SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

Caltech-CS-TR-90-14

1 October 1990

Reporting Period:	16 March 1990 – 30 September 1990
Principal Investigator:	Charles L. Seitz
Faculty Investigators:	K. Mani Chandy
	Alain J. Martin
	Charles L. Seitz
	Stephen Taylor
	Jan van de Snepscheut

Sponsored by the
Defense Advanced Research Projects Agency
DARPA Order Number 6202

Monitored by the
Office of Naval Research
Contract Number N00014-87-K-0745

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 research activities and results for the six-and-one-half-month period, 16 March 1990 to 30 September 1990, under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. Previous semiannual technical reports and other technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

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

1.3 Highlights

- Mosaic C (section 2.1).
- Mosaic programming system (section 3.1).
- The Page Kernel demonstated (section 3.3).
- Self-timed designs (section 4.1–4.8).

2. Architecture Experiments

2.1 Mosaic Project

Chuck Seitz, Nanette J. Boden, Jakov Seizovic, Don Speck, Wen-King Su

Our previous semiannual technical report includes a detailed description of the development of the Mosaic C, an experimental *fine-grain multicomputer* based on single-chip nodes and a reactive-process programming model.

Our previous report occurred just before the MOSIS 1.2μm SCMOS run that closed on 20 March 1990. Fast turnaround has allowed us to complete 2.5 iterations of design, fabrication, and testing of the Mosaic silicon in this six-and-one-half-month period.

The Mosaic project has proceeded in accordance with or faster than the schedule outlined in the previous report.

Mosaic C dRAM

A 64KB ($32K\times16$) Mosaic C dRAM operated correctly on first silicon, and over an exceptionally wide range of operating conditions. The only anomaly discovered in testing this 1T dRAM was one-to-zero errors in several locations in the outside columns. These errors were traced to negative charge injected into the substrate by input-protection structures on pads located several hundred μm away. The input-protection structures were functioning correctly; ground bounce was causing the low input to appear as a voltage less than ground, and correcting the ground bounce in the test fixture cured the problem. The input-protection structures were replaced with an annular design that will collect the negative charges with the structure, and a guard structure was added to the outside columns of the dRAM.

This chip was also tested with a variety of deliberate disturbances, including light, alpha particles, and wide power-supply variations. The speed is right on the design target: 11MHz/V, *eg,* 44MHz at 4V operation.

The second silicon of the dRAM behaved in the same way as the first except for its susceptability to substrate charge. The yield, however, was significantly lower, but we have reason to believe that this was due to the run rather than to the changes in the design.

Memoryless Mosaic 2.1

MM2.1 is a 1.2μ version of the MM2.0 with a minor microcode change. It uses the original 3D, 4-bit-wide, synchronous router. Chips returned from fabrication in late April 1990, and are completely functional with a yield of 48/50. They have been exercised extensively in our first generation of program-development boards.

A prototype board was also made, consisting of MM2.1, our new 64KB, 1T dRAM, and two 15ns off-the-shelf EPROMs, to verify that there were no oversights in the design of the memory interface. The setup is functional up to 27MHz at 4V, limited by the EPROM timing.

Memoryless Mosaic 3.0

MM3.0 is our first attempt at incorporating the 2D, 8-bit-wide asynchronous router into the Mosaic. The MM3 chip is assembled from the same processor as MM2.1, a version of the FMRC2.3 mesh-routing chip with several modifications (such as a 7-bit rather than a 6-bit field in the header flit to represent the relative distance), and an almost completely redesigned packet interface.

The new packet interface had to deal with a different message format – 2D *vs* 3D routing; a different protocol at the router interface – 8-bit, 2-cycle asynchronous *vs* 4-bit synchronous; and a higher data-rate at the router interface – 80MB/s *vs* 20MB/s. The scope of the required changes called for a new design rather than numerous local patches. Only the interrupt and bus-arbitration logic remained unchanged in the packet interface from the MM2.1 version. The packet interface amounts to about 30% of the active area of the MM3.0.

We received the chips in mid-August, and have been testing them extensively on our second-generation program-development boards. Two minor design errors were discovered during the testing. The first error was the result of an oversight in the optimization of a special case in the arbitration for storage. After this error was discovered, 12 otherwise functional chips were sent to HP to have this bug eliminated by cutting one second-metal wire with a laser. This repair was 100% successful, and allowed us to look for deeper troubles. The second design error was causing some 1 bits of packets to be received as 0s. The problem was eventually traced to the lack of a sufficient timing margin between the request and data lines at the interface between the router and the packet interface.

