SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

5235:TR:86

5 December 1986

Reporting Period:	16 March 1986 to 15 November 1986
Principal Investigator:	Charles L Seitz
Faculty Investigators:	James T Kajiya
	Alain J Martin
	Robert J McEliece
	Martin Rem
	Charles L Seitz

Sponsored by the
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-0597

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document is a summary of the research activities and results for the eight month period, 16 March 1986 to 15 November 1986, under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. Technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes. Our work is focused on VLSI architecture experiments that involve the design, construction, programming, and use of experimental message-passing concurrent computers, and includes related efforts in concurrent computation and VLSI design.

Additional background information can be found in previous semiannual technical reports [5052:TR:82], [5078:TR:83], [5103:TR:83], [5122:TR:84], [5160:TR:84], [5178:TR:85], [5202:TR:85], [5220:TR:86].

1.3 Highlights

Some highlights of the previous 8 months are:

- All sections of the Mosaic C designed and laid out (sections 2.2.3 and 4.1).
- Cantor programming language operational and in use (sections 2.2.2 and 3.1).
- Joint projects initiated with two companies to build prototype second generation "cubes" (section 2.3).
- New results in compiling multiprocess programs to self-timed VLSI designs (section 4.2).

2. Architecture Experiments

2.1 Cosmic Cube Project

W C Athas, Michael Lichter, Wen-King Su, Chuck Seitz

The Cosmic Cubes and Intel iPSC continue to run very reliably, with researchers at Caltech and at other DARPA sites using them for application programming projects. Usage has been moderately heavy.

Our own most intensive use has been for event-driven simulations of designs for the second generation message-passing multicomputers (see sections 2.3 and 3.4).

A system supporting Cantor, a fine grain concurrent object-oriented programming language, is now available and in regular use on the Cosmic Cubes and iPSC (see section 3.1).

2.1.1 Hardware and System Software Status

Neither the 64-node nor 8-node Cosmic Cubes has exhibited a hard failure in this 8-month period. The cubes have now logged 2.3 million node-hours with only 3 hard failures. The calculated MTBF of the nodes of 100,000 hours reported before these machines were constructed was conservative at above the 99% confidence level. A node MTBF in excess of 500,000 hours is probable, and can be stated now at a 70% confidence level. The systems will have to operate for another 2 years with a similar or smaller failure rate for us to be able to state a 500,000 hour MTBF with 90% confidence.

A 500,000 hour MTBF for hard failures in a node corresponds to an MTBF for a 64-node system of only approximately a year, which is hardly a problem in a system in which the mean time to repair is a few minutes. The reason for our interest in the *post facto* reliability experience with the Cosmic Cubes is to accumulate and share some experience that may be applicable to our own and other researchers's experimental prototypes. The three hard failures observed have been in chips, two in 64K dRAMs and one in a 74C-series line driver. There have been no connector failures. Systems with comparably complex nodes in current technology, and at a 1024-node scale, would exhibit an MTBF of about three weeks, which we regard as acceptable for a system of that scale and performance.

There have been no significant changes made in the cosmic cube system software in this period, except for the Intel iPSC compatibility package discussed below. There have been a large number of network problems reported by users outside of Caltech, but very few cube problems.

2.1.2 Intel Cube

Our Intel iPSC I d7 (128 nodes) is proving to be a valuable resource for the project. This machine was contributed to the Submicron Systems Architecture Project as a part of the license agreement between the Caltech and Intel, and are accessible via the ARPAnet to other DARPA researchers who may wish to experiment with them. To request an account, please contact `chuck@vlsi.caltech.edu`.

Intel has gone through several revisions of their hardware and software since we recieved our iPSC last year, with improvements being made at each step in reliability, speed, and usability.

Since this last summer we have been running as a user option a beta release of the R3.0 node kernel, and are now running the production release. This new kernel is written in C, and is accordingly much easier for us to experiment with than was the R2.0 kernel. It is also much faster. The largest component of the message-passing latency on the iPSC cubes is startup time in the node operating system, and it has been significantly reduced in R3.0. The iPSC nodes are for integer computations about 4 times faster than the cosmic cube nodes. The iPSC message latency for short messages is now about 1ms, about half that of the cosmic cubes, and asymtotically one fourth as large for long messages. The cosmic cube was faster than the iPSC with R2.0 for short messages, but the relationship between communications and computing performance in these two systems is now similar.

Computing problems run on the iPSC show performance up to about 100 times faster than a VAX11/750 when good load balance and message locality are achieved in the concurrent formulation.

The cosmic environment has been generalized to support space-sharing on the iPSC and compatibility with the cosmic cube message functions. Except for programs that use the Cosmic Cube's more elaborate process spawning mechanisms, the same programs can be compiled to run on either the Cosmic Cubes or iPSC. These new system features are described in the July 1986 edition of the "C Programmer's Guide to the Cosmic Cube" [5203:TR:85].

Wen-king Su is now writing a new version of the cosmic environment package based on the new "reactive" message primitives discussed in section 2.3. Michael Lichter will be doing the same with the cosmic kernel and/or the iPSC kernel.

2.1.3 Applications

Many new applications have been written to run on our cubes. The Cantor system runs on both types of cubes, and quite a few of the newest applications have been written in Cantor (see section 3.1).

We continue to make the cubes available to selected guest researchers for problems that appear to be particularly interesting or difficult.

Researchers in the Aeronautics department at Caltech, under the direction of Professors Brad Sturtevant and Tony Leonard, continue to report very good results on some difficult fluid mechanics computations. One of their most successful applications employs a Monte Carlo method for computing local interactions for molecular dynamics, rarified gas (such as jet plumes), and granular flow, with a non-uniform grid to balance the load. They have also gotten very good results in experiments with the same lattice gas automaton approach to fluid mechanics computations that has been developed for and applied to the Connection Machine.

Researchers at the University of Washington, under the direction of Professor Larry Snyder, have ported the Poker programming system to the cubes.

Researchers at USC/ISI, under the direction of Dr David Mizell, have been regular users of the cubes. Chuck Seitz, Bill Athas, and Wen-King Su gave a one-day cube programming class at ISI on 10 July 1986.

2.2 Mosaic Project

Bill Athas, Charles Flaig, Fritz Nordby, Steve Rabin, Steve Roskowski, Don Speck, Wen-King Su, Chuck Seitz

A large fraction of the project's resources and effort have been devoted to the Mosaic project over the past eight months, and a great deal has been accomplished.

