SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

5240:TR:87

2 April 1987

Reporting Period: 16 November 1986 – 31 March 1987

Principal Investigator: Charles L Seitz

Faculty Investigators: 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 $4\frac{1}{2}$ month period, 16 November 1986 to 31 March 1987, 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], [5235:TR:86].

1.3 Highlights

Some highlights of the previous $4\frac{1}{2}$ months are:

- Joint projects with two companies to build prototype second generation "cubes" (section 2.3).
- All sections of the Mosaic C designed and laid out, and first parts returned from fabrication (section 4.1).
- Prototypes of software for 2nd generation cubes complete for the host environment (section 3.2) and nearing completion for the node operating system (section 3.3).
- Fully automatic compilation of multiprocess programs to self-timed VLSI designs (section 4.2).
- A modular design for self-timed routing chips (section 4.5).

2. Architecture Experiments

2.1 Cosmic Cube Project

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

This section summarizes the current usage and the hardware and software status of the Cosmic Cubes and Intel iPSC. Research developments are described in other sections of this report (see sections 3.2, 3.3).

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. David Sosha of the University of Washington visited in January 1987 and brought up the Poker programming system on the cubes. David Mizell and his colleagues at USC/ISI have been particularly heavy users. Overall usage has been moderately heavy.

Neither the 64-node nor 8-node Cosmic Cubes has exhibited a hard failure in this $4\frac{1}{2}$-month period. The cubes have now logged over 2.5 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 75% confidence level.

A new version of the host runtime system for the Cosmic Cubes and iPSC, called the cosmic environment, has now replaced the v6 system that had served us well for the previous two years. The primitives of the v7 cosmic environment are the `xsend` and `xrecv` functions (see section 3.2) of the second generation "cubes" currently in development (see section 2.3). The entire set of Cosmic Cube and iPSC primitives are implemented as library functions that call the "x" primitives. Existing user programs are recompiled to run under the v7 system, and no incompatibilities have been reported to date.

A new operating system that implements wormhole routing on the cosmic cube's program-controlled channels and the same message primitives as the second-generation "cubes" is in development. A skeleton of this "Reactive Kernel" runs now on the cosmic cube, and will replace the "Cosmic Kernel" as soon as a full complement of system services is added (see section 3.3).

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

2.2 Mosaic Project

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

This section is a summary of the architecture of the Mosaic C, and our plan for developing a 16K-node Mosaic C system. The status of the Mosaic C chip design is described in section 4.1, and current work on the Cantor programming system that we shall use for programming the Mosaic is described in section 3.1.

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 up to 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 channels 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 microcycle simulations to an instruction rate for a single node of about 12 MIPS. The channels send 4 bits per clock period, corresponding to a channel rate of 20 MB/s. In aggregate, the 16K-node Mosaic will have up to 256MB of storage, a peak performance of 200,000 MIPS (6,000 Mflops in floating point subroutines), and a bilateral bisection communication bandwidth of 10,240 MB/s. The significance of the bisection bandwidth is that a 16K-node Mosaic C is able to perform an arbitrary data rearrangement in $256/10240 = 25$ms.

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 4KB per node. Clock rate based on circuit simulations will be 20MHz.

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 a routing section in the Mosaic C chip.

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 of even 4K byes is adequate to support Cantor programs. Messages are typically short, 20 bytes or less, so the message system is optimized for short messages.

2.3 Second Generation Cosmic Cubes*

Chuck Seitz, Alain Martin, Bill Athas, 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]. In September 1986 we started simultaneous joint projects with two companies, Intel Scientific Computers and Ametek

* 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).

Computer Research Division, to build prototype second generation message-passing multicomputer systems. The companies are providing all necessary parts, assembly, and logistical support for constructing the prototypes to our specifications and designs, and are working 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 ongoing work at Caltech. They also have non-exclusive resale licenses for the Cosmic Cube system software. 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. 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.

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.

We are already well underway with these projects. However, because of proprietary concerns, the schedules and certain details of the designs, such as the processors used, are not included in this report, but have been reported separately to the DARPA management.

The "reactive" message primitives used in these systems are described in section 3.2. As mentioned in section 3.2, the latest version (v7) of the cosmic environment is a prototype of the host runtime system for the second generation "cubes," and implements these primitives directly.

