



SUBMICRON SYSTEMS ARCHITECTURE
SEMIANNUAL TECHNICAL REPORT

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

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

5220:TR:86

Computer Science Department
California Institute of Technology

March 1986

SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

*Department of Computer Science
California Institute of Technology*

5220:TR:86

March 1986

Reporting Period: 16 September 1985 to 15 March 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 six month period 16 September 1985 to 15 March 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, 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].

1.3 Highlights

Some highlights of the previous 6 months are:

- Working Mosaic A elements (2.2 and 4.1)!!!
- Torus Routing Chips work on first silicon (4.2).
- Mosaic C – an SCMOS Mosaic element – being designed (2.2.2 and 4.3).
- Compilation method for designing self-timed circuits (4.5).
- 128-node Intel iPSC in routine operation and available for use via the ARPAnet (2.1.3).
- New results in parallel execution of logic programs (3.1).
- A new flow framework for concurrent programming (3.3)
- Event-driven simulations chew up endless cycles on the cosmic cubes (3.5).
- A surprise about network topologies (3.6).

2. Architectural Experiments

2.1 Cosmic Cube Project

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

The cosmic cubes are working reliably, and researchers are using them for application programming projects and for event-driven simulations of other message-passing architectures. Usage has been moderately heavy, and there have been very few problems reported.

2.1.1 Hardware Status

The 6-cube interface was moved from `sol.caltech.edu`, our main SUN file-server, a SUN-2/170, to `ceres`, a SUN-2/120, in order to lighten the load on the already overloaded file server. The 3-cube interface was moved from `ceres` to `venus`, another SUN-2/120.

Neither the 6-cube nor the 3-cube has failed in this 6-month period. The cubes have now logged 1.85 million node-hours with 3 hard failures. The calculated MTBF of the nodes of 100,000 hours reported before these machines were constructed can now be stated to be conservative at the 99% confidence level. A node MTBF in excess of 500,000 hours is probable, and can be stated now at a 55% confidence level. The systems will have to operate for another $2\frac{1}{2}$ years with a similar or smaller failure rate for us to be able to state a 500,000 hour MTBF with 90% confidence.

2.1.2 System Software Status

There have been no significant changes made in the cosmic cube system software in this period. Progress continues in converting some additional parts of the operating system kernel from 8086 assembly code to C to assist in bringing this kernel up quickly on future machines.

2.1.3 Intel cube

In October 1985 our first Intel cube, an iPSC d6 (64 nodes), was installed. In December it was upgraded to a d7 (128 nodes). These machines were 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 ARPA researchers who may wish to experiment with them. The iPSC host is `ipsc.caltech.edu` (or `ipsc.arpa`) on the ARPAnet. In order to request an account, please contact `chuck@vlsi.caltech.edu`.

The Intel iPSC systems are quite similar to the cosmic cubes, but have faster hardware and more primary storage per node. The system software does not include

all the features of the cosmic cube due to limitations of the Intel 286/310 host, which runs Xenix, while the cosmic cube hosts run Berkeley Unix.

Each of the iPSC nodes consists of an Intel 80286, an 80287 floating point processor, eight ethernet channels, an RS422 channel, and 512K bytes of memory. Seven of the ethernet channels are used for bidirectional point-to-point cube connections, while the eighth is used as a global channel for communication directly to the host. This global channel is an improvement over the cosmic cube's scheme, in which only node 0 has a connection to the cube host. (In the cosmic cube, node 0 and other nodes on the path from a process performing I/O to the host bear an extra load of routing a substantial volume of messages.) The RS422 channel in the iPSC is used for initializing the nodes, and for running diagnostics.

We have seen the performance of programs ported from the cosmic cube to the iPSC increase by a factor of three per node. For example, Chuck Seitz's Collatz sieve program, which spends most of its time in arbitrary precision integer arithmetic routines, runs six times faster on the Intel 7-cube than it does on the cosmic 6-cube. This performance on the iPSC d7 is more than 60 times faster than the best sequential program version of the Collatz sieve running on a VAX11/780.

Other benchmarks comparing the cosmic cube with the Intel cube show that the cosmic cube is faster for short (less than 80 bytes) messages, but asymptotically about one fourth as fast for long messages. These differences are due to the greater bandwidth but higher overhead in starting up the Intel cube's DMA-driven ethernet channels. Because of the higher message latency on the iPSC, relative to instruction rate, applications that use message passing intensively and tend towards shorter messages are best run on the cosmic cube, while problems involving coarser grain concurrency run much faster on the iPSC. A detail characterization of message-passing performance on both the cosmic cube and iPSC is currently being written up.

The Intel system software at present resembles an early version of the cosmic environment, but both performance and convenience continue to improve with later releases. In order to accommodate differences between the iPSC and cosmic cube functions, a compatibility library was written that allows cosmic cube programs to run on the iPSC without changes in the source code. This library will track expected future changes in Intel's software.

The Intel hardware has been reliable, and the system software has been reasonably free of bugs. We have encountered several problems with the TCP/IP implementation for the 286/310 host, but the latest release appears to work reliably. We have sources for all of the system software, and have been able to make experimental modifications to and reassemble the node operating system.

2.1.4 Application programs

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

Several researchers from the Aeronautics department at Caltech, under the direction of Professors Brad Sturtevant and Tony Leonard, report very good results on some difficult fluid mechanics computations. Some of these computations are based on a vortex model in which the interaction of vortices is computed in a way that is essentially similar to an n -body computation. Others of these computations employ 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. These results and others were reported recently in a "Workshop on Applications of the Cosmic Cube to Selected Problems in Fluid Mechanics" organized by the Caltech Aeronautics Group.

A program written by Craig S. Steele of our research group is suggestive of the way in which these machines are used on problems that come up in our research. The Intel iPSC was used to support a conjecture in information theory. Mutual information is a measure of the maximum information capacity of a set of noisy communication channels. Essential mutual information is the information transfer possible if the probabilistic input/output function is replaced by the best possible "forced choice" of an output for each input wire. For most input probability distributions, the essential mutual information is a large fraction of the mutual information of the original noisy configuration. The question of just how bad the best "forced choice" can be seems analytically intractable.

Recently the problem was attacked by a randomized search of possible configurations. After considerable effort on conventional computers, a concurrent version was produced for the iPSC. For this particular problem, the iPSC d7 proved to be about 80 times faster than a VAX-11/750 under UNIX. It was possible to conveniently search 100 times more cases than previous efforts. A new lower bound on the ratio of the essential mutual information to mutual information was found, and the results supported the conjecture that the ratio is low only for configurations of extremely low information capacity.

Another problem that would be practically out of reach is our ongoing evaluations by simulation of preliminary designs for future message-passing concurrent computers. In most cases analytical techniques are not adequate to model blocking in the message-passing networks. An implementation of event-driven simulation (see section 3.5) using a technique similar to the Chandy-Misra-Bryant distributed simulation algorithm has been used to "benchmark" machines not yet constructed. Simulations of message-flow performance done in 1981 in connection with the cosmic cube design, and for machines up to 96 nodes, required about 1000 hours of DEC-2060 time. These more ambitious simulations, performed for machines of various sizes up to 1024 nodes, are expected to consume a similar amount of time on

the cosmic cubes.

2.2 Mosaic Project

Over the past six months, the Mosaic project has been the focus of our system-building activities.

- With many trials and tribulations past, and more no doubt to come, we now have working prototype Mosaic A elements. This work is described in section 4.1.
- A number of ideas have developed – thanks to the work of several of the senior graduate students in the group – to the point where 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 sections 2.4 and 3.3.
- Some recent results on message routing and network topologies suggest that Mosaic systems with large numbers of elements can use a low-dimension (2D or 3D) mesh connection. More complex topologies such as shuffles and binary n -cubes are unnecessary. See section 3.6.
- The design of a Mosaic C, in MOSIS SCMOS technology, is underway. See sections 2.2.2 and 4.3.

2.2.1 Mosaic software tools

Steve Rabin, Chuck Seitz

The Mosaic A assembler has been upgraded to be essentially compatible with the Mosaic B assembler, and several programming conveniences have been added to both assemblers. Modifying the assembler to accommodate the Mosaic C is expected to be straightforward.

Because bugs in Mosaic programs can garble the message system flow control, it is difficult to debug programs directly on Mosaic hardware. The Mosaic A element, which lacks interrupt hardware and clears its state on RESET, is unable to restore system context in the event of a serious error in the message system. For this reason most of our software development must be performed on a simulated Mosaic ensemble rather than on Mosaic hardware.

The current Mosaic simulation environment supports the simulation of Mosaic A and B 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 – written in MAINSAIL – executes roughly 50 simulated Mosaic cycles per VAX11/780 cpu-second. At this rate the Mosaic A v1.2 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 bootstrap programs we are interested primarily in ensembles of one element.

Work is now under way to provide a concurrent simulation environment for Mosaic ensembles, intended 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.

2.2.2 Mosaic C

Bill Athas, Lounette Dyer, Fritz Nordby, Steve Rabin, Don Speck, Wen-King Su, Chuck Seitz, and the 1985-86 Caltech VLSI Design Class

The design of a new Mosaic element, called Mosaic C, was started in January 1986. Some of the details of the chip design are described in section 4.3, and an overview of the architecture is summarized here.

- Target technology: MOSIS SCMOS with $0.6\mu\text{m} \leq \lambda \leq 1.5\mu\text{m}$. Target maximum chip size is 36mm^2 , or $100\text{M}\lambda^2$ with $\lambda = 0.6\mu\text{m}$, and $16\text{M}\lambda^2$ with $\lambda = 1.5\mu\text{m}$. Speed, storage size, and top-level floorplan will necessarily vary with feature size.
- Storage: dynamic RAM, with refresh accomplished by the processor microcode as in Mosaic A & B, and small bootstrap ROM. 2T, 3T, and 4T RAMs are being designed. The primary storage is expected to require between 420 and $675 \lambda^2$ per bit including peripheral circuits, depending on which design is used. The 2T RAM, with only $420\lambda^2$ and one contact per bit, is looking very promising.
- Processor: 16-bit, 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 sets of registers to be able to switch instantly between task and channel processing contexts.
- Channels: memory mapped, bit serial, two virtual channels per physical channel, with hardware assist for cut-through routing. These channels are a compro-

mise between the performance of channels such as those on the Torus Routing Chip (TRC) (see section 4.2) and generality. The TRC channels also require more pins. Routing will be controlled by software, but after the routing decision is made the flow control units (flits) are shuttled from incoming to outgoing channel without processor intervention.

The processor, channels, ROM, and peripheral circuits on the Mosaic C are expected to consume about $10M\lambda^2$. The amount of space left for storage depends strongly on feature size: about $6M\lambda^2$ or 1.5K bytes at $3\mu\text{m}$ feature size – similar to Mosaic A –, about $26M\lambda^2$ or 8K bytes at $2\mu\text{m}$ feature size, and up to $90M\lambda^2$ or 24K bytes at $1.2\mu\text{m}$ feature size. The $3\mu\text{m}$ version can be used to prove the design. The $2\mu\text{m}$ version with 8K bytes of storage would be nearly ideal for experiments with message-driven programming styles (see section 2.4).

2.3 Future Architecture Experiments

Chuck Seitz, Bill Athas, Bill Dally, Craig Steele, Wen-King Su

We are developing a number of designs for second generation medium grain size message passing systems to combine the performance of commercial 32-bit processor chips and the low-latency communication of devices such as the Torus Routing Chip (TRC). Some of the critical design decisions are being investigated by simulations (see sections 2.1.4 and 3.5).

Since many cosmic cube programs display their results as images, and because these programs are I/O limited, we intend to include in these second generation systems a distributed frame buffer to allow the system to generate graphic displays directly.

2.4 Reflections on Models of Concurrent Computation

Chuck Seitz

In our experiments of the past year or so with message-passing concurrent computers, we have started to introduce an object discipline into the use of the concurrent process model on which the cosmic cube and Mosaic are based.

In the basic process model, concurrent processes communicate by asynchronous, non-blocking *send* and *receive* operations, and messages are queued while they traverse the message system. These aspects of the process model appear to be dictated by physical design considerations in VLSI technologies. The way in which this model is captured in programming notations has its roots both in Hoare's Communicating Sequential Processes (CSP) notation and in Lang's work in our own group on concurrent extensions of object-oriented programming notations [5014:TR:82] such as Simula.

We like to think of CSP as the FORTRAN of concurrent programming notations, because in CSP programs are static entities. The process structure – the set of processes that participate in the computation and their references to other processes – remains static for the duration of a computation. Processes communicate with operations in which (by definition) the number of completed *send* and *receive* operations on a channel are identical, much like signaling on a wire. CSP is then, to us, a satisfactory notation for describing either hardware (*c.f.*, section 4.5) or quite simple programs, but is not able to express programs of a more dynamic and interesting character.

Our process model, as expressed by adding extra primitives to programming languages such as C, is much more “liberal” than CSP. The synchronization between *send* and *receive* is potentially unbounded. The number of completed *receive* operations of a given type is not more than the number of completed *send* operations of the same type. Our model is also liberal in its notion of a “channel”. All that a process requires to send a message to another process is reference to the other process, and reference can be passed in messages. Finally, we permit processes to spawn or to kill other processes. One can express programs with the dynamic character that we prefer. However, all of these capabilities are invoked explicitly.

This process model has a number of virtues. It is more than adequate for many types of scientific computations, and is compatible with familiar concepts from operating systems based on processes and message-based interprocess communication mechanisms. It is extremely powerful: any *trigger* mechanism [1] can be programmed. For example, it is relatively easy to build an object-oriented programming environment on top of the process model with the methods run under a receive-dispatch-execute loop.

The object-oriented programming notations used by Bill Dally (Concurrent Smalltalk, or CST) to express concurrent data structures, and by Bill Athas (Experimental Concurrent Programming Language, or XCPL) for compilation experiments, forego the use of an explicit *receive* in favor of the objects being *reactive* or *message-driven*. In this respect these programming notations are Actor [2] languages. An Actor similarly *reacts* to an incoming message by *sending* a (possibly empty) set of messages, creating a (possibly empty) set of *new* Actors, and changing its state (or behavior, or, in Actor semantics, it *becomes* another Actor). This trio of *send*, *new*, *becomes* is then the foundation of Actor semantics. One important difference in CST or XCPL objects is that they can exercise *discretion* in which messages they may react to. This concept was also present in Lang’s work [5014:TR:82] of four years ago. We have persisted in the belief that the message system rather than the objects is right place to buffer messages.

It appears that there are sound reasons both with respect to programming style and architecture to use an object rather than process model for our next round of experimental machines – both those machines based on commercial processors and

the Mosaic C. From the programming standpoint, the object model introduces a discipline into the use of *send* and *new*. From the architecture standpoint, recently developed message routing techniques reduce communication latency (see 3.6 and 4.2) to an extent that puts pressure on the node processor to reduce the overhead associated with *send* and *receive* operations. There is some speed advantage for a node to schedule process (object) execution based on received messages – that is, to make the processes (objects) reactive and the processors message-driven (see 3.7). It also appears that the operating system of a node can be made simpler than, for example, the cosmic kernel, if its entire job is (1) to run a process (or a particular method, if the method lookup is in the operating system) according to the messages received, and (2) to be able to spawn new processes. The organization of Mosaic C as a two-sequence computer is motivated in part by the desire to make it a good testbed for this model. The communication sequence, in addition to controlling message routing, would be able to provide the desired front-end services for message-driven operation of the node with no context-switching delay.

References:

- [1] see Kotov's classification of trigger mechanisms in chapter 5 of *Algorithms, Software, and Hardware of Parallel Computers*, edited by J Miklosko and V E Kotov, Springer-Verlag, 1984.
- [2] Gul A Agha, "Actors: A Model of Concurrent Computation in Distributed Systems", Technical Report 844, MIT AI Lab, 1985.

3. Concurrent Computation

3.1 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 multiprocessor system. The Sync Model constructs a dynamic tree of processes to represent the AND/OR proof tree of a logic program and performs breadth-first search on the tree. AND parallelism is implemented through the construction of dynamic data flow graph of the literals in the clause body by an ordering algorithm. OR parallelism is achieved by adding special Synchronization signals to the stream of partial solutions and synchronizing the multiple streams with a merge algorithm.

In the past six months the Sync Model has been improved in several aspects: (1) Besides AND and OR parallelism, stream parallelism is also implemented in the model. Stream parallelism is the pipelining of the shared list data structure between two sibling AND processes. It is implemented by identifying the “stream variables” in the program automatically and transmitting the partial solutions of the stream variables with special communication protocol. (2) Tail recursion is optimized in two ways: by using a *map* operator to transform a tail recursive clause into a set of parallel AND processes to preserve the inherent parallelism in the definition, and by looping back and repeatedly using the same process for the recursive calls of the same goal. (3) The data flow graph is refined to avoid unnecessary computations. “Selective channels” are added to the literals that have zero outdegree in the graph to transmit the input values that make the literals true. The refined graph is also maintained acyclic to prevent deadlock.

The performance of the mapping of the Sync Model onto the Sneptree is analyzed in terms of the load factor and the communication overhead. The simulation results show that the *Exchange Sneptree* is one of the best connection patterns for the Sync Model, where the Exchange Sneptree is a Sneptree in which all the outgoing links of the leaf nodes in the left half are directed to the incoming links of the nodes in the right half, and vice versa. We also found that load balancing heuristics do not improve the performance of the mapping. After comparing this mapping with the mapping of an arbitrary tree onto other networks, such as a mesh or a binary n -cube, we have found that the Sneptree is indeed the ideal architecture for the Sync Model.

