF:0)) |
                   (sel2 & ~sel1 & ~sel0 &
                    (((rop >> 3) & 1)? 0xFFFFFFFF:0)) |
                      (sel2 & ~sel1 & sel0 &
                       (((rop >> 2) & 1)? 0xFFFFFFFF:0)) |
                         (sel2 & sel1 & ~sel0 &
                          (((rop >> 1) & 1)? 0xFFFFFFFF:0)) |
                            (sel2 & sel1 & sel0 &
                             ((rop & 1) ? 0xFFFFFFFF:0));
/* assign results back into register file */
/* if idle bit is set then we won't write to any addresses */
      idlbits=*(reg+IDLE_ADDR);
/* get old values */
      ltmp=*(reg+lw);
      rtmp=*(reg+rw);
/* write left registerfile results */
/* allow writing to idle_addr bit all the time */
      if(lw == IDLE_ADDR) *(reg+IDLE_ADDR)=lwd;
      else *(reg+lw)=(lwd & ~idlbits) | (ltmp & idlbits);
/* write to right registerfile results */
/* allow writing to idle_addr bit all the time */
      if(rw == IDLE_ADDR) *(reg+IDLE_ADDR)=rwd;
      else *(reg+rw)=(rwd & ~idlbits) | (rtmp & idlbits);
/* debug */
/*      printf("lad %x lbd %x rad %x rbd %x lwd %x rwd %x\n",
             lad, lbd, rad, rbd, lwd, rwd); */
/* increment pointers per row */
      reg += 64; nn++;
    }
/* increment newsedge pointer per chip */
    nein2+=4;
```

```
  }
  if(slocus_verbose==1) puts("alu done");
  return 0;
}

long int process_special()
{
  long int special_op,c,m,i,k,cnt,*nein;
  long int itmp,outtmp,*ioptr,*ipin,*ipout;
  special_op=(slocus_instr0 >> 8) & 0x07;

/* load control registers */
  if(special_op==1) {
/* get eight control bits - just in case */
    c=(slocus_instr0 >> 13) & 0xFF;
/* get mask control */
    m=(slocus_instr0 >> 11) & 0x03;
    if(m==0) slocus_control=c; /* copy */
    else if(m==1) slocus_control &=c; /* set bits */
    else if(m==2) slocus_control &= ~c; /* clear bits */
    else if(m==3) slocus_control ^= ~c; /* toggle bits */
  }

/* set DRAM address register */
  else if(special_op==2) {
    slocus_dramareg=(slocus_instr0 >> 11) & 0x0FFFFF;
  }

/* load immediate constant */
  else if(special_op==4) {
    cnt=XCHIPS*YCHIPS;
    nein=slocus_newsedgein+1; /* select east edge */
    c = ((slocus_instr0 >> 11) & 0x01FFFFF) +
      ((slocus_instr1 & 0x07FF) << 21) ;
    while(cnt-->0) {
      *nein=c;
      nein += 4; }
  }

/* load and store IO port (option to shift) */
  else if((special_op==5) || (special_op==6)){
    if((slocus_instr1 >> 12) & 0x01) {
      cnt=XCHIPS*YCHIPS;
      ioptr=slocus_regfile+IOP_ADDR;
      ipin=slocus_iopedgein;
      ipout=slocus_iopedgeout;
      while(cnt-->0) {
        itmp=*ipin;
/* go down rows and shift all bits left*/
        for(k=0;k<32;k++) {
/* do next bit of outward register */
          outtmp = (outtmp << 1) + ((*(ioptr+k*64) >> 31) & 1);
/* shift bit in from input register */
/* make sure to and off the top bit because of arithmetic shift */
          *(ioptr+k*64)=(*(ioptr+k*64) << 1) | ((itmp >> 31 ) & 0x01);
/* shift to next bit in input register */
          itmp <<=1; }
/* store final result in output register */
        *ipout=outtmp;
        ipin++; ipout++; ioptr += 32*64;
      }
    }
  }

/* bad special instruction */
  else if(slocus_verbose==1) puts("Bad special instruction.");
}


/* this routine parses the slocus_membits variable and */
/* does things accordingly */
long int dpm_comp()
{
  long int cnt,*dpptr,*reg,*dpin,*dpout,k,outtmp,intmp;
  long int dpclk,dpsel;

/* increment dram address register if needed */
  if(slocus_membits & SLOCUS_ADDRINC_MASK) slocus_dramareg++;

/* parse dram plane clock */
  dpclk=slocus_membits & SLOCUS_DPCLK_MASK;
  if(dpclk==0) return 0;   /* return if nothing */

  dpptr=slocus_regfile+DPM_ADDR; /* pointer into register file */
  if(slocus_membits & SLOCUS_WORDSEL_MASK) {
    dpin=slocus_drampedgeinhi;
    dpout=slocus_drampedgeouthi; }
  else {
    dpin=slocus_drampedgeinlo;
    dpout=slocus_drampedgeoutlo; }
```

```
/* cycle through all chips */
  cnt=XCHIPS*YCHIPS;
  while(cnt-->0){
    intmp=*dpin;
/* go down rows and shift all bits left (dpsel=0)*/
    for(k=0;k<32;k++) {
/* do next bit of outward register */
      outtmp = (outtmp << 1) + ((*(dpptr+k*64) >> 31) & 1);
/* shift bit in from input register */
/* make sure to and off the top bit because of arithmetic shift */
      *(dpptr+k*64)=(*(dpptr+k*64) << 1) | ((intmp >> 31 ) & 0x01);
/* shift to next bit in input register */
      intmp <<=1; }
/* store final result in output register */
    *dpout=outtmp;
    dpin++; dpout++; dpptr += 32*64;
  }
  if(slocus_verbose==1)puts("dpm done");
  return 0;
}

/* we don't do news precharge before!  we write all values */
/* of the newsregisters out after the alu_comp cycle if someone */
/* writes to a location */
/* we must make sure not to do this during special instructions */
news_precharge()
{
  long int cnt,*nn,*reg;
  if(slocus_special==1) return;
  cnt=XCHIPS*YCHIPS*32;
  nn=slocus_news;
  reg=(slocus_regfile+NEWS_ADDR);
  while(cnt-->0) {
    *(nn++)=*reg;
    reg +=64;
  }
}

/* subroutine to compute the news network after each cycle */
/* number of nodes travelled defined in ITERATIONS */
/* this number must be even, or the swapping doesn't work */
long int news_comp()
{
#ifdef FPGAHARDWARE
```

```
  volatile long int *alu_ldrive=(long int *)ALU_LDRIVE;
  volatile long int *alu_lnode =(long int *)ALU_LNODE;
  volatile long int *alu_rdrive=(long int *)ALU_RDRIVE;
  volatile long int *alu_rnode =(long int *)ALU_RNODE;
  volatile long int *alu_sel1  =(long int *)ALU_SEL1;
  volatile long int *alu_sel0  =(long int *)ALU_SEL0;
  volatile long int *alu_mynode=(long int *)ALU_MYNODE;
  volatile long int *alu_break =(long int *)ALU_BREAK;
  volatile long int *alu_result=(long int *)ALU_RESULT;
#endif
  long int i,k,mynode,swap,lw,rw;
  long int cnt,*lreg,*rreg,*breg,*nn;
  long int *pnews,*snews;
  long int brkfile,lselfile,rselfile;
/* these local arrays are for swapping the NEWS network in SRAM */
  long int swap1[32],swap2[32],lregtmp[32],rregtmp[32],brktmp[32];

/* if current instruction writes to NEWS register */
/* then copy NEWS registers to NEWS network (precharge) */
  lw=(slocus_instr0 >> 18) & 0x001F;
  rw=((slocus_instr1 >> 9) & 0x001F) + 32;
  if((lw==NEWS_ADDR) ||
     (rw==NEWS_ADDR)) news_precharge();

/* ITERATIONS must be an even number due to swapping !! */
  swap=0;
/* cycle through all the chips */
  cnt=XCHIPS*YCHIPS;
/* load address registers with proper offset */
  breg=slocus_regfile+BRK_ADDR;
  lreg=slocus_regfile+SEL_ADDR1;
  rreg=slocus_regfile+SEL_ADDR0;
  nn=slocus_news;
  while(cnt-->0){
    if(slocus_breaksig==1) {
      slocus_breaksig=0;
      return 1;
    }
/* get values from large array into local small arrays */
    for(i=0;i<32;i++) {
      swap1[i]=*(nn+i);
      lregtmp[i]=*lreg; lreg+=64;
      brktmp[i]=*breg; breg+=64;
      rregtmp[i]=*rreg; rreg+=64;
      }
```

```
/* perform 16 iterations */
    for(i=0; i< ITERATIONS;i++) {
      if(swap==0){
        pnews=swap1;
        snews=swap2;
        swap=1;}
      else{
        snews=swap1;
        pnews=swap2;
        swap=0;}
/* if FPGAHARDWARE flag not set, do all logic internally */
#ifndef FPGAHARDWARE
/* msbit is at the top of the column */
/* first iteration is for row zero */
      brkfile=~(brktmp[0]);
      lselfile=lregtmp[0];
      rselfile=rregtmp[0];
      mynode=*pnews;
/* start with self */
      *(snews++) = mynode |
/* check for below node driving up on us */
      (~(lregtmp[1]) & ~(rregtmp[1]) & *(pnews+1) & ~(brktmp[1])) |
/* check for seeking downward */
      (lselfile & rselfile & brkfile & *(pnews+1)) |
/* check left node driving right on us */
/* AND off the top bit to correct arithmetic shift */
      (((~lselfile & rselfile & mynode & brkfile) >> 1)
        & 0x7FFFFFFF) |
/* check seeking left */
/* AND off the top bit to correct arithmetic shift */
      ((lselfile & ~rselfile & ((mynode >> 1) & 0x7FFFFFFF)
        & brkfile)) |
/* check right node driving left on us */
      ((lselfile & ~rselfile & mynode & brkfile) << 1) |
/* check right seeking */
      (~lselfile & rselfile & brkfile & (mynode << 1));
/* increment pointer */
      pnews++;
/* do rest of internal rows... 1 to 30 (not 0 or 31) */
      for(k=1;k<31;k++) {
        brkfile=~(brktmp[k]);
        lselfile=lregtmp[k];
        rselfile=rregtmp[k];
        mynode=*pnews;
/* start with self */
```