2.2.1 Working Toward a 16K-node Mosaic C System

Mosaic is a message-passing MIMD multicomputer similar to the Cosmic Cubes, but with the stipulation that the node be integrated onto a single chip. We regard this node size as the most interesting design point for this architecture from the viewpoint of exploiting VLSI technology. The stipulation of single-chip nodes limits the storage for each node so that relatively fine grain concurrent programming techniques must be used.

We are working toward building a 16K-node Mosaic system using nodes fabricated in 1.2μm CMOS technology, with a near-term milestone of a 1K-node system with nodes fabricated in 2μm CMOS.

The 16K-node system will be built as a 3D routing mesh ($32 \times 32 \times 16$). Each node will include 16K bytes of storage, a 16-bit instruction processor, and channels that perform wormhole routing on the mesh. The mesh topology allows the nodes and their communication to operate in a locally bounded skew clocking scheme. Our circuit simulations show that we can achieve a clock rate of 40MHz for this design in 1.2μm CMOS, which translates in the microcycle simulations to an instruction rate of about 12 MIPS. The channels send 2 bits per clock period, corresponding to a channel rate of 10 MB/s. In aggregate, the 16K-node Mosaic will have 256MB of storage, a peak performance of 200,000 MIPS (6,000 Mflops in floating point subroutines), and a bilateral bisection communication bandwidth of 5,120 MB/s.

The 1K-node system is both a near-term milestone and a programming development system. It will be organized as a 32×32 2D routing mesh. We will include as much storage per node as the 2μm fabrication technology and the quality of our RAM designs allows, most likely about 6KB per node. Clock rate based on circuit simulations will be 20MHz.

We have calculated an MTBF for the 16K-node system of about 400 hours (three weeks). Although this MTBF is actually quite good for a system with such phenomenal performance, it is a source of some concern in that we estimate that the repair time for this 3D assembly will be an hour or so. It will be possible to use the working nodes of the system in spite of faulty nodes under the same "space-sharing" mode of operation that we have developed for the Cosmic Cubes.

The calculated MTBF of the 1K-node 2D system, essentially the same circuit and packaging technology, but with fewer package pins per chip, is nearly 10,000 hours. It is accordingy unlikely that we will observe enough failures in its operation to give us a better estimate for the 16K-node system. However, if we observe more than 2-3 failures in its first year of operation, we will know that we have a more serious problem in the 16K-node system than we have calculated.

2.2.2 Mosaic Programming Systems

The Cantor Model

The concurrent programming model for the Mosaic is a fine grain object-oriented model. Since the development of the Cantor programming system, based on this model, we are confident not only that we can program relatively fine grain systems such as the Mosaic without extraordinary efforts, but that these fine grain systems are efficient and surprisingly general. See section 3.1 for a discussion of Cantor.

The Cantor execution model fits message-passing ensembles very well, and has also dictated many features of the Mosaic C node architecture. Although Cantor is an "Actor" language, we do not always use Actor terminology to describe this model. Cantor execution deals with objects and messages. A Cantor object is normally at rest, and executes its program code, called its definition, in reaction to receiving messages. The object takes a finite time to complete a series of actions that may include sending messages, spawning new objects, and executing code with possible side-effects that change the object's persistent variables.

The message-driven or reactive character of object scheduling allows a very streamlined run-time system that must be resident in each node. This system manages incoming and outgoing message queues, and runs objects according to the contents of the incoming message queue. This system can be implemented in about 1000 bytes of Mosaic code. In order to be able to perform this message handling efficiently, the Mosaic C processor is a "two sequence" machine with two program counters and two overlapping register banks, so that it can process interrupts from the message system with zero context switching overhead.

Although the run-time system is assisted by the compiler in determining the placement of new objects, object placement is assumed to be little better than random. The performance of the message network with non-localized message traffic is accordingly very important, and is accomplished by the "routing automata" channels (see sections 4.1 and 4.2).

Typical Cantor programs that have been run on the Cosmic Cubes and iPSC have many thousands of objects. The typical size of an object is much less than 100 bytes in the object table to represent the persistent variables, and several hundred bytes of code for each object definition. Thus the storage size even of the nodes of the 1K-node Mosaic C is adequate to support Cantor programs. Messages are typically short, 20 bytes or less, so the message system is optimized for short messages.

Mosaic C Simulators

The Mosaic detail simulation environment, meant principally for system development, supports the simulation of Mosaic ensembles with arbitrary connectivity, and with elements that are loaded with variable microcode and bootstrap programs. There is a simulated host interface to the ensemble, and provisions for loading data into or unloading data from the ensemble. Debug commands allow one to monitor communications and to examine the state of each computer in the ensemble. The simulation is, to the best of our knowledge, absolutely accurate on a clock cycle by clock cycle basis, in order to be able to replicate precisely any behavior of a Mosaic ensemble.

The cost of this precision is performance. The current implementation of the Mosaic simulator executes roughly 50 simulated Mosaic cycles per VAX11/780 cpu-second. At this rate the Mosaic A v1.2 self-test and bootstrap program, which moves a march pattern through memory, requires roughly 6 VAX11/780 cpu-minutes multiplied by the number of machines in the ensemble being simulated. Fortunately, for self-test and bootstrap programs we are interested in ensembles of one element.

We have been working on providing a concurrent simulation environment for Mosaic ensembles, intended for application programming development, and able to run either on the cosmic cubes or Intel iPSC computers. Our goal, now that the Mosaic processors are so well characterized, is to perform macroinstruction rather than microinstruction level simulation, and be able to simulate full size (*e.g.*, 1024 node) ensembles on the 128-node iPSC at no worse than 1,000 times slower than a hardware ensemble.

This new Mosaic simulator has been written in C. It currently models the Mosaic B processor instruction set and Mosaic C communications network. The node architecture being simulated has the following attributes:

- Mosaic B instruction set (without channel addressing modes).

- Special memory locations control node initialization, interrupt control, and I/O, as in the Mosaic C. Channel I/O is done entirely by DMA operations.

The communications network simulated is the Mosaic C wormhole routing mesh.

The simulator is written to run on concurrent hardware. Each node simulates a submesh of Mosaics. The same distributed event-driven simulation algorithm reported in section 3.4 is used to schedule the node simulations in order to avoid simulating Mosaics that are in a busy-wait state. The simulator operates in several modes. At one extreme, *realistic* mode models communication hardware faithfully, including blocking in the network, whereas in *high-performance* mode message order is preserved but message delays are arbitrary.

The simulator is not yet availible for general use.

2.2.3 Mosaic C Chip Design

The continued poor yield experience with the nMOS Mosaic A and the rapid progress in the design and superior performance of the CMOS Mosaic C have persuaded us to drop the Mosaic A design in favor of the Mosaic C.

Some additional details of the Mosaic C chip design are reported in section 4.1. This section is only a summary.