Full details can be found in “A Parallel Execution Model for Logic Programming”, Peggy Li’s Ph.D thesis (in preparation).

3.2 A descriptive model of computing agents

Kevin S. Van Horn, Alain J. Martin

One indication of how well we understand some phenomenon is our ability to provide an adequate mathematical model of it. Such a model provides a firm basis for reasoning about the phenomenon; in its absence we are vulnerable to the treacheries of informal reasoning, and we are severely limited in our ability to analyze complex instances of the phenomenon. What is the appropriate *general* model of computing systems? This model should be applicable to both hardware and software systems, and must of course be applicable to concurrent systems. As a candidate for this role, we have developed a general model of computing agents, called Complete and Infinite Traces, or CIT.

We take the view that a computing agent is an object which may perform various *actions*, thus exhibiting some discrete *behavior*, which may be influenced by the actions of other agents. In the case of a digital circuit the relevant actions are voltage transitions. A computer program may perform such actions as assigning a value to a variable or writing a character to the user's screen, and may respond to actions such as the user hitting a key on the keyboard.

The mathematical structure used to model a computing agent is called a *process*, and is a set of traces (sequences of actions) plus an input and an output alphabet (sets of actions performable by exterior entities or the agent itself, respectively). If we imagine an observer who records the entire sequence of actions which are performed during the operation of the computing agent, every such sequence which he might record (called a *complete* trace) will be in the trace set of the process modeling the agent. If the agent has a cyclic behavior and may continue to operate forever, then the trace set will contain infinite traces.

To resolve some difficulties associated with infinite traces, we assume that all processes are either *sequential* (i.e., they model an agent which performs its output actions one at a time) or may be formed as the parallel composition of a set of sequential processes. Since we use recursion only in defining sequential processes, we may define parallel composition using the weave operator from trace theory [1] and thus avoid the problems associated with defining parallel composition as some form of "fair merge".

A detailed presentation of this model of computing agents may be found in [5207:TR:86].

At present, CIT is being applied to the semantics of concurrent programming languages. Concurrently (of course), compositional techniques for verifying such programs are being investigated, based on CIT.

Reference:

- [1] J. L. A. van de Snepscheut. *Lecture Notes in CS, vol. 200: Trace Theory and*

3.3 Computation Flow

William C. Athas, Chuck Seitz

Research into compilation techniques for concurrent message passing computers (ensemble machines) has progressed from the definition and demonstration of a high level programming notation for describing concurrent objects [5196:TR:85] to research into program flow analysis. The program flow analysis is based upon a finite state process model for describing the basic or primitive behavior of a computation. The processes are capable of sending messages and spawning new processes. The finite state process shares many properties with the actor model of computation, although for our purposes it has been specifically tailored for program flow analysis. As a result of basing the program flow framework on this model, two additional frameworks of future and after flow are derived.

Future flow allocates new processes (objects) before the processes are actually needed by a computation. This allocation can be performed statically at compile time to build an initial process graph for placement on the ensemble machine, or can be done concurrently with the computation. By concurrently performing future flow analysis with program execution, the interval between the specification of a process and its subsequent use is increased. This additional time may be used advantageously in finding better locations for new processes with respect to load balancing and message traffic.

After flow is akin to the task of garbage collection as performed in LISP systems. Whereas future flow attempts to identify processes that are to be used in the future of a computation,, after flow identifies those processes that are still relevant to a computation, though not necessarily active. By determining which processes are relevant, or other processes are thereby irrelevant to the outcome of the computation and can be safely reclaimed. Like future flow, after flow can be concurrent with program flow. The concurrent execution of all three is called computation flow.

3.4 Concurrent Simulated Annealing

Craig S. Steele, Chuck Seitz

Preliminary experiments indicate that the previously reported simulated annealing technique, used for process placement, is tolerant of minor database inconsistency. Some such "staleness" of data is likely in a highly concurrent computer due to communications latency. A concurrent implementation of the process placement optimization program is being designed, and is targeted for the Intel iPSC. In addition to the goals of greatly increased optimization speed and larger maximum problem size, it is intended that the program should be written in such a way

that it may improve its own communications performance by optimizing its own placement.

With the above goals in mind, a “quasi-static” programming environment (QSE) is being designed. This environment combines medium-sized data structures with C code operator modules, similar to small subroutines. Programs are written by specifying the logical connection between specific code and data components. Messages required to update remote copies of data are generated by the runtime system rather than by the programmer’s explicit use of *send* services.

This approach removes the low-level details of message addressing from the code and allows relocation of computation components during execution. Debugging is eased because most components of system state are externalized and visible to the debugger via the runtime environment. Success of the QSE will provide a convenient environment for scientific programming applications based on iterative methods, and should be a significant step in the evolution of a fully dynamic concurrent programming model.

3.5 Event-driven Simulation on Message-Passing Concurrent Computers

Wen-King Su, Chuck Seitz

Event-driven simulation is an important class of computations in which different parts of the system being simulated interact by scheduling events to take place at some time in the future. Each event may in turn cause more events to be scheduled at a later time. On sequential computers the events are performed in chronological order by a scheduler that maintains a single forward moving clock.

A subset of such simulations map extremely well onto message-passing concurrent computers. This subset includes simulations in which the system being simulated includes a sufficiently large number of elements and activities going on at once to keep a concurrent machine busy. On a large concurrent machine, however, maintaining a universal clock is difficult. We are currently exploring an approach to event-driven simulation on message-passing concurrent computers in which the elements are connected by logical channels through which the events are delivered in the form of messages.

The basic technique, which is closely related to the distributed simulations algorithm suggested by Chandy and Misra, and by Bryant, is that an element can produce its outputs up to a future time $t + s$ if it can predict its output s time units in the future based on the inputs up to time t . Any element that does not respond instantaneously – which includes all physically realizable elements, has this property. In such a simulation the available slack, s , contributed by the elements is used increase the performance of the simulation. Messages are tagged with times, and all scheduling is local. The algorithm permits different elements of the simulation to get as much out of step as is permitted by their mutual dependencies, thus allowing

a maximal number of elements to be scheduled for evaluation while not doing any redundant work.

A prototype simulator has been written and is currently giving the cosmic cubes a real workout. We are using the prototype simulation package to evaluate the performance of alternative network topologies and routing schemes under several program models. These simulations are part of our efforts at designing second generation message-passing systems (see section 2.3).

3.6 Interconnection Network Topology

William J. Dally, Chuck Seitz

A comparison of the latency of k -ary n -cube networks as a function of dimension gives the surprising result that, holding wiring bisection width constant, relatively low dimension networks (*e.g.*, the 2-dimensional torus) achieve lower latency than do high-dimensional networks (*e.g.*, binary n -cubes). The minimum latency is achieved when the component of latency due to message length is approximately equal to the component of latency due to distance. For networks of fewer than 1000 processing nodes, minimum latency is achieved by two-dimensional networks.

3.7 Message-Driven Processor

William J. Dally

A message-driven processor reduces message latency by directly executing messages rather than interpreting them with a series of instructions. In a concurrent computer built around a conventional instruction processor, interpreting a message is a time-consuming process. The processor must respond to an interrupt, save its state, fetch the message, look up the method, and finally (after executing about 100 instructions) execute the method.

Instead of nesting the fetch-decode-execute loop of a conventional instruction processor inside the receive-dispatch-execute loop required to process a message, a message-driven processor directly interprets messages. The reception and buffering of messages is performed by hardware so that no interrupt is required to receive a message. Method lookup is accelerated by using an instruction-translation lookaside buffer. Primitive methods are executed directly by the processor as if they were instructions. Defined methods are executed by creating a context object and performing a series of *sends*.

Current work involves specifying the architecture of a message-driven processor and constructing a simulation model of this architecture. The goal of this work is to integrate a message-driven processor with a TRC-like communications device and memory on a single chip to form a building block for a future fine-grain concurrent computers.

3.8 Concurrent Smalltalk

William J. Dally

The Concurrent Smalltalk (CST) programming language is being developed to combine the data abstraction and late binding features of Smalltalk-80 with the message-passing semantics of actor languages. CST includes features that allow the programmer to create abstractions for concurrency. A key feature of this language is the ability to specify *distributed objects*. The state of a distributed objects is distributed among a collection of *constituent objects* each of which can receive messages on behalf of the distributed object. Thus distributed objects can perform many operations simultaneously. They are the foundation upon which concurrent data structures are built.

3.9 A General Purpose Concurrent Architecture

John Y. Ngai, Chuck Seitz

This research, still in its very early stages, is an investigation of computing structures that can support both concurrent and general purpose computing. The ideas involved have been discussed in the research group under the heading of "soft computers", and are based on programming arrays of very small computing elements in a way that is specializes the ensemble to interpret a program representation that is tailored to the program itself.

The idea is also akin to that of silicon compilation. Instead of compiling the high level description of a computation directly into physical circuits elements such as wires and transistors, it would be translated into processes or objects, each accomplishing a certain part of the computation being compiled. The analogies between component or cell placement and process placement to achieve locality of communication is clear. These processes execute concurrently on a fine grain computing mesh.

Certain elements of this idea are very old, and were investigated by John von Neumann [1] and Arthur Burks [2] in the late 1950s in their study of cellular automata theory. One can build fine grain structures that can be programmed and composed together to form larger architectural blocks. Exactly how much concurrency one can achieve depends on the algorithmic formulation of the computation. As with XCPL [5196:TR:85], the source text is necessarily concurrent, and the compilation can at best preserve the concurrencies in execution.

Our current investigation focuses on finding suitable *soft* fine grain structures used in composition, and the corresponding simulation techniques. It appears, as in [1], that the ability to form communication paths is a necessary primitive. Future work will include a study of compilation techniques and composition rules, especially those needed to support an object-oriented programming style.

References:

- [1] John von Neumann, *Theory of Self-Reproducing Automata*, Urbana, Univ. of Illinois Press, 1966.
- [2] Arthur W. Burks, *Programming and the Theory of Automata, Essays on Cellular Automata*, Univ. of Illinois Press, 1970.

4. VLSI Design

4.1 Mosaic A Elements

Don Speck, Chuck Seitz

The following narrative description of the final stages of a long-term effort on the Mosaic A elements may be of interest to other VLSI program contractors who have gone through or expect to go through similar experiences.

As discussed in previous semi-annual technical reports, the two major parts of the Mosaic A element, the processor and dynamic RAM, were fabricated and tested separately. The processor gave us little trouble, but the RAM, with its exotic circuit design style, required many fabrication runs and extensive circuit simulation to achieve a robust design.

With the major parts designed, there remained only to design a ROM and to integrate those parts into a single chip. The parts all hang off of a central address/data bus, so predictably enough, that is where what few interface problems there were showed up. In designing the address drivers, it eventually became clear that the highly capacitive address lines – forced by the RAM layout to run more than twice the length of the chip – could never rise as fast as the ROM design had assumed. Timing pressure from the slow address inputs forced the ROM decoder to mutate into the same extreme hot-clock design style as the RAM decoder, sacrificing some of the desired implementation independence of the ROM from the more suspect RAM.

Since Mosaic elements are stored-program computers, they must have an initial program, contained in the ROM. A bootstrap loader is the minimum, but if any subsystem proves inoperative, the loader will likewise be inoperative, and it becomes impossible to load any diagnostic programs. In particular, we did not want to depend on the RAM, the most process-sensitive part of the chip, in performing RAM tests. Therefore, the diagnostic programs, particularly the RAM test, must already reside in the ROM if one hopes to gain any diagnostic information from partially-working chips.

In hopes of getting some early feedback on yield, a “snapshot” of the evolving fully assembled chip was sent to MOSIS even before full simulations could be attempted. In this “snapshot” version, Mosaic 1.0, the ROM code performs a purposely simplified port handshake (designed to be satisfied by a passive pullup), computes the ROM checksum, scans the RAM for errors, and reports both results before attempting to download a program. In this way the chip provides enough information to forecast yield, even if it proves unable to download any programs.

Subsequent simulation revealed that the “snapshot” version was, in fact, too defective to download a program, but still able to run its ROM/RAM tests. The downloader routine contained an error despite the simple-and-safe coding style, and

an inverter was omitted from the input ports, causing the flow control circuits to ground out the input. The latter defect could be bypassed for test purposes by shorting the input to Vdd, in place of the passive pullup.

The first batch of these chips received from MOSIS yielded 4 of 7 chips with all 96 words of ROM and 512 words of RAM working correctly. Thus the "snapshot" and the paranoia in the test code were well justified. However, the chips only worked over a limited range of clock amplitudes (4.1 volts \pm 5%) and clock speeds (1.5 to 5 MHz). These same limitations were characteristic of an earlier version of the RAM, and may indicate some residual process sensitivity even in our improved version. Future fabrication results may help explain these test results.

In the meanwhile, we have gone through the extraction, switch-level simulation, and bug-fixing cycle through fabricated versions 1.2, 1.3, and 1.4 with the 1.3 and 1.4 versions being free of any known bugs. The full suite of switch-level simulation tests with Mossim II requires more than 20 days on a VAX11/750.

The ROM code in Mosaic 1.3 & 1.4 has been made flexible enough to use for diagnosis, testing, and program loading. The size has quadrupled with the addition of specific tests for every subsystem except the microcode. Together, the tests are so comprehensive that they diagnosed 4 bugs in the instruction simulator. None of those bugs exist in the chip itself, as determined by simulation with Mossim II.

The Mosaic 1.3 is expected to be producible in quantity. It has 768 words (1.5K bytes) of RAM, 392 words of ROM, 65,000 transistors, a die of 4.6 by 7.1 mm² (including the MOSIS test structures) in a 3 μ m MOSIS nMOS process, and 17 pads. The pads are all located on the 4.6mm edges, so that the chip fits perfectly in a standard 18-pin dRAM package.

The use of a 2D mesh connection allows very simple packaging of Mosaic systems on PCBs with 256 node elements and 32 clock driver chips per board.

4.2 The Torus Routing Chip

William Dally, Chuck Seitz

First silicon of the Torus Routing Chip (TRC) was received in December. These chips functioned properly; however performance was below expectations. Channel period was measured to be 250ns and channel delay 150ns. Even with this disappointing performance – due to an oversight in the location of the critical timing path – the TRC offers a substantial improvement in performance over existing communication networks. For example in a 1024 processor 32-ary 2-cube a message across half the diameter of the machine crosses 31 channels with a latency of only 4.65 μ s (31 \times 150ns). In contrast, store and forward networks used by machines such as the cosmic cube have single channel latencies in the order of a millisecond. The torus routing chip reduces these millisecond latencies to microseconds, making possible much finer-grain concurrency.

Continuing work aims at improving the performance of the TRC. A $1.4\mu\text{m}$ feature size version of the TRC was submitted for fabrication in January, and a redesign of the TRC to improve performance and function is in progress.

A preprint of a paper describing the Torus Routing Chip is reproduced as an appendix to this report.

4.3 Mosaic C

Bill Athas, Lounette Dyer, Fritz Nordby, Steve Rabin, Don Speck, Wen-King Su, Chuck Seitz, and the 1985-86 Caltech VLSI Design Class

A new version of the Mosaic element, called Mosaic C, was started in January 1986. The target technology is MOSIS SCMOS with $0.6\mu\text{m} \leq \lambda \leq 1.5\mu\text{m}$. The architecture is discussed in section 2.2, and some of the rationale behind the unusual features of the element is found in sections 2.4, 3.3, 3.6, and 4.2.

A first cut logic design and layout is being done by the 32 "hard-core" members of the full-year VLSI design class, all of whom have already done projects in the fall quarter. The design effort is being led by seven of our most experienced chip designers. The class has been broken into four groups, each group led by two experienced designers:

RAM/ROM: (Group leaders: Don Speck, Fritz Nordby) This group has developed designs and done extensive simulations for 2-, 3-, and 4-transistor dynamic RAMs. The 2T RAM cell, essentially two 1T cells written with complementary data and sensed differentially, is both the densest possibility and – because it uses only one contact per bit – is likely to have the highest yield. The density, including decoding circuits, is about $400\lambda^2$ per bit. For this RAM we are aiming for medium speed, approximately 10 MHz operation at $3\mu\text{m}$ feature size, and relatively low (dynamic) power consumption.

Channels: (Group leaders: Steve Rabin, Chuck Seitz) The channels group has had to start at a higher level of design than the other groups, in that it was decided for architectural reasons that the Mosaic C channels should be substantially different from the Mosaic A channels. The channels designed are memory mapped: they appear to the programmer as special storage locations. Each channel is bidirectional, synchronous, and bit serial, with one wire in each direction. Unlike the Mosaic A channels, the wires are always driven. The flow control unit (flit) is one Mosaic word, and there are two virtual channels per physical channel. Each transmission starts with a lead bit, followed by a bit to distinguish a message from an acknowledge, and followed by a bit to specify the virtual channel. If the transmission is a message rather than an acknowledge, this preamble is followed by a tail

bit, one or more programmable control bits, and 16 bits of data. The channel interface also includes a hardware assist for wormhole and cut-through routing.