```
        *(snews++) = mynode |
/* check for above node driving down on us*/
        (lregtmp[k-1] & rregtmp[k-1] & *(pnews-1) & ~(brktmp[k-1])) |
/* check for seeking upward */
        (~lselfile & ~rselfile & brkfile & *(pnews-1)) |
/* check for below node driving up on us */
        (~(lregtmp[k+1]) & ~(rregtmp[k+1]) & *(pnews+1)
          & ~(brktmp[k+1])) |
/* check for seeking downward */
        (lselfile & rselfile & brkfile & *(pnews+1)) |
/* check left node driving right on us */
/* AND off the top bit to correct arithmetic shift */
        (((~lselfile & rselfile & mynode & brkfile) >> 1)
          & 0x7FFFFFFF) |
/* check seeking left */
/* AND off the top bit to correct arithmetic shift */
        ((lselfile & ~rselfile & ((mynode >> 1) & 0x7FFFFFFF)
          & brkfile)) |
/* check right node driving left on us */
        ((lselfile & ~rselfile & mynode & brkfile) << 1) |
/* check right seeking */
        (~lselfile & rselfile & brkfile & (mynode << 1));
/* increment pointer */
        pnews++;
      }
/* do last row */
      brkfile=~(brktmp[31]);
      lselfile=lregtmp[31];
      rselfile=rregtmp[31];
      mynode=*pnews;
/* start with self */
      *(snews++) = mynode |
/* check for above node driving down on us*/
      (lregtmp[30] & rregtmp[30] & *(pnews-1) & ~(brktmp[30])) |
/* check for seeking upward */
      (~lselfile & ~rselfile & brkfile & *(pnews-1)) |
/* check left node driving right on us */
/* AND off the top bit to correct for arithmetic shift */
      (((~lselfile & rselfile & mynode & brkfile) >> 1)
        & 0x7FFFFFFF) |
/* check seeking left */
/* AND to correct arithmetic shift */
      ((lselfile & ~rselfile & ((mynode >> 1) & 0x7FFFFFFF)
        & brkfile)) |
/* check right node driving left on us */
```

```
      ((lselfile & ~rselfile & mynode & brkfile) << 1) |
/* check right seeking */
      (~lselfile & rselfile & brkfile & (mynode << 1));
#else
/* do all computations using external FPGA - how elegant!! */
/* FPGA thinks rows are vertical - alter the sel bits slightly */
/* first row */
      *alu_ldrive=0;
      *alu_lnode=*(pnews+1);
      *alu_rdrive=(~(lregtmp[1]) &
                  ~(rregtmp[1]) &
                  *(pnews+1) &
                  ~(brktmp[1]));
      *alu_rnode=0;
      *alu_sel1=lregtmp[0];
      *alu_sel0=~rregtmp[0];
      *alu_mynode=*(pnews++);
      *alu_break=brktmp[0];
      *(snews++)=*alu_result;
/* middle rows */
      for(k=1;k<31;k++){
        *alu_ldrive=(lregtmp[k-1] &
                    rregtmp[k-1] &
                    *(pnews-1) &
                    ~(brktmp[k-1]));
        *alu_lnode=*(pnews+1);
        *alu_rdrive=(~(lregtmp[k+1]) &
                    ~(rregtmp[k+1]) &
                    *(pnews+1) &
                    ~(brktmp[k+1]));
        *alu_rnode=*(pnews-1);
        *alu_sel1=lregtmp[k];
        *alu_sel0=~rregtmp[k];
        *alu_mynode=*(pnews++);
        *alu_break=brktmp[k];
        *(snews++)=*alu_result;
      }
/* last row */
      *alu_ldrive=(lregtmp[30] &
                  rregtmp[30] &
                  *(pnews-1) &
                  ~(brktmp[30]));
      *alu_lnode=0;
      *alu_rdrive=0;
      *alu_rnode=*(pnews-1);
      *alu_sel1=lregtmp[31];
      *alu_sel0=~rregtmp[31];
      *alu_mynode=*pnews;
      *alu_break=brktmp[31];
      *snews=*alu_result;
#endif
    }
/* write array back into global array */
/* increment news pointer as well */
    for(i=0;i<32;i++) *(nn++)=swap1[i];
  }
  news_edge();
  if(slocus_verbose==1)puts("news done");
  return 0;
}

/* propagate edges of NEWS network accross chip */
/* boundaries and then accept new values off network */
news_edge()
{
  long int i,j,cnt,nrtmp,nltmp,*nn,*nein,*neout;
/* compute two latency registers */
/* cycle through all chips */
  cnt=XCHIPS*YCHIPS;
  nein=slocus_newsedgein;
  neout=slocus_newsedgein2;
/* cycle and do all four edges */
  while(cnt-->0){
    *(neout++)=*(nein++);
    *(neout++)=*(nein++);
    *(neout++)=*(nein++);
    *(neout++)=*(nein++);
  }
/* compute cycle after output */
  nein=slocus_newsedgein;
  neout=slocus_newsedgeout;
/* cycle through all the chips */
  for(j=0;j<YCHIPS;j++)
    for(i=0;i<XCHIPS;i++) {
/* inout register direction (0=up,1=right,2=down,3=left) */
/* ndir=1 is NW while ndir=0 is SE */
      if(slocus_control & SLOCUS_NDIR_MASK){
        *(nein++)=*neout;
        *(nein++)=*(neout+((i==(XCHIPS-1))?
                              (7-4*XCHIPS):
```

```
                              7));
          *(nein++)=*(neout+((j==(YCHIPS-1))?
                              (-4*XCHIPS*(YCHIPS-1)):
                              (4*XCHIPS)));
          *(nein++)=*(neout+3);
        }
        else {
          *(nein++)=*(neout+((j==0)?
                              (2+4*XCHIPS*(YCHIPS-1)):
                              (2-4*XCHIPS)));
          *(nein++)=*(neout+1);
          *(nein++)=*(neout+2);
          *(nein++)=*(neout+((i==0)?
                              (4*XCHIPS-3):
                              -3));
        }
        neout+=4;
      }
  /* get new values off of network edge */
  /* cycle through chips */
    cnt=XCHIPS*YCHIPS;
    neout=slocus_newsedgeout;
    nn=slocus_news;
    while(cnt-->0){
      if((slocus_control & SLOCUS_PADPWR_MASK)==0) {
        for(i=0;i<32;i++){
          nltmp<<=1;
          nltmp |= (*(nn+i) & 0x80000000)? 1:0;
          nrtmp<<=1;
          nrtmp |= *(nn+i) & 1;
        }
        *(neout++)=*nn;
        *(neout++)=nrtmp;
        *(neout++)=*(nn+31);
        *(neout++)=nltmp;
        nn+=32;
      }
      else { /* powerdown if one */
        *(neout++)=0;
        *(neout++)=0;
        *(neout++)=0;
        *(neout++)=0;
      }
    }
  }
```

```
/* compute global-OR function */
long int or_comp()
{
  long cnt,*nn,orflag;
  nn=slocus_news;
  orflag=0;
/* cycle through all chips */
  cnt=XCHIPS*YCHIPS;
/* check upper right bit */
  if((slocus_control & SLOCUS_GLOBORPWR_MASK)==0) {
    while(cnt-->0){
      if(*nn & 0x01){ orflag=1; break;}
      nn+=32;}
    if(slocus_verbose==1)
      if(orflag==1) puts("Global OR returned TRUE");
      else puts("Global OR returned FALSE");
  }
  else if(slocus_verbose==1) puts("Global OR disabled");
  slocus_result=((orflag << 7) | (slocus_control &0x7F));
  return orflag;
}
```

A.1.2 Sloconv Nanocode Converter

```
/*  the Sloconv Slocus convertor */
/*  By Tim Kutscha 12/1/93 */
/*  produces slocus script files from */
/*  assembler output files from the slocus assember */