The target technology for the Mosaic C is MOSIS SCMOS with $0.6\mu m \leq \lambda \leq 1.5\mu m$. Target maximum chip size is $36mm^2$, or $100M\lambda^2$ with $\lambda = 0.6\mu m$, and $16M\lambda^2$ with $\lambda = 1.5\mu m$. Speed, storage size, and top-level floorplan will necessarily vary with feature size.

The Mosaic C is composed of 3 main parts: RAM & ROM, channels, and processor.

- *RAM & ROM* – Of the preliminary designs investigated by layouts of critical parts and circuit simulations, 2T and 4T fast dynamic RAMs emerged as the most promising. The 2T RAM cell, essentially two 1T cells written to complementary values and sensed differentially, can be implemented in about $400\lambda^2$ per bit including decode and sense circuits, and with only one contact per bit in the RAM cell. As a backup, since the sense circuitry introduces some risk of having to go through several fabrication iterations on the 2T design, we first laid out a safer 4T RAM that requires about $600\lambda^2$ per bit. The ROM is straightforward and requires less than $100\lambda^2$ per bit.

- *Channels* – The channels section of the Mosaic C designed in the spring 1986 was abandoned because of its size and performance relative to a new organization based on "routing automata" (see section 4.3). Routing automata channels were laid out over the summer 1986. A unidirectional channel requires 4 wires, two data, one control escape, and one in the reverse direction for flow control. A 2D mesh requires two unidirectional channels for each bidirectional channel, and channels in each of 4 directions, for a total of 32 wires. Except for power, clock, reset, and LED drivers, all of the I/O from a Mosaic is through these channels. The channels convey 2 bits on each clock period. Layout size is about $3M\lambda^2$.

- *Processor* – The Mosaic C processor is a 16-bit machine similar to the Mosaic B processor in layout and microcode style, except that the channels are implemented separately. Also, the Mosaic C processor is a "two-sequence" machine: It has two program counters and two overlapping sets of registers to be able to switch instantly between task and channel processing contexts. In addition to the two program counters, the processor has 5 other addressing registers that share the same incrementer: refresh counter, channel receive pointer and limit register, channel send pointer and limit register. Also like the Mosaic A and B, the Mosaic C is controlled with a single large PLA, which in the current design is 96 implicants. Layout size is about $5M\lambda^2$.

The processor, channels, ROM, and peripheral circuits on the Mosaic C consume about $10M\lambda^2$. The amount of space left for storage depends strongly on feature size under our assumption of not wanting to push the die size above about $36mm^2$. The area remaining for RAM is about $6M\lambda^2$ or 1.5K bytes at 3μm feature size with the 2T RAM, $26M\lambda^2$ or 6K bytes at 2μm feature size, and up to $90M\lambda^2$ or 24K bytes at 1.2μm feature size. The amount of 4T that will fit is about 2/3 of these amounts. The 3μm version is not interesting for system building, but only for chip design verification. The 2μm version even with 4K bytes of memory is more than adequate for system building and programming experiments.

2.3 Second Generation Cosmic Cubes*

Chuck Seitz, Alain Martin, Bill Athas, Bill Dally†, Charles Flaig, Steve Rabin, Craig Steele, Wen-King Su

Over the past three years we have developed a number of new ideas for second generation medium grain size message passing systems based on commercial processor chips and RAM, low-latency communication devices such as the Torus Routing Chip (TRC) [5208:TR:86], [5209:TR:86], and a reactive programming model [5196:TR:85], [5209:TR:86], [5232:TR:86]. Over the past eight months these ideas have come together into a coherent design, and in September 1986 we started simultaneous joint projects with two companies, Intel and Ametek, to build prototype second generation message-passing multicomputer systems. The companies are both

* This segment of our research is sponsored jointly by DARPA and by grants from Intel Scientific Computers (Beaverton, Oregon) and Ametek Computer Research Division (Arcadia, California).

† At Caltech through June 1986; now Assistant Professor of EECS at MIT.

prepared to provide all necessary parts, assembly, and logistical support for constructing the prototypes to our specifications and designs, and to work with us in developing the system software.

Intel and Ametek have non-exclusive licenses to patents on the Cosmic Cube architecture and message passing mechanisms, several later patents, and non-exclusive resale licenses for the Cosmic Cube system software, and ongoing work at Caltech. Both companies are manufacturers of first generation message-passing multicomputers based on the Cosmic Cube, and are licensed to produce the second generation machines commercially. Because of proprietary concerns, the schedules and certain details of the designs are not included in this report, but have been reported elsewhere to the DARPA management both by our project and by the companies.

We are already well underway with these projects. Machines of 64 nodes will be contributed to the project, and will be on the ARPAnet just as our Cosmic Cubes and iPSC are currently.

Since the two machines come from the same basic architectural and system software design, they will be sufficiently similar that it will be possible to compile the same programs to run on either machine. In fact, because of the formulation of the low level message send and receive primitives, the user can provide a library of message handling routines that allows these machines to execute programs written for *any* multicomputer systems.

The two machines differ in their processors, hosts, what accelerators will become available, details of their software implementations, programming language support, and application packages.

2.3.1 Hardware Characteristics

The general hardware characteristics of these machines is that the node processors are about 10 times faster than the first generation machines, but the message performance for non-localized message traffic is about 1000 times faster. The design is tuned to minimize latency for short messages. The message network uses a low dimension graph rather than the binary n-cube, and wormhole routing. The machines can be scaled over a wide range of N, but a 256-node system is the "centerline" design point.

The nodes are an unsymmetrical two-processor design with a general-purpose instruction processor in the 5-10 MIPS range for user processes, and a microcode-driven message interface processor. Except for the message interface, the characteristics of a node are expected to be similar to those of an advanced workstation of the same period. Scalar floating point performance is in the 1-2 Mflop range, and can be extended by vector accelerators to the 10-20 Mflop range. Minimum memory size per node will be 1 MB, but expandable to 16 MB or more. We expect that most commercial machines would have 4 MB per node. Thus the "centerline" 256-node system would have a peak performance of almost 2,000 MIPS, 400 scalar

Mflops, and as much as 5 Gflops with vector accelerators and vectorization in the process code. Total storage size of the 256-node system with the typical 4 MB per node is 1 GB.

We have the encouragement of both companies to suggest standards that would not compromise either company's proprietary approaches, including:

- The format of the functions that invoke operating system services.
- The format of error reports and error recovery routines.
- Host functions for allocating machines and submachines in space sharing modes.
- "Open" hardware and software interfaces for accelerators and I/O.

The second generation machines will readily accommodate multiple hosts and disks.