Datapath: (Group leaders: Wen-King Su, Chuck Seitz) The processor datapath has the same general floorplan as that in the Mosaic B, but in SCMOS rather than n MOS, it quite different in its logical design and layout. The bit pitch is a generous 60λ , so that the 16-bit datapath is 960λ high. At this bit pitch the datapath is approximately 2500λ wide. The datapath employs a number of very nice logic tricks, including a double-rail carry chain to allow the precharged shift network to be cascaded with the precharged adder. Since the Mosaic C processor is a “two-sequence” machine, it has two program counters and two sets of registers.

Microcode: (Group leaders: Bill Athas, Lounette Dyer) The microcode group is using the Mosaic B microcode as a model, and have devoted their initial efforts to developing simulators to be able to test the somewhat more complex microcode to operate the Mosaic C.

We expect to finish all of the Mosaic C parts by June, and plan to fabricate the processor, RAM & ROM, and channel sections independently before combining them onto one chip.

4.4 SCMOS Pad Designs

Fritz Nordby, Chuck Seitz

All SCMOS chip projects need pads. The designs available six months ago, developed at MIT, were perceived as deficient in several respects, including inadequate or misdirected static protection structures, improperly sized output transistors, and over-large real estate requirements. A new set of SCMOS pads was designed to fill a perceived need.

The first versions of these pads were designed for $3.0\mu\text{m}$ SCMOS. These pads employ a polysilicon resistor for input protection, and output transistors sized to drive a 10nH lead inductance into a 50pF load without ringing. These pads have been fabricated both in a test configuration and in a larger chip (the TRC). The pads function properly in both cases; their performance is now being characterized, and will be used to evaluate possible design changes.

In addition to the $3.0\mu\text{m}$ pads, a set of pads were developed for what Ms. Mosis refers to as “1.2 micron” SCMOS, that is, the technology for which $\lambda = 0.7\mu\text{m}$. In this case, the primary concern was to reduce the pad pitch to avoid wasting chip area in pad-limited designs. The pads can be arranged in two rows with staggered pad centers, a pattern fairly commonly used in industry. Also, a different static

protection strategy was employed in this design, since the space available to these structures is considerably smaller than in the case of a single row of pads. These pads are currently being fabricated as part of the $1.4\mu\text{m}$ (sorry, $1.2\mu\text{m}$) version of the TRC.

4.5 Compiling Programs into Self-timed VLSI Circuits

Steve Burns, Alain J. Martin, Kevin Van Horn

As previously reported, we are developing a method for deriving self-timed VLSI circuits from a high-level description of a computation – a set of communicating processes. An overview of the method can now be found in: A J Martin, “Compiling communicating processes into delay-insensitive VLSI circuits”, *Journal of Distributed Computing*, vol.1, no.3, 1986, a preprint of which is reproduced in the appendix to this report.

Recently we have found a solution to the delay-insensitive fair arbiter problem that is simpler than the one described in [5193:TR:85]. This new solution also presents the important advantage of being readily extensible to an arbitrary number of requesting processes. The solution can be found in [5219:TR:86].

California Institute of Technology
Computer Science Department, 256-80
Pasadena CA 91125

Technical Reports
Spring 1986

Available from the Computer Science Department Library
Prices include postage and help to defray our printing and mailing costs.

Publication Order Form

If you wish to order any of the reports listed, complete this form and return it with your check or international money order (in U.S. dollars) payable to CALTECH. Prepayment is required for all materials.

___5216:TR:86	\$4.00	<i>Coling 80,</i> Bozena H. Thompson and Frederick B. Thompson
___5215:TR:86	\$2.00	<i>How to Get a Large Natural Language System into a Personal Computer,</i> Thompson, Bozena H. and Frederick B. Thompson
___5214:TR:86	\$2.00	<i>ASK is Transportable in Half a Dozen Ways,</i> Tompson, Bozena H. and Frederick B. Thompson
___5208:TR:86	\$2.00	<i>The Torus Routing Chip,</i> Dally, William and Charles L Seitz
___5207:TR:86	\$2.00	<i>Complete and Infinite Traces: A Descriptive Model of Computing Agents,</i> van Horn, Kevin
___5206:TR:86	\$2.00	<i>Deadlock Free Message Routing in Multiprocessor Interconnection Networks,</i> Dally, William J, and Charles L Seitz
___5205:TR:85	\$2.00	<i>Two Theorems on Time Bounded Kolmogorov-Chaitin Complexity,</i> Schweizer, David and Yaser Abu-Mostafa
___5204:TR:85	\$3.00	<i>An Inverse Limit Construction of a Domain of Infinite Lists,</i> Choo, Young-Il
___5203:TR:85	\$9.00	<i>C Programmer's Guide to the Cosmic Cube,</i> Su, Wen-King, Reese Faucette and Chuck Seitz
___5202:TR:85	\$15.00	<i>Submicron Systems Architecture,</i> ARPA Semiannual Technical Report
___5200:TR:85	\$18.00	<i>ANIMAC: A Multiprocessor Architecture for Real-Time Computer Animation,</i> PhD thesis Whelan, Dan
___5198:TR:85	\$8.00	<i>Neutral Networks, Pattern Recognition and Fingerprint Hallucination,</i> PhD thesis Mjølness, Eric
___5197:TR:85	\$7.00	<i>Sequential Threshold Circuits,</i> MS thesis Platt, John
___5196:TR:85	\$5.00	<i>ECL: An Experimental Concurrent Language,</i> Athas, Bill
___5195:TR:85	\$3.00	<i>New Generalization of Dekker's Algorithm for Mutual Exclusion,</i> Martin, Alain J
___5194:TR:85	\$5.00	<i>Sneptree - A Versatile Interconnection Network,</i> Li, Pey-yun Peggy and Alain J Martin
___5193:TR:85	\$4.00	<i>Delay-Insensitive Fair Arbiter,</i> Martin, Alain J
___5190:TR:85	\$3.00	<i>Concurrency Algebra and Petri Nets,</i> Choo, Young-il
___5189:TR:85	\$10.00	<i>Hierarchical Composition of VLSI Circuits,</i> PhD Thesis Whitney, Telle
___5185:TR:85	\$11.00	<i>Combining Computation with Geometry,</i> PhD Thesis Lien, Sheue-Ling

Caltech Computer Science Technical Reports

- 5184:TR:85 \$7.00 *Placement of Communicating Processes on Multiprocessor Networks*, Ms Thesis
Steele, Craig
- 5178:TR:85 \$9.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report
- 5177:TR:85 \$4.00 *Hot-Clock nMOS*, Proc. 1985 Chapel Hill Conference on VLSI, pp 1-17
Seitz, Charles, A H Frey, S Mattisson, S D Rabin, D A Speck, and J L A van de Snepscheut
- 5174:TR:85 \$7.00 *Balanced Cube: A Concurrent Data Structure*,
Dally, William J and Charles L Seitz
- 5172:TR:85 \$6.00 *Combined Logical and Functional Programming Language*,
Newton, Michael
- 5168:TR:84 \$4.00 *Object Oriented Architecture*,
Dally, Bill and Jim Kajiya
- 5165:TR:84 \$4.00 *Customizing One's Own Interface Using English as Primary Language*,
Thompson, B H and Frederick B Thompson
- 5164:TR:84 \$13.00 *ASK French - A French Natural Language Syntax*, MS Thesis
Sanouillet, Remy
- 5160:TR:84 \$7.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report
- 5158:TR:84 \$6.00 *VLSI Architecture for Sound Synthesis*,
Wawrzynek, John and Carver Mead
- 5157:TR:84 \$15.00 *Bit-Serial Reed-Solomon Decoders in VLSI*, PhD Thesis
Whiting, Douglas
- 5148:TR:84 \$4.00 *Fair Mutual Exclusion with Unfair P and V Operations*,
Martin, Alain and Jerry Burch
- 5147:TR:84 \$4.00 *Networks of Machines for Distributed Recursive Computations*,
Martin, Alain and Jan van de Snepscheut
- 5143:TR:84 \$5.00 *General Interconnect Problem*, MS Thesis
Ngai, John
- 5140:TR:84 \$5.00 *Hierarchy of Graph Isomorphism Testing*, MS Thesis
Chen, Wen-Chi
- 5139:TR:84 \$4.00 *HEX: A Hierarchical Circuit Extractor*, MS Thesis
Oyang, Yen-Jen
- 5137:TR:84 \$7.00 *Dialogue Designing Dialogue System*, PhD Thesis
Ho, Tai-Ping
- 5136:TR:84 \$5.00 *Heterogeneous Data Base Access*, PhD Thesis
Papachristidis, Alex
- 5135:TR:84 \$7.00 *Toward Concurrent Arithmetic*, MS Thesis
Chiang, Chao-Lin
- 5134:TR:84 \$2.00 *Using Logic Programming for Compiling APL*, MS Thesis
Derby, Howard
- 5133:TR:84 \$13.00 *Hierarchical Timing Simulation Model for Digital Integrated Circuits and Systems*, PhD Thesis
Lin, Tzu-mu
- 5132:TR:84 \$10.00 *Switch Level Fault Simulation of MOS Digital Circuits*, MS Thesis
Schuster, Mike
- 5130:TR:84 \$3.00 *LOG The Chipmunk Logic Simulator User's Guide*,
Gillespie, Dave
- 5129:TR:84 \$5.00 *Design of the MOSAIC Processor*, MS Thesis
Lutz, Chris
- 5128:TM:84 \$3.00 *Linguistic Analysis of Natural Language Communication with Computers*,
Thompson, Bozena H
- 5125:TR:84 \$6.00 *Supermesh*, MS Thesis
Su, Wen-king

Caltech Computer Science Technical Reports

___5124:TR:84	\$4.00	<i>Probe: An Addition to Communication Primitives,</i> Martin, Alain
___5123:TR:84	\$14.00	<i>Mossim Simulation Engine Architecture and Design,</i> Dally, Bill
___5122:TR:84	\$8.00	<i>Submicron Systems Architecture,</i> ARPA Semiannual Technical Report
___5120:TM:84	\$1.00	<i>Mathematical Approach to Modeling the Flow,</i> Johnsson, Lennart and Danny Cohen
___5119:TM:84	\$1.00	<i>Integrative Approach to Engineering Data and Automatic Project Coordination,</i> Segal, Richard
___5118:TR:84	\$2.00	<i>SMART User's Guide,</i> Ngai, John
___5114:TM:84	\$3.00	<i>ASK As Window to the World,</i> Thompson, Bozena, and Fred Thompson
___5113:TR:84	\$4.00	<i>WoLery,</i> Mead, Carver A
___5112:TR:83	\$22.00	<i>Parallel Machines for Computer Graphics,</i> PhD Thesis Ulner, Michael
___5106:TM:83	\$1.00	<i>Ray Tracing Parametric Patches,</i> Kajiya, James T
___5105:TR:83	\$2.00	<i>Memory Management in the Programming Language ICL,</i> Wawrzynek, John
___5104:TR:83	\$9.00	<i>Graph Model and the Embedding of MOS Circuits,</i> MS Thesis Ng, Tak-Kwong
___5103:TR:83	\$7.00	<i>Submicron Systems Architecture,</i> ARPA Semiannual Technical Report
___5102:TR:83	\$2.00	<i>Experiments with VLSI Ensemble Machines,</i> Seitz, Charles L
___5101:TM:83	\$1.00	<i>Concurrent Fault Simulation of MOS Digital Circuits,</i> Bryant, Randal E
___5099:TM:83	\$1.00	<i>VLSI and the Foundations of Computation,</i> Mead, Carver
___5098:TM:83	\$2.00	<i>New Techniques for Ray Tracing Procedurally Defined Objects,</i> Kajiya, James T
___5097:TR:83	\$4.00	<i>Design of a Self-timed Circuit for Distributed Mutual Exclusion,</i> Martin, Alain J
___5094:TR:83	\$2.00	<i>Stochastic Estimation of Channel Routing Track Demand,</i> Ngai, John
___5093:TR:83	\$1.00	<i>Design of the MOSAIC Element,</i> Lutz, Chris, Steve Rabin, Chuck Seitz and Don Speck
___5092:TM:83	\$2.00	<i>Residue Arithmetic and VLSI,</i> Chiang, Chao-Lin and Lennart Johnsson
___5091:TR:83	\$2.00	<i>Race Detection in MOS Circuits by Ternary Simulation,</i> Bryant, Randal E
___5090:TR:83	\$9.00	<i>Space-Time Algorithms: Semantics and Methodology,</i> PhD Thesis Chen, Marina Chien-mei
___5089:TR:83	\$10.00	<i>Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circuits</i> Lin, Tzu-Mu and Carver A Mead
___5086:TR:83	\$4.00	<i>VLSI Combinator Reduction Engine,</i> MS Thesis Athas, William C Jr
___5084:TM:83	\$3.00	<i>Tree Machine: An Evaluation of Strategies for Reducing Program Loading Time,</i> Li, Pey-yun Peggy, and Lennart Johnsson

Caltech Computer Science Technical Reports

- ___5082:TR:83 \$10.00 *Hardware Support for Advanced Data Management Systems*, PhD Thesis
Neches, Philip
- ___5081:TR:83 \$4.00 *RTsim - A Register Transfer Simulator*, MS Thesis
Lam, Jimmy
- ___5080:TR:83 \$4.00 *Distributed Mutual Exclusion on a Ring of Processes*,
Martin, Alain
- ___5079:TR:83 \$2.00 *Highly Concurrent Algorithms for Solving Linear Systems of Equations*,
Johnsson, Lennart
- ___5078:TR:83 \$5.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report
- ___5075:TR:83 \$2.00 *General Proof Rule for Procedures in Predicate Transformer Semantics*,
Martin, Alain
- ___5074:TR:83 \$10.00 *Robust Sentence Analysis and Habitability*,
Trawick, David
- ___5073:TR:83 \$12.00 *Automated Performance Optimization of Custom Integrated Circuits*, PhD Thesis
Trimberger, Steve
- ___5068:TM:83 \$1.00 *Hierarchical Simulator Based on Formal Semantics*, Proc Third Caltech Conf on VLSI
Chen, Marina and Carver Mead
- ___5065:TR:82 \$3.00 *Switch Level Model and Simulator for MOS Digital Systems*,
Bryant, Randal E
- ___5055:TR:82 \$5.00 *FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems*, MS Thesis
Ng, Charles H
- ___5054:TM:82 \$3.00 *Introducing ASK, A Simple Knowledgeable System*, Conf on App'l Natural Language Processing
Thompson, Bozena H and Frederick B Thompson
- ___5052:TR:82 \$8.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report
- ___5051:TM:82 \$2.00 *Knowledgeable Contexts for User Interaction*, Proc Nat'l Computer Conference
Thompson, Bozena, Frederick B Thompson, and Tai-Ping Ho
- ___5047:TR:82 \$3.00 *Torus: An Exercise in Constructing a Processing Surface*, Proc 2nd Caltech Conference on VLSI
Martin, Alain
- ___5046:TR:82 \$3.00 *Axiomatic Definition of Synchronization Primitives*, Acta Informatica 16, pp 219-235 (1981)
Martin, Alain
- ___5045:TM:82 \$3.00 *Distributed Implementation Method for Parallel Programming*, Proc Information Processing '80
Martin, Alain
- ___5044:TR:82 \$10.00 *Hierarchical Nets: A Structured Petri Net Approach to Concurrency*,
Choo, Young-Il
- ___5038:TM:82 \$4.00 *New Channel Routing Algorithm*,
Chan, Wan S
- ___5035:TR:82 \$9.00 *Type Inference in a Declarationless, Object-Oriented Language*, MS Thesis
Holstege, Eric
- ___5034:TR:82 \$12.00 *Hybrid Processing*, PhD Thesis
Carroll, Chris
- ___5033:TR:82 \$4.00 *MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual*,
Schuster, Mike, Randal Bryant and Doug Whiting
- ___5029:TM:82 \$4.00 *POOH User's Manual*,
Whitney, Telle
- ___5021:TR:82 \$5.00 *Earl: An Integrated Circuit Design Language*, MS Thesis
Kingsley, Chris
- ___5018:TM:82 \$2.00 *Filtering High Quality Text for Display on Raster Scan Devices*,
Kajiya, Jim and Mike Ullner
- ___5017:TM:82 \$2.00 *Ray Tracing Parametric Patches*,
Kajiya, Jim