#include <stdio.h>

int offset, interval,perchip;
char tmp;

main(int argc, char *argv[])
{
/* set up variables for counters, chip count and files */
  long int i,j,k,cnt;
  unsigned long int instr1,instr0;
  long int xc,yc,reghex,binary;
  FILE *fp1,*fp2,*fp3;
  char str[256],linp[65600],*ptr;
  int flags[77];
/* parse command line arguments */
  if (argc ==1) {
    printf("No arguments given\n");
    printf("Must specify assembler <filename> \n");
    exit(1); }
  if (argc !=2) {
    printf("Too many arguments specified\n");
    printf("Only supply assembler <filename>\n");
    exit(1); }
  strcpy(str,argv[1]);
  fp1=fopen(str,"r");
  if(fp1 ==NULL){
    printf("Can't open %s for input!\n",str);
    exit(1);}
  strcpy(str,argv[1]);
  strcat(str,".slocus");
  fp2 = fopen(str,"w");
  if(fp2 ==NULL){
    printf("Can't open %s for output!\n",str);
    fclose(fp1);
    exit(1); }
  fp3 = fopen("slocus.def","w");
  if(fp3 ==NULL){
    printf("Can't open slocus.def for output!\n");
    fclose(fp1);
    fclose(fp2);
    exit(1); }
/* get first line - PE array count */
  fgets(linp,25600,fp1);
  ptr=linp+1;
  if(linp[0]!='m') {
    printf("Error in first line (not mesh definition):\n");
    printf("%s\n",linp);
    exit(1); }
  sscanf(ptr,"%dx%d",&yc,&xc);
  if((xc % 32) !=0 || (yc %32) !=0) {
    printf("Input mesh is not a factor of 32: %s\n",linp);
    exit(1);}
  xc=xc/32; yc=yc/32;
  if((xc < 0) || (yc < 0) || (xc*yc > 800)) {
    printf("Mesh parameters are negative or out of range: %s\n",linp);
    exit(1);}
  puts("");
  puts("Welcome to SLOCONV, the Slocus assembler converter.");
  printf("Using an array of %d by %d chips.\n",xc,yc);
/* set up line counter and start parsing lines in the file */
  i=0;
  for(;;){
/* fgets returns NULL at end of file */
    if(fgets(linp,65600,fp1)==NULL) break;
    i++;  /* increment line count */
    ptr=linp+1;
/* execute command */
    if(linp[0]=='o'){
      instr1=0; instr0=0;
/* parse high 24 bits */
      for(j=0;j<24;j++){
        instr1 <<=1;
        if(*ptr=='1') instr1++;
        else if (*ptr!='0'){
          printf("Instruction error in line %d. Continuing...\n",i);
          break;
        }
        ptr++;
      }
/* parse low 32 bits */
      for(j=0;j<32;j++){
        instr0 <<=1;
        if(*ptr=='1') instr0++;
        else if(*ptr!='0'){
          printf("Instruction error in line %d. Continuing...\n",i);
          break;}
        ptr++;
      }
/* output to file */
      fprintf(fp2,"i %08x %08x\n",instr1,instr0);
      fprintf(fp2,"p Executing Instruction %08x %08x\n",
              instr1,instr0);
/* ADD stuff for printing out current state of machine !!!*/
/* numbers are as follows: */
/* 0-63 register files (left first, top to bottom) */
/* 64 top news in port */
/* 65 right news in port */
/* 66 bottom news in port */
/* 67 left news in port */
/* 68 top news out port */
/* 69 right news out port */
/* 70 bottom news out port */
/* 71 left news out port */
/* 72 DPM in port */
/* 73 DPM out port */
/* 74 Global or status (read only) */
/* 75 News network nodes */
/* 76 Data-plane network nodes */

      for(j=0;j<77;j++)
        if(flags[j]) {
          if (j==76) {
            fputs("p Global OR flag",fp2);
            fputs("rs g",fp2); }
          else {
            get_param(j);
            fprintf(fp2,"p Register %d status:\n",j);
            for(k=offset;k<xc*yc*interval*perchip;k+=interval)
              fprintf(fp2,"rs %c %x\n",tmp,k);
          }
        }
    }
/* NOTE:  All values are loaded on a per chip basis */
/* i.e. the first 32 bit words in a line will go into all the rows */
/* of the first chip of the array and not along the top row */
/* for each chip in line !! This will cause major problems when */
/* simulating a greater than 32x32 array !! */
/* set up general register */
    else if(linp[0]=='g') {
      sscanf(ptr,"%d",&j);
      if(j>=0 || j<=73) {
        ptr=(char *)strchr(ptr,'=');
        ptr++;
        tmp=*(ptr++);
        if(tmp=='b') binary=1;
        else if(tmp=='h') binary=0;
        else {
          printf("Register in line %d has incorrect type.\n",i);
          fclose(fp1);
          fclose(fp2);
          fclose(fp3);
          exit(1);
        }
        get_param(j);
        for(cnt=offset; cnt<interval*xc*yc*perchip;cnt+=interval){
          if(binary==1)
            for(k=0;k<32;k++){
              reghex <<=1;
              if(*(ptr++)=='1') reghex++;
            }
          else {
            strncpy(str,ptr,8);
            ptr+=8;
            *(str+8)=0;
            sscanf(str,"%x",&reghex);
          }
          fprintf(fp2,"ws %c %08x %08x\n",tmp, cnt, reghex);
        }
      }
    }
/* set up flagged register */
    else if(linp[0]=='f') {
      sscanf(ptr,"%d",&j);
      if(j>=0 && j<=76){
        flags[j]=1;
        ptr=(char *)strchr(ptr,'=');
        if(ptr!=NULL && j !=76) {
          ptr++;
          tmp=*(ptr++);
          if(tmp=='b') binary=1;
          else if(tmp=='h') binary=0;
          else {
```

```
          printf("Register in line %d has incorrect type.\n",i);
          fclose(fp1);
          fclose(fp2);
          fclose(fp3);
          exit(1);
        }
        get_param(j);
        for(cnt=offset; cnt<interval*xc*yc*perchip;cnt+=interval){
          if(binary==1)
            for(k=0;k<32;k++){
              reghex <<=1;
              if(*(ptr++)=='1') reghex++;
            }
          else {
            strncpy(str,ptr,8);
            ptr+=8;
            *(str+8)=0;
            sscanf(str,"%x",&reghex);
          }
          fprintf(fp2,"ws %c %08x %08x\n",tmp, cnt, reghex);
        }
      }
    }
  }
/* unflag a given register */
  else if(linp[0]=='u') {
    sscanf(ptr,"%d",&j);
    if(j>=0 && j<=76) flags[j]=0;
    else printf("Tried to unflag bad register %d, line %d\n",j,i);
  }
/* send command directly to slocus simulator (sans initial 'c') */
  else if(linp[0]=='c') {
    fputs(ptr,fp2);
  }
/* print out comment in output file by preceeding with p */
  else if(linp[0]==';') {
    strcpy(str,"p ");
    strcat(str,linp);
    fputs(str,fp2); }
/* everything else is an error - break out and finish */
  else {
    printf("Error in file...finishing anyway.\n");
    break;
  }
 }
```

```
  printf("Processed %d lines.\n",i);
  fputs("quit\n",fp2);
  fprintf(fp3,"#define XCHIPS %d\n",xc);
  fprintf(fp3,"#define YCHIPS %d\n",yc);
  fprintf(fp3,"#define VERBOSELEVEL 0\n\n");
  fclose(fp1);
  fclose(fp2);
  fclose(fp3);
  printf("Make sure to recompile the Slocus program with\n");
  printf("XCHIPS=%d, YCHIPS=%d and VERBOSELEVEL=0\n",xc,yc);
  printf("These parameters are in the slocus.def file.\n");
  puts("");
}

get_param(int num)
{
  if(num>=0 && num<=63){tmp='r'; offset=num; interval=64; perchip=32;}
  else if(num>=64 && num<=67){tmp='e'; offset=num-64;
                              interval=4; perchip=1;}
  else if(num>=68 && num<=71){tmp='t'; offset=num-68;
                              interval=4; perchip=1;}
  else if(num==72){tmp='i'; offset=0; interval=1; perchip=1;}
  else if(num==73){tmp='o'; offset=0; interval=1; perchip=1;}
  else if(num==75){tmp='n'; offset=0; interval=1; perchip=32;}
  else if(num==76){tmp='d'; offset=0; interval=1; perchip=32;}
}
```

A.1.3 Slorun Execution Program

```
#include <stdio.h>
#include <string.h>
#define SLOCUSSLAVE 1
#define VERBOSELEVEL 0
#define XCHIPS 16
#define YCHIPS 16
#include "slocus.c"

#ifndef sparc
#include "_filbuf.c"
#endif

main()
{
  char fio[120],*charptr,filstr[100],cmdstr[200],*tmpptr,buf[129];
  unsigned char chunk;
  long int sresponse,regnum, i,j,k,l,cnst,*regptr,v1,v2;
  int fd ;
  FILE *filecmd,*fileslocus,*fileout;

#ifndef sparc
  filecmd=
    fopen("/home/ar/urop/kutscha/slocus/i960/slorun.cmd","r");
#else
  filecmd=fopen("slorun.cmd","r");
#endif
  if(filecmd == NULL)
    { puts("error opening command file.."); exit(1); }

#ifndef sparc
  fileout=
    fopen("/home/ar/urop/kutscha/slocus/i960/slorun.output","w");
#else
  fileout=fopen("slorun.output","w");
#endif
  if(fileout == NULL)
    { puts("error opening output.."); exit(2); }

  while(1)
    {
      if(fgets(fio,120,filecmd) == NULL) break;
      charptr=fio;
      if(*charptr=='q') break;
      if(*charptr=='c') reset_slocus();
      if(*charptr=='r') {
        tmpptr=next_space(charptr);
        tmpptr=clear_space(tmpptr);
        sscanf(tmpptr,"%s",filstr);
        fileslocus=fopen(filstr,"r");
        if(fileslocus != NULL) {