A prototype of the node operating system for the second generation "cubes," the Reactive Kernel (RK), is being developed and tested using the original Cosmic Cubes (see section 3.3).

3. Concurrent Computation

Our research efforts in concurrent computation over the past $4\frac{1}{2}$ months have been concentrated on the Cantor programming system, the system software for the second generation cosmic cubes, and more advanced versions of our object-oriented event-driven simulators.

3.1 Cantor

Bill Athas, Chuck Seitz

Cantor, a fine grain concurrent object-oriented programming language, has been updated to version 2.0. A revised and improved *Cantor User Report* [5232:TR:86] was released in January, 1987. The user report along with the Cantor compiler and interpreters are now available as a package to non-commercial organizations outside of Caltech.

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.

Because of the deliberate omission of control structures for iteration within objects, Cantor programs must be expressed in terms of small objects. Thus Cantor is particularly well suited for programming the Mosaic, although it runs now on cubes and on sequential computers.

Cantor software development has continued and now includes a small flow analyzer that is used to remove redundant type checks inside individual object definitions. An interpreter written in 68000 assembly has been written and tested. The native code interpreter project is a precursor to the native code generation project. The performance of the 68000 interpreter is roughly twice the performance of the interpreters written in C.

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.

3.2 Reactive primitives for the Cosmic Environment

Wen-King Su, Chuck Seitz

A new formulation for communication primitives has been adopted for the second generation cubes that are now in development. In order to test the ideas and to develop the code necessary to support these "reactive" primitives on the second generation machines, we have implemented them in the host runtime system, called the cosmic environment, for the Cosmic Cube and iPSC. These primitives are also supported in a new operating system being developed for the Cosmic Cube (see section 3.3).

The old (Cosmic Cube) primitives treat messages as logical entities held in message buffers. When a process calls the `recvq` function, it gives the kernel the address of the buffer to accept the message. When a process calls the `sendq` function, it gives the kernel the address of the buffer containing a message. In each case, the process is notified when the message action is completed by a lock variable in a shared message descriptor. The kernel and programs using these primitives are complicated because the message operations are not atomic.

The new primitives treat each message as a physical entity including both the message buffer and its content. When a message is sent by a process, the process releases the buffer. When a message is received by a process, a message buffer containing the message is given to the process. The new communication primitives are:

`xsend(buf,dst)`

> `buf` is a pointer to a message buffer previously allocated by a storage allocation function called `xmalloc`, which is identical to the Unix `malloc`, and `dst` is the destination process. During the `xsend` call, the buffer is released by the process and the buffer is placed in the node's send queue.

`buf = xrecv()`

In theory, when `xrecv` is called the process is suspended, and the process specified by the message at the head of the receive queue is activated by returning the message buffer to that process. In practice, in order avoid context switches, the system may search back along the message queue for another message for the same process. This optimization is allowed because message order preservation is assured only between pairs of processes. If the input queue is empty when it is checked, a process is selected (nominally round-robin) and `NULL` is passed to it. When the process no longer requires the message, it releases the buffer with an `xfree` function.

The Cosmic Environment has been revised from v6 to v7 to support these new primitives, and v7 is now in use for the Cosmic Cubes and Intel iPSC. Currently it talks to our old style cubes through a layer of library routines that emulate the old primitives using the new primitives. This changeover was accomplished without causing problems to *any* user programs!

3.3 Reactive Kernel

Jakov Seizovic, Bill Athas, Wen-King Su, Chuck Seitz

Work has started on the operating system for the second generation Cosmic Cubes. A new operating system called the Reactive Kernel (RK) is being written in C, and is being tested on the Cosmic Cubes. The RK runs reliably now, and is expected to be available for Cosmic Cube users within a month. RK will eventually replace the Cosmic Kernel. Except for a small number of communication hardware dependent functions, it will be directly portable to the second generation machines.

The message-driven organization [5220:TR:86] has made the Reactive Kernel less complicated and easier to maintain than the Cosmic Kernel. The scheduler is reduced to a simple dispatch loop that normally activates processes (objects) according to (although not necessarily strictly according to – see previous section) the order of messages in the receive queue.