Both errors were fixed, and the MM3.1 was submitted for fabrication in mid-September 1990. The chips are expected in the beginning of November. Since we believe that the MM3.1 will be fully functional, we have already started the final phase of assembling the full Mosaic element, and will have it ready by the time MM3.1 chips are back.

Yield Observations

The yield for the MM3.0 was 38/50, much lower than the usual 45/50 to 48/50. The yield for the memory on this same run was 16/50 rather than the previously observed yield of 22/50.

We have tried to localize every fault to make sure that the fault is caused by fabrication rather than a marginal design. After extensive testing and numerous hours of observing chips under the microscope, 8 of the 12 bad chips have been positively identified as containing fabrication errors, and the other 4 contain probable but invisible fabrication errors.

Plot of the Memoryless Mosaic 3.1

Program-Development/Host-Interface Boards

Mosaic Processor Development Board R1.0: In order to allow meaningful software development and more comprehensive testing of the Mosaic processor, we designed a double-height (6U) VME board that holds 4 Mosaic processors connected in a two-by-two array. The board contains 128 Kbytes of SRAM per processor, and the SRAM is shared with the Sun 3/260 host by cycle stealing. The board and the processors were shown to be operating correctly to a processor clock frequency of 20 MHz — the maximum speed achievable with the 25ns external SRAM. We have fabricated ten boards and populated six of them. One of the six is used as a showpiece, and the other five are installed in various Sun 3/260s around the department. The ability to run realistic programs allowed us to detect several logic errors in the Mosaic processor that would otherwise have been missed.

Mosaic Development/Interface Board R2.0 Our decision to switch from a 3-D synchronous message network to a 2-D asynchronous network made it necessary to design a new development board. We have also taken the opportunity to modify the processor-memory and processor-VME interface to increase the clock speed achievable using our existing stock of 25ns SRAM chips. By putting the tri-state buffers needed in R1.0 on the CPU chip itself, we have also halved the total number of IC chips needed, thus making room available for installing external connectors to bring out the uncommiteed channels of the four MM3 chips. The R2.0 can thus be used as a host interface for the Mosaic multicomputer and as part of a test structure during the manufacturing of the multicomputer modules.

The 25ns SRAM allows the board and the processor to run reliably at a speed of 30 MHz. With a set of 15ns SRAM, we were able to run one of the MM3 chips at 35 MHz. The development board allowed us to discover and study a few problems with the router-processor interface. It also allowed us to discover a logic error in the condition-code register — an error that is manifested during context switching — that would never have been discovered under normal testing procedures.

Mosaic C Compiler

We have customized the Gnu C Compiler (gcc) kit to produce Mosaic assembly language code and a new assembler to produce Mosaic machine code to support the development of a compiled and dynamically-linked run time environment. As the authors of gcc claimed, the target for gcc is a CPU with 32 bit integers. For the 16 bit Mosaic, the compiler produces sub-optimal codes. We are in the process of refining the compiler to produce better code. We also need a new assembler to support the dynamic linking of object codes and to handle a set of compiler-generated directives.

Current Activities

With all of the silicon parts now tested, we are assembling the full Mosaic C node, a chip that will be approximately 9mm$\times$10mm in 1.2μm MOSIS SCMOS. Much of the effort is in developing the built-in-test code rather than in assembling the geometry.

Negotiations with HP have been completed for the chip fabrication and packaging development for a first run of three 8$\times$8 Mosaic boards.

A complete report on the packaging, manufacturing, testing, and early use of the Mosaic C is anticipated for the next semiannual technical report.

2.2 Second-Generation Medium-Grain Multicomputers*

Chuck Seitz, Joe Beckenbach, Christopher Lee, Jakov Seizovic, Craig Steele, Wen-King Su

Our Caltech project continues to work closely with the DARPA-supported Touchstone project at Intel Scientific Computers. The principal research activities in this period were concerned with mesh-routing chips for the Delta prototype (see section 4.8).

