SUBMICRON SYSTEMS ARCHITECTURE PROJECT
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Submicron Systems Architecture Project

Semiannual Technical Report

Caltech Computer Science Technical Report

Caltech-CS-TR-93-37

1 November 1993

The research described in this report was sponsored by
the Advanced Research Projects Agency of the Department of Defense,
and monitored by the Office of Naval Research.

SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

Caltech-CS-TR-93-37

1 November 1993

Reporting Period:	1 April 1993 — 31 October 1993 (7 months)
Principal Investigator:	Charles L. Seitz
Faculty Investigators:	Alain J. Martin Charles L. Seitz Jan L. A. van de Snepscheut

Sponsored by the
Advanced Research Projects Agency
of the Department of Defense

Monitored by the
Office of Naval Research

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This report is a summary of research activities and results for the seven-month period, 1 April to 31 October 1993, under the Advanced Research Projects Agency (ARPA) 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 multicomputers (message-passing concurrent computers), and includes related efforts in concurrent computation and