Caltech Computer Science Technical Reports

___5016:TR:82	\$4.00	<i>Bristle Blocks - Scrutinized and Analyzed,</i> McNair, Richard and Monroe Miller
___5015:TR:82	\$15.00	<i>VLSI Computational Structures Applied to Fingerprint Image Analysis,</i> Megdal, Barry
___5014:TR:82	\$15.00	<i>Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture,</i> PhD Thesis Lang, Charles R Jr
___5012:TM:82	\$2.00	<i>Switch-Level Modeling of MOS Digital Circuits,</i> Bryant, Randal
___5001:TR:82	\$2.00	<i>Minimum Propagation Delays in VLSI ,</i> IEEE J Solid State Circuits Mead, Carver, and Martin Rem
___5000:TR:82	\$6.00	<i>Self-Timed Chip Set for Multiprocessor Communication,</i> MS Thesis Whiting, Douglas
___4777:TR:82	\$7.00	<i>Techniques for Testing Integrated Circuits,</i> PhD Thesis DeBenedictis, Erik P
___4724:TR:82	\$2.00	<i>Concurrent, Asynchronous Garbage Collection Among Cooperating Processors,</i> Lang, Charles R
___4716:TM:82	\$4.00	<i>Rectangular Area Filling Display System Architecture,</i> Whelan, Dan
___4684:TR:82	\$3.00	<i>Characterization of Deadlock Free Resource Contentions,</i> Chen, Marina, Martin Rem, and Ronald Graham
___4675:TR:81	\$7.00	<i>Switching Dynamics,</i> MS Thesis Lewis, Robert K
___4655:TR:81	\$20.00	<i>Proc Second Caltech Conf on VLSI,</i> Seitz, Charles, ed.
___4654:TR:81	\$12.00	<i>Versatile Ethernet Interface,</i> MS Thesis Whelan, Dan
___4653:TR:81	\$10.00	<i>Toward A Theorem Proving Architecture,</i> MS Thesis Lien, Sheue-Ling
___4618:TM:81	\$5.00	<i>Tree Machine Operating System,</i> Li, Peggy
___4600:TM:81	\$3.00	<i>Notation for Designing Restoring Logic Circuitry,</i> Proc Second Caltech Conf on VLSI Rem, Martin, and Carver Mead
___4530:TR:81	\$20.00	<i>Silicon Compilation,</i> PhD Thesis Johannsen, Dave
___4527:TR:81	\$11.00	<i>Communicative Databases,</i> PhD Thesis Yu, Kwang-I
___4521:TR:81	\$8.00	<i>Lambda Logic,</i> MS Thesis Rudin, Leonid
___4517:TR:81	\$7.00	<i>Serial Log Machine,</i> MS Thesis Li, Peggy
___4407:TM:82	\$3.00	<i>Experimental Composition Tool,</i> Mosteller, Richard C
___4332:TR:81	\$3.00	<i>RLAP, Version 1.0, A Chip Assembly Tool,</i> Mosteller, R
___4320:TR:81	\$7.00	<i>Hierarchical Design Rule Checker,</i> MS Thesis Whitney, Telle
___4317:TR:81	\$10.00	<i>REST - A Leaf Cell Design System,</i> MS Thesis Mosteller, Richard C
___4298:TR:81	\$7.00	<i>From Geometry to Logic,</i> MS Thesis Lin, Tzu-mu
___4204:TR:78	\$8.00	<i>16-Bit LSI Digital Multiplier,</i> EE Thesis Masumoto, R T

Caltech Computer Science Technical Reports

- ___4191:TR:81 \$4.00 *Towards A Formal Treatment of VLSI Arrays*, Proc Second Caltech Conf on VLSI
Johnsson, Lennart S, Uri Weiser, D Cohen, and Alan L Davis
- ___4128:TM:81 \$2.00 *Shifting to a Higher Gear in a Natural Language System*,
Thompson, Fred and B Thompson
- ___4116:TR:79 \$25.00 *Toward A Mathematical Theory of Perception*, PhD Thesis
Kajiya, Jim
- ___3975:TM:80 \$3.00 *Rapidly Extendable Natural Language*,
Thompson, B H and Fred B Thompson
- ___3762:TR:80 \$8.00 *Software Design System*, PhD Thesis
Hess, Gideon
- ___3761:TR:80 \$7.00 *Fault Tolerant Integrated Circuit Memory*, PhD Thesis
Barton, Tony
- ___3760:TR:80 \$10.00 *Tree Machine: A Highly Concurrent Computing Environment*, PhD Thesis
Browning, Sally
- ___3759:TR:80 \$10.00 *Homogeneous Machine*, PhD Thesis
Locanthi, Bart
- ___3710:TR:80 \$10.00 *Understanding Hierarchical Design*, PhD Thesis
Rowson, James
- ___3364:TR:79 \$8.00 *Stack Data Engine*,
Efland, G and R C Mosteller
- ___2276:TM:78 \$12.00 *Language Processor and a Sample Language*,
Ayes, Ron

Please fill in your name, address and amount enclosed below:

name _____

Address _____

City _____ State _____ Zip _____ Country _____

Amount enclosed \$ _____

_____ Please check here if you wish to be included on our mailing list

_____ Please check here for any change of address

Return this form to: Computer Science Library, 256-80, Caltech, Pasadena CA 91125

The Torus Routing Chip

William J. Dally
Charles L. Seitz

Computer Science Department
California Institute of Technology

5208:TR:86

January 24, 1986

To be published in *Journal of Distributed Computing* vol. 1, no. 3, 1986.

Abstract

The torus routing chip (TRC) is a self-timed chip that performs deadlock-free *cut-through* routing in k -ary n -cube multiprocessor interconnection networks using a new method of deadlock avoidance called *virtual channels*. A prototype TRC with byte wide self-timed communication channels achieved on first silicon a throughput of 64Mbits/s in each dimension, about an order of magnitude better performance than the communication networks used by machines such as the Caltech Cosmic Cube or Intel iPSC. The latency of the cut-through routing of only 150ns per routing step largely eliminates message locality considerations in the concurrent programs for such machines. The design and testing of the TRC as a self-timed chip was no more difficult than it would have been for a synchronous chip.

The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597, in part by Intel Corporation, and in part by an AT&T Ph.D. fellowship.

Copyright © California Institute of Technology, 1986.

1 Introduction

Message-passing concurrent computers such as the Caltech Cosmic Cube [13] and Intel iPSC [6] consist of many processing *nodes* that interact by sending messages over communication channels between the nodes. We designed the torus routing chip (TRC) as a building block to construct high-throughput, low-latency k -ary n -cube interconnection networks for message-passing concurrent computers.

The TRC is a self-timed VLSI circuit that provides deadlock-free packet communications in k -ary n -cube (torus) networks [12] with up to $k = 256$ processors in each dimension. While intended primarily for $n = 2$ -dimensional networks, the chips can be cascaded to handle n -dimensional networks using $\lceil \frac{n}{2} \rceil$ TRC chips at each processing node. A prototype TRC has been laid out, fabricated, and tested.

Even if only two dimensions are used, the TRC can be used to construct concurrent computers with up to 2^{16} nodes. It would be very difficult to distribute a global clock over an array of this size [4]. To avoid this problem, the TRC is entirely self-timed [11], thus permitting each processing node to operate at its own rate with no need for global synchronization. Synchronization, when required, is performed by arbiters in the TRC.

To reduce the latency of communications that traverse more than one channel, the TRC uses *cut-through* [7] routing rather than *store-and-forward* routing. Instead of reading an entire packet into a processing node before starting transmission to the next node, the TRC forwards each byte of the packet to the next node as soon as it arrives. Cut-through routing thus results in a message latency that is the *sum* of two terms, one of which depends on the message length, L , and other of which depends on the number of communication channels traversed, D . Store-and-forward routing gives a latency that depends on the product of L and D . Another advantage of cut-through routing is that communications do not use up the memory bandwidth of intermediate nodes. A packet does not interact with the processor or memory of intermediate nodes along its route. Packets remain strictly within the TRC network until they reach their destination.

The TRC uses *virtual channels* to perform deadlock-free routing in torus networks. By splitting each physical channel into two virtual channels and making routing dependent on the virtual channel on which a message arrives, the TRC converts the cycle of channel dependencies in each dimension into a spiral.

This paper describes the considerations that went into the design of the TRC in a “top-down” order that starts with a formal discussion of the deadlock problem in Section 2. We develop a model of communications in multiprocessor interconnection networks and prove a strong theorem about deadlock. Based on this model, the concept of virtual channels is presented in Section 3. Sections 4 and 5 present the design of the TRC at the system and logical levels. Experimental results are reviewed in Section 6.

2 Deadlock-Free Routing

Deadlock in the interconnection network of a concurrent computer occurs when no message

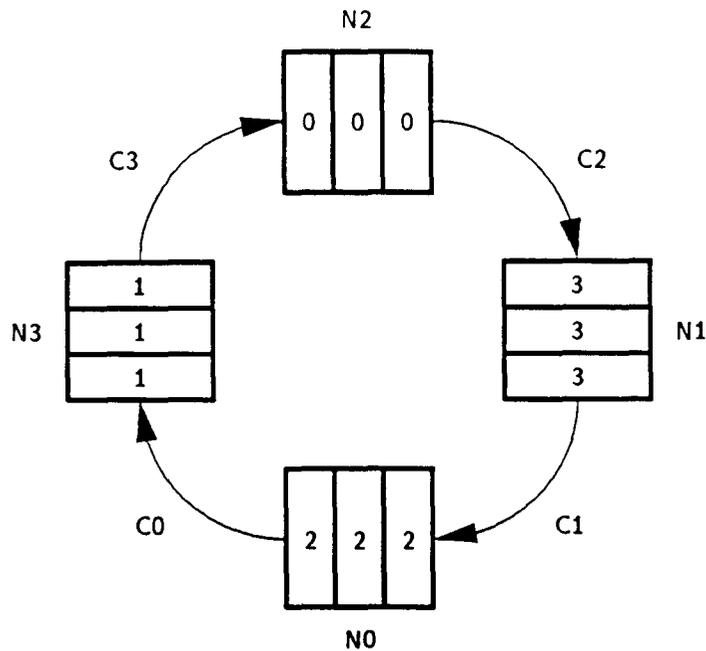


Figure 1: Deadlock in a 4-Cycle

can advance toward its destination because the queues of the message system are full [8]. Consider the example shown in Figure 1. The queues of each node in the 4-cycle are filled with messages destined for the opposite node. No message can advance toward its destination; thus the cycle is deadlocked. In this locked state, no communication can occur over the deadlocked channels until exceptional action is taken to break the deadlock.

The technique of virtual channels allows deadlock-free routing to be performed in any strongly-connected interconnection network [2]. This technique involves splitting physical channels on cycles into multiple virtual channels and then restricting the routing so the dependence between the virtual channels is acyclic.

Definition 1 A flow control digit or *flit* is the smallest unit of information that a queue or channel can accept or refuse. Generally a *packet* consists of many flits. Each packet carries its own routing information.

We have adopted this complication of standard terminology to distinguish between those flow control units that always include routing information - viz. packets - and those lower level flow control units that do not - viz. flits. The literature on computer networks [16] has been able to avoid this distinction between packets and flits because most networks include routing information with every flow control unit; thus the flow control units are packets. That is not the case in the interconnection networks used by message-passing concurrent computers such as the Caltech Cosmic Cube [13].

We assume the following:

1. Every packet arriving at its destination node is eventually consumed.

2. A node can generate packets destined for any other node.
3. The route taken by a packet is determined only by its destination, and not by other traffic in the network.
4. A node can generate packets of arbitrary length. Packets will generally be longer than a single flit.
5. Once a queue accepts the first flit of a packet, it must accept the remainder of the packet before accepting any flits from another packet.
6. An available queue may arbitrate between packets that request that queue space, but may not choose amongst waiting packets.
7. Nodes can produce packets at any rate subject to the constraint of available queue space (source queued).

The following definitions develop a notation for describing networks, routing functions, and configurations.

Definition 2 An *interconnection network*, I , is a strongly connected *directed graph*, $I = G(N, C)$. The vertices of the graph, N , represent the set of processing nodes. The edges of the graph, C , represent the set of communication channels. Associated with each channel, c_i , is a queue with capacity $\text{cap}(c_i)$. The source node of channel c_i is denoted s_i and the destination node d_i .

Definition 3 A *routing function* $\mathbf{R} : C \times N \mapsto C$ maps the current channel, c_c , and destination node, n_d , to the next channel, c_n , on the route from c_c to n_d , $\mathbf{R}(c_c, n_d) = c_n$. A channel is not allowed to route to itself, $c_c \neq c_n$. Note that this definition restricts the routing to be memoryless in the sense that a packet arriving on channel c_c destined for n_d has no memory of the route that brought it to c_c . However, this formulation of routing as a function from $C \times N$ to C has more memory than the conventional definition of routing as a function from $N \times N$ to C . Making routing dependent on the current channel rather than the current node allows us to develop the idea of channel dependence. Observe also that the definition of \mathbf{R} precludes the route from being dependent on the presence or absence of other traffic in the network. \mathbf{R} describes strictly deterministic and non-adaptive routing functions.

Definition 4 A *channel dependency graph*, D , for a given interconnection network, I , and routing function, \mathbf{R} , is a directed graph, $D = G(C, E)$. The vertices of D are the channels of I . The edges of D , are the pairs of channels connected by \mathbf{R} :

$$E = \{(c_i, c_j) | \mathbf{R}(c_i, n) = c_j \text{ for some } n \in N\}. \quad (1)$$

Since channels are not allowed to route to themselves, there are no 1-cycles in D .

Definition 5 A *configuration* is an assignment of a subset of N to each queue. The number of flits in the queue for channel c_i will be denoted $\text{size}(c_i)$. If the queue for channel c_i contains a flit destined for node n_d , then $\text{member}(n_d, c_i)$ is true. A configuration is legal if

$$\forall c_i \in C, \text{size}(c_i) \leq \text{cap}(c_i). \quad (2)$$

Definition 6 A *deadlocked configuration* for a routing function, \mathbf{R} , is a non-empty legal configuration of channel queues such that

$$\forall c_i \in C, (\forall n \ni \text{member}(n, c_i), n \neq d_i \text{ and } c_j = \mathbf{R}(c_i, n) \Rightarrow \text{size}(c_j) = \text{cap}(c_j)) \quad (3)$$

In this configuration no flit is one step from its destination, and no flit can advance because the queue for the next channel is full. A routing function, \mathbf{R} , is *deadlock-free* on an interconnection network, I , if no deadlock configuration exists for that function on that network.

Theorem 1 A routing function, \mathbf{R} , for an interconnection network, I , is deadlock-free iff there are no cycles in the channel dependency graph, D .

Proof:

\Rightarrow Suppose a network has a cycle in D . Since there are no 1-cycles in D , this cycle must be of length two or more. Thus one can construct a deadlocked configuration by filling the queues of each channel in the cycle with flits destined for a node two channels away, where the first channel of the route is along the cycle.

\Leftarrow Suppose a network has no cycles in D . Since D is acyclic one can assign a total order to the channels of C so that if $(c_i, c_j) \in E$ then $c_i > c_j$. Consider the least channel in this order with a full queue, c_l . Every channel, c_n , that c_l feeds is less than c_l , and thus does not have a full queue. Thus, no flit in the queue for c_l is blocked, and one does not have deadlock. ■

3 Virtual Channels

Now that we have established this if-and-only-if relationship between deadlock and the cycles in the channel dependency graph, we can approach the problem of making a network deadlock-free by breaking the cycles. We can break such cycles by splitting each physical channel along a cycle into a group of *virtual channels*. Each group of virtual channels shares a physical communication channel; however, each virtual channel requires its own queue.

Consider for example the case of a unidirectional four-cycle shown in Figure 2A, $N = \{n_0, \dots, n_3\}$, $C = \{c_0, \dots, c_3\}$. The interconnection graph I is shown on the left and the dependency graph D is shown on the right. We pick channel c_0 to be the dividing channel of the cycle and split each channel into high virtual channels, c_{10}, \dots, c_{13} , and low virtual channels, c_{00}, \dots, c_{03} , as shown in Figure 2B.

Packets at a node numbered less than their destination node are routed on the high channels, and packets at a node numbered greater than their destination node are routed on the low channels. Channel c_{00} is not used. We now have a total ordering of the virtual channels

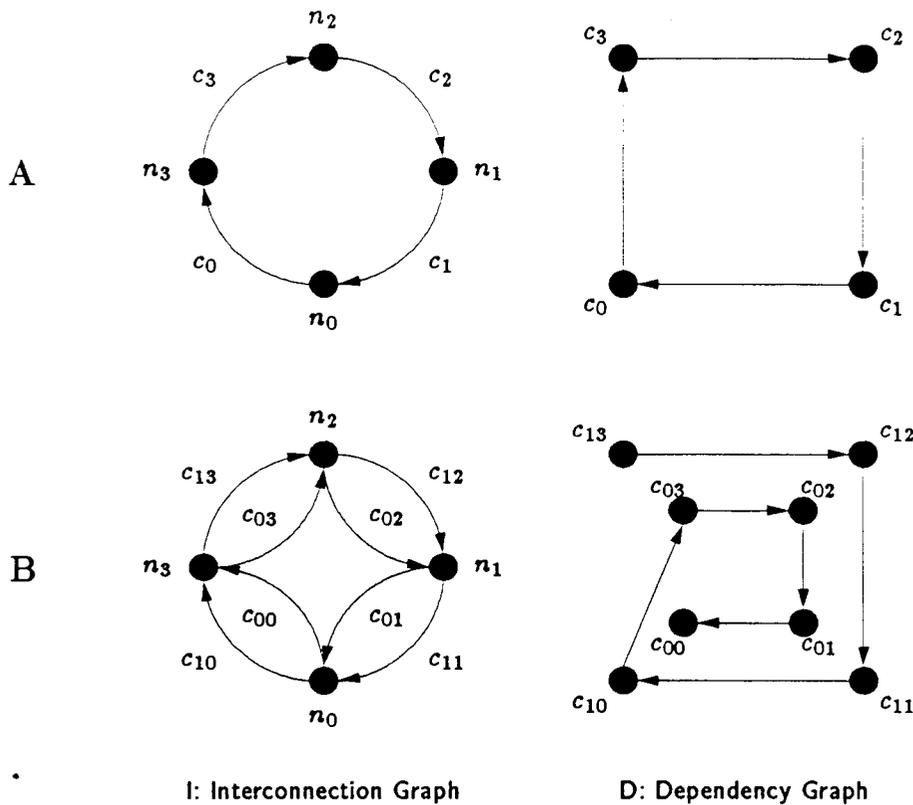


Figure 2: Breaking Deadlock with Virtual Channels

according to their subscripts: $c_{13} > c_{12} > c_{11} > c_{10} > c_{03} > c_{02} > c_{01}$. Thus, there is no cycle in D , and the routing function is deadlock-free. In [2] this technique is applied to construct deadlock-free routing functions for k -ary n -cubes, cube-connected cycles, and shuffle-exchange networks. In each case virtual channels are added to the network and the routing is restricted to route packets in order of decreasing channel subscripts. In the next two sections, the routing function for k -ary n -cubes is developed into a chip.