The project currently operates the following multicomputers: 8-node and 64-node Cosmic Cubes, a 128-node Intel iPSC/1, a 16-node Intel iPSC/2, and 32-node and 192-node Symult S2010 systems. The 192-node S2010 system is, of course, the preferred machine for users, and is accessed through the Caltech Concurrent Supercomputer Facilities. Utilization has been at a level of approximately 90% of the available node-hours.

* This segment of our research is sponsored jointly by DARPA and by grants from Intel Scientific Computers (Beaverton, Oregon) and Symult Systems (Monrovia, California).

3. Concurrent Computation

3.1 Fine-Grain-Multicomputer Programming Systems

Nanette J. Boden, Chuck Seitz

Significant advances in several areas of fine-grain multicomputer software have been made during the past six months.

Removal of Undue Restrictions

We are continuing our investigations into approaches that remove perceived "restrictions" on fine-grain multicomputer programming methods and on program execution that, uncorrected, could limit the application space of these machines. In previous reports we commented upon the apparent difficulty of permitting message discretion and functions in programs without perhaps introducing violation of the guarantee of message delivery. The argument is as follows: When a process is waiting for the arrival of a particular message, messages received during the interim must be buffered. Since the resources available on a node for this process are quite limited, physical space may not be available that will allow the awaited message to be received. Because we believed that unwanted messages could not be buffered within the constraints of our reactive programming model, we suggested in the last report that the need for the programming abstraction of message discretion justified an engineering solution. During the last six months we discovered a queueing formulation that successfully buffers unwanted messages while using only reactive semantics and our process-creation mechanism. This solution to the queueing problem enables a fine-grain multicomputer node to selectively receive messages *without danger of overflowing its small receive queue.* Thus, we have implemented an extremely convenient programming mechanism that uses only the reactive semantics that are ideally suited for fine-grain machines.

Fine-grain programming is clearly facilitated by the addition of a selective receive mechanism. Many functional programs can be directly translated into fine-grain programs — each function call results in the creation of a new process that eventually responds with the function value. In addition, the selective receive mechanism can be used to remove some of the simplifying assumptions that were made in early runtime systems [Oct 1989 report]. In these runtime systems, process creation was greatly simplified by assuming that if an available reference value exists for the creation of a new process on a remote node, then enough resources exist on that node for the new process. We also assumed that the code for each process resides on each node. The selective receive mechanism can be used to remove each of these restrictions. During process creation, the selective receive mechanism enables a node to wait indefinitely for a reference value to be returned by the physical node that is the eventual host for the new process. If the required code for a process is not available on a particular node, the node can use the selective receive mechanism to postpone processing of messages until the code has been dynamically linked.

Runtime System Development

Since a major goal of the Mosaic project is to provide the user with completely automatic resource management, the most recent runtime systems have focused on exploring different algorithms for process placement, code distribution, node local memory management, remote node memory management, etc. These systems incorporate much of the fundamental elements of the Cantor runtime system that was developed for the Mosaic and briefly described in our April 1989 report. In contrast to the Cantor runtime system, however, the Mosaic runtime systems have been designed so that memory and other resource demands are distributed throughout the multicomputer's available nodes. If the demands on a single node's resources threaten to overflow the available resource, the node can forward the requests or can free some of its own resources by *exporting* data structures. A design goal of this family of runtime systems is that a computation should not fail due to lack of resource until a very high percentage of the physical resources of the entire machine is actually unavailable.

Two runtime systems with different approaches to node local memory management have been developed and written in C. Using existing multicomputer nodes simulating the behavior of a Mosaic node, a Mosaic ensemble simulator has been used to partially debug these runtime systems. Further debugging, analysis, and experimentation will be performed using the Mosaic Software development boards, pending completion of a Mosaic C compiler.

Experimental Programming Notation

Since evaluation of the various runtime algorithm choices depends heavily on the original coding of the user program and on the capability of the compilers, we have been experimenting with a new notation for expressing fine-grain multicomputer programs. Although use of the fine-grain language Cantor provided much insight into the nature of fine-grain programming, the complex compiler and intermediate code of Cantor do not facilitate experimentation with such issues of interest as compiler-assisted resource management. Consequently, we are developing a C-based notation that segments a program into a collection of *definitions* that encapsulate information concerning processes and a collection of *C functions* that express conventional code. The definitions are initially written in a C language extension; a simple compiler extracts information about the processes that may be helpful in process placement and other resource management tasks, and then converts the definition code to conventional C. The C functions and the converted definitions are then compiled together using a conventional C compiler. The purpose of this effort is not to develop another fine-grain programming language, but rather to facilitate experimentation with the compilation and runtime levels of computation. This experimental programming notation is still in the design phase.