#ifndef sparc
          sprintf(cmdstr,
                  "echo Running i960 %s >> slorun.log",filstr);
          systemd(cmdstr);
          systemd("date >> slorun.log");
#else
          sprintf(cmdstr,
                  "echo Running Sparc %s >> slorun.log",filstr);
          system(cmdstr);
          system("date >> slorun.log");
#endif

          while(1)
            {
              if(fgets(fio,120,fileslocus) == NULL) break;
              if(*fio=='q') break;
              if(*fio=='p') fputs((fio+2),fileout);
              *(fio+strlen(fio)-1)=0; /* clear the newline  */
              sresponse=slocus(fio);
              if(*fio=='r') fprintf(fileout,"%08X\n",sresponse);
            }
#ifndef sparc
          systemd("date >> slorun.log");
          sprintf(cmdstr,
                  "grep -c Executing %s >> slorun.log",filstr);
          systemd(cmdstr);
#else
          system("date >> slorun.log");
          sprintf(cmdstr,
                  "grep -c Executing %s >> slorun.log",filstr);
          system(cmdstr);
#endif
          fclose(fileslocus);
        }
        else printf("Error opening file %s\n",filstr);
      }
      if(*charptr=='s') {
        tmpptr=next_space(charptr);
        tmpptr=clear_space(tmpptr);
        sscanf(tmpptr,"%d %s",&regnum, filstr);
        fd=creat(filstr,0664);
        if(fd != -1) {
#ifndef sparc
          sprintf(cmdstr,
                  "echo Saving register %d to %s >> slorun.log",
                  regnum,filstr);
          systemd(cmdstr);
#else
          sprintf(cmdstr,
                  "echo Saving register %d to %s >> slorun.log",
                  regnum,filstr);
          system(cmdstr);
#endif
          sprintf(buf,"P5\n128 128\n255\n");
          i = write(fd,buf,15);
          if(i!=15) printf("error writing file header %s\n",filstr);
          printf("Saving register %d to %s\n",regnum,filstr);
          for(i=0; i<16;i++) { /* down 16 chips */
            printf("%2d\n",(15-i));
            for(j=0; j<8; j++) { /* down 8 numbers */
              for(k=0; k<16;k++) {/* accross 16 chips */
                regptr=slocus_regfile +
                  (regnum + (k*64*32+j*64*4+64*32*16*i));
                v1=*regptr; v2=*(regptr+64);
                for(l=7;l>=0;l--){ /* accross 16 snakes */
                  chunk=((v2 >> (l*4)) & 0x0F) << 4 |
                    flip4((v1 >> (l*4)) & 0x0F);
                  *(buf+k*8+(7-l))=chunk;
                }
              }
              k=write(fd,buf,128);
              if(k!=128) printf("write error %s i %d\n",filstr,i);
            }
          }
        }
        else printf("Error writing picture file %s\n",filstr);
      }
      if(*charptr=='l') {
        tmpptr=next_space(charptr);
        tmpptr=clear_space(tmpptr);
        sscanf(tmpptr,"%d %s",&regnum,filstr);
        fd=open(filstr,0);
        if(fd != -1) {
#ifndef sparc
          sprintf(cmdstr,
                  "echo Loading register %d from %s >> slorun.log",
                  regnum,filstr);
          systemd(cmdstr);
#else
          sprintf(cmdstr,
                  "echo Loading register %d from %s >> slorun.log",
                  regnum,filstr);
          system(cmdstr);
#endif
          i=read(fd,buf,15);
          if((*buf != 'P') || (i != 15))
            printf("bad input file %s\n",filstr);
          printf("Loading register %d from %s\n",regnum,filstr);
          for(i=0; i<16;i++) { /* down 16 chips */
            printf("%2d\n",(15-i));
            for(j=0; j<8; j++) {/* down 8 numbers */
              k=read(fd,buf,128);
              if(k != 128) printf("error reading i %d\n",i);
              for(k=0; k<16;k++) {/* accross 16 chips */
                cnst=(regnum + (k*64*32+j*64*4+64*32*16*i));
                regptr=slocus_regfile + cnst;
                v1=0; v2=0;
                for(l=0;l<8;l++){ /* accross 16 snakes */
                  chunk=*(buf+k*8+l);
                  v1 <<=4; v1 |= flip4((int)chunk & 0x0F);
                  v2 <<=4; v2 |= ((int)chunk & 0x0F0) >> 4;
                }
/*              if(i==0) printf("v1 %8x v2 %8x\n",v1,v2); */
                *regptr=v1; *(regptr+64)=v2;
                *(regptr+128)=0; *(regptr+192)=0;
              }
            }
          }
        }
        else printf("Error reading picture file %s\n",filstr);
      }
    }
  fclose(fileout);
  fclose(filecmd);

  exit(0);
```

```
}


flip4(n)
{
  switch(n)
    {
    case 0:  return 0;
    case 1:  return 8;
    case 2:  return 4;
    case 3:  return 12;
    case 4:  return 2;
    case 5:  return 10;
    case 6:  return 6;
    case 7:  return 14;
    case 8:  return 1;
    case 9:  return 9;
    case 10: return 5;
    case 11: return 13;
    case 12: return 3;
    case 13: return 11;
    case 14: return 7;
    case 15: return 15;
    }
}
```

A.2 Vision Algorithms

The following pages list the Abacus assembler code utilizing optimized procedures to implement the four following vision algorithms:

1. `gtest.asm` - Performs a single Gaussian convolution by convolving the image with a triangle in both the horizontal and vertical direction.

2. `edgetest.asm` - Applies two Gaussians to an image, then a Laplacian kernel, then searches for resulting zero-crossings in the image.

3. `surfacetest.asm` - Uses surface reconstruction equations to approximate a surface given noisy information. Approximates the height, X-slope and Y-slope of a given surface.

4. `optiflowtest.asm` - Takes two binary images and determines the most-likely displacement of the objects in the second image relative to the first. Finds a maximum positive displacement of two on both axes and calculates a voting sum over a 3x3 area using a spiral technique.

A.2.1 Gaussian

```
include /projects/abacus/src/asm/all-lib.asm ;

globals : image = 11 , output = 12 ;

mesh : 32 , 32 ;

procedure main ( none )
{
        call ( init-config ) ;
        comment shifting precision up ;
        call ( orient-high ) ;
        call ( asl4  image image ) ;
        comment performing gaussian ;
        call ( gauss image image ) ;
        comment shifting precision down ;
        call ( orient-low ) ;
        call ( asr image output ) ;
        call ( asr2 output output ) ;
        call ( asr4 image image ) ;
        comment done ;
}
;
```

A.2.2 Edge Detection

```
include /projects/abacus/src/asm/all-lib.asm ;

globals : i1 = 11 , tmp1 = 44 , tmp2 = 45 , tmp3 = 46 ,
tmp4 = 47 , tmp5 = 48 , tmp6 = 49 , tmp7 = 50 , tmp8 = 51 ,
tmp9 = 52 , tmp10 = 53 , lp = 12 , cy = 13 , horiz = 54 ,
vert = 55 , diag = 56 , g1 = 14 ;

mesh : 32 , 32 ;


procedure main ( none )
{
! image loaded in i1 already
        call ( init-config ) ;
        comment shifting precision up ;
        call ( orient-high ) ;
        call ( asl4  i1 i1 ) ;
        comment performing gaussian 1 2 ;
        call ( gauss i1 i1 ) ;
        call ( gauss i1 i1 ) ;
        comment shifting precision down ;
        call ( orient-low ) ;
        call ( asr2 i1 g1 ) ;
        call ( asr g1 g1 ) ;
        call ( asr4  i1 i1 ) ;
        comment doing laplace kernel ;
        call ( orient-high ) ;
        call ( asl2 i1 tmp4 ) ;
        call ( orient-north ) ;
        call ( shift-cluster-north i1 tmp1 ) ;
        call ( shift-cluster-north tmp4 tmp5 ) ;
        call ( orient-south ) ;
        call ( shift-cluster-south i1 tmp2 ) ;
        call ( shift-cluster-south tmp4 tmp6 ) ;
        call ( orient-east ) ;
        call ( shift-cluster-east tmp4 tmp7 ) ;
        call ( shift-cluster-east tmp2 tmp9 ) ;
        call ( shift-cluster-east tmp1 tmp10 ) ;
        call ( orient-west ) ;
        call ( shift-cluster-west tmp4 tmp8 ) ;
        call ( shift-cluster-west tmp2 tmp2 ) ;
        call ( shift-cluster-west tmp1 tmp1 ) ;
        cy = ( zero ) ;
        lp = tmp5 ;  ! initial number
        comment final laplacian ;
        call ( orient-high ) ;
        call ( acc-start lp tmp6 cy lp ) ;
        call ( accumulate lp cy tmp7 lp ) ;
        call ( accumulate lp cy tmp8 lp ) ;
        call ( accumulate lp cy tmp1 lp ) ;
        call ( accumulate lp cy tmp2 lp ) ;
        call ( accumulate lp cy tmp9 lp ) ;
        call ( accumulate lp cy tmp10 lp ) ;
        call ( add cy lp lp ) ;
        call ( asl4 i1 tmp1 ) ;
        call ( minus lp tmp1 lp ) ;
        call ( minus lp tmp4 lp ) ;
        comment find-zero-crossings ;
        call ( sign lp tmp1 ) ;
        call ( orient-north ) ;
        call ( shift-cluster-north tmp1 tmp2 ) ;
        horiz = ( xor tmp1 tmp2 ) ;
        call ( orient-west ) ;
        call ( shift-cluster-west tmp1 tmp4 ) ;
        vert = ( xor tmp4 tmp1 ) ;
        diag = ( or horiz vert ) ;
        comment done ;
}
;
```

A.2.3 Surface Approximation