Many deadlock-free routing algorithms have been developed for store-and-forward computer communications networks [5]. These algorithms are all based on the concept of a *structured buffer pool*. The packet buffers in each node of the network are partitioned into classes, and the assignment of buffers to packets is restricted to define a partial order on buffer classes. The structured buffer pool method has in common with the virtual channel method that both prevent deadlock by assigning a partial order to resources. The two methods differ in that the structured buffer pool approach restricts the assignment of buffers to packets while the virtual channel approach restricts the routing of messages. Either method can be applied to store-and-forward networks, but the structured buffer pool approach is not directly applicable to cut-through networks, since the flits of a packet cannot be interleaved.

4 System Design

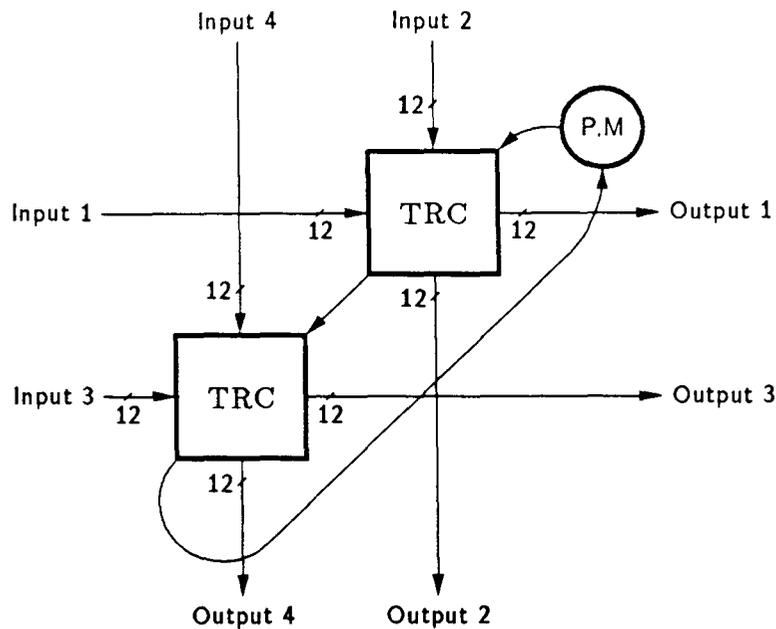


Figure 3: A Dimension 4 Node

The torus routing chip (TRC) can be used to construct arbitrary k -ary n -cube interconnection networks. Each TRC routes packets in two dimensions, and the chips are cascadable as shown in Figure 3 to construct networks of dimension greater than two. The first TRC in each node routes packets in the first two dimensions and strips off their address bytes before passing them to the second TRC. This next chip then treats the next two bytes as addresses in the next two dimensions and routes packets accordingly. The network can be extended to any number of dimensions.

A block diagram of a two-dimensional message-passing concurrent computer constructed around the TRC is shown in Figure 4. Each node consists of a processor, its local memory, and a TRC. Each TRC in the torus is connected to its processor by a processor input channel and a processor output channel. Connections on the edges of the torus wrap around to the opposite edge. One can avoid the long end-around connection by folding the torus, as shown in Figure 5.

A *flit* in the TRC is a byte whose 8 bits are transmitted in parallel. The X and Y channels each consist of 8 data lines and 4 control lines. The 4 control lines are used for separate request/acknowledge signal pairs for each of two virtual channels. The processor channels are also 8 bits wide, but have only two control lines each.

The packet format is shown in Figure 6. A packet begins with two address bytes. The bytes contain the relative X and Y addresses of the destination node. The relative address in a given direction, say X, is a count of the number of channels that must be traversed in the X direction to reach a node with the same X address as the destination. After the address comes the data field of the packet. This field may contain any number of *non-zero* data bytes. The packet is terminated by a zero tail byte. Later versions of the TRC may use an extra bit to tag the tail of a packet, and might also include error checking.

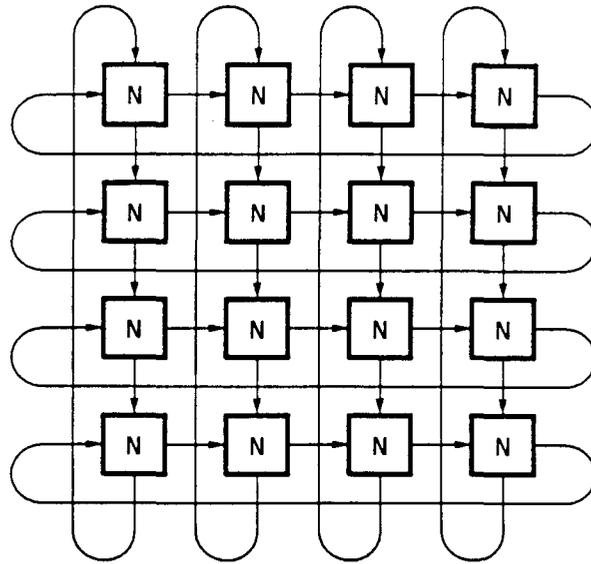


Figure 4: A Torus System

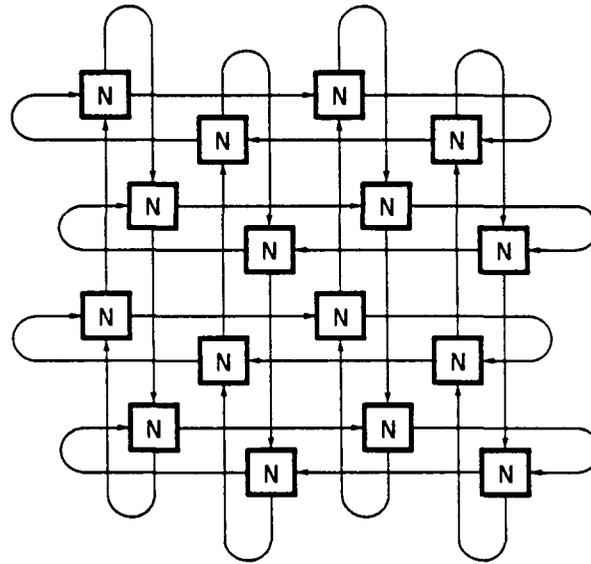


Figure 5: A Folded Torus System

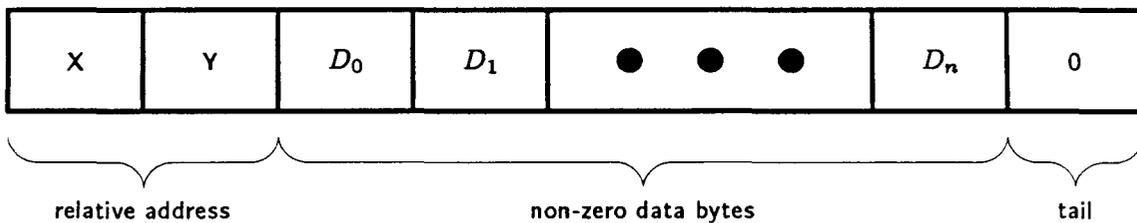


Figure 6: Packet Format

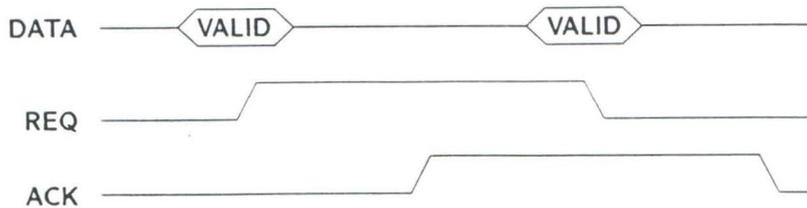


Figure 7: Virtual Channel Protocol

The TRC network routes packets first in the X direction, then in the Y direction. Packets are routed in the direction of decreasing address, decrementing the relative address at each step. When the relative X address is decremented to zero, the packet has reached the correct X coordinate. The X address is then stripped from the packet, and routing is initiated in the Y dimension. When the Y address is decremented to zero, the packet has reached the destination node. The Y address is then stripped from the packet, and the data and tail bytes are delivered to the node.

Each of the X and Y physical channels is multiplexed into two virtual channels. In each dimension packets begin on virtual channel 1. A packet remains on virtual channel 1 until it reaches its destination or address zero in the direction of routing. After a packet crosses address zero it is routed on virtual channel 0. The address 0 origin of the torus network in X and Y is determined by two input pins on the TRC. The effect of this routing algorithm is to break the channel dependency cycle in each dimension into a two-turn spiral similar to that shown in Figure 2. Packets enter the spiral on the outside turn and reach the inside turn only after passing through address zero.

Each virtual channel in the TRC uses the 2-cycle signaling convention shown in Figure 7. Each virtual channel has its own request (*R*) and acknowledge (*A*) lines. When $R = A$, the receiver is ready for the next flit (byte). To transfer information, the sender waits for $R = A$, takes control of the data lines, places data on the data lines, toggles the *R* line, and releases the data lines. The receiver samples data on each transition of *R* line. When the receiver is ready for the next byte, it toggles the *A* line.

The protocol allows both virtual channels to have requests pending. The sending end does not wait for any action from the receiver before releasing the channel. Thus, the other virtual channel will never wait longer than the data transmission time to gain access to the channel. Since a virtual channel always releases the physical channel after transmitting each byte, the arbitration is fair. If both channels are always ready, they will alternate bytes on the physical channel.

Consider the example shown in Figure 8. Virtual channel X1 gains control of the physical channel, transmits one byte of information, and releases the channel. Before this information is acknowledged, channel X0 takes control of the channel and transmits two bytes of information. Then X1, having by then been acknowledged, takes the channel again.

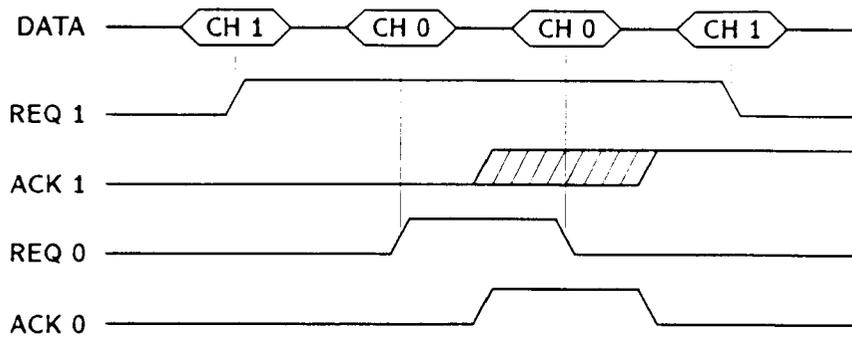


Figure 8: Channel Protocol Example

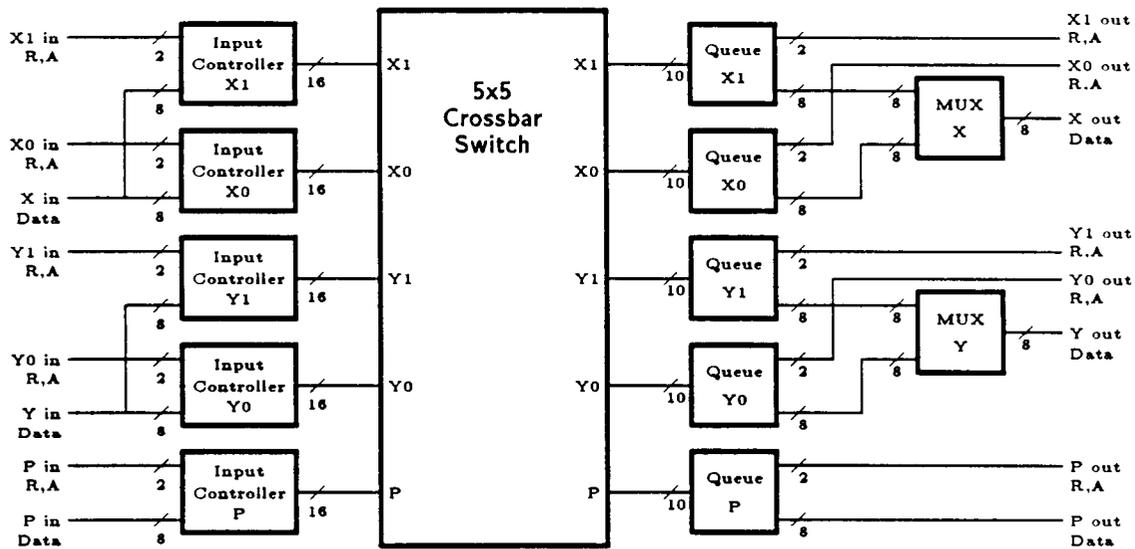


Figure 9: TRC Block Diagram

5 Logic Design

As shown in Figure 9, the TRC consists of five input controllers, a five by five crossbar switch, five output queues, and two output multiplexers. There is one input controller and one output controller for each virtual channel. The output multiplexers serve to multiplex two virtual channels onto a single physical channel.

The input controller is responsible for packet routing. When a packet header arrives, the input controller selects the output channel, adjusts the header by decrementing and sometimes stripping the byte, and then passes all bytes to the crossbar switch until the tail byte is detected.

The input controller, shown in Figure 10, consists of a datapath and a self-timed state machine. The datapath contains a latch, a zero checker, and a decremter. A state

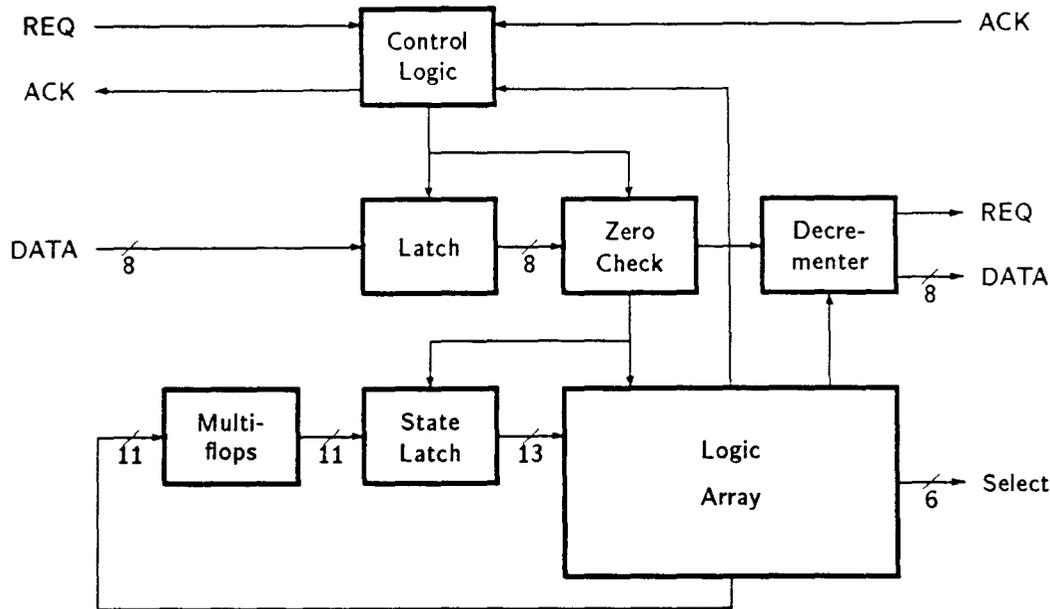


Figure 10: Input Controller Block Diagram

latch, logic array, and control logic comprise the state machine. When the request line for the channel is toggled, data is latched, and the zero checker is enabled. When the zero checker makes a decision, the logic array is enabled to determine the next state, the selected crossbar channel, and whether to strip, decrement, or pass the current byte. When the required operation has been completed, possibly requiring a round trip through the crossbar, the state and selected channel are saved in cross-coupled multi-flops and the logic array is precharged.

The input controller and all other internal logic operates using a 4-cycle self-timed signaling convention [11]. One function of the state machine control logic is to convert the external 2-cycle signaling convention into the on-chip 4-cycle signaling convention. The signaling convention is converted back to 2-cycle at the output pads.

The crossbar switch performs the switching and arbitration required to connect the five input controllers to the five output queues. A single crosspoint of the switch is shown in Figure 11. A two-input interlock (mutual-exclusion) element in each crosspoint arbitrates requests from the current input channel (row) with requests from all lower channels (rows). The interlock elements are connected in a priority chain so that an input channel must *win* the arbitration in the current row and all higher rows before gaining access to the output channel (column).

The output queues buffer data from the crossbar switch for output. The queues are each of length four. While a shorter queue would suffice to decouple input and output timing, the longer queue also serves to smooth out the variation in delays due to channel conflicts.