3.2 A Pascal Compiler for the Mosaic

Jan L.A. van de Snepscheut, Johan J. Lukkien

We have implemented a Pascal compiler for the Mosaic. The compiler takes a Pascal source and generates code for a single Mosaic chip. The language Pascal has been extended with primitives (derived from CSP) to support the execution of multiple processes on one processor. Communication can be performed between pairs of processes, either on the same processor or on different processors, and is synchronous. In this way we avoid the assignment of bufferspace to communicating processes.

The compiler has been used since the first Mosaics became available. A monitor program, running on a Sun workstation, loads the code into the Mosaics and communicates with the Mosaics in order to implement basic input/output functions (file I/O, terminal I/O). We have used this system to perform some fluid-flow computations.

3.3 The Page Kernel

Craig S. Steele, Chuck Seitz

The previously-described "page kernel" (PK) concurrent programming environment is now operating on the Symult S2010 multicomputer, and several test and example programs have demonstrated the functionality of its concepts. The PK is an evolution of the now-familiar reactive programming model which uses the virtual-memory capabilities of second-generation multicomputers to implement data-sharing mechanisms supporting multiple overlapping address spaces. The programmer accesses shared data structures much as in a shared-memory machine, but without the need for explicit locking to control the problems of concurrent access. The execution of the light-weight reactive processes, called *actions*, implicitly induces atomicity and consistency of data modifications. Poorly coded programs will generally run correctly but with limited effective concurrency; efficiency is improved by eliminating unnecessarily broad data consistency conditions, which may result from naive use of shared data structures. A program formulation that avoids indiscriminate writing to widely-shared data structures maximizes realizable concurrency under the PK.

While performance optimization may require careful design, many details of multicomputer programming are considerably simplified in the PK environment. Message transmission becomes implicit, as does mutual exclusion of concurrent writers to a single datum. The placement of actions and data on multicomputer nodes is handled transparently by the kernel. The physical configuration of the multicomputer hardware is hidden from the programmer; the programmer's only essential concern is to avoid reducing the problem's logical concurrency, as expressed in the program, beneath the physical concurrency, as provided by the available hardware resources. A simple triggering scheme allows actions to be scheduled when associated data structures are changed. Actions are coded in C++, allowing definition of libraries of sharable data types of general utility, such as queue classes.

While both the kernel and the program suite are still under development, preliminary results have demonstrated near-linear speedup for problems with dozens of nodes, hundreds of actions, and thousands of shared data structures.

3.4 Multicomputer C

Marcel van der Goot, Alain Martin

Multicomputer C is a C-based concurrent programming language for message-passing multicomputers. A program consists of concurrent processes connected by channels, and communication and synchronization are done with CSP-like communication actions via the channels. During the past half-year we have worked on a manual and on a revision of the compiler. The manual describes the language design and gives implementations of the new language constructions (like communication actions). It also outlines techniques or alternatives for mapping processes on machine nodes, using time-outs in the selection of communication actions, prioritizing processes, sharing data, handling interrupts, and implementing I/O.

One change was made to the language, with the introduction of multi-sender channels. Such channels are useful to collect results computed by multiple processes in a central point, and they can often be used as an alternative to shared variables.

The compiler was reviewed and updated to allow for better diagnostics and, in particular, to facilitate code generation for a wider range of machines. The original compiler generated ANSI C for a SUN workstation. The new version is able to deal with some machine dependencies in the generated C (useful because for many machines no C compilers that implement the full standard are available), and can generate code for true multicomputers. Generating code for multicomputers involves additional difficulties when not all processors are identical, as different code may be required for different processors. Multicomputers also require a special effort to implement a global namespace for functions and processes; this can only be done at link-time. We expect the compiler to be running by the end of October, generating code for SUN workstations and for multicomputers running CE/RK. Adaptation to other medium-grain multicomputers should be straightforward.