```
include /projects/abacus/src/asm/all-lib.asm ;

globals : image = 11 , i1 = 10 , tmp1 = 44 , tmp2 = 45 , tmp3 = 46 ,
tmp4 = 47 , tmp5 = 48 , tmp6 = 49 , tmp7 = 50 , tmp8 = 51 ,
tmp9 = 52 , tmp10 = 53 , lp = 12 , cy1 = 13 , u2 = 54 , p2 = 55 ,
q2 = 56 , u1 = 57 , p1 = 58 , q1 = 59 ;

mesh : 32 , 32 ;

procedure iteration ( u1 , u2 , p1 , p2 , q1 , q2 )
{
        comment computing height ;
        call ( orient-east ) ;
        call ( shift-cluster-east p1 tmp1 ) ;
        call ( shift-cluster-east u1 tmp3 ) ;
        call ( orient-north ) ;
        call ( shift-cluster-north q1 tmp2 ) ;
        call ( shift-cluster-north u1 tmp4 ) ;
        call ( orient-west ) ;
        call ( shift-cluster-west u1 tmp6 ) ;
        call ( orient-south ) ;
        call ( shift-cluster-south u1 tmp5 ) ;
        call ( orient-high ) ;
        call ( acc-start tmp6 tmp2 cy1 tmp6 ) ;
        call ( accumulate tmp6 cy1 tmp3 tmp6 ) ;
        call ( accumulate tmp6 cy1 tmp4 tmp6 ) ;
        call ( accumulate tmp6 cy1 tmp5 tmp6 ) ;
        call ( accumulate tmp6 cy1 tmp1 u2 ) ;
        call ( add cy1 u2 cy1 ) ;
        call ( minus cy1 q1 cy1 ) ;
        call ( minus cy1 p1 cy1 ) ;
        call ( orient-low ) ;
        call ( asr2 cy1 u2 ) ;
        comment computing slope1 ;
        call ( orient-west ) ;
        call ( shift-cluster-west u2 tmp6 ) ;
        call ( shift-cluster-west p1 tmp5 ) ;
        call ( orient-north ) ;
        call ( shift-cluster-north p1 tmp3 ) ;
        call ( orient-south ) ;
        call ( shift-cluster-south p1 tmp4 ) ;
call ( orient-high ) ;
call ( acc-start tmp6 tmp1 cy1 tmp6 ) ;
call ( accumulate tmp6 cy1 tmp3 tmp6 ) ;
call ( accumulate tmp6 cy1 tmp4 tmp6 ) ;
call ( accumulate tmp6 cy1 tmp5 p2 ) ;
call ( add cy1 p2 cy1 ) ;
call ( minus cy1 u2 cy1 ) ;
comment dividing by five ;
call ( asl cy1 tmp3 ) ;
call ( add cy1 tmp3 cy1 ) ;
call ( orient-low ) ;
call ( asr4 tmp3 cy1 ) ;
call ( orient-high ) ;
call ( add cy1 tmp3 cy1 ) ;
call ( orient-low ) ;
call ( asr4 cy1 cy1 ) ;
call ( orient-high ) ;
call ( add cy1 tmp3 cy1 ) ;
call ( add cy1 lsb cy1 ) ;
call ( orient-low ) ;
call ( asr4 cy1 p2 ) ;
comment computing slope2 ;
call ( orient-south ) ;
call ( shift-cluster-south u2 tmp3 ) ;
call ( shift-cluster-south q1 tmp4 ) ;
call ( orient-east ) ;
call ( shift-cluster-east q1 tmp5 ) ;
call ( orient-west ) ;
call ( shift-cluster-west q1 tmp6 ) ;
call ( orient-high ) ;
call ( acc-start tmp6 tmp2 cy1 tmp6 ) ;
call ( accumulate tmp6 cy1 tmp3 tmp6 ) ;
call ( accumulate tmp6 cy1 tmp4 tmp6 ) ;
call ( accumulate tmp6 cy1 tmp5 q2 ) ;
call ( add cy1 q2 cy1 ) ;
call ( minus cy1 u2 cy1 ) ;
comment dividing by five ;
call ( asl cy1 tmp3 ) ;
call ( add cy1 tmp3 cy1 ) ;
call ( orient-low ) ;
call ( asr4 cy1 cy1 ) ;
call ( orient-high ) ;
call ( add cy1 tmp3 cy1 ) ;
call ( orient-low ) ;
call ( asr4 cy1 cy1 ) ;
```

```
        call ( orient-high ) ;
        call ( add cy1 tmp3 cy1 ) ;
        call ( add cy1 lsb cy1 ) ;
        call ( orient-low ) ;
        call ( asr4 cy1 q2 ) ;
}

procedure main ( none )
{
! image, p1 and q1 already loaded
        call ( init-config ) ;
        comment getting landscape ;
        u1 = image ;
        call ( iteration u1 u2 p1 p2 q1 q2 ) ;
        image = u2 ;
        p1 = p2 ;
        q1 = q2 ;
        comment done ;
}
;
```

A.2.4 Optical Flow

```
include /projects/abacus/src/asm/all-lib.asm ;

globals : R1 = 5 , R2 = 44 , R3 = 6 , R4 = 45 , R5 = 7 , R6 = 46 ,
R7 = 8 , R8 = 47 , R9 = 9 , R10 = 48 , R11 = 10 , R12 = 49 ,
R13 = 11 , R14 = 50 , R15 = 12 , R16 = 51 , VAL = 13 , image = 14 ;

mesh : 32 , 32 ;

procedure main ( none )
{
! image loaded into "image" and moved into R5
  call ( init-config ) ;
  comment generating initial mover ;
  R1 =  ( zero ) ; ! max
  R7 =  ( zero ) ; ! best x
  R8 =  ( zero ) ; ! best y
  R4 = image ;

!  call ( orient-north ) ;
!  call ( shift-cluster-north image R5 ) ;
!  call ( shift-cluster-north R5 R5 ) ;
!  call ( orient-east ) ;
!  call ( shift-cluster-east R5 R5 ) ;

  comment doing loop 0 0;
  R3 =  ( zero ) ;
  R2 = R4 ;
  call ( orient-high ) ;
  call ( equal R2 R5 R2 ) ;
  R2 = ( and R2 lsb ) ; ! set matching things to one

  call ( add R2 R3 R3 ) ;
  call ( orient-south ) ;
  call ( shift-cluster-south R2 R2 ) ;
  call ( orient-high ) ;
  call ( add R2 R3 R3 ) ;
  call ( orient-east ) ;
  call ( shift-cluster-east R2 R2 ) ;
  call ( orient-high ) ;
  call ( add R2 R3 R3 ) ;
  call ( orient-north ) ;
  call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( greater-than R3 R1 IDLE ) ;
call ( logic-not IDLE IDLE ) ;
R1 = R3 ;
R7 = ( zero ) ;
R8 = ( zero ) ;
IDLE =  ( zero ) ;

comment doing loop 0 1 ;
R3 =  ( zero ) ;
call ( orient-north ) ;
call ( shift-cluster-north R4 R2 ) ;
call ( orient-high ) ;
call ( equal R2 R5 R2 ) ;
R2 = ( and R2 lsb ) ; ! set matching things to one

call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-east ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( greater-than R3 R1 IDLE ) ;
call ( logic-not IDLE IDLE ) ;
R1 = R3 ;
R7 = ( zero ) ;
R8 = lsb ;
call ( numgen-zero R8 ) ;
IDLE =  ( zero ) ;

comment doing loop 0 2 ;
R3 =  ( zero ) ;
call ( orient-north ) ;
call ( shift-cluster-north R4 R2 ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( equal R2 R5 R2 ) ;
R2 = ( and R2 lsb ) ; ! set matching things to one

call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-east ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( greater-than R3 R1 IDLE ) ;
call ( logic-not IDLE IDLE ) ;
R1 = R3 ;
R7 = ( zero ) ;
R8 = lsb ;
call ( numgen-zero R8 ) ;
IDLE =  ( zero ) ;

comment doing loop 1 0 ;
R3 =  ( zero ) ;
call ( orient-east ) ;
call ( shift-cluster-east R4 R2 ) ;
call ( orient-high ) ;
call ( equal R2 R5 R2 ) ;
R2 = ( and R2 lsb ) ; ! set matching things to one
```

```
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-east ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( greater-than R3 R1 IDLE ) ;
call ( logic-not IDLE IDLE ) ;
R1 = R3 ;
R7 = lsb ;
R8 = ( zero ) ;
IDLE =  ( zero ) ;

comment doing loop 1 1 ;
R3 =  ( zero ) ;
call ( orient-east ) ;
call ( shift-cluster-east R4 R2 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( equal R2 R5 R2 ) ;
R2 = ( and R2 lsb ) ; ! set matching things to one

call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-east ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( greater-than R3 R1 IDLE ) ;
call ( logic-not IDLE IDLE ) ;
R1 = R3 ;
R7 = lsb ;
R8 = lsb ;
IDLE =  ( zero ) ;
```