Each output multiplexer performs arbitration and switching for the virtual channels that share a common physical channel. As shown in Figure 12, a small self-timed state machine sequences the events of placing the data on the output pads, asserting request, and removing

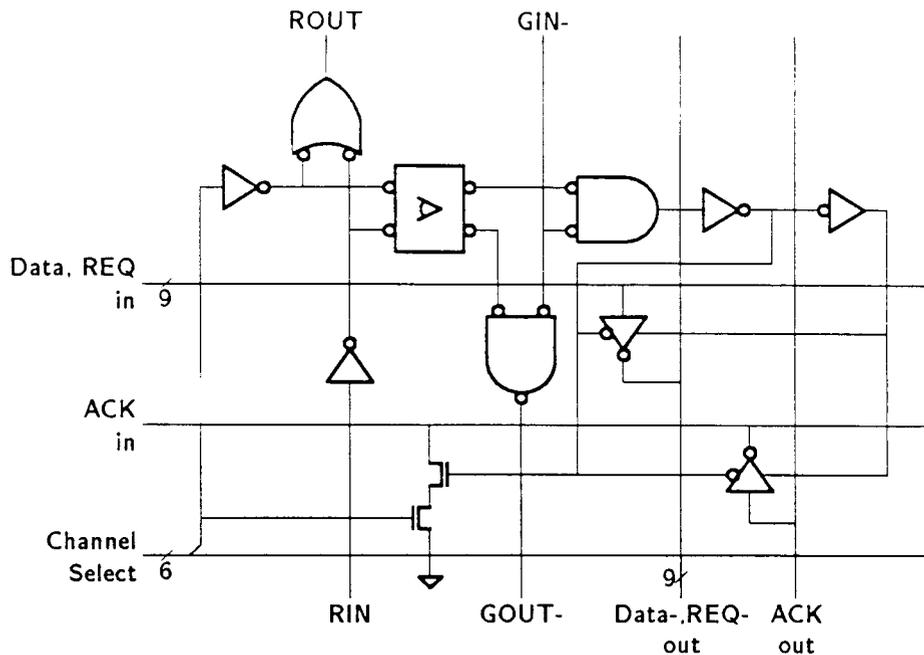


Figure 11: Crosspoint of the Crossbar Switch

the output data. An interlock element is used to resolve conflicts between channels for the data pads.

To interface the on-chip equipotential region to the off-chip equipotential region that connects adjacent chips, self-timed output pads (Figure 7.22 in [11]) are used. A Schmidt Trigger and exclusive-OR gate in each of these pads signals the state machine when the pad is finished driving the output. These completion signals are used to assure that the data pads are valid before the request is asserted and that the request is valid before the data is removed from the pads and the channel released.

6 Experimental Results

The design of the TRC began in August 1985. The chip was completely designed and simulated at the transistor level before any layout was performed. The circuit design was described using CNTK, a language embedded in C [3], and was simulated using MOSSIM [1]. A subtle error in the self-timed controllers was discovered at the circuit level before any time-consuming layout was performed. Once the circuit design was verified, the TRC was laid out in the new Mosis scalable CMOS technology [17] using the Magic system [10]. A second circuit description was generated from the artwork and six layout errors were discovered by simulation of the extracted circuit. The verified layout was submitted to Mosis for fabrication in September 1985.

The first batch of chips was completed the first week of December but failed to function because of fabrication errors. A second run of chips (same design), returned the second week of December, contained some fully functional chips.

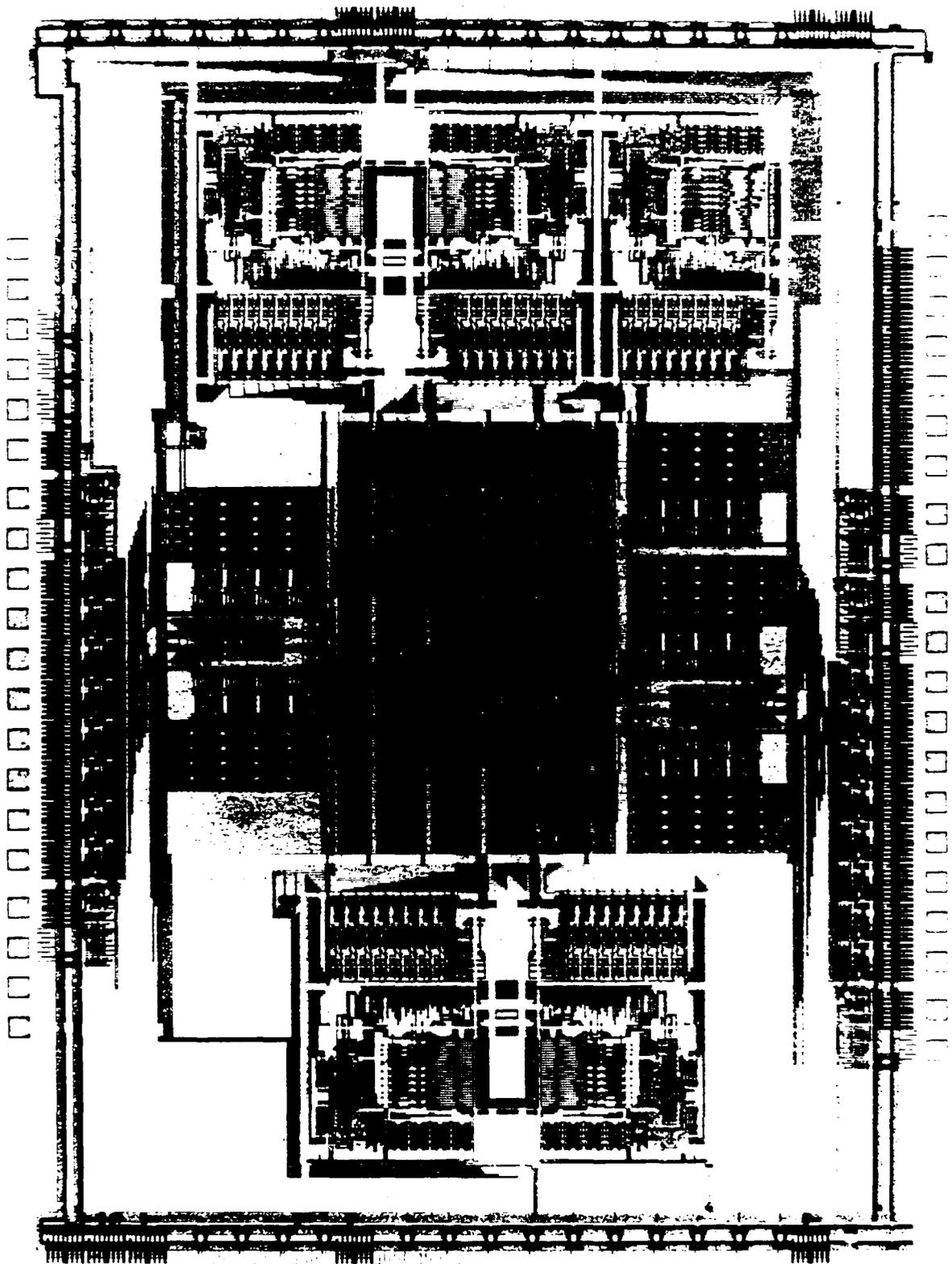
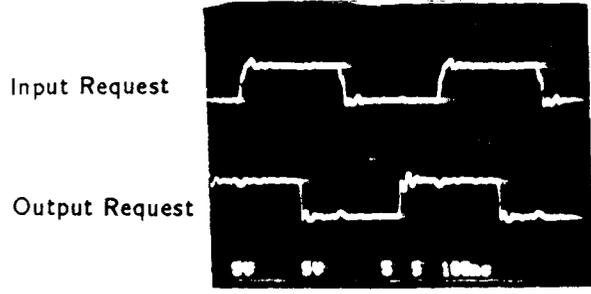
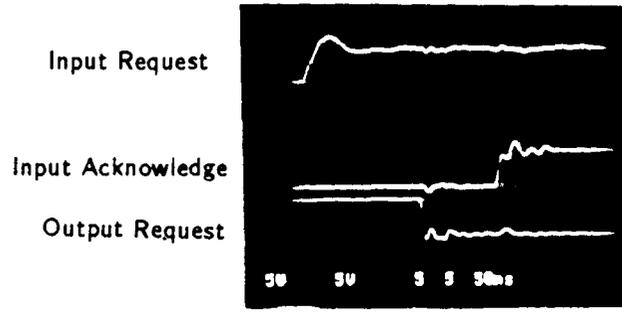


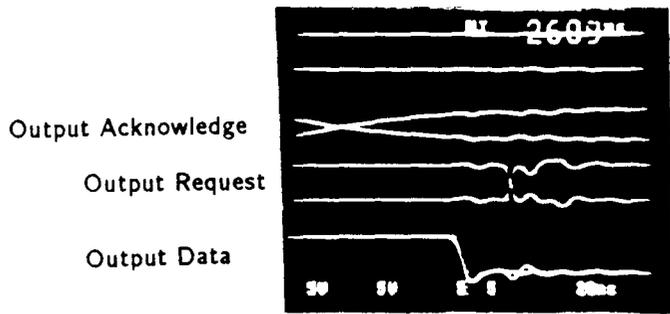
Figure 13: The Torus Routing Chip



A



B



C

Figure 14: TRC Performance Measurements

tial computer. By using byte-wide datapaths and cut-through routing, the TRC provides node-to-node communication times that approach main memory access times of sequential computers. Communications across the diameter of a network, however, require substantially longer than a memory access time.

In spite of our past success in building machines using binary n -cube interconnection networks, there are some compelling reasons to experiment with machines using a torus network. First, the torus is easier to wire. Any network topology must be embedded in the plane for implementation. The torus maps naturally into the plane with all wires the same length; the cube maps into the plane in a less uniform way. Second, the torus more evenly distributes load to communication channels. When a cube is embedded in the plane, a saturated communication channel may run parallel to an idle channel. In the torus, by grouping these channels together to make fewer but higher bandwidth channels, the saturated channel can use all of the idle channel's capacity.

Compare, for example, a 256-node binary 8-cube with a 256-node 16-ary 2-cube (16×16 torus) constructed with the same bisection width. If the 8-cube uses single bit communication channels, 256 wires will pass through a bisection of the cube, 128 in each direction. Thus, with the same amount of wire we can construct a torus with 8-bit wide communication channels. Assuming the channels operate at the same rate ¹, by choosing the torus network we trade a 4-fold increase in diameter (from 8 to 32) for a 8-fold increase in channel throughput. In general, for a $N = 2^n$ node computer we trade a $\frac{2\sqrt{N}}{n}$ increase in diameter for a $\frac{\sqrt{N}}{2}$ increase in channel throughput.

We plan to use the TRC and its successors in future experimental concurrent computers. Our first machine will use the TRC along with commercial microprocessors and memory parts to construct a 2-dimensional torus of several hundred processors. In $3\mu\text{m}$ scalable CMOS technology the TRC measures $4.5\text{mm} \times 6.5\text{mm}$ with pads. After scaling to $1.6\mu\text{m}$ technology there will room on a single die to combine both the TRC and a simple processor. With further scaling some of the processor's local memory may be moved on-chip.

The TRC serves as still another counterexample to the myth that self-timed systems are more complex than synchronous systems. The design of the TRC is not significantly more complex than a synchronous design that performs the same function. As for speed, the TRC will certainly be faster than a synchronous chip since each chip can operate at its full speed with no danger of timing errors. A synchronous chip is generally operated at a slower speed that reflects the timing of a worst-case chip and adds a timing margin.

The real challenge in concurrent computing is software. The development of concurrent software is strongly influenced by available concurrent hardware. We hope that by providing machines with higher performance internode communication we will encourage concurrency to be exploited at a finer grain size in both system and application software.

¹This assumption favors the cube since some of its channels are quite long while the torus channels are uniformly short.

Acknowledgements

The authors thank Craig Steele, Don Speck and Bill Athas for their many helpful suggestions, Fritz Nordby for designing the pads used on the TRC, and Keith Solomon for his work on the layout of the input controller datapath.

References

- [1] Bryant, Randy, Schuster, Mike and Whiting, Doug, *MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual*, Caltech Technical Report 5033:TR:82, January 1983.
- [2] Dally, William J. and Seitz, Charles L., *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*, Dept. of Computer Science, California Institute of Technology, Technical Report 5206:TR:86, 1986.
- [3] Dally, William J. *CNTK: An Embedded Language for Circuit Description*, Dept. of Computer Science, California Institute of Technology, Display File, in preparation.
- [4] Fisher, A.L. and Kung, H.T., "Synchronizing Large VLSI Processor Arrays," *IEEE Transactions on Computers*, C-34(8), August 1985, pp. 734-740.
- [5] Gunther, Klaus D., "Prevention of Deadlocks in Packet-Switched Data Transport Systems," *IEEE Transactions on Communications*, Vol. COM-29, No. 4, April 1981, pp. 512-524.
- [6] Intel iPSC User's Guide, Intel Document No. 175455-001, August 1985.
- [7] Kermani, Parviz and Kleinrock, Leonard, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, Vol 3., 1979, pp. 267-286.
- [8] Kleinrock, Leonard, *Queueing Systems*, Wiley, 1976, Vol. 2, pp. 438-440.
- [9] Lang, Charles R., *The Extension of Object-Oriented Languages to a Homogeneous Concurrent Architecture*, Dept. of Computer Science, California Institute of Technology, Technical Report, 5014:TR:82, 1982, pp. 118-124.
- [10] Ousterhout, John K. et. al., "The Magic VLSI Layout System," *IEEE Design and Test of Computers*, 2(1), February 1985, pp. 19-30.
- [11] Seitz, Charles L., "System Timing", Chapter 7 in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Addison Wesley, 1980.
- [12] Seitz, Charles L., "Concurrent VLSI Architectures," *IEEE Transactions on Computers*, C-33(12), December 1984, pp. 1247-1265.
- [13] Seitz, Charles L., "The Cosmic Cube," *CACM*, 28(1), January 1985, pp. 22-33.
- [14] Seitz, Charles L. et. al., *The Hypercube Communications Chip*, Dept. of Computer Science, California Institute of Technology, Display File 5182:DF:85, March 1985.

- [15] Steele, Craig S., *Placement of Communicating Processes on Multiprocessor Networks*, Dept. of Computer Science, California Institute of Technology, Technical Report 5184:TR:85, 1985.
- [16] Tanenbaum, A. S., *Computer Networks*, Prentice Hall, 1981, pp. 15-21.
- [17] Trotter, D., *Miss MOSIS Scalable CMOS Rules*, Version 1.2, 1985.

Compiling Communicating Processes
into
Delay-Insensitive VLSI Circuits

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

5210:TR:86

31 December 1985

to appear in: *Journal of Distributed Computing*, vol.1, no.3, 1986

1. Introduction

If VLSI is an adequate technology to implement highly concurrent computations [7], it should be possible to apply to VLSI the already well-established design methods for distributed programming. Ideally, a distributed computation should be described in a notation that can be compiled into a VLSI-circuit as well into code for a stored-program computer. The method described in this paper is a step in that direction. At the moment, the term "compiling" means a "systematic, semantics-preserving transformation". The ultimate goal of the transformation being carried out automatically has not yet been achieved, although we believe that it is not remote.

In the method we propose, the computation is initially described as a set of communicating processes in the notation of [3], which is somewhat similar to C.A.R. Hoare's CSP [2]. This first description is the reference solution, which has to be proved correct. The program is then compiled into a delay-insensitive circuit by applying a series of semantics-preserving transformations. Hence the circuit obtained is correct by construction: all semantic properties that can be proved of the program hold for the circuit as well.

Following [11], a circuit is called *delay-insensitive* when its correct operation is independent of any assumption on delays in operators and wires, except that the delays are

finite. Consequently, such circuits do not use a clock signal: sequencing is enforced entirely by communication mechanisms. Delay-insensitive circuits have been known and used for their elegance, versatility, and robustness, which result from the ideal separation of concerns they provide between the mathematical and physical aspects of circuit design.

The first modern survey on the topic is [10], where such circuits are called *self-timed*. A different approach—the *macro-module* approach—is described in [8]. Closer to our method is the recent work at Eindhoven University of Technology, a good survey of which is [9].

A circuit is a network of elementary operators (*and, or, C-element, arbiter, synchronizer, wire, fork*). The specification of an operator is a so-called *production rule set*, where a production rule is a “weaker” form of guarded command, and a production rule set a “weaker” form of repetition. The compilation relies essentially on the four-phase (also called four-cycle) handshaking expansion of the communications. After expansion, the program of each process is compiled into a production rule set from which all explicit sequencing has been removed. By matching those production rules to those describing the operators, the programs are identified with networks of operators.

The method has already been applied to a whole spectrum of problems, some of them, such as distributed mutual exclusion [4], and fair arbitration [5], being quite difficult. The results are beyond our original expectations. For many circuits, especially complex ones, the compiled circuits are superior to their “hand-designed” counterparts, which are often more complex and not entirely delay-insensitive.

We first present the program notation and the VLSI operators that constitute the “object code”. We then describe the four steps of the compilation and illustrate the method with a number of simple examples.

2. The program notation

Sequential part

For the sequential part of the algorithm, we use a subset of Edsger W. Dijkstra’s guarded command language [1], with a slightly different syntax. In this introductory paper we give only a very informal definition of the semantics of the constructs used.