The Reactive Kernel is structured in two main layers. The inner layer consists of the reactive communication primitives, dispatch loop, and message interface. The message interface implemented for the Cosmic Cube imitates the routing hardware of the second generation cubes in interrupt routines, and implements the wormhole routing of the Cosmic Cube 64-bit hardware flow control units (flits). This message software is the only machine-dependent part of RK, and will be drastically reduced on the second generation machines, since the routing is accomplished in those machines in the hardware. The message system implemented in the inner layer of RK is tuned for short messages. Long messages will be fragmented by the library routines at a higher level.

The outer layer supports a particular environment, and thus can be optimized for the environment without interfering with the message-passing mechanism. This kind of layered approach provides, for example, for running a standard Cosmic Cube style environment on one sub-cube, and an environment optimized for Cantor (see section 3.1) on another sub-cube.

To provide for maintaining system services even when a user process is in an error state, a class of priority messages has been introduced with a separate receive queue for each class. Their effect is analogous to the interrupt mechanism intrinsic to the instruction processor. Interrupt messages also provide a means of time-driven scheduling, should that be desired for certain applications.

Except for the interrupt messages, which are supposed to be infrequent, context switches usually occur only at well defined places in programs (at `xrecv()`), so that the necessary overhead is reduced. Although second generation cube software rather than a new cosmic cube operating system is the principal goal of this effort, the reactive approach, combined with the latency advantage of wormhole versus store-and-forward routing, is expected to improve the performance of latency-sensitive Cosmic Cube programs.

3.4 Object-oriented event-driven simulation

Wen-King Su, Chuck Seitz

The changes in message primitives reported in section 3.2 have been influenced by and in turn inspired a new approach to distributed event-driven simulations. The `xrecv` primitive, as opposed to a blocking receive, will return `NULL` when every process in the node is waiting for a message and the node has no messages queued for any process. While this property does not help or hurt reactive programming, which inspired these primitives, the latest incarnation of the simulator makes use of this property to assure progress.

One goal in refining our simulator is to reduce the number of messages needed in doing simulation. A lazy simulator approaches this goal by queueing output state information that otherwise could be delivered immediately. An overly lazy reactive simulator can deadlock due to data dependencies. This deadlock condition and the theoretical limit on laziness has been explored in our and the earlier research of Chandy, Misra, and Bryant. The method employed by others to overcome this limit is to do a global deadlock detection and to re-trigger the processes when a deadlock is detected. However, a global deadlock detection is very expensive on a large network and can be very costly for simulations that run into a deadlock situation frequently. With the new communication primitives, we can re-trigger the processes based on a simple test on local conditions.

When a deadlock does occur during a simulation, the message queue in each node must be empty. Thus if we re-trigger the processes in a node whenever the input queue for that node is empty, the simulation will not deadlock. The trigger is `xrecv` returning `NULL`. Although a node with an empty queue does not imply a deadlock, an empty queue implies an idle node. In this simulation approach, the idle time is profitably spent in formating and sending output messages.

When using the indefinite lazy evaluation in the form:

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

there are a number of heuristics that may be used to decide in cases in which `xrecv` returns a non-NULL pointer whether it is likely to improve simulator *performance* (progress is assured) to send outputs as messages immediately, or whether the messages can be queued. For example, consider an output that lies on any circuit of the graph of processes for which the minimum delay around the circuit is D_{min}, and the number of elements is k. We suspect that the process should send outputs whenever the queued messages correspond to a delay comparable to D_{min}/k.

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 architecture of the Mosaic C system is discussed in section 2.2.

The Mosaic C chip is composed of 3 main parts: RAM & ROM, channels, and processor. Our strategy for verifying the design of this very complex chip and to characterize its yield on MOSIS runs is initially to fabricate and test the 3 main parts separately. After the parts are well characterized, their layouts will be combined onto a single chip.

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 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 size above about $36mm^2$. The area remaining for RAM is about $6M\lambda^2$ at $3\mu m$ feature size, $26M\lambda^2$ at $2\mu m$ feature size, and up to $90M\lambda^2$ at $1.2\mu m$ feature size. The RAM thus represents about 70% of the area of the $2\mu m$ version of the Mosaic C, and up to 90% of the area of the $1.2\mu m$ 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.

4.1.1 Mosaic C dRAM

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.