```
comment doing loop 1 2 ;
R3 =  ( zero ) ;
call ( orient-east ) ;
call ( shift-cluster-east R4 R2 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( equal R2 R5 R2 ) ;
R2 = ( and R2 lsb ) ; ! set matching things to one

call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-east ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( greater-than R3 R1 IDLE ) ;
call ( logic-not IDLE IDLE ) ;
R1 = R3 ;
R7 = lsb ;
R8 = lsb ;
call ( numgen-zero R8 ) ;
IDLE =  ( zero ) ;

comment doing loop 2 0 ;
R3 =  ( zero ) ;
call ( orient-east ) ;
call ( shift-cluster-east R4 R2 ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( equal R2 R5 R2 ) ;
R2 = ( and R2 lsb ) ; ! set matching things to one

call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-east ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( greater-than R3 R1 IDLE ) ;
call ( logic-not IDLE IDLE ) ;
R1 = R3 ;
R7 = lsb ;
call ( numgen-zero R7 ) ;
R8 = ( zero ) ;
IDLE =  ( zero ) ;

comment doing loop 2 1 ;
R3 =  ( zero ) ;
call ( orient-east ) ;
call ( shift-cluster-east R4 R2 ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( equal R2 R5 R2 ) ;
R2 = ( and R2 lsb ) ; ! set matching things to one

call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-east ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-west ) ;
call ( shift-cluster-west R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( greater-than R3 R1 IDLE ) ;
call ( logic-not IDLE IDLE ) ;
R1 = R3 ;
R7 = lsb ;
call ( numgen-zero R7 ) ;
R8 = lsb ;
IDLE =  ( zero ) ;

comment doing loop 2 2 ;
R3 =  ( zero ) ;
call ( orient-east ) ;
call ( shift-cluster-east R4 R2 ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( equal R2 R5 R2 ) ;
R2 = ( and R2 lsb ) ; ! set matching things to one

call ( add R2 R3 R3 ) ;
call ( orient-south ) ;
call ( shift-cluster-south R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-east ) ;
call ( shift-cluster-east R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
call ( shift-cluster-north R2 R2 ) ;
call ( orient-high ) ;
call ( add R2 R3 R3 ) ;
call ( orient-north ) ;
```

```
  call ( shift-cluster-north R2 R2 ) ;
  call ( orient-high ) ;
  call ( add R2 R3 R3 ) ;
  call ( orient-west ) ;
  call ( shift-cluster-west R2 R2 ) ;
  call ( orient-high ) ;
  call ( add R2 R3 R3 ) ;
  call ( orient-west ) ;
  call ( shift-cluster-west R2 R2 ) ;
  call ( orient-high ) ;
  call ( add R2 R3 R3 ) ;
  call ( orient-south ) ;
  call ( shift-cluster-south R2 R2 ) ;
  call ( orient-high ) ;
  call ( add R2 R3 R3 ) ;
  call ( orient-south ) ;
  call ( shift-cluster-south R2 R2 ) ;
  call ( orient-high ) ;
  call ( add R2 R3 R3 ) ;
  call ( greater-than R3 R1 IDLE ) ;
  call ( logic-not IDLE IDLE ) ;
  R1 = R3 ;
  R7 = lsb ;
  call ( numgen-zero R7 ) ;
  R8 = lsb ;
  call ( numgen-zero R8 ) ;
  IDLE =  ( zero ) ;
  comment done ;
}
;
```

A.3 Optimized Assembly Procedures

The following pages contain listings of optimized assembly procedures to help the compiler abstract away from the internals of the Abacus architecture. Some procedures are not fully optimized due to bugs in the assembler. Procedures assume that operands come from either register file bank and may have to take extra cycles to move registers around. Procedure groups include the following:

1. `all-lib.asm, registers.asm` - A general include file for programs and a file to tell the assembler all the globally assigned registers.

2. `orient.asm` - Procedures to orient the NEWS network for all procedures.

3. `add.asm` - Contains adding, subtracting, accumulation and decummulation routines.

4. `multdiv.asm` - Contains multiplication. (division and remainder not implemented yet)

5. `shift.asm` - Provides for inter-chip shifting of clusters as well as logical and arithmetic shifts for intra-cluster functions.

6. `logic.asm` - Gives the Ascheme compiler an abstract bitwise logic library to use. (Not really needed at the assembler level)

7. `compare.asm` - Performs greater-than, less-than and equal comparisons of numbers.

8. `rebroadcast.asm` - Broadcasts a register to the global-OR pin on each chip. Also contains a procedure to determine the sign (polarity) of a number.

9. `gauss.asm` - Performs a triangle convolution on a register in both the horizontal and vertical direction to approximate a Gaussian function. Used by `gtest.asm` and `edgetest.asm` in the vision algorithms.

A.3.1 Library Listing and Global Registers

```
include /projects/abacus/src/asm/registers.asm ;
include /projects/abacus/src/asm/config.asm ;
include /projects/abacus/src/asm/add.asm ;
include /projects/abacus/src/asm/compare.asm ;
include /projects/abacus/src/asm/orient.asm ;
include /projects/abacus/src/asm/shift.asm ;
include /projects/abacus/src/asm/rebroadcast.asm ;
include /projects/abacus/src/asm/logic.asm ;
include /projects/abacus/src/asm/multdiv.asm ;
include /projects/abacus/src/asm/gauss.asm ;
```

```
!FILE OF REGISTER GLOBALS - counterpart of registers.sc

globals : in = 0 , out = 0 , inselect0 = 1 , inselect1 = 2 ,
          dp-in/out = 32 , break = 33 , idle = 35 ,
          msb = 36 , lsb = 37 , pid0 = 3 , pid1 = 4 ,
          pid2 = 38 , pid3 = 39 , news-in-north = 64 ,
          news-in-east = 65 , news-in-south = 66 ,
          news-in-west = 67 , news-out-north = 68 ,
          news-out-east = 69 , news-out-south = 70 ,
          news-out-west = 71 , data-plane-in = 72 ,
          data-plane-out = 73 , global-or-flag = 74 ,
          news-network = 75 , data-plane-network = 76 ,
          top-edge = 40 , right-edge = 41 ,
          bottom-edge = 42 , left-edge = 43 ;
```

A.3.2 Orientation Procedures

```
!       Orient to one direction along snake, or reverse thereof
!necessary : pid0,1,2 not all in same,  in-select1,0 with majority
!possible arrangement : (pid2) (pid1 pid0 in_select1 in_select0)
!
! The major problem with these orientations is the the msb of one
! cluster points to the lsb of an adjacent one so we get very long
! propagation lines.  This must be corrected....

!read from lower id number ->flow to msb
procedure orient-high-connect ( none )
{
        inselect1 = ( not-and-or pid2 pid1 pid0 ) , break = ( one ) ;
        inselect0 = ( and-or pid2 pid1 pid0 ) ;
}
!read from higher id number ->flow to lsb
procedure orient-low-connect ( none )
{
        inselect1 = ( or-and pid2 pid1 pid0 ) , break = ( one ) ;
        inselect0 = ( not-or-and pid2 pid1 pid0 ) ;
}

! The two following procedures try to correct this problem at
! expense of a clock cycle.
!read from lower id number ->flow to msb
procedure orient-high ( none )
{
        inselect1 = ( not-and-or pid2 pid1 pid0 ) , break = ( one ) ;
        inselect0 = ( and-or pid2 pid1 pid0 ) ;
        inselect0 = ( or inselect0 lsb ) ;
}
!read from higher id number ->flow to lsb
procedure orient-low ( none )
{
        inselect1 = ( or-and pid2 pid1 pid0 ) , break = ( one ) ;
        inselect0 = ( not-or-and pid2 pid1 pid0 ) ;
        inselect1 = ( and-not inselect1 msb ) ;
}

! These procedures allow image shifting and orient the
! chip boundaries as well.  The break bit is turned on to save
! cycles since it is easy to turn it off and let values propagate
```

```
procedure orient-south ( none ) !read from processor above
{
        controlbits copy 0 ;
        inselect1 = ( zero ) , break = ( one ) ;
        inselect0 = ( zero ) ;
}

procedure orient-north ( none ) !read from processor below
{
        controlbits copy 8 ;
        inselect1 = ( one ) , break = ( one ) ;
        inselect0 = ( one ) ;
}

procedure orient-east ( none ) !read from processor to left
{
        controlbits copy 0 ;
        inselect1 = ( one ) , break = ( one ) ;
        inselect0 = ( zero ) ;
}

procedure orient-west ( none ) !read from processor to right
{
        controlbits copy 8 ;
        inselect1 = ( zero ) , break = ( one ) ;
        inselect0 = ( one ) ;
}
procedure orient-read-from-north ( none )
{
        controlbits copy 0 ;
        inselect1 = ( zero ) , break = ( one ) ;
        inselect0 = ( zero );
}

procedure orient-read-from-east ( none )
{
        controlbits copy 8 ;
        inselect1 = ( zero ) , break = ( one ) ;
        inselect0 = ( one );
}

procedure orient-read-from-west ( none )
{
        controlbits copy 0 ;
        inselect1 = ( one ) , break = ( one ) ;
        inselect0 = ( zero );
}

procedure orient-read-from-south ( none )
{
        controlbits copy 8 ;
        inselect1 = ( one ) , break = ( one ) ;
        inselect0 = ( one );
}

;
```

A.3.3 Addition and Accumulation Routines

```
!This performs a manchester-carry-chain addition
!again, watch for direction of read-set-up - again, I think it
!should be toward the lsb...  orient-high  (flow to msb)
!
!The add functions do a full carry-propagate addition of
!registers "a" and "b", storing the result in "s"
!
!When adding large numbers of registers, use the accumulate
!option instead.
!