Another of the new features we plan to include as an option in these machines is a distributed frame buffer. For example, a 16×16 node machine will be able to refresh a 1024×1024 pixel video image from 64×64 pixel frame buffers in each node. This distributed frame buffer can be used both to create some visibility into the activities of the whole machine – to "animate" algorithms in execution –, as well as to provide an extremely high performance graphics output from computations performed by the system.

2.3.2 Software

One of the aspects of the design that we can discuss fairly freely, as we hope that this user interface will become a standard for message-passing multicomputers, is the system software. The very small network component of the message latency makes it essential that we streamline the message handling in the nodes. The way in which this is accomplished is to use a "message-driven" or "reactive" node kernel.

All messages are sent and received from a heap maintained by the kernel, and accessed both by the main processor and the message interface processor. Messages are sent by allocating message space in the heap by:

```
p = xmalloc(len);
```

then sending the message built in that space by:

```
xsend(p,node,pid);
```

which also deallocates the space. That is, `xsend(p,...)` is like `free(p)` except that it also sends a message. Thus there is no need for the time-consuming "feedback," as for example via the message descriptor lock variable in the Cosmic Cube.

Messages are received by:

```
p = xrecv();
```

which is semantically like `malloc`, except that the length of the block allocated is determined by the length of the message received. Space thus allocated can later be freed after the message is used, either by the `xfree(p)` function, or by building another message in the space and freeing the space by the `xsend(p,...)` function.

Just as the `malloc` function in the Unix environment may return the NULL pointer if there is no space available, `xrecv` function will return the NULL pointer if no message is present. However, `xrecv` returns the NULL pointer only if there are no messages in the node's entire receive queue. Calling `xrecv` when the next queued message is for another object (process) causes the kernel to save the state of this object and start running the other object. Thus the appearance of `xrecv` in the code marks "choice points" for switching execution to another object, and it is in this sense that the scheduling is "message-driven." It is still possible for an object to do work while waiting for a message, as in:

```
if (p = xrecv()) digest(p);
else do_other_work();
```

So long as an object (process) is making progress, one does not force a context switch. The blocking receive is expressed as a busy-wait loop:

```
while ( !(p = xrecv()) );
```

which has no inefficiency, since if the NULL pointer is returned the node does not have anything else to do anyway.

Although this set of low-level primitives encourages object-oriented programming styles, it is easy to express any other message primitives in terms of these. For example, we have written a library of functions that exactly duplicates the Cosmic Cube message primitives, which are much more complex in including a type mechanism and allowing a process to exercise discretion in the messages it receives.

The results of this formulation of the message primitives are a very fast and simple node kernel that is also compatible with all existing "cube" programs.

A substantial part of the software necessary for the second generation Cosmic Cubes has been derived from the Cosmic Cube software. A new version of the cosmic environment based on these new primitives has been written and will be used as the host runtime system. We are working on a portable kernel that will need only to be adapted for each model of the new hardware. We are also writing libraries of the host message and process spawning functions for each of the older types of cubes.

These reactive primitives, although illustrated here as called from C processes, are essentially identical to those used in Cantor's intermediate code. Cantor will be particularly efficiently supported on these second generation cubes.

One of the advantages of these projects that use commercial microprocessors in the nodes is the ability to exploit software tools – compilers, interpreters, debuggers

– already developed for that particular microprocessor. Although these machines do not offer the ultimate in performance per cost, as a system such as Mosaic does, their large storage per node and extensive software tools result in the most highly evolved and advanced software development environments that will be available on message-passing concurrent computers. In supporting fine grain concurrency on these machines, we exploit the leverage in the software development environments in a way that will also be applicable to fine grain machines.

Taken together, the computing and communication performance, scalability, open interfaces, I/O capability, new features, and system software of these second generation machines represent to us fulfilling an "IOU" – a working demonstration of the capabilities we have said would be possible to include in a well engineered message-passing concurrent computer.

2.3.3 Other Multicomputers

We have developed under DARPA sponsorship a base design and set of portable system software into which one can drop other processors, such as fast RISC processors, and hosts. Our licenses with Ametek and Intel are non-exclusive, and it is possible that in the future that we may form joint efforts with other companies to develop multicomputers based on other processors.

3. Concurrent Computation

3.1 Cantor

Bill Athas, Chuck Seitz

The programming notation Cantor represents the confluence of our ideas about programming idioms and compilation techniques for fine-grain message-passing concurrent computers. The results of this work have been twofold. The reactive model for object-oriented programming first presented in the XCPL report [5196:TR:85] and subsequently refined in the Cantor experiment, has been cast in the well-known programming language C to be used as the basis for the next generation of message driven kernels for the "Cube" machines. The Cosmic Kernel and its ilk are easily expressed in terms of the reactive or message-driven kernel.

Secondly, Cantor exists as a programming system comprised of a compiler, code generator, and interpreter for a variety of computers, both sequential and concurrent. Cantor is a lexically scoped, dynamically typed, object-oriented programming notation. The objects of Cantor share data and synchronize exclusively by message passing. The delivery of messages between objects and the assignment of objects to nodes is handled jointly by the Cantor compiler and the interpreter. The programmer does not need to know the topology or number of nodes in the computer on which a Cantor program is executed.

The Cantor compiler is a recursive-descent compiler written in C, and is portable to any machine that hosts a C compiler. The code generator is also written in C and is a table-driven program. The interpreters exist in a variety of forms. For concurrent computers, the interpreter is partitioned into two parts, the node part and the host part. Both parts run under the auspices of the cosmic environment. Each node of the Intel iPSC or the Cosmic Cube hosts a node interpreter as one process. The loading and initialization of the the node interpreters is handled by the host interpreter, which is also a process. Managing I/O between the cube and user is also handled by the host interpreter.

For sequential computers there is a version of the interpreter for debugging purposes. There is also an interpreter designed to be coupled to the statistical analysis tool S. This interpreter is used for profiling Cantor programs under ideal conditions, to measure quantities such as the amount of concurrency present, the amount of message traffic present, the average proximity between objects, etc. These quantities are measured as a function of discrete time quanta called sweeps. Although the ideal conditions are most likely not representative of actual conditions, if a program performs poorly under profiling, then almost assuredly, it will perform poorly on an actual concurrent machine. The profiler is also useful for evaluating different strategies for assigning objects to processors.

A variety of applications programs have been written in Cantor including: a Gaussian elimination program, an N-Queens program, two prime sieves, a circuit simulator, a finite field package, the game of Life, optimal ordering for matrix multiplication (dynamic programming), and Floyd's all points shortest path computation.