4T dRAM

The 4-transistor dRAM is described in detail in our previous semiannual technical report [5235:TR:86]. A first prototype was completed and sent to MOSIS for fabrication, and first silicon arrived from MOSIS (run M71J) on 31 March 1987. This prototype is an 8K-bit (1K byte, 512 16-bit word organization) slice.

Preliminary testing (we have only had 3 days between the receipt of the chips and the compilation of this report) demonstrates that the prototype reads and writes correctly, but the yield is disappointing. The testing is being performed using a Tektronics DAS9100 logic analyser and a simple test fixture that provides timing.

Of the 12 chips received, only 3 of them pass all of the limited set of tests that we have subjected them to so far. Six of the chips that do not work have scattered bad bits, while three of them have massive faults. The source of the yield problem will be investigated, but it is evidently a fabrication yield problem. This kind of yield is unsatisfactory for system experiments that involve chips even as complex as parts of the full the Mosaic C elements. If we have a yield of only 0.25 on this 1K-byte RAM, we could expect a (naive) yield of only $0.25^4 = 0.004$ for a 4K-byte RAM, which is about the smallest size that would be interesting for a system experiment.

2T dRAM

Since the last report, work has focused on analyzing the statics and dynamics of sensing, and deriving a suitable driving current, for the 2T dRAM. The general circuit configuration is described in the last report [5235:TR:86].

The sense amplifier spends most of its time in saturation, where the differential current is quadratic in differential voltage. Solving the differential equation, the bit-line separation goes as $-1/t$ (for negative t, until it reaches one threshold). In light of this equation, sense amplifiers can be characterized by a delay-sensitivity product, which is determined by the ratio of bit line capacitance to amplifier capacitance, and by τ.

The implicit assumption in the above is that one transistor of the cross-coupled pair is conducting while the other is at the verge of conduction. Using dependent sources in Spice2G.6 to simulate this condition, the sense amplifier traces out its own optimum current waveform, and we found that our speed target requires an initial bit-line separation at least 50 mV more than the threshold mismatch.

Additional dependent sources allowed us to solve for the gate voltage waveform that would produce this optimum current. The closest approximation to this wave-shape was to use positive feedback from the node being driven, but the resulting rate of current increase depended much too strongly on thresholds and power supply voltage. The nMOS Mosaic taught us the necessity of wide power-supply tolerance, so this approach was abandoned.

The current from each ramp generator must be divided among many sense amplifiers, but it doesn't necessarily divide evenly. With a current mirror, the mirror transistors would be as large as the sense transistors, operate at similar voltage levels, and thus be similarly affected by threshold mismatches – compounding the effect of those mismatches. With the sense amplifiers connected common-source, degraded voltages from a weak bit causes that sense amplifier to get less than its fair share of current, which can be a serious problem if the fair share is already small, *i.e.* if one aims for extreme sensitivity. Conversely, the effect is manageable if the current is high. The current can safely be set as high as it would be in the absence of this effect, so it doesn't slow down the worst case the way the current mirror would. Setting the current for a sensitivity of half the initial bit-line swing seems about right.

Data retention is adversely affected by the heat and power-supply noise from the parts of the chip that operate at full logic swings. Upon calculating the power budget, a majority of this power comes from the dozens of output pads. The only way to reduce their heat and noise is to dictate a reduced supply voltage. Once this decision was made, threshold drops were obviously "out," greatly simplifying the design choices and clarifying the problem with our previous approach, wherein we had assumed that sensing speed was independent of supply voltage. The rest of the chip slows down as the voltage is decreased, so it's natural for the RAM to do so too, and quite helpful: the delay-sensitivity product gives us increased sensitivity under the conditions where it's needed most. Thus, setting the current with pulldowns like those in the rest of the chip means that the minimum detectable signal is specified as some percentage of the supply voltage (about 2-4%) instead of an absolute value. The bit-line separation is likewise specified as a percentage of V_{dd} (about 6-8%). Threshold mismatch is an unfortunate exception to scaling, but even this has a component that depends on supply voltage, due to the body effect.

At about this point we discovered that the default for `lambda` in the Spice2G.6 level=2 model gave unreasonable currents at high fields, so we redid the analyses with level=3 models.