3.5 A Concurrent Wire-Routing Program

Su-Lin Wu, Chuck Seitz

We are attempting to use multicomputers to generate wire routings of circuit boards and VLSI chips. To produce nearly optimal routes, the program will use a cost function based on physical considerations, and will also allow interaction with the user.

We have adapted the Lee-Moore algorithm for finding the shortest path between two points to a method of finding good (cheap) routes of n points. By taking advantage of existing electrically equivalent wires, this heuristic gives better routes than simply applying the two-point algorithm repeatedly. As with any attempt to solve an NP-complete problem, the n-point Lee-Moore algorithm has pathological cases, but wastes an acceptably small amount of wire in routing these.

There are easily exploitable concurrencies in this method. In the Lee-Moore algorithm's propagation phase, the parts that make up the expanding wavefront are independent and their activities can be computed concurrently. To disperse the wavefront rapidly to the nodes of the multicomputer, a wrap mapping is used. The nets and sub-nets that comprise the netlist may also be routed independently if they are confined to areas that do not intersect. The user may specify the order of the nets to be routed, but within that order the program will have some latitude to maximize concurrency. This is the classical manager–multiple-worker formulation in which the boss gives instructions to a manager who must then work within those constraints to divide the set of tasks among additional workers so that the work is completed in the shortest possible time.

Cost is based on the idea that there are limited resources available. The cost function adapts the value assigned to area, vias, and other structure to enforce behaviors desired by the user. The user assigns such costs to reflect the extent that allowing a wire to pass through that area will deplete the user's supplies.

4. VLSI Design

4.1 The Asynchronous-VLSI Project

Alain J. Martin, Dražen Borković, Steve Burns, Pieter Hazewindus, Tony Lee, José Tierno

As the project is entering its second phase, it may be appropriate to recapitulate its objectives and current status.

We have developed a novel design method for high-performance asynchronous VLSI systems. There are two main directions to the research: The first one is a high-level synthesis approach to the design of digital VLSI circuits. In our implementation of this idea, a circuit is first described as a concurrent computation in a high-level notation. It is then "compiled" into a circuit by semantics-preserving transformations. Consequently, the circuits obtained are correct by construction. (Typically, the object code is a network of CMOS pull-up and pull-down cells connected to a pad frame.)

The second aspect of the research is a novel approach to asynchronous design. We have now a complete design methodology that includes general techniques for both control and datapath, as well as a repertoire of basic cells that includes synchronizers and arbiters, generalized C-elements, bus controllers, registers, sequencing cells (D-elements), *etc.*

Although CAD was not originally a main objective of the project, an important CAD activity has developed in support of the rest of the research, since it has always been a (self-imposed) requirement on the project to test the proposed ideas by actual chip design. The set of CAD programs developed include tools for design (automatic compilation), analysis (simulation and critical path analysis), optimization for performance (transistor sizing), and physical layout (cell generation, placement and routing).

First Results

We now have a general method for designing asynchronous (and quasi delay-insensitive) circuits for any type of digital computation. We have demonstrated the practicality of the method on a series of actual MOSIS CMOS designs. All fabricated chips have been found correct on first silicon. The main chips designed include stacks, queues, routing automata, multiplier, distributed mutual exclusion (arbitration), special-purpose processor ($3X + 1$ engine); and culminated in two designs of a general-purpose 16-bit microprocessor running at 18 MIPS in 1.6μm CMOS. Since the design of this microprocessor included all main aspects of digital design (except arbitration, which was demonstrated in previous chips), the completion of the processor design was understood to be the completion of the first phase of this project.

The results of the first phase of the project can be summarized as follows: First, at the system design level, the design experiments (in particular the microprocessor) have demonstrated the flexibility and versatility of the high-level notation that we have developed. The conclusion we have drawn is that most high-level design issues are indeed concurrency issues that are best solved with the techniques and notation of concurrent computation. These results anticipate a unique design methodology for digital systems across an increasingly moveable hardware/software boundary.

Second, it is possible to design asynchronous circuits that are efficient both in area and speed. (At this point we believe that there is an irreducible area penalty compared to synchronous design, but it falls well within acceptable margins given the abundance of real-estate provided by modern technology.) With respect to speed efficiency, our experiment with demanding designs like the control part of the microprocessor indicate that rather sophisticated techniques have to be used in order to reduce the penalty due to asynchronous sequencing (completion trees, handshaking, *etc*) to an overhead comparable to that of clock skew. However, once this objective has been achieved (which is the case of the control part of the microprocessor), the asynchronous design can reap the benefits (when compared to synchronous design) of the flexible execution times and extensive concurrency provided by the concurrent computation approach.