Plans for Cantor include a native code generator for a variety of 32-bit commercial microprocessors as well as generating native code for Mosaic C. A flow analysis program for Cantor to perform future flow, as described in our previous semiannual technical report [5220:TR:86], is also in preparation. Currently the compiler is able to "assist" the interpreter in placing objects by supplying pertinent information about the behavior of new objects to the interpreter. This information includes the maximum number of new objects the new object could in turn potentially create, called the New Factor (NF) , and the number of messages the new object could potentially send, called the Send Factor (SF). Both the Send and New Factor are measured on a per single message received basis. Simple heuristics based upon SF and NF can appreciably increase the locality between communication objects with only negligible impact on the load balancing.

The next step is to have the compiler "direct" placement rather than merely assist. The problem of future flow is an instance of value or constant propagation used by optimizing compilers. If the types of values used in a program are constant, then the result is type inferencing. Type inferencing is useful in Cantor since values can be type checked once at compile time, thus removing unnecessary type checking at runtime. If the values used in a program are constant, then any expression consisting completely of constants can be evaluated at compile time and replaced by a constant.

Expressions in Cantor can evaluate to new objects, which are called "futures". Propagating reference values at compile time is future flow.

3.2 The Sync Model for Parallel Execution of Logic Programming

Pey-yun Peggy Li, Alain J. Martin

The Sync Model is a multiple solution data driven model which realizes AND-parallelism and OR-parallelism in a logic program assuming a message-passing multicomputer system. Our initial studies of this model are now complete and reported in "A Parallel Execution Model for Logic Programming" [5227:TR:86].

A paper about the Sync Model is included as an appendix to this report.

3.3 Concurrent Simulated Annealing

Craig S. Steele, Chuck Seitz

The process placement optimization system previously reported is being implemented in a concurrent version for the Intel iPSC. A preliminary sequential version of the event-driven programming environment is presently running. Current work is extending the system to support concurrency and data distributed among several processing nodes.

A simulation of the concurrent optimization has demonstrated remarkable insensitivity to database inconsistency. The data upon which process relocation decisions are based are local to the node upon which the process resides. The local record of the positions of distant but communicating processes may become "stale" as those processes move. Concurrent optimization coupled with communications system latency induces some error in cost calculations. Another deleterious effect of concurrency is the possibility of conflicting simultaneous attempts to move a particular process to other nodes. For a given size task, both problems are accentuated by increasing the number of nodes concurrently attempting optimization. With increasing concurrency, some threshold will be reached where the errors arising from data staleness and the inefficiency due to aborted conflicting moves will preclude any optimization.

Both effects have been simulated sequentially by delaying the actual application of process relocation decisions for some time after the decision is made. By the time the action is taken, the environment may have changed significantly due to previously queued relocations. Typical problems do not show significant deterioration in convergence rate until the simulated concurrency exceeds one optimizing node per process being optimized. Since any real application must have at least one process per node to fully utilize the hardware, this result is very favorable to the development of a concurrent programming environment which dynamically optimizes its own placement at runtime.

3.4 Object-oriented event-driven simulation

Wen-King Su, Chuck Seitz

Since March 1986, a canonical model of event driven simulation has been formulated. The model includes a set of reactive processes that represent the elements in a simulation, and a set of messages that represent the influences exerted by the elements. Though it remains to be proven that all event driven simulators can be derived from this model, we have derived all the simulators that we have studied so far, including those based on the fundamental algorithm developed by Bryant, and independently by Chandy and Misra, as well as the "time warp" scheme developed by Jefferson.

The condition for ensuring progress of the canonical simulator has been proven. Though specialized simulators may have a more relaxed progress condition, satisfying the progress condition of the canonical simulator is sufficient to ensure the progress of the simulator.

The model has a intimate relationship with object-oriented reactive programming. It was this study and the study done by Bill Athas on the Cantor language that prompted us to re-think our concurrent programming environment, and which resulted in the message primitives described in section 2.3.2. Amongst other advantages, this formulation of the primitives allows an indefinite lazy evaluation strategy that still assures progress, typically expressed as:

```
if (p = xrecv()) simulate_and_optionally_send_outputs(p);
else send_accumulated_outputs();
```

A simulation package originally written to evaluate the performance of alternative network topologies and routing schemes for the second generation cubes is developing into a more general package for a broad class of discrete event simulations.

3.5 Adaptive Routing in Fine Grain Multicomputers

John Y. Ngai, Chuck Seitz

Our research continues the investigation of system architectures to support general purpose computing in massively concurrent machines. As was pointed out in previous reports, for progressively finer grain message passing concurrent machines, the performance of the underlying communication network becomes increasingly significant in determining the overall performance of the machines. In the extreme, the sending and receiving of messages *is* the computation, while the results of the computation are the corresponding *side effects* of message passing.

Our current investigation focuses in experimenting with strategies that enhance and sustain the performance of communication networks in moderate to heavy message loading situations. The routing networks used in the Mosaic and second generation cubes have now reached a limit of being as fast as possible for a given bisection while using deterministic (oblivious) routing. Any improvements on these networks will require an *adaptive* utilization of available network bandwidth to prevent buildup of local congestion (hot spots). Message routes are no longer deterministic, but are continuously perturbed by local message loading. Messages will tend to follow their optimal routes to destinations in light traffic loading but adaptively detour to longer but less loaded routes as local hot spots spring up. This approach is in contrast to *randomized* approaches suggested in the literature, and which completely destroy any message locality present in the original process placements.

Current work involves :

- The derivation of an analytic framework for the understanding of these adaptive routing strategies.

- Investigation of several related variants such as *cut-through* and adaptive *fragmentation* schemes.

- Measuring performance statistics through extensive simulation.

- Derivation and simplification of automata for the hardware implementation of these schemes.

In the future, we are also interested in investigating adaptive routing strategies to support efficent message delivery in arbitrary networks. These schemes would then help our understanding in devising routing schemes to handle message forwarding in faulty networks.

4. VLSI Design

4.1 Mosaic Elements

Bill Athas, Charles Flaig, Fritz Nordby, Steve Rabin, Steve Roskowski, Don Speck, Wen-King Su, Chuck Seitz

The overall organization of the Mosaic C chip was described in section 2.2. Here we fill in some details on our final attempts with the Mosaic A chips, and the various sections of the Mosaic C. The parts of the Mosaic C are being switch simulated and fabricated separately for design verification before assembling them into a single chip.

4.1.1 Mosaic A

The yield and voltage sensitivity problems with the nMOS Mosaic A reported in our last report still remain. We've determined that the voltage sensitivity occurs even on chips without RAM, and is specific to the amplitude of one clock phase. That phase is used in few places outside of the RAM, none of which should be intolerant of large variations in clock voltage swings. With MOSIS phasing out nMOS, and with the Mosaic C design being far along and a better system chip, it has become a moot point.