In the foregoing analyses, one transistor of each cross-coupled pair is assumed to be on the verge of conducting. However, since saturation current is quadratic in voltage, it pays to let it conduct some, so that the other transistor can conduct much more. Capacitance matching is a limiting factor; with a 20% mismatch, the differential current peaks at a 2:1 ratio of currents, which seems like a safe place to draw the line.

With the initial current finally determined, attention turns to dynamic analyses to select a safe schedule of current increase. Since an increasing current is more difficult to control than a static one, a more conservative definition of "safe" is necessary, and we return to the initial assumption that one transistor of each pair is (at most) barely conducting. Since a ramp voltage on the gate of a transistor does not produce an especially close match to the optimum waveform, the delay-sensitivity product is a little higher than what would otherwise be attainable, though it looks like the speed target can still be met.

4.1.2 Channels

The Mosaic C channels are 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.

The 1-dimensional router described in the last report underwent the full extractsimulate-modify cycle and was sent off for fabrication in early February. MOSIS estimates that these test chips should be returned by the middle of April.

Each dimension of this router is about $1200\lambda \times 750\lambda$ in area. The estimated area for a complete 2D router, including the bus interface, is about $2000\lambda \times 1500\lambda$, assuming we still use the minimum width flit. This minimum flit allows for 2 data bits, 1 control bit, and 1 acknowledge bit for each half of a bidirectional channel. So a the minimum 2D router would require (8 wires)$\times$(2 directions)$\times$(2 dimensions) = 32 channel I/O pins, plus extra ground pins to minimize noise. It is quite likely that the flit width will be expanded to 4 data bits, something which can easily be done without any real layout modification. The number of channel I/O pins then increases to 48.

Work has also been started on the interface between the processor, RAM, and the channels. It will require arbitration between the output channel request, the input channel request, and a refresh request from RAM, and will grant a free bus cycle to one of these. The major details yet to be worked out involve timing and responsibilities for the different sections, after which layout can begin. The design is straight-forward.

4.1.3 Processor

The Mosaic C processor datapath layout is complete, and switch simulates correctly with MOSSIM. The dimensions of this 16-bit datapath including 24 general registers, ALU, shifter, condition flags, 7 addressing and address limit registers, and address arithmetic, is approximately $1000\lambda \times 3500\lambda$.

The general registers are addressed as two overlapping banks of 16 registers for each "sequence." The address registers of Mosaic C include 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 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 microcode is currently being tested using a detailed instruction-level simulator for the Mosaic C microcode and datapath. Several microcode bugs have been found. Final assembly of the processor is expected within the next two months.

4.2 Automatic Compilation of Programs into Self-Timed VLSI Circuits

Steve Burns, Pieter Hazewindus, Alain J. Martin

We have developed a simple automatic compiler for translating CSP programs into self-timed circuits. The basic constructs of the source language are directly translated into self-timed circuit pieces. The syntax of CSP directs the reconstruction of these sub-circuits into a circuit semantically equivalent to the original program.

This compilation method differs from our previous schemes by using translation rules produced for each grammar rule of the source language. Instead of redoing it for each individual program, the transformations to handshaking expansions, production rule sets, and operators sets, the grammar rules are transformed once, producing translation rules that may be applied to individual programs. This meta-compilation of grammar rules produces a systematic means of translating CSP source code directly into operator sets.

By avoiding the internal representations, the meta-compilation strategy quickly produces operators sets for every CSP program. Because the compiler does not examine the structure of the individual programs, the circuits produced by the compiler are not yet as good as those produced by a human designer performing transformations through the intermediate forms. However, the compiler is useful as an existence proof of an automatic compilation scheme and as a benchmark with which to compare future compilation procedures.

4.3 Compilation of Programs into Self-Timed VLSI Circuits – State Variables

Steve Burns, Pieter Hazewindus, Alain J. Martin

One of the problems in automating the compilation of CSP-like programs into VLSI circuits is the introduction of so-called state variables. So far it has been the designer's task to choose state variables, and the states in which they are changed. We have tried to apply structure theory to this problem, in particular the theory of decompositions of finite state machines.

The handshaking expansion is translated into a machine description. This machine now has to be decomposed in order to get a circuit. We view each gate in a circuit as a finite state machine, and we define the behavior of a composition of gates. These compositions are different in self-timed circuits from the classical theory using synchronous machines. Consequently, the rules for decomposing a machine are also different. We consider partitions of the set of states of a machine. A partition is said to have the *substitution property* if the next-state function maps equivalent states into equivalent states. If a machine has a partition with substitution property, then (part of) the circuit can be realized loop-free.