Third, quasi delay-insensitive VLSI design exhibits remarkably robust behaviors. As previously reported, the microprocessor is operational across an unusually broad range of VDD voltages and range of temperature. Another remarkable feature of this type of asynchronous designs is that the power consumption is about an order of magnitude smaller than that of an equivalent synchronous design. This characteristic is of course very attractive for the design of future multicomputers that will require packaging a very large number of chips in a small volume, and also for battery-operated applications.

Designer-Assisted Compilation

The second phase of the project will concentrate on the system-level design, with a redesign of an improved version of the processor as the first step towards an entirely asynchronous system. However, we will focus first on improving the CAD tools, for the following reasons:

Our attitude towards automatic compilation has changed significantly during the project. Whereas we originally thought that we would soon use an automatic compiler for chip synthesis, we are now convinced that entirely automatic compilation will not produce high-performance design in the near future. We have an automatic compiler that has been operational for several years already. The compilation is "syntax-directed," *ie,* the compiler produces a standard circuit implementation for each syntactic construct of the language. The final design is improved by "peep-hole" optimizations. Coupled with a standard cell-layout-generation program, the compiler has been used for several automatic designs, *eg,* for a torus-routing chip. Although such a compiler is an excellent tool for rapid prototyping, our first attempt at using it for the control part of the microprocessor convinced us that we will never get the performance we were aiming for if we follow the route of automatic compilation: The performance of the critical path of a chip like the microprocessor is just too sensitive to minor optimizations that an automatic compiler cannot even generate, let alone evaluate.

Our approach now is that of *designer-assisted compilation.* Each step of the synthesis method is applied automatically to produce a number of alternative designs. These different solutions are compared and the best (according to some criterion decided by the designer) is selected for the next step of the compilation. The procedure also includes backtracking. This approach relies on tools for performance evaluation and optimization.

The second generation of synthesis tools that we envision will integrate simulation, performance evaluation, and optimization (transistor sizing). The designer will be able or perhaps even required to make choices at different stages of the synthesis based on the results of the previous stage. As a first step toward such a system, we are designing a program for the synthesis of a straightline program into CMOS chips. The final program will include automatic cell synthesis, transistor sizing, placement and routing.

4.2 Testing Self-Timed Circuits

Pieter Hazewindus, Alain Martin

A self-timed circuit is described as a production rule set, implementing a handshaking expansion of a high-level program. For testing purposes, we use the single stuck-at model. For this model, an input or an output of a gate is either permanently at a high voltage (stuck-at-1) or at a low voltage (stuck-at-0). A circuit is tested by executing the handshaking expansion that it implements.

We are currently analyzing the testability of the control part of the first self-timed microprocessor. We have added the required testing circuitry. The revised circuit will be sent off for fabrication shortly.

4.3 Gallium Arsenide and Self-Timed Circuits

Alain J. Martin, José A. Tierno

The same techniques used for designing self-timed circuits in silicon can be applied to gallium arsenide (GaAs). However, the basic gates that are used in the implementation have to be carefully designed for reliability, noise immunity, power consumption, etc. A design style and a whole family of gates was developed so that they can be used in an "oblivious" manner, that is, requiring minimal concern for the electrical characteristics of the circuit.

A special set of pad drivers and receivers was designed to interface with this technology on chip and similar pads or $CMOS$ circuits off chip. Work is in progress now on two chips, one already in the fabrication queue and the second to be submitted before October 24th. The first circuit contains several different buffers, the basic synchronization structure for self-timed circuits, as well as smaller test features for gates and pad drivers and receivers. The second circuit contains a self-timed register file. These are being fabricated using Vitesse's enhancement depletion mode process.

4.4 Automatic Compilation of Straightline Handshaking Expansion

Dražen Borković, Alain J. Martin

As a first step towards the next generation of synthesis tools, we are designing a program for the synthesis of straightline program into CMOS chips. The final program will include automatic cell synthesis, transistor sizing, placement and routing.