4.1.2 Mosaic C RAM

The RAM represents about 70% of the area of the 2μm version of the Mosaic C, and about 85% of the area of the 1.2μm version. The RAM is also determines the speed of the chip. Thus we have made the most concerted effort to minimize size and delay in the RAM.

The basic strategy is to develop a 4T dRAM that is a low risk design with a relatively large area, and a 2T dRAM that is a higher risk design but a relatively small area.

Of the 3 dRAM designs attempted by the VLSI class last year, the 4T design was the easiest, and gave us some usable layout, primarily the data bus interface. Addition of addressing logic and the test interface completed the layout. Verification began with visual inspection, leading to correction of myriad electrical problems, such as lack of substrate contacts, and progressing to MOSSIM, with the 35,000-transistor size exposing weaknesses in our tool suite at every step. It is nearly ready for first fabrication. Because it was designed by committee, the area on this version is larger than we predicted, about $800\lambda^2$ per bit.

Meanwhile, attempts to locate the voltage-swing sensitivity in Mosaic A pointed out a capacitive coupling problem in 3T dRAM cells that was exacerbated in double-metal CMOS. Reducing the coupling in the CMOS design entailed a large area increase. This destroyed the appeal of 3T dRAM, so we've abandoned that design. It had other disadvantages, such as tiny stored charges, large amounts of peripheral circuitry, and timing differences from the 4T and 2T designs.

This leaves the 2T design as our only alternative for denser storage, which is convenient because the 2T dRAM is nearly compatible with the 4T dRAM, the main difference being the need for sense amplifiers. The part that gives us the most trouble is generating a sense ramp that must maintain the right rate and shape over a reasonably wide range of supply voltage. Our current design for the sense ramp generators works over only a narrow range, so must be improved.

The 2T dRAM is a variation of the popular 1T dRAM that we do not think we could build reliably in a logic process. The 2T dRAM cell is just two 1T cells with a differential pair of storage capacitors per bit, eliminating the need for a dummy cell and providing a more reliable 0.2 volt difference between the bit lines. It also lends itself to folded bit lines, resulting in easier I/O and a convenient layout pitch. Thus while the 2T dRAM is not the ultimate in density, at $\approx 450\lambda^2$ per bit, it is much easier to make work in a logic process than the 1T dRAM.

4.1.3 Channels

After abandoning the router design for the Mosaic C which was generated by the VLSI design class, work was started on an improved version based on routing automata in which the header represents offsets in a prefix encoding that spans multiple flits. This approach lowers the latency in each node, and also the area of the routing circuits.

The new channels design is synchronous, with a minimum of 4 wires in each direction between nodes. Two wires are used for data, one for control, and one for an acknowledge signal. The control wire is used to specify the data lines as containing a digit, or encoded signals for a null flit, the tail of a message, or turn instructions. The number of data bits per flit, and the number of dimensions of the mesh, can easily be expanded.

Each dimension of the router is made up of three parallel data paths, one for data from the + direction, one for data from the - direction, and a cross-dimension path for data from the previous dimension. Each data path has a controller and feeds into a switch matrix that can switch data flow to any of the matching output data paths. If the first flit is a turn control flit, a request is generated to switch data flow to one of the +/- output paths or to the path leading to the next higher dimension. In the +/- data paths, if the first flit is a digit it is decremented and a request is generated to switch data flow to the opposite output path. Input data continues to flow through these connections until the arrival of a tail flit, at which time the connection is closed.

The subcells of a complete router have been laid out and individually tested. They have also been composed into a minimum width 1-dimensional router in preparation for a fabrication test run. This composed layout is currently being checked for errors and will proceed to fabrication after a it passes simulation.

The channels will be connected to the rest of the Mosaic C via a pair of simple finite state machines that make use of a simple cycle-stealing DMA scheme to keep up with the high data transfer rate of the channels. The first of these FSMs accepts 16-bit words from memory, converts them into the flit format required by the channels, and feeds them into the cross-dimension routing path of the first dimension of the router. The second FSM accepts the flits which emerge from the last dimension of the router and converts them into 16-bit words for transfer to memory. These FSMs have been specified but not yet designed.

4.1.4 Microcode

The microcode for Mosaic C implements an instruction set closely resembling the Mosaic B instruction set, however, the Mosaic C microcode supports the two-sequence processor architecture. The Mosaic C microcode also permits memory-mapped "devices" to access the address generation logic section of the the Mosaic C datapath during microcycles where the microcontroller is unable to use the storage cycle. The memory-mapped devices include the refresh counter and the channels interface.

The data registers of Mosaic C are split between 8 data registers that are common to both the system and user context, and a set of 8 data registers private to each context. The address registers of Mosaic C, contained in the address generation logic section, includes a program counter for each context (PC0 and PC1), a refresh address register (RA), a send message pointer (SMP) and a receive message pointer (RMP), and a send message end (SME) and a receive message end (RME) . The register pairs, (SMP, SME) and (RMP, RME) are specific to the channels interface. These registers are accessed from the instruction set by using a new source and destination mode for the move instructions called CONTROL.

The register pairs (SMP, SME) and (RMP, RME) are loaded under program control but incremented and tested inside of the address generation logic section under channel interface control. When the microcode is unable to use a storage cycle, control of the address generation logic section and the address bus is released to one of the memory-mapped devices. The device, *eg,* the channels interface, can perform a memory read or write, increment a pointer, and compare the new pointer value to one of the message end registers. If the end of a message has been released, an interrupt is generated to place the instruction processor into system context.

This scheme solves the problem of arbitrating the use of the address bus and allows the refresh counter and channels interface to share the program counter incrementer.

An instruction-level simulator for the Mosaic C microcode and datapath has been written. This simulator will soon be used to program the message-driven kernel for Mosaic.

4.1.5 Datapath

Major pieces of the of the Mosaic C datapath were laid out as a VLSI design class project before June. Since then, the datapath has gone through several redesigns. The address generation part of the data channels has been moved into the datapath as a set of DMA control registers. Arbitration of memory access is simplified because the datapath is the only source of memory address. Bus arbitration has been moved into the microcode engine which will now handle requests from the channels, the processor, and the refresh counter.

To accomodate this change, the address section of the datapath has been expanded and generalized. It now includes two pairs of start/stop address registers for the channels, one for the output DMA, and one for the input DMA. A read request from the channel causes the microcode engine to interleave a memory cycle at a point in microcode execution when memory access is not required, then fetch a word from memory and stuff it into the channel, and update the associated address registers. A write request behaves similarly, but would move a word from channel to memory instead. Since the PC is not being used at the time DMA occurs, the incrementer for PC is used to increment the start address register. In order to improve the robustness of the first silicon, efforts are made to recycle similar pieces through out the datapath at a small expense of area.