If, on the other hand, no partition has the substitution property, then there is no loop-free realization of the circuit. In this case we resort to partition pair algebras. It is possible to derive circuits using partition pairs. It has not yet been shown that every circuit can be thus derived.

The possible advantages of this method are that a machine description seems to be a convenient way to represent a handshaking expansion, and that it is easier to find state variables yielding small circuits using analysis tools from the classical decomposition theory. Moreover, the number of partitions to be considered is relatively small since we need to consider only partitions into two parts (all gates are binary), and since most machines have a very regular structure.

4.4 Adaptive Routing in Multicomputer Networks

John Y. Ngai, Chuck Seitz

Our investigation of adaptive routing strategies is aimed at ways to improve and sustain the performance of communication networks in moderate to heavy message loading situations. As a first step towards developing an analytic framework for the understanding of such adaptive strategies, we have focused in addressing a number of issues fundamental to any communication network supporting reliable concurrent computation:

(1) *The problem of avoiding communication deadlock.* If there is any advantage in making adaptive routing decisions over making deterministic ones, it must come from the flexibility to exploit alternative routes. Such flexibility must be introduced in ways that still guarantee freedom from communication deadlock. The concept of *virtual channels* [5231:TR:86] provides a way to develop deadlock-free routing algorithms in arbitrary direct networks. While it may be possible in certain special cases to allow for multiple alternative routes using virtual channels, the task becomes progressively more difficult in more complex routing schemes and network topologies.

The concept of a *structured buffer pool*[1] developed for store-and-forward computer communication networks provides a general approach to introduce alternate deadlock-free routes into an arbitrary network. However, these schemes in general require a buffer pool that grows with the network size, and hence are not feasible in fine grain multicomputer communication networks where resources at the node level are finite and scarce, and the network is arbitrarily extensible. To

[1] P. Merlin and P. Schweitzer, "Deadlock Avoidance in Store-and-Forward Networks – I : Store-and Forward Deadlock", *IEEE Transactions on Communications*, Vol. COM-28, No. 3, March 1980.

cope with this problem, we decided to adopt the *exchange* model[2] commonly employed in synchronous sorting networks, rather than the usual *block until clear then forward* model used in conventional *store-and-forward* and *wormhole* routing, as our basic node to node message transfer mechanism.

The exchange model avoids the necessity of having to block a message and wait until the requested resource is available, by having the two communicating parties agree to preempt the requested resources if necessary. In the event where both parties are full, the preemption is accomplished through an exchange. Message transfer is guaranteed locally between every pair of adjacent nodes acting as partners in the exchange operations.

(2) *The problem of guaranteeing message delivery.* The ability to guarantee eventual message delivery is another basic requirement for the communication network supporting reliable concurrent computation. Although the message exchange model eliminates the possibility of communication deadlock, it replaces it with the need to demonstrate eventual delivery of every message. The problem is that preemption of a message can push it to a node further away from its own destination. We cope with this problem by assigning a priority to each message and let the messages compete among themselves for access to the physical channels. In order to deal with the dynamic situation where new messages are continuous generated, we let the messages *age* while in transit inside the network, and assign a higher priority to an *older* message than to a *younger* one. This priority based message assignment scheme allows us to assure eventual message delivery for every message inside the network.

(3) *The problem of guaranteeing initial message injection.* In addition to the necessity in demonstrating eventual message delivery, the exchange model also requires one to be able to assure eventual message injection into the routing network. The problem here is that a node located at the cross point of heavy message traffic can be denied injection of its own message into the routing network for an arbitrary period. Roughly, this happens because the exchange model prevents a node from *blocking* its through traffic to make room for its own message. We cope with this problem by imposing either a message delivery or a message injection discipline that ensures room for message injection when one is needed. These methods directly force the occurrence of empty slots in the through message traffic whenever a node has a message waiting for injection. By using these methods we are able to assure eventual message injection for every node inside the network.