The problem of positioning the state variable transitions for programs containing conditional branches ("IF" statements) was proven to be NP complete. An algorithm that solves the problem in $O(n^k)$ was developed, where k is the number of guarded commands in the "IF" statement, and n is the length of the longest guarded command.

A program for automatic generation of minimal production rules for straight-line handshaking expansions was developed, as well as one for the reset of the generated circuits. The program allows the designer to explore different options and backtrack in order to achieve the desired performance. It can also be coupled with number of other tools: inverter reshuffling, performance analysis, and cell-layout.

4.5 Automatic Custom Cell Generation and Layout

Tony Lee, Alain J. Martin

We have developed a program which generates CMOS magic cells for implementing a given set of production rules. The input production rules must be in disjunctive-normal form and the sizes of the transistors in the production rules may be specified. The output generated by this program can be used directly by *gladys*, our placement and routing program. Thus, we now have tools that will take an arbitrary set of production rules (provided it is in disjuctive-normal form) and generate a complete layout for it.

4.6 Self-Timed Arithmetic

Tony Lee, Alain J. Martin

Consider the simple shift-and-add method of multiplying two n-bit integers. If we ignore additions by zeros, then the number of partial-sum additions performed in the multiplication is determined by the number of ones in the binary representation of the multiplier. Furthermore, for each addition, the length of the longest carry-chain is a function of the partial sum and the multiplicand. In general, the time involved in performing an arithmetic operation is greatly affected by the values of the input data. Nevertheless, this inherent variance in the latency of arithmetic operations is usually not exploited by simple synchronous systems that, for the sake of timing uniformity, operate under the worst-case delay assumption. Such a pessimistic assumption is not needed for asynchronous systems since they function properly regardless of the actual time it takes for them to perform a given computation.

Thus, we believe that efficient self-timed arithmetic circuits can be designed so that they can take advantage of the shorter latency for cases of favorable inputs and thereby yield better average performance than synchronous systems.

Our approach is to start with a high-level description of the arithmetic algorithm and then apply our synthesis method to transform the description into self-timed circuits. We have had encouraging results with the $3X + 1$ engine and the simple ALU used in the microprocessor. Currently, we are working on a multiplier that implements the shift-and-add algorithm. The layout of the multiplier has been completed and its functionality has been verified. We are now working on increasing its performance by using our timing analysis tools to size the transistors.

4.7 Performance Analysis of Linear Arrays of Asynchronous Processes

Steve Burns, Alain Martin

We have developed a method for determining the performance of linear arrays of repetitive asynchronous processes. The complexity of the procedure is related to the size of the single replicated process and not to the size of the collection of instantiated processes. This method is of great help in designing optimal pipeline stages for a computing engine, as well as *FIFO*s and stack stages for memory systems.

The method is an extension of the performance analysis techniques described in the last semi-annual report, at *TAU '90*, and in Steve Burns' forthcoming PhD thesis. Linear timing functions — the principle tool used to reduce the analysis of an infinite repetitive computation into the analysis of a finite structure — can also be used to reduce the analysis of the computation performed by an infinite array of processes into the analysis of a finite structure. Thus the performance of very large systems can be determined with very little computational effort.

These techniques have been used to compare the performance of several possible implementations of buffer processes. The implementations that achieve the highest performance have been cataloged for future use. The techniques have also been used to show that particular designs developed by other researchers are not optimal. We suggest changes to these designs which improve performance.

4.8 Fast Self-Timed Mesh-Routing Chips

Chuck Seitz

Two design-fabricate-test iterations of the Frontier mesh-routing chips (FMRC) for the Intel Touchstone Delta prototype were completed in this period. These FMRC2.2 and FMRC2.3 chips incorporated a number of improvements, described in our previous report, and aimed at increasing the reliability of high-speed data transfer on the channels.

The first mesh-routing chips fabricated in 1.2μm CMOS, the FMRC2.2, functioned correctly, but, due to the designer's misunderstanding of correcting for velocity saturation, the output drive was excessively asymmetrical. A small investigation of the output characteristics allowed the pad drivers to be corrected in the FMRC2.3 — this chip has been tested extensively both at Caltech and at Intel, and appears to be completely adequate for the Touchstone Delta prototype.

These same lessons about pad drivers have been incorporated into the pad frame of the Mosaic chips that are fabricated on these same runs.