- i) $b \uparrow$ stands for $b := \text{true}$, $b \downarrow$ stands for $b := \text{false}$.
- ii) The execution of the *selection* command $[G_1 \rightarrow S_1 \mid \dots \mid G_n \rightarrow S_n]$, where G_1 through G_n are Boolean expressions, and S_1 through S_n are program parts, (G_i is called a “guard”, and $G_i \rightarrow S_i$ a “guarded command”) amounts to the execution of an arbitrary S_i for which G_i holds. If $\neg(G_1 \vee \dots \vee G_n)$ holds, the execution of the command is suspended until $(G_1 \vee \dots \vee G_n)$ holds. \vee
- iii) For atomic actions x and y , “ x, y ” stands for the execution of x and y in any order.
- iv) $[G]$ where G is a Boolean, stands for $[G \rightarrow \text{skip}]$, and thus for “wait until G holds”. (Hence, “ $[G]; S$ ” and $[G \rightarrow S]$ are equivalent.)
- v) $*[S]$ stands for “repeat S forever”.
- vi) From ii) and iii), the operational description of the statement $*[[G_1 \rightarrow S_1 \mid \dots \mid G_n \rightarrow S_n]]$ is “repeat forever: wait until some G_i holds; execute an S_i for which G_i holds”.

Communicating processes

A concurrent computation is described as a set of processes composed by the usual parallel composition operator \parallel . Processes communicate with each other by communication actions on channel; they do not share variables. When no messages are transmitted, communication on a channel is reduced to synchronization signals. The name of the channel is then sufficient for identifying a communication action.

If two processes $p1$ and $p2$ share a channel named X in $p1$ and Y in $p2$, at any time the number of completed X -actions in $p1$ equals the number of completed Y -actions in $p2$. In other words, the completion of the n -th X -action “coincides” with the completion of the n -th Y -action. If, for example, $p1$ reaches the n -th X -action before $p2$ reaches the n -th Y -action, the completion of X is suspended until $p2$ reaches Y . The X -action is then said to be *pending*. When thereafter $p2$ reaches Y , both X and Y are completed. The predicate “ X is pending” is denoted qX . If, for an arbitrary command A , cA denotes the number of completed A -actions, the semantics of a pair (X, Y) of communication commands is expressed by the two axioms:

$$cX = cY \quad (A1)$$

$$\neg qX \vee \neg qY. \quad (A2)$$

Probe

Instead of the usual selection mechanism by which a set of pending communication actions can be selected for execution, we provide a general Boolean command on channels, called the *probe*. The definition of the probe given in [3] states that in process $p1$, the probe command \bar{X} has the same value as qY . Here, we use a weaker definition, namely:

$$\begin{aligned} \bar{X} &\Rightarrow qY \\ qY &\Rightarrow \diamond \bar{X}, \end{aligned}$$

where $\diamond P$ means *P holds eventually*.

Hence the guarded command $\bar{X} \rightarrow X$ guarantees that the X -action is not suspended. And a construct of the form $[\bar{X} \rightarrow X \mid \bar{Y} \rightarrow Y]$ can be used for selection. (For a more rigorous definition of the communication mechanism and the probe, see [3].)

3. The “Object Code”

The set of operators with which we want to build our circuits is not unique. In this introduction, we will use the simple set consisting of *and*, *or*, *C-element*, *wire*, and *fork*. We believe that this simple set extended with an *arbiter* and a *synchronizer* is sufficient for compiling any program. Each operator is described by a set of production rules. A production rule is similar to a guarded command, and we shall therefore use a similar syntax. There are, however, important semantic differences. Consider the production rule $G \mapsto S$:

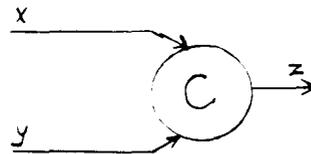
- S is either a simple assignment or of the form “ $s1, s2$ ” where $s1$ and $s2$ are each a simple assignment.

- If G holds, the correct execution of S is guaranteed only if G remains invariantly true until the completion of S . We say that G must be *stable*.
- Unlike the guarded commands of a selection or a repetition, the mutual exclusion among the different production rules of a set is not guaranteed automatically. It has to be enforced by the semantics of the program.
- If stability of the guards and mutual exclusion among guards are guaranteed, the production rule set PRS is semantically equivalent to the repetition $*[[GCS]]$, where GCS is the guarded command set syntactically identical to PRS .

The description of the five operators used in this paper in terms of their production rules and their logic symbols are as follows.

The C-element:

$$(x, y) \underline{C} z \equiv x \wedge y \mapsto z \uparrow \\ \neg x \wedge \neg y \mapsto z \downarrow.$$



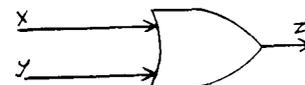
The “and”:

$$(x, y) \underline{\Delta} z \equiv x \wedge y \mapsto z \uparrow \\ \neg x \vee \neg y \mapsto z \downarrow.$$



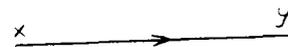
The “or”:

$$(x, y) \underline{\vee} z \equiv x \vee y \mapsto z \uparrow \\ \neg x \wedge \neg y \mapsto z \downarrow.$$



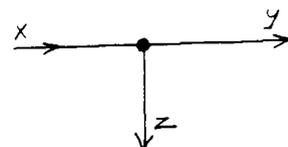
The wire:

$$x \underline{w} y \equiv x \mapsto y \uparrow \\ \neg x \mapsto y \downarrow.$$



The fork:

$$x \underline{f} (y, z) \equiv x \mapsto y \uparrow, z \uparrow \\ \neg x \mapsto y \downarrow, z \downarrow.$$



Any input or output variable of an operator may be negated. In particular, a wire with its input or its output negated—but not both—is an inverter. A negated input or output is represented in the figures by a small circle on the corresponding line.

4. The Compilation Method

Process Decomposition

The first step of the compilation, called “process decomposition”, consists in replacing a process by several semantically equivalent processes. The purpose of the decomposition is to obtain a process representation of the program in which the right-hand side of each

guarded command is a straight-line program, i.e. consists only of simple assignments and communication commands, composed by semi-colons and commas.

Decomposition rule: A process P containing an arbitrary program part S is semantically equivalent to two processes $P1$ and $P2$, where $P1$ is derived from P by replacing S by a communication action C on the newly introduced channel (C, D) between $P1$ and $P2$, and $P2 \equiv *[[\bar{D} \rightarrow S; D]]$.

Observe that the above decomposition does not introduce concurrency. Although $P1$ and $P2$ are potentially concurrent processes, they are never active concurrently: $P2$ is activated from $P1$, much as a procedure or a coroutine would be. The only purpose of this transformation is to simplify the structure of each command. As an example, consider the process:

$$P \equiv *[[\dots A; [B_1 \rightarrow S_1 \mid B_2 \rightarrow S_2]; \dots]].$$

Applying the decomposition rule, P is replaced by the two processes $P1$ and $P2$. Channel (C, D) is introduced between $P1$ and $P2$.

$$\begin{aligned} P1 &\equiv *[[\dots A; C; \dots]] \\ P2 &\equiv *[[\bar{D} \wedge B_1 \rightarrow S_1; D \\ &\quad \mid \bar{D} \wedge B_2 \rightarrow S_2; D \\ &\quad]]. \end{aligned}$$

Observe that the newly created processes $P1$ and $P2$ may share variables. Since the processes are never active concurrently, there is no conflicting access to the shared variables. Process decomposition is applied repeatedly until the right-hand side of each guarded command is a straight-line program.

Handshaking Expansion

The implementation of communication, called “handshaking expansion”, replaces each channel by a pair of wire-operators and each communication action by its implementation. Channel (X, Y) is implemented by the two wires $(x_0 \underline{w} y_i)$ and $(y_0 \underline{w} x_i)$.

If X belongs to process $p1$ and Y to process $p2$, x_0 and x_i belong to $p1$, and y_0 and y_i belong to $p2$. Initially, x_0 , x_i , y_0 , and y_i —which we will call the “handshaking variables of (X, Y) ”—are false. Assume that the program has been proved to be deadlock-free and that we can identify a pair of matching actions X and Y in $p1$ and $p2$ respectively. We replace X and Y by the sequences U_x and U_y respectively, with:

$$\begin{aligned} U_x &\equiv x_0 \uparrow; [x_i] \\ U_y &\equiv [y_i]; y_0 \uparrow. \end{aligned}$$

The formal proof that U_x and U_y fulfil axioms $A1$ and $A2$ is omitted. The following is an informal argument that relies on a definition of completion of an action different from the usual one. Since the argument is not essential to the comprehension of the method, it may be skipped at first reading.

Assume that we know what the *initiation* and *termination* of an atomic action mean. A non-atomic action is initiated when its first atomic action is initiated. A non-atomic action is terminated when its last atomic action is terminated.

A non-atomic action is said to be completed when it is initiated and it is guaranteed to terminate.

(An atomic action is completed when it is terminated.) Between initiation and completion, an action is *suspended*.

Obviously, U_x and U_y are guaranteed to terminate if and only if they are both initiated, which establishes A1 and A2.

It is essential to observe that these definitions of completion and suspension are valid because they satisfy the semantic properties of completion and suspension that are used in correctness arguments, namely:

$$\{cX = x\} X \{cX = x + 1\}$$

$$qX \Rightarrow pre(X)$$

where $pre(X)$ is any precondition of X in terms of the program variables and auxiliary program variables.

(This completes the argument.)

Unfortunately, when the communication terminates, all handshaking variables are true. Hence, we cannot implement the next communication with U_x and U_y . However, the complementary implementation can be used for the next matching pair, namely:

$$D_x \equiv x0 \downarrow; [\neg xi]$$

$$D_y \equiv [\neg yi]; y0 \downarrow.$$

The solution consisting in alternating U_x and D_x as an implementation of X , and U_y and D_y as an implementation of Y is essentially the so-called "two-phase handshaking", or "two-cycle signaling". However, it is in general not possible to determine syntactically which X - or Y -actions are following each other in an execution. In general, two-phase handshaking implementations require testing the current value of the variables. In this paper, we shall use a simpler but less efficient solution known as "four-phase handshaking", or "four-cycle signaling".

In a four-phase handshaking protocol, all X -actions are implemented as " $U_x; D_x$ " and all Y -actions as " $U_y; D_y$ ". Observe that the D -parts in X and Y introduce an extra communication between the two processes whose only purpose is to reset all variables to false. The synchronization introduced by this extra communication is unnoticeable since the immediately preceding communication implemented by U_x and U_y sees to it that both processes reach a matching D_x and D_y "at the same time".

Both protocols have the property that for a matching pair (X, Y) of actions, the implementation is not symmetrical in X and Y . One action is called *active* and the other one *passive*. The four-phase implementation with X active and Y passive is:

$$X \equiv x0 \uparrow; [xi]; x0 \downarrow; [\neg xi] \tag{1}$$

$$Y \equiv [y\uparrow]; y\downarrow; [\neg y\uparrow]; y\downarrow \quad (2)$$

When no action of a matching pair is probed, the choice of which one should be active and which one passive is arbitrary, but a choice has to be made. The choice can be important for the composition of identical circuits. A simple rule is that for a given channel (X, Y) , all actions at one side are active and all actions at the other side passive. If \bar{X} is used, all X -actions are passive—with the obvious restriction that \bar{Y} cannot be used in the same program.

The implementation of the probe is simply:

$$\begin{aligned} \bar{X} &\equiv x\uparrow \\ \bar{Y} &\equiv y\uparrow \end{aligned} \quad (3)$$

Given our definition of suspension, the proof that this implementation of the probe fulfils the definition of Section 2 is straightforward and is omitted.

A probed communication action $\bar{X} \rightarrow \dots X$ is implemented:

$$x\uparrow \rightarrow \dots x\downarrow; [\neg x\uparrow]; x\downarrow.$$

Basic properties

The following properties of the handshaking protocol play an important role in the compilation method.

Property 1: *For the pair of wires $(x\downarrow \underline{w} y\uparrow)$ and $(y\downarrow \underline{w} x\uparrow)$, used together as in (1) and (2), and all variables false initially, the following sequence of transitions is guaranteed to occur if the system is deadlock-free:*

$$*[x\downarrow; y\uparrow; y\downarrow; x\uparrow; x\downarrow; y\uparrow; y\downarrow; x\uparrow]. \quad (4)$$

Hence, the following postconditions hold:

$$\begin{aligned} x\downarrow &\{\diamond x\uparrow\} \\ x\downarrow &\{\diamond \neg x\uparrow\} \\ y\downarrow &\{\diamond \neg y\uparrow\} \end{aligned} \quad (5)$$

Property 2: *Consider the handshaking expansion of a program p according to (1), (2), and (3). Provided that the cyclic order of the four handshaking actions of a communication command is respected, the last two actions of this command—the two actions of D_x or D_y —can be inserted at any place in p without invalidating the semantics of the communication involved. However, modifying the order of these two actions relatively to other actions of p may introduce deadlock.*

Property 2 is a direct consequence of the way in which we have introduced the sequences D_x and D_y . We will see examples of how to use Property 2. In this paper, we will ignore the deadlock issue when we re-order handshaking actions.

First example: stack element

Consider the simple process S , which we call a "stack element":

$$S \equiv *[\bar{L} \rightarrow R; L],$$

where L and R are channels. Since L is probed, it must be passive, and if we want to compose S -processes together, R must be active, since it will match a passive L . The handshaking expansion gives:

$$*[[li]; ro \uparrow; [ri]; ro \downarrow; [\neg ri]; lo \uparrow; [\neg li]; lo \downarrow]. \quad (6)$$

5. Production-rule expansion

The next step is to compile the handshaking expansion of the program into a set of production rules from which all explicit sequencing has been removed. By matching those production rules to those describing the semantics of operators, the programs can be identified with networks of operators. We use the compilation of S to illustrate the different steps of the expansion.

We start with the production rule set syntactically derived from the program. In the case of S , it is the set derived from (6), namely:

$$\begin{aligned} li &\mapsto ro \uparrow \\ ri &\mapsto ro \downarrow \\ \neg ri &\mapsto lo \uparrow \\ \neg li &\mapsto lo \downarrow. \end{aligned}$$

The execution of a production rule is called *effective* if it changes the value of a variable. Otherwise, it is called *vacuous*. We ignore vacuous executions of production rules.

For each guarded command of the program, the production rule set representation is semantically equivalent to the program representation if and only if the order of execution of effective production rules is the same as the order of the corresponding transitions in the program—we call it the *program order*. (As a clue to the reader we list the production rules of a set in program order.)

In general, we have to strengthen the guards of some rules to enforce execution in program order. This is the case in our example: Since $\neg ri$ holds initially, the third production rule can be executed first. It is also true for the fourth production rule; but the execution of the fourth rule in the initial state is vacuous.

Because all handshaking variables of R are back to false when R is completed, we cannot find a guard for the transition $lo \uparrow$. (Hence, the transitions following a semi-colon that can be identified with a semi-colon of the original program are likely to be difficult to deal with.)

Direct implementation

In order to define uniquely the state in which the transition $lo \uparrow$ is to take place, the first technique consists in introducing a state variable, say x , initially false. S becomes

$$*[[li]; ro \uparrow; [ri]; x \uparrow; [x]; ro \downarrow; [\neg ri]; lo \uparrow; [\neg li]; x \downarrow; [\neg x]; lo \downarrow]. \quad (7)$$

Now, the production-rule expansion can be performed:

$$\neg x \wedge li \mapsto ro \uparrow \{\diamond ri\} \quad (S1)$$

$$ri \mapsto x \uparrow \{x\} \quad (S2)$$

$$x \mapsto ro \downarrow \{x \wedge \diamond \neg ri\} \quad (S3)$$

$$x \wedge \neg ri \mapsto lo \uparrow \{\diamond \neg li\} \quad (S4)$$

$$\neg li \mapsto x \downarrow \{\neg x\} \quad (S5)$$

$$\neg x \mapsto lo \downarrow. \quad (S6)$$

(Why is the conjunct $\neg x$ necessary in the first rule?) Using the postconditions indicated between braces—these conditions rely on (5)—, it is easy to verify that the production rules of the set are executed in program order. Hence, the execution of the production rule set is equivalent to the execution of (7).

Re-ordering implementation

Another way to find a valid guard for $lo \uparrow$ is to use Property 2, to re-order the actions of (6). For instance, we can postpone the second half of the handshaking expansion of S —i.e., the sequence $ro \downarrow; [\neg ri]$ —until after $[\neg li]$. We get:

$$*[[li]; ro \uparrow; [ri]; lo \uparrow; [\neg li]; ro \downarrow; [\neg ri]; lo \downarrow]. \quad (8)$$

The syntactic production rule expansion is already “program ordered”:

$$li \mapsto ro \uparrow$$

$$ri \mapsto lo \uparrow$$

$$\neg li \mapsto ro \downarrow$$

$$\neg ri \mapsto lo \downarrow.$$

6. Operator reduction

The last step of the compilation, called *operator reduction*, consists in identifying sets of production rules in the program with sets of production rules describing operators. The program can then be identified with a set of operators. We group pairs of production rules that modify the same variable.

If a given group cannot be directly identified with the production rule set of an operator, we perform on this group a last transformation called *symmetrization*: we transform the guards of the production rules—again under invariance of the semantics—so as to make them “look like” the guards of operators. In case a guard contains too many variables, this step may also involve decomposing a production rule into several production rules by introducing new internal variables.