All but a few "glue" pieces of this new datapath has been completed and assembled together, and have been shown to be correct with the help of MOSSIM.

4.2 Compiling Programs into Self-Timed Circuits

Steve Burns, Peter Hazewindus, Alain J. Martin

Our main task during the past half year has been to experiment with the method we have developed for compiling CSP-like programs into self-timed VLSI circuits. The collection of algorithms that we have compiled into self-timed circuits now covers the whole spectrum of applications that one could possibly envisage for the method. No fundamental flaw or limitation of the method has been uncovered by these experiments.

However, before addressing the issue of automatic compilation, *ie,* before embarking on writing a compiler, we still have to solve two main problems. The first and most difficult one is related to the introduction of so-called "state variables." In all but very simple programs, the variables provided by the original CSP program and by the handshaking expansions of communication actions are not sufficient to identify certain states of the system. In such cases, extra variables called state variables have to be introduced. So far, the technique we have been using for the introduction of state variables still relies on the insight of the designer and may require a certain amount of backtracking. We are now looking for more systematic algorithms for the introduction of those variables. As the reader suspects, this problem is strongly related to the state assignment problem in the theory of sequential machines.

The second problem is related to the last phase of the compilation *ie,* the identification of the program with gates and the implementation of the gates in VLSI. So far the method works as follows: once the program has been compiled into a "production rule set," we try to identify sets – mostly pairs – of production rules with the sets of production rules that define our basic repertoire of gates. When the set of production rules that controle a certain variable cannot be directly identified with the rules of any gate, we have to further massage the set of rules to make it fit. In particular, this massaging may consist of decomposing the rules into more elementary ones, by introducing extra variables. This is a step that we want to eliminate in an automatic compilation because it involves backtracking. At the moment, we think that we can easily eliminate this step if we don't restrict ourselves to a fixed set of gates. After all, one of the main advantages of VLSI is that we can create new operators at no cost. We can therefore take any set of production rules that control a certain set of output variables and implement it directly rather than in terms of more elementary gates.

The above refinement will require that we also refine our assembly and layout programs. So far the chips that we have fabricated have been assembled and laid out using a "home-made" assembler, designed by students as a project in Chuck Seitz's VLSI class. This assembler uses a standard cell library of self-timed elements in SCMOS technology. We have also started experimenting with using the MOSIS 'fusion' service in cooperation with Ron Ayres at ISI. In both cases, the refinement mentioned above will require from the assembler that it is able to assemble generic gates, that we call 'generalized C-elements'.

Once these two problems have been solved, the construction of an automatic compiler can be confidently considered.

4.3 Routing Automata

Chuck Seitz, Alain J. Martin

The switching elements of the brain, neurons, have large numbers of input and output connections, but employ simple signaling. VLSI switching elements, by contrast, tend to be limited in "fan-in" and "fan-out", but can interpret complex signals. If one decides to build complex VLSI networks in a way that exploits their strong point, one is led to the idea of a set of switching elements that we refer to as "routing automata."

A typical routing automaton accepts serial streams with prefix headers of high information density, such as the packets in message-passing concurrent computers, and produces streams that can be interpreted in the same way. For the purposes of describing the way in which the message handling and routing circuits for message-passing concurrent computers can be developed from routing automata, let us illustrate this organizational approach with the same 2-dimensional bidirectional mesh network that we use in the Mosaic. The mesh network is acyclic in each direction and dimension, so a simple dimension order routing function – first x then y – will suffice to provide deadlock-free routing without recourse to virtual channels. However, one can employ the same organizational approach for k-ary n-cube (torus) networks, in which there is a cycle in each dimension, by using two virtual channels per physical channel.

We can assume for the time being that the packet consists of a sequence of flow control units, called *flits*, in which one can specify Δx in the first flit, Δy in the second flit, followed by message flits, the last of which is marked at the *tail* flit. Because this network is bidirectional, the Δx and Δy specification are signed numbers.

Because the routing proceeds first in x and then in y, one can compose a 2D routing chip out of two cascaded 1D routing automata. The first 1D routing automaton accepts a packet from its node computer. If the Δx read in the first flit is non-zero, it decrements the value and sends it in the $+x$ or -x direction according to the sign. If the header value is 0, the Δx part of the header is stripped and the remainder of the message is passed along to the y routing automaton, which operates identically. A message entering the x routing automaton at, for example, the $+x$ input is processed similarly. If the Δx value in the first flit is 0, the Δx flit is stripped and the remainer of the message is passed to the y routing automaton. If the Δx value is non-zero, it is decremented as the message is sent out the $+x$ output.

This structure would be nothing more than a nice organization for making routing chips but for the observation that the 1D routing automaton can in turn be composed of more elementary routing automata. One can build the 1D routing automaton out of two types of elementary routing automata that we refer to as decision (D) and merge (M). The D elements are typically required in several forms. The D and M elements are duals that can be thought of as similar to railroad switches. The M element is 2-input, 1-output, and passes entire packets from its inputs to outputs on a first-come first-serve basis. The flits of a packet can be thought of like coupled train cars, and only after the tail flit (caboose) of a packet has cleared the M element can another full packet be admitted from an input. The D element is 1-input, 2-output, and the most essential type for building routing networks processes the first flit of packet header as described above, stripping the flit and sending the packet to the side if the first flit is 0, and decrementing the first flit and passing the packet straight if the first flit is non-zero. A D_s (sign decision) is another 1-input 2-output elementary routing automaton that simply sends its input message to one of its two outputs according to the sign of the relative address in the first flit, and is needed to make routers for bidirectional networks.

The implementation of the elementary routing automata can be either synchronous or self-timed. There appears to be no limit to the variety of routing structures – including structures with properties of fault-tolerance, freedom from deadlock, and ability to perform message broadcast – that can be implemented as systems composed of elementary routing automata, each of which has only a few inputs and outputs.

One of the interesting properties of these networks is that the complex signals *at every point* have an identical sequential interpretation.

In addition to variations in the internal structure of these routing systems, one can employ different conventions for the sequential interpretation of signals. For example, in multicomputer communication networks, one might use different formats for header information.