The use of a 5-mil pad pitch in this 4492$\times$4492, 132-pin, semi-standard frame, an experiment that Wes Hansford at MOSIS encouraged, has caused no problems

4.9 A Silicon Architecture for Adaptive Cut-Through Routing

Mike Pertel, Chuck Seitz

Previous theoretical studies have shown that the performance of multicomputer networks can be increased by using adaptive cut-through routing in place of oblivious techniques (see Ngai and Seitz, 1988). State-of-the-art oblivious routers, such as the FMRC routers described in the preceeding section, can route a packet between a given pair of nodes along only one path, regardless of the state of the network. Routers that can choose any of several paths exhibit greater utilization of network bandwidth, better traffic balancing, and increased fault tolerance. To implement the ideas from the earlier work, we have developed a simple architecture for performing multipath routing. The architecture confines the design space to allow detailed simulation, but does not appear to limit flexibility.

A routing algorithm for multicomputer networks must be deadlock-free to be practical. The oblivious routers avoid deadlock by using dimension-order routing; a multipath router requires another mechanism. A key idea from the theoretical studies is to avoid deadlock by misrouting. Deadlock is impossible if a router never blocks its input channels. By using any available output channel, it can rid itself of packets that it cannot buffer. The earlier studies showed that even if misrouting is used to avoid deadlock, it can be made very rare by throttling the network traffic. The architecture supports deadlock avoidance by being able to misroute a packet from any input to any output. The congestion control required to cause misrouting to be required only rarely is handled by requiring packet sources to await an acknowledge message from the destination before further sending. This technique also assures packet-order preservation despite the existence of multiple paths between source and destination.

Once the problem of deadlock is resolved, we are free to consider any number of ways to route packets. The path of a packet in an oblivious routing network is fixed by the deadlock-avoidance scheme, but misrouting eliminates this restriction. The adaptive router can forward an incoming packet along any profitable output channel. The exact definition of what constitutes a profitable channel assignment depends on the specific routing algorithm, but in general a channel is profitable if it reduces the packet's distance from its destination. We can avoid making misrouting a special case by regarding any output assignment as profitable when input blocking becomes imminent. Other than this, the definition of a profitable assignment is left open, thus we maintain the flexibility to implement virtually any specific algorithm. Since there will generally be more than one profitable output for a packet, it is necessary to choose one assignment from multiple candidates. Moreover, output assignments must be made fairly in the sense that any packet awaiting an output eventually gets one.

An architecture to support this framework must be able to connect any input to any output for misrouting. It must also have buffering to allow packets to wait for profitable outputs to become available without blocking. This suggests a simple structure with FIFOs and a crossbar. By placing the FIFOs on the input channels, we can use the filling of the input queue to trigger misrouting. The requirement that access to output channels be fair between the inputs, and the necessity of ensuring that each input is connected to at most one output (and each output to at most one input) suggest a central decision structure. An important lesson from the theoretical studies was that simultaneous arrival of multiple new packets needing output assignments is very rare. This suggests that the hardware for reading/writing packet headers and computing/making profitable assignments can be shared by all inputs, rather than duplicated, with negligible increase in average assignment latency.

The incoming packets awaiting output assignments are serviced sequentially. This eliminates the need to duplicate logic for computing assignments, and trivializes the problem of mutual exclusion between assignments. More importantly, by serving the inputs round-robin, we guarantee fair access to the output channels. When an input is served, we compute the profitability of each output in parallel. If no profitable output is free, no assignment is made. If at least one profitable assignment is possible, then one is chosen. In the case where the profitability of an assignment is discrete (eg, binary), we arbitrarily select one of the most profitable assignments. If profitability is continuous, we merely choose the most profitable. We assume that the determination of output profitability can be done in one cycle. Based upon the theoretical studies, this is a reasonable assumption. Given that it only takes one cycle to service an input, and that simultaneous header arrivals are rare, sequential service does not compromise efficiency, and it solves the problems associated with doing channel assignment quite cleanly.

We have developed a simulator for the architecture described. The profitability of an assignment is determined by a small C function. We are proceeding to compare the performance of several definitions of profitability under different traffic conditions to select the best alternative. The simple architecture we have described has the flexibility to implement the promising algorithms developed during the earlier theoretical studies, yet it dramatically reduces the design space to explore. In its simplicity, the architecture demonstrates that it is not difficult to design a practical adaptive router.