These issues and their resolution are summarized in much greater detail in a technical report under preparation. We regard it as the first steps towards laying

[2] A. Borodin and J. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation", *Journal of Computer and System Sciences* **30**, 130-145 (1985).

down a foundation and demonstrating the feasibility of the adaptive routing approach. Our next effort in the immediate future will be focused in developing a concise analytic framework that allows us to discuss the various adaptive control strategies and to carry out performance analyses and simulations.

4.5 Self-timed Routing Automata (Modular Design)

Charles Flaig, Chuck Seitz

Instead of designing a single routing chip for second generation cubes, we have developed a general scheme for designing routing chips based on routing automata, as described in our previous report [5235:TR:86], and have now developed CMOS designs and layouts and a stylized way of assembling them to put together in a day or two whatever kind of routing chip we might like.

These designs are based on data flow through self-timed FIFO (queue) stages, with intermediate logic to operate on the headers, and to split and merge paths.

A small FIFO section was designed and sent in for test fabrication in February, but since then we have changed the design of the FIFO to be in

Large sections of the data-path and some control sections have been designed and laid out for the updated version. These include a symmetrical switch section that can be used with various different control structures for both the split and merge operations. Each of the data paths is 10-bits wide, 8 for data, 1 for control, and 1 as yet undefined (included for symmetry). Only 6 bits are needed for displacement information (maximum of 64 nodes in each dimension) so we have enough bits to encode all other control information for that dimension on the same flit. Some work still has to be done on the control structures for the switches and the decrementer.

As far as possible, each of the sections of the data path and control structures are being designed to be modularly composed into routing automata with different structures, for example they could also be used directly to implement a router for a hypercube structure by including switches only in the 'down' direction.

A simple schematic for 1 dimension of the router is shown below. The center path comes from the previous dimension and proceeds to the next dimension. A message may either switch to an output path in the + or - direction or proceed to the next dimension. The outside paths are for messages going in the + and - directions. A message here may either continue in the same direction or switch to the center path to proceed to the next dimension.

```
- FIFO section        >----O-------X------D--->
| bus connect              |       |
O merge section       >----X---X---O---O------>
X split section                |       |
D decrementer         >--------O-------X--D--->
```

Each dimension has an area based on layout done so far of $1000\lambda \times 1500\lambda$, plus area for extra FIFO stages. Each channel taken off chip has 8 data bits, 1 control bit, a request line, and an acknowledge line for a total of 11 wires in each direction of a bidirectional channel.

Cycle time for the FIFO stages is estimated by the τ model to be less than 5ns for 3μm SCMOS technology. Switch and decrementer delays will probably be about twice that. The largest delays in this highly pipelined design will be in driving the pads and sensing the outputs to generate a request signal, so we expect an overall cycle time of about 40ns (25MHz) in 3μm SCMOS.

4.6 Cantor Engine

Bill Athas, Nanette Jackson, Jakov Seizovic, Chuck Seitz

A hardware architecture for a Cantor engine has been designed as a VLSI chip project. The engine is a "Harvard" architecture with separate data and instruction streams. The instruction bus is a dedicated high performance bus that is private to each Cantor engine. The data bus is a shared, high latency bus that is shared with an object manager and possibly several other engines. The object manager is responsible for sending and receiving messages. Because the data bus can exhibit long access times, "hardware futures" are used instead of a cache. Hardware futures are special registers that are asynchronously loaded by an autonomous data fetch unit. The Cantor compiler can emit load commands early so that the processing of instructions and the load of data registers is performed concurrently.

The datapath for the engine is split into the *tagpath* and the *itempath*. The tagpath is used for dynamic type checking. The itempath is a slightly modified Mosaic C datapath. The major modification to the Mosaic datapath is the addition of a second data bus between the ALU and registers. This second bus allows ALU transfers to occur every two clock cycles. Because the Cantor engine is specified for 32-bit data values and the Mosaic ALU produces 16-bit results, the ALU is modified slightly to produce a 32-bit result every two clock cycles.

4.7 SCMOS Pad Frames

Fritz Nordby, Chuck Seitz

We continue to design input and output pad circuitry, and pad frames. The current driver for this effort is the new MOSIS 132PGA standard package, which we need to use for prototype Mesh Routing Chips for the 2nd generation cubes. It is surprisingly possible to surround the 6.8mm×6.9mm MOSIS project with 132 pads, and we will propose this pad frame as a standard frame.