The form of header with Δx in the first flit and Δy in the second flit requires that the number of bits communicated in a single flit be adequate to represent the maximal address difference between nodes in each dimension. Suppose one had a relatively large network, such as a 64 by 64 mesh of 4096 nodes, in which the communication bandwidth requirements between nodes required only 2 data bits to be sent with each flit. However, to be able to address any node it would be necessary to employ a flit of $\log_2 64 = 6$ bits for Δx or Δy, plus a sign bit. However, one may extend the representation of Δx and Δy (and additional dimensions) over several flits with the following technique.

Consider the case of 2 data bits, D0 and D1. These two bits are able to represent a radix-4 integer. An additional signal is required to distinguish "punctuation" symbols in the message stream, including the tail. Thus a minimum of 3 signals is required, for which we shall use the following of many possible encodings:

```
C  D0  D1  meaning
------------------
0   0   0  blank  (represented as ".")
0   0   1  plus   (represented as "+")
0   1   0  minus  (represented as "-")
0   1   1  unused
1   0   0  data 0
1   0   1  data 1
1   1   0  data 2
1   1   1  data 3
```

In addition, it is necessary to have one flow control signal communicated in the reverse direction of the channel in order to force transmission of data to pause in the event that the head of the message has become blocked.

A message that is to be conveyed by +5 in the x dimension and -2 in the y dimension is then encoded with the following string of characters from this alphabet:

```
...+0123+2-11+...
```

The message should be visualized as traveling from left to right, with its header on the right. It is surrounded by blank characters, and could also have blank characters intermixed inside the message as may occur for timing reasons.

The initial "+" indicates that the message is to travel in the $+x$ direction. The following "11" is the radix-4 representation for 5, as the message is to traverse 5 channels in the $+x$ direction. The following "-" indicates that the message is to travel next in the -y direction. The following "2" is the distance in the -y direction, in radix-4 notation. The following "+" indicates the beginning of the data part of the message. The following "0123" is radix-4 data corresponding to the 8-bit binary message 00011011. The following "+" indicates the tail of the message.

What is remarkable about this encoding is that by processing overlapping pairs of symbols from this alphabet, a routing automaton is both able to decrement the remaining distance and detect when the remaining distance has reached 0. For example, when this packet enters the x routing automaton, it is sent out the $+x$ channel, and has the following representation on successive channels:

```
...+0123+2-11+...   entering 1st router
...+0123+2-10...    leaving +x output of 1st router
...+0123+2-3...     leaving +x output of 2nd router
...+0123+2-2...     leaving +x output of 3rd router
...+0123+2-1...     leaving +x output of 4th router
...+0123+2-0...     leaving +x output of 5th router
...+0123+1...       leaving -y output of 6th router
...+0123+0...       leaving -y output of 7th router
...+0123...         entering node at 8th router
```

There are many variations on this basic scheme, one of which is used in the Mosaic C routing system.

We are at the stage in this work where we have interesting examples of routing systems constructed from routing automata, and a deep interest from our architecture experiments in developing more advanced routing systems. What is needed is a theory of and systematic methods for transforming routing specifications into routing structures. We believe that, in addition to their application to the communication networks of message-passing concurrent computers, this framework is suggestive of new approaches to the design of VLSI chips with multiple pipelines.

4.4 Fast Mesh Routing Chips

Charles Flaig, Chuck Seitz

Work has started on a next generation of self-timed routing chips to provide a message system for message-passing concurrent computers whose nodes are based on commercially available processors and memory.

The internal structure will consist of many stages of ultra-fast self timed FIFO pipeline, interspersed with slightly slower switching and pipelined header processing circuits, which operate in the manner of the Mosaic C router (section 4.1.3) and routing automata (section 4.3). The largest delay will probably be encountered in synchronizing data at the outputs, driving the pads, and waiting for an acknowledge signal. The internal pipelining will help relieve congestion in the system without adding an overly large amount of latency. It is expected that the chip will have a flit rate at least twice that of the synchronous router in the Mosaic C (which uses the same clock as the rest of the chip) and will also have the advantage of not being sensitive to clock distribution.

The routing chip will be optimized for an 8-bit wide data path, which requires at the interface 3 more lines for control, request, and acknowledge, for a total of 11 wires going in each direction between nodes. The low cycle time and wide data path will require a very fast buffering system between the routing chip and the connected processor in order to avoid slowing down the system.

4.5 SCMOS Pad Designs

Fritz Nordby, Chuck Seitz

As chip geometry scales down, the requirements of pad structures change. In chips with small feature sizes, the sizes of the output drivers decrease until they are small compared to the input protection structures. Because of this, output pads, traditionally protected by the sheer size of their associated driver transistors, are found to be in as much danger from input transients as the circuits connected to input pads, and the amount of chip area devoted to input protection structures increases to the point of near-absurdity.

Investigations of protection structures to attack these problems of chip area and output protection have yielded a new strategy for input and output protection, and which promise to allow wider and more varied on-chip use of input signals. This new strategy is based on the idea that static protection cannot be fully realized on-chip: the magnitude of the problem is simply too large to allow it to be handled without devoting a large area to protection structures.

Thus, rather than use an input resistor to *dissipate* the energy of the transient, a pair of large clamping transistors is placed adjacent to the pad so that any input transient is clamped directly to the power rail and its energy is *kept out* of the chip. This clamping structure is used on input and output pads alike, thus providing more protection for the outputs than would be provided by their drivers alone without significantly increasing the size of the output pad or interfering with the action of the output drivers. Inputs are further protected by a resistor and a second, smaller, clamp structure. This allows the pair of input clamps to act as a current divider, and greatly decreases the size of any transient; and this decrease in transient size allows inputs to be connected to diffusions internal to the chip, since the size of any transient that gets past the input structure can be kept below the 0.6 volts turn-on potential of the diffusion-substrate or diffusion-well diode.

These new pad structures are presently being layed out, and will soon be fabricated and tested.

4.6 Polygon Rendering Chip

Rajiv Gupta, Al Barr, Chuck Seitz

This chip is designed to render polygons at speeds of 8M polygons per second. The rendering process consists of z-depth buffering and Gouraud shading of polygons on a raster scan display. The future version of the design will also permit us to perform Bit-blt (creating copies of arbitrary portions of the screen with one command) operations.

The chips will have 18 pins with a die size of approximately 36mm^2. The chip is organized with on-chip memory to reduce communication bottlenecks, and 24 processors per chip, each handling a 20 pixel sliver of the screen. As is apparent, a large number of such chips would be required to render large screens. Thus with minimal additional support circuitry we will be able to sub-pixel – the only way of mimicking anti-aliasing in z-depth buffer systems.

We decided to partition the processor into 6 parts to ease in testing. Of these, 4 have been fabricated in 3μm SCMOS and have passed tests at 20 Mhz. The others along with the pre-processor and the post-processor are in the process of fabrication and testing.