! This procedure makes sure the input carry bit is zero so we don't
! have a carry from an adjacent number
! altered to handle all register banks
procedure add ( src1 , src2 , dst )
{
        locals-a : a , s ;
        locals-b : b , blarg ;
        a = src1 , b = src2 ;
! make sure lowest bit is broken
        break = ( or-same lsb a b ) , out = ( and a b ) ;
! not (A+B) & ~(A&B) wierdly enough...
        blarg = ( xor a b ) ;  ! also delay
        s = ( and-not in lsb ) , break = ( one ) ;
        dst = ( xor blarg s ) ;
}

!this may be slightly faster if you don't need to worry about the
!carry bit propagating from nearby clusters
!necessary : a, b NOT in same bank.
!if a, s or a, b in same bank
!break, s not in same bank (although this is fixable)
!possible setup : (break a) (b s in/out)
procedure addshort ( a , b , s )
{
        break = ( or-same lsb a b ) ;
!not (A+B) & ~(A&B) wierdly enough...
        out = ( and a b ) ;      !wait for break before writing to net
        delay ;                  !wait for propagation
        s = ( sum in a b ) , break = ( one ) ;
}
```

```
!               Accumulate
! This function adds the register "n" to the accumulation register "s"
! Register "c" is the carry register which stores the state of the
! system.  After all is accumulated, the user should use a carry-prop
! addition to add the "c" and "s" registers for the final result.
! This function is good for multiplication functions which require
! the sum of many numbers.
!
!
! We should add a subtract accumlate in the future....
!
!necesary : s, c, n not all in same bank.
! at least two of above in bank with out.
! If in, lsb in same bank, c in that bank as well
!
! orient-high required
! altered to handle all banks (for 7 or fewer "add" is faster)
! assume acc-start
procedure accumulate ( src , cy , nextnum , dst )
{
        locals-a: s , c1 ;
        locals-b: n , c2 , d ;
        s = src , n = nextnum ;
        c1 = cy , c2 = cy ;
        out = ( carry s n c1 ) , d = ( sum s n c2 ) ;
        cy = ( and-not in lsb ) ;
        dst = d ;
}

! src and nextnum in different banks
! cy in left and dst in right
! assumes orient-high and acc-start
procedure accumulate-short ( src , cy , nextnum , dst )
{
        locals-b : c2 ;
        out = ( carry src nextnum cy ) , c2 = cy ;
        cy = ( and-not in lsb ) , dst = ( sum src nextnum c2 ) ;
}

! this acc-start instruction must be used to start ALL accumulates
! it sets the break bit and sets up the carry register
! assumes orient high
procedure acc-start ( src1 , src2 , cy , dst )
{
        locals-a : s1 ;
        locals-b : s2 ;
        s1 = src1 , s2 = src2 ;
        out = ( carry s1 s2 ) , break = ( one ) ;
        cy = ( and-not in lsb ) ;
        dst  = ( sum s1 s2 ) ;
}
procedure acc-start-short ( src1 , src2 , cy , dst )
{
        out = ( carry src1 src2 ) , break = ( one ) ;
        cy = ( and-not in lsb ) , dst  = ( sum s1 s2 ) ;
}

! this function subtracts two numbers  s=a-b
! we accomplish this by inverting b and asserting the carry bit
! assumes orient-high-isolate
procedure minus ( src1 , src2 , diff )
{
        locals-a : a , s , cdrv , tmpcar ;
        locals-b : b ;
        a = src1 , b = ( not src2 ) ;
! COMPILER CREATES AN ERROR IF WE TRY TO COMBINE THE NEXT TWO LINES
! FIX THIS!!
        break = ( or-same lsb a b ) ;
        cdrv = lsb ;
        out = ( sel-and-xor cdrv a b ) ;
        delay ;
        tmpcar = ( or in lsb ) ;
        s = ( sum tmpcar a b ) , break = ( one ) ;
        diff = s ;
}
procedure minus-short ( a , b , s )
{
        locals-a : cy , tmpcar ;
! should be or-xor
        cy = lsb , s = ( not b ) ;
! break if a and b are different
! should be sel-andnot-same
        out = ( sel-and-xor cy a s ) , break = ( or-same lsb a s ) ;
        delay ;
        tmpcar = ( or in lsb ) ;
        s = ( same tmpcar a b ) , break = ( one ) ;
}

! this is the reverse of accumulate and subtracts a number
! from a stockpile by inverting the number and asserting the
! carry bit on the way out .
! assume orient-high-isolate
! altered to handle all registers (use minus if fewer than 7 )
! assumes unacc-start (break is one )
procedure unaccumulate ( src , cy , nextnum , dst )
{
        locals-a : s , c ;
        locals-b : n , d , c2 ;
        s = src , n = ( not nextnum ) ;
        c = cy , c2 = cy ;
        out = ( carry s n c ) , d = ( sum s n c2 ) ;
        cy = ( or in lsb ) ;
        dst = d ;
}
! src, nextnum in different banks, cy in left , dst in right
procedure unaccumulate-short ( src , cy , nextnum , dst )
{
        locals-b : c2 ;
        out = ( uncarry src nextnum cy ) , c2 = cy ;
        cy = ( or in lsb ) , dst = ( same src nextnum cy ) ;
}

! these procedures start off the unaccumulate instructions
! they assume orient high
procedure unacc-start ( src , subtract , cy , dst )
{
        locals-a : s1 ;
        locals-b : s2 ;
        s1 = src , s2 = subtract ;
        out = ( and-not s1 s2 ) , break = ( one ) ;
        cy = ( or in lsb ) ;
        dst  = ( sum s1 s2 ) ;
}

procedure unacc-start-short ( src1 , src2 , cy , dst )
{
        out = ( and-not src1 src2 ) , break = ( one ) ;
        cy = ( or in lsb ) , dst  = ( same s1 s2 ) ;
}

;
```

A.3.4 Multiplication Code

```
! multiply function
! result = a * b.  a and b in either bank.  result in left bank
! (in/out/result) (lsb/break)
! assuming orient-high-isolate
procedure mult16 ( a , b , output )
{
        locals-a : tmp2 , decbit , cy ;
        locals-b : tmp , result , cy2 ;
        decbit = lsb , tmp = b ;
        out = ( and decbit a ) , break = ( zero ) ;
        result = ( zero ) ;  ! do delay here..
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 1
        out = ( carry result tmp2 ) , result = ( sum result tmp2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 2
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 3
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 4
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 5
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 6
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 7
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 8
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 9
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 10
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 11
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
```

```
! accumulate 12
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 13
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 14
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 15
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;
        out = tmp ;
        tmp = ( and-not in lsb ) , out = decbit ;
        decbit = ( and-not in lsb ) ;

        out = ( and decbit a ) , break = ( zero ) ;
        delay ;
        tmp2 = ( and in tmp ) , break = ( one ) ;
! accumulate 16
```

```
        out = ( carry result tmp2 cy ) ,
         result = ( sum result tmp2 cy2 ) ;
        cy = ( and-not in lsb ) , cy2 = ( and-not in lsb ) ;

! final add
! should use or-same
        out = ( and result cy ) , break = ( or-same lsb result cy ) ;
        delay ;
        tmp2 = ( and-not in lsb ) ;
        result = ( sum tmp2 result cy2 ) ;
        output = result ;
}


! assumes orient-high-isolate
! calculate a/b , produces result and remainder
;
```

A.3.5 Cluster, Logical and Arithmetic Shifting

```
!               Arithmetic Shift Right
!note, reads should be organized from the MSB down... orient-low
!necessary : if in, msb in same bank, src there as well
!all asr's assume orient-low-isolate
procedure asr ( src , dst )     ! an arithmetic shift right...
{
        locals-a : d , s ;
        s = src ;
        out = src , break = ( one ) ;
        d = ( mux in s msb ) ;
        dst = d ;
}
procedure asr2 ( src , dst )    ! an arithmetic shift right...
{
        locals-a : d , s ;
        s = src ;
        out = src , break = ( one ) ;
        out = ( mux in s msb ) ;
        d = ( mux in s msb ) ;
        dst = d ;
}
procedure asr4 ( src , dst )    ! an arithmetic shift right...
{
        locals-a : d , s ;
        s = src ;
        out = src , break = ( one ) ;
        out = ( mux in s msb ) ;
        out = ( mux in s msb ) ;
        out = ( mux in s msb ) ;
        d = ( mux in s msb ) ;
        dst = d ;
}
procedure asr-short ( src , dst )       ! an arithmetic shift right...
{
        out = src , break = ( one ) ;
        dst = ( mux in src msb ) ;
}
procedure asr2-short ( src , dst )      ! an arithmetic shift right...
{
        out = src , break = ( one ) ;
        out = ( mux in src msb ) ;
        dst = ( mux in src msb ) ;
}
procedure asr4-short ( src , dst )      ! an arithmetic shift right...
{
        out = src , break = ( one ) ;
        out = ( mux in src msb ) ;
        out = ( mux in src msb ) ;
        out = ( mux in src msb ) ;
        dst = ( mux in s msb ) ;
}
! Do logic shift rights, assume orient-low
procedure lsr ( src , dst )
{
        out = src , break = ( one ) ;
        dst = ( and-not in msb ) ;
}
procedure lsr2 ( src , dst )
{
        out = src , break = ( one ) ;
        out = ( and-not in msb ) ;
        dst = ( and-not in msb ) ;
}
procedure lsr4 ( src , dst )
{
        out = src , break = ( one ) ;
        out = ( and-not in msb ) ;
        out = ( and-not in msb ) ;
        out = ( and-not in msb ) ;
        dst = ( and-not in msb ) ;
}

! these are also logic shift lefts...
! all shift left procedures assume orient-high-isolate
procedure asl ( src , dst )     ! an arithmetic shift left...
{
        out = src , break = ( one ) ;
        dst = ( and-not in lsb ) ;
}
procedure asl2 ( src , dst )
{
        out = src , break = ( one ) ;
        out = ( and-not in lsb ) ;
        dst = ( and-not in lsb ) ;
}
procedure asl4 ( src , dst )
```