Consider $S1$ and $S3$. No operator corresponds to these rules. But, if we replace x by $\neg li \vee x$ in $S3$, the value of the guard of $S3$ is not changed since li holds as precondition

of $S3$, and now the two production rules represent the operator $(\neg x, li) \triangle ro$. Since we have weakened the guard of $S3$, we have to check that we have not enlarged the set of states in which $S3$ can be effectively executed. No such state has been added, hence the transformation is safe.

In the case of $S2$ and $S5$, no guard can be weakened. We therefore strengthen both of them as

$$\begin{aligned} ri \wedge li &\mapsto x \uparrow \\ \neg ri \wedge \neg li &\mapsto x \downarrow, \end{aligned}$$

which corresponds to the C -element $(ri, li) \underline{C} x$. Observe that strengthening the guards in this way is always possible since the guards are mutually exclusive by construction. Hence it is always possible to implement a pair of guards with a C -element. Why then bother about weakening the guards? The answer is that introducing a disjunction is the only transformation leading to combinatorial operators—*and*, *or*—, which are usually less “expensive” than C -elements—a C -element is a state-holding operator.

For the direct implementation of S , the symmetrization of the set $S1$ through $S6$ gives:

$$\neg x \wedge li \mapsto ro \uparrow \quad (S1)$$

$$ri \wedge li \mapsto x \uparrow \quad (S2)$$

$$\neg li \vee x \mapsto ro \downarrow \quad (S3)$$

$$x \wedge \neg ri \mapsto lo \uparrow \quad (S4)$$

$$\neg ri \wedge \neg li \mapsto x \downarrow \quad (S5)$$

$$ri \vee \neg x \mapsto lo \downarrow. \quad (S6)$$

The identification with operators is now straightforward.

$(S1, S3)$ corresponds to $(\neg x, li) \triangle ro$.

$(S2, S5)$ corresponds to $(li, ri) \underline{C} x$.

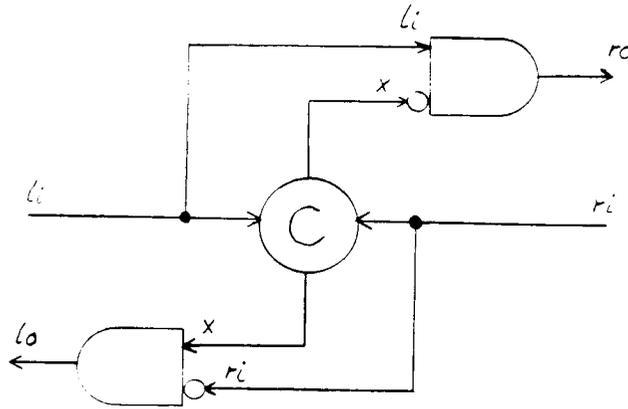
$(S4, S6)$ corresponds to $(x, \neg ri) \triangle lo$.

Isochronic forks

In the previous operator reduction, li is input to the C -element $(li, ri) \underline{C} x$, and to the *and*-operator $(li, \neg x) \triangle ro$. Formally, in order to compose the circuit we have to introduce the fork $li \underline{f} (l1, l2)$ and replace li by $l1$ in the C -element and by $l2$ in the *and*-operator.

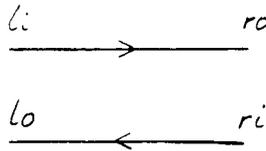
Since the fork is delay-insensitive, $l1$ and $l2$ are not guaranteed to have the same value in all states, whereas the two operators are constructed with the same input variable li . We solve this problem by making a simplifying assumption: we assume that the forks used to connect operators inside a process are *isochronic*, i.e. the delays in these forks are short enough, compared to the delays in all operators other than forks and wires, to assume that the two outputs of an isochronic fork have the same value at any time.

The resulting circuit is shown in Fig. 1.



-Figure 1-

For the second implementation of S —with re-ordering of actions—the production rule set can be reduced directly: the first and third rules specify the wire $li \underline{w} ro$, the second and fourth rules specify the wire $ri \underline{w} lo$. The circuit is shown in Fig. 2.



-Figure 2-

Comparing the circuits of Figs. 1 and 2, we observe that the re-ordering of handshaking actions leads to a simpler implementation. This observation is true in general, although the gain is not always as drastic as in this case. We also observe that re-ordering handshaking actions modifies the behavior of the circuit concerning its synchronization with its environment. This is not surprising since the second half of a handshaking sequence—the part that we shift from its place—is an extra synchronization action. Placed just after the first half, this second synchronization has no noticeable effect. But its synchronization effect becomes noticeable when the action is shifted away from the first half of the handshaking sequence. Hence the choice to re-order actions is a choice in favor of a simpler circuit at the cost of modifying the original synchronization behavior of the circuit—in general for the worse.

7. Second example: one-place buffer

Our second example is the simple “one-place buffer” process

$$B \equiv *[L; R],$$

where L and R are two channels. The handshaking expansion of B gives:

$$B \equiv *[[li]; lo \uparrow; [-li]; lo \downarrow; ro \uparrow; [ri]; ro \downarrow; [-ri]]. \quad (9)$$

Here the difficult transition is $ro \uparrow$. In this example we construct only the solution obtained by re-ordering of actions. The construction of the solution with introduction of a state variable is more difficult and is left as an exercise to the reader. (It is described in [6].) If we postpone the second half of the handshaking expansion of L until after $[ri]$, we get:

$$*[[li]; lo \uparrow; ro \uparrow; [ri]; [\neg li]; lo \downarrow; ro \downarrow; [\neg ri]],$$

which we can also re-order as:

$$*[[\neg ri]; [li]; lo \uparrow; ro \uparrow; [ri]; [\neg li]; lo \downarrow; ro \downarrow]. \quad (10)$$

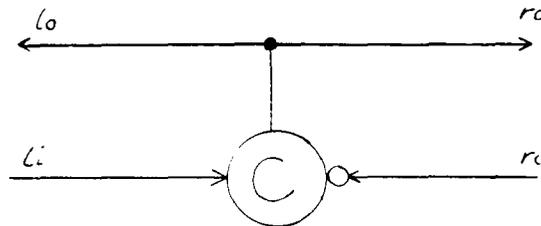
The order between two successive transitions on output variables—like $lo \uparrow; ro \uparrow$ —is irrelevant. Hence the production-rule expansion of (10) gives:

$$\begin{aligned} \neg ri \wedge li &\rightarrow lo \uparrow, ro \uparrow \\ ri \wedge \neg li &\rightarrow lo \downarrow, ro \downarrow. \end{aligned}$$

After introducing the auxiliary variable u , the production rule expansion is straightforward:

$$\begin{aligned} ((\neg ri, li) \underline{C} u) \\ (u \underline{f} (lo, ro)). \end{aligned}$$

The corresponding circuit is shown in Fig. 3.



-Figure 3-

8. Message communication

So far, we have only considered the synchronization aspect of the communication actions: no message was passed. The last two examples describe implementations of communications that entail transmissions of messages. We consider the transmission of Boolean variables only; the generalization to other types is relatively straightforward.

Third example: Queue (FIFO) element

Queues (FIFO) play an important role in pipeline computations for increasing throughput when processing times are variable. A queue consists of the linear composition of a number of buffer-elements of the type:

$$E \equiv * [L?(x); R!(x)] . \quad (11)$$

($L?(x)$ is an input action assigning to internal variable x the value received on L . $R!(x)$ is an output sending the value of x on channel R .)

We are going to implement the transmission of true messages and of false messages on two independent channels. We shall construct a circuit for each type of messages, and then compose the two circuits. Such a technique is called the "double-rail" technique [10]. We get:

$$* [[\bar{L}_t \rightarrow L_t; R_t \\ | \bar{L}_f \rightarrow L_f; R_f \\]] ,$$

where $\neg \bar{L}_t \vee \neg \bar{L}_f$ holds at any time.

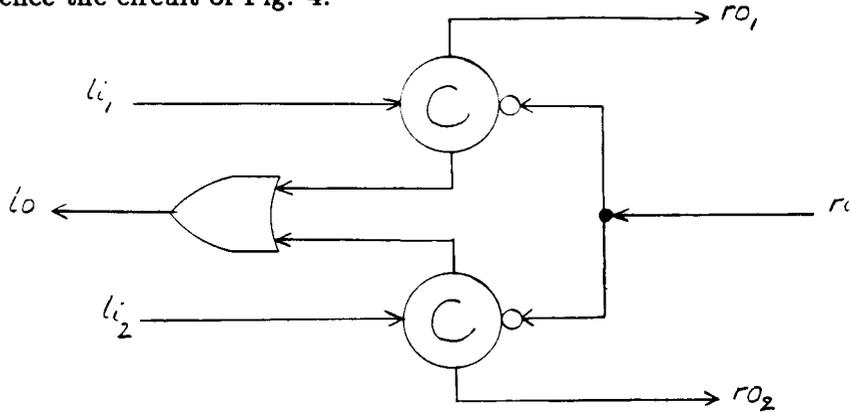
If we let channels L_t and L_f share variable lo , and channels R_t and R_f share variable ri , the handshaking expansion gives the two guarded commands:

$$* [[li_1 \rightarrow lo \uparrow; [\neg li_1]; lo \downarrow; ro_1 \uparrow; [ri]; ro_1 \downarrow; [\neg ri] \\ | li_2 \rightarrow lo \uparrow; [\neg li_2]; lo \downarrow; ro_2 \uparrow; [ri]; ro_2 \downarrow; [\neg ri] \\]] . \quad (12)$$

The production rule expansion of (12) has to guarantee mutual exclusion between the two guarded commands. Since $\neg li_1 \vee \neg li_2$ holds at any time, it is easy to see that mutual exclusion is guaranteed if we re-order the actions of each guarded command as in the implementation of B . We get:

$$* [[\neg ri \wedge li_1 \rightarrow lo \uparrow, ro_1 \uparrow; [ri \wedge \neg li_1]; lo \downarrow, ro_1 \downarrow \\ | \neg ri \wedge li_2 \rightarrow lo \uparrow, ro_2 \uparrow; [ri \wedge \neg li_2]; lo \downarrow, ro_2 \downarrow \\]] . \quad (13)$$

Since each of the two guarded commands of (13) is identical to (10), the circuit for (12) consists of two copies of the circuit of Fig. 3 composed in the obvious way so as to share lo and ri . Hence the circuit of Fig. 4.



-Figure 4-

Fourth example: single variable

Consider the following process that provides read and write access to a simple Boolean variable x :

$$\begin{aligned}
 & *[[\overline{P} \rightarrow P?x \\
 & \quad | \overline{Q} \rightarrow Q!x \\
 & \quad]],
 \end{aligned} \tag{14}$$

where $\neg\overline{P} \vee \neg\overline{Q}$ holds at any time.

Again, according to the double-rail technique, each guarded command of (14) is expanded to two guarded commands. But now the values true and false have to be explicitly assigned to x , in the following way:

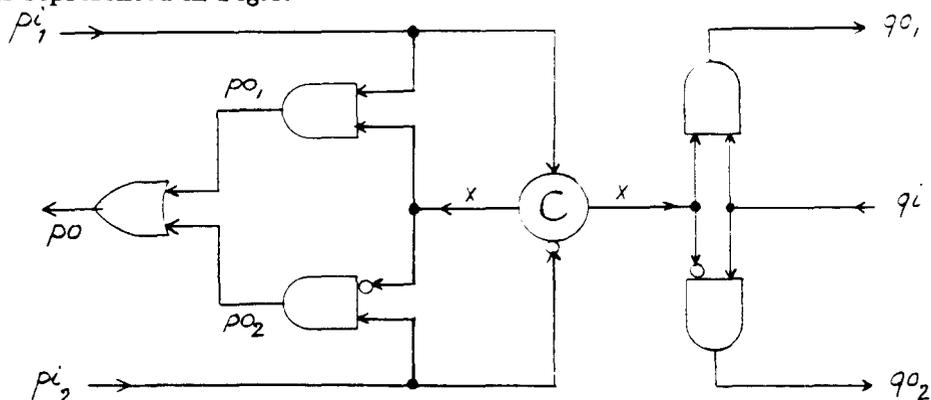
$$\begin{aligned}
 & *[[pi_1 \rightarrow x \uparrow; [x]; po \uparrow; [\neg pi_1]; po \downarrow \\
 & \quad | pi_2 \rightarrow x \downarrow; [\neg x]; po \uparrow; [\neg pi_2]; po \downarrow \\
 & \quad | x \wedge qi \rightarrow qo_1 \uparrow; [\neg qi]; qo_1 \downarrow \\
 & \quad | \neg x \wedge qi \rightarrow qo_2 \uparrow; [\neg qi]; qo_2 \downarrow \\
 & \quad]].
 \end{aligned} \tag{15}$$

The rest of the compilation is now straightforward and is left as an exercise to the reader. (Hint: don't forget to ensure mutual exclusion between the guarded commands.)

The operator reduction gives:

$$\begin{aligned}
 & (pi_1, \neg pi_2) \underline{C} x \\
 & (pi_1, x) \underline{\Delta} po_1 \\
 & (pi_2, \neg x) \underline{\Delta} po_2 \\
 & (po_1, po_2) \underline{\vee} po \\
 & (x, qi) \underline{\Delta} qo_1 \\
 & (\neg x, qi) \underline{\Delta} qo_2.
 \end{aligned}$$

The circuit is represented in Fig.6.



-Figure 6-

9. Conclusion

We have described a method for implementing a high-level concurrent algorithm (a set of communicating processes) as a network of digital operators that can be directly mapped into a delay-insensitive VLSI-circuit. The circuit is derived from the program by a series of systematic, semantics-preserving, transformations that we have compared to compiling.

Since the circuits are correct by construction, and in particular, since the guards of the production rules are stable by construction, the circuits are free from “hazards”.

The choice between active and passive implementations is usually clear from the context. For instance, the choice to implement input as passive and output as active is most of the time safe. Furthermore, in the case the wrong choice has been made and it turns out that two active or two passive commands have to be paired, an “adaptor” process can be used. An adaptor is a one-place buffer with L and R both active—a “double-A”—or both passive—a “double-P”. A double-A is used to pair two passive commands, a double-P to pair two active commands.

The simplifying assumption of isochronic forks is not severe, since such a fork is always confined to a very small circuit part. In fact, it is even weaker than the usual isochronic assumption used in self-timed design, where a whole circuit part is assumed isochronic. We believe that isochronic forks can be avoided, but doing so would complicate the circuits without real advantage in return.

We also believe that the basic sets of operators used in this paper, extended with an arbiter and a synchronizer to implement mutual exclusion among independent commands, is sufficient for all purposes. (Obviously, having both *and* and *or* is redundant.) However, there is no interest in confining the designer to a minimal set of operators. On the contrary, since one of the advantages of VLSI is the possibility to create operators at no cost, introducing other operators—like, e.g., *and* and *or* with more than two inputs, or *exclusive-or*—may often simplify a circuit drastically.

We have illustrated the method with four simple—sometimes deceptively so—but characteristic examples that embody very standard control and data structures. The method has also been tested on quite difficult examples like the distributed mutual exclusion circuit described in [4]. In [5], we have used the method to solve an open problem: It had been conjectured that it is impossible to construct a delay-insensitive fair arbiter. We have disproved the conjecture by constructing such an arbiter applying our method.

The most encouraging aspect of the method is that it is really a synthesis technique: it allows a designer to construct solutions that he would never have found had he not applied the method.

10. Acknowledgement

I am indebted to Martin Rem, Chuck Seitz, Peggy Li, and Kevin Van Horn for their comments on the manuscript. Kevin Van Horn also contributed to the definition of the completion of a communication. Acknowledgements are also due to the Eindhoven VLSI Club, in particular Huub Schols, for their comments and criticisms during an oral presentation of this material in September 1985.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and was monitored by the Office of Naval Research under contract number N00014-79-C-0597.

11. References

- [1] Dijkstra, Edsger W., *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs NJ (1976)
- [2] Hoare, C.A.R. "Communicating Sequential Processes". *Comm. ACM* 21,8, pp 666-677 (August 1978)
- [3] Martin, A.J., "The Probe: an Addition to Communication Primitives", *Information Processing letters* 20, pp 125-130 (1985)
- [4] Martin, A.J., "The Design of a Self-Timed Circuit for Distributed Mutual Exclusion", *Proc. 1985 Chapel Hill Conf. VLSI*, ed. Henry Fuchs, pp 247-260 (1985)
- [5] Martin, A.J., "A Delay-Insensitive Fair Arbiter", Caltech Computer Science Technical Report 5193:TR:85 (1985)
- [6] Martin, A.J., "FIFO: an Exercise in Compiling Programs into Circuits", Caltech Computer Science Technical Memo (1985)
- [7] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980)
- [8] Molnar, C.E., et al. "Synthesis of Delay-Insensitive Modules", *Proc. 1985 Chapel Hill Conf, VLSI*, ed. Henry Fuchs, pp 67-86, (1985)
- [9] Rem, M., "Concurrent Computations and VLSI Circuits", in "Control Flow and Data Flow: Concepts in Distributed Programs", ed. M.Broy, pp 399-437 Springer-Verlag Berlin Heidelberg (1985).
- [10] Seitz, C.L., "System Timing", Chapter 7 in Mead & Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980)
- [11] Snepscheut, J.v.d., "Trace Theory and VLSI Design" LNCS 200, Springer-Verlag Berlin Heidelberg (1985).