```
{
        out = src , break = ( one ) ;
        out = ( and-not in lsb ) ;
        out = ( and-not in lsb ) ;
        out = ( and-not in lsb ) ;
        dst = ( and-not in lsb ) ;
}

! number generation shifts, these shift left and add a one or zero
procedure numgen-zero ( dst )   ! an arithmetic shift left...
{
        out = dst , break = ( one ) ;
        dst = ( and-not in lsb ) ;
}
procedure numgen-one ( dst )    ! an arithmetic shift left...
{
        out = dst , break = ( one ) ;
        dst = ( or in lsb ) ;
}



! cluster shifts - shift a 16 bit number up, down, left or right
! destination must be in B bank and edge flags must be in B!!
! assumes orient-north with break set
procedure shift-cluster-north ( src , dst )
{
        locals-b : d ;
        out = src ;
        out = in ;
        out = in ;
        out = in ;
        out = in , d = in ;
! take care of boundary shifting
        out = in, d = ( mux d in bottom-edge ) ;
        d = ( mux d in bottom-edge ) ;
        dst = d ;
}
! assumes orient-south
procedure shift-cluster-south ( src , dst )
{
        locals-b : d ;
        out = src ;
        out = in ;
        out = in ;
        out = in ;
        out = in , d = in ;
! take care of boundary shifting
        out = in, d = ( mux d in top-edge ) ;
        d = ( mux d in top-edge ) ;
        dst = d ;
}
! assumes orient east
procedure shift-cluster-east ( src , dst )
{
        locals-b : d ;
        out = src ;
        out = in ;
        out = in ;
        out = in ;
        out = in , d = in ;
! take care of boundary shifting
        out = in, d = ( mux d in left-edge ) ;
        d = ( mux d in left-edge ) ;
        dst = d ;
}
! assumes orient west
procedure shift-cluster-west ( src , dst )
{
        locals-b : d ;
        out = src ;
        out = in ;
        out = in ;
        out = in ;
        out = in , d = in ;
! take care of boundary shifting
        out = in, d = ( mux d in right-edge ) ;
        d = ( mux d in right-edge ) ;
        dst = d ;
}

!  a and s in same bank as break
! assume the orient command sets break to one
! This DOES NOT handle chip boundary conditions!!
procedure image-shift-same-bank ( src , dst )
{
        out = src ;
        out = in ;
        out = in ;
        out = in ;
        dst = in ;
}

;
```

A.3.6 Bitwise Logical Operations

```
! computes general logic functions
! needed for the compiler to handle registers in both banks
! this takes twice as long but comes out correctly
! this also prevents the compiler from handling primitives.

procedure logic-not ( src , dst )
{
        dst = ( not src ) ;
}

procedure logic-and ( src1 , src2 , dst )
{
        locals-a : a ;
        locals-b : b ;
        a = src1 , b = src2 ;
        dst = ( and a b ) ;
}

procedure logic-or ( src1 , src2 , dst )
{
        locals-a : a ;
        locals-b : b ;
        a = src1 , b = src2 ;
        dst = ( or a b ) ;
}

procedure logic-nand ( src1 , src2 , dst )
{
        locals-a : a ;
        locals-b : b ;
        a = src1 , b = src2 ;
        dst = ( nand a b ) ;
}

procedure logic-nor ( src1 , src2 , dst )
{
        locals-a : a ;
        locals-b : b ;
        a = src1 , b = src2 ;
        dst = ( nor a b ) ;
}
```

```
procedure logic-same ( src1 , src2 , dst )
{
        locals-a : a ;
        locals-b : b ;
        a = src1 , b = src2 ;
        dst = ( same a b ) ;
}

procedure logic-xor ( src1 , src2 , dst )
{
        locals-a : a ;
        locals-b : b ;
        a = src1 , b = src2 ;
        dst = ( xor a b ) ;
}

;
```

A.3.7 Comparison Functions

```
! this routine compares two numbers
!    A > B ?
! orient-high-isolate is assumed
! NOTE: This ignores the state of the topmost bit!!
! returns all ones if A > B otherwise all zeros

!necessary : if a, b in same bank, so is break.  diff and break not in
!       same bank
!possible arrangement : (a break) (b diff in/out)
procedure greater-than ( src1 , src2 , diff )
{
        locals-a : a , d ;
        locals-b : b ;
        a = src1 , b = src2 ;
! deal with lsb bit!!  or-xor lsb a b
        d = ( xor a b ) ;
        out = ( and-not a b ) , break = ( or lsb d ) ;
        delay ;
! combin next instructions
        d = ( and-not in lsb ) ; ! get bits minus lsb
        out = ( and d msb ) , break = lsb ;
        delay ;
        d = in , break = ( one ) ;
        diff = d ;
}
procedure greater-than-short ( a , b , diff )
{
! use or-xor lsb a b
        out = ( and-not a b ) , break = ( xor a b ) ;
        delay ;
! combine next two lines into one using special func
! or-andnot msb in lsb , move msb to left bank first
        diff = ( and-not in lsb ) ;
        out = ( and diff msb ) , break = lsb ;
        delay ;
        diff = in , break = ( one ) ;
}
procedure less-than ( src1 , src2 , diff )
{
        locals-a : a , d ;
        locals-b : b ;
        a = src1 , b = src2 ;
! use or-xor lsb a b
        out = ( and-not b a ) , break = ( xor a b ) ;
        delay ;
! combine next two instructions
        d = ( and-not in lsb ) ; ! get bits minus lsb
        out = ( and d msb ) , break = lsb ;
        delay ;
        d = in , break = ( one ) ;
        diff = d ;
}
procedure less-than-short ( a , b , diff )
{
! user or-xor lsb a b
        out = ( and-not b a ) , break = ( xor a b ) ;
        delay ;
! combine next two instructions
        diff = ( and-not in lsb ) ; ! get bits minus lsb
        out = ( and diff msb ) , break = lsb ;
        delay ;  ! propagate difference
        diff = in , break = ( one ) ;
}

!a procedure which sends a one out if the bits are not equal,
!then inverts - (so a one indicates that things are okay)
!Assumes an orient-high-isolate

procedure equal ( src1 , src2 , result )
{
        locals-a : a ;
        locals-b : b ;
        a = src1 , b = src2 ;
        out = ( xor a b ) , break = lsb ;
        delay ;         !wait for broadcast to all processors
        result = ( not in ) ;
}
procedure equal-short ( a , b , result )
{
        out = ( xor a b ) , break = lsb ;
        delay ;         !wait for broadcast to all processors
        result = ( not in ) ;
}
;
```

A.3.8 Global-OR and Polarity Functions

```
!rebroadcast a result back over the net, in one of two flavors...
!in most applications, you will want to make source and dest the same
!location...
!This function makes <dest> have all ones if there are ANY ones
!in <src>.  If break is zero, orienting high or low should have the
!same effect. (i.e. it doesn't matter)
!You may orient the select registers orient-high or orient-low
!altered to handle all registers
procedure broadcast-low ( source , dest )
{
        locals-a : d ;
        break = ( zero ) , out = source ;
        delay ;
        d = ( or source in ) , break = ( one ) ;
        dest = d ;
}

procedure broadcast-high ( source , dest )
{
        locals-a : d ;
        break = ( zero ) , out = source ;
        delay ;
        d = ( or source in ) , break = ( one ) ;
        dest = d ;
}

! this procedure computes the global-OR function and propagates
! the or-out throughout the chip and reads back the or-in after
! full propagation

procedure global-or ( or-out , or-in )
{
        inselect0 = ( one ) , break = ( zero ) ;
        inselect1 = ( one ) ;
        out = or-out ;
        controlbits clear 2 ; ! enable OR output pad
        delay ;
        inselect1 = ( zero ) ;
        delay ;
        delay ;
        or-in = in ;
}
```

```
! this procedure takes the msb bit from a cluster and broadcasts it
! to all other processors in the cluster.  This lets everyone know
! what the sign bit for the current number is. The bit is loaded
! negated so that zero or a positive number yields all ones and
! negative number produces all zeros.
! assumes orient-high with isolation

procedure sign ( src , dst )
{
        locals-a : s ;
        s = src , break = ( zero ) ;
        out = ( and s msb ) ;
        delay ;
        dst = ( not in ) ;
}

;
```

A.3.9 General Gaussian Algorithm

```
! Gaussian computation program
! input must be in left register
! include all-lib.asm just to be sure
! should work with either config32x32 or config512x512

procedure gauss ( src , out )
{
        locals-a : dst ;
        locals-b : left , right ;
        dst = src ;
        comment horizontal shifting ;
        call ( orient-east ) ;
        call ( shift-cluster-east dst left ) ;
        call ( orient-west ) ;
        call ( shift-cluster-west dst right ) ;
        call ( triangle dst left right ) ;
        comment vertical shifting ;
        call ( orient-north ) ;
        call ( shift-cluster-north dst left ) ;
        call ( orient-south ) ;
        call ( shift-cluster-south dst right ) ;
        call ( triangle dst left right ) ;
        out = dst ;
}

procedure triangle ( a , lnum , rnum )
{
        comment performing triangle ;
        call ( orient-high ) ;
        call ( asl a a ) ;
        comment add1 ;
        call ( add lnum a a ) ;
        comment add2 ;
        call ( add rnum a a ) ;
        call ( orient-low ) ;
        call ( asr2 a a ) ;
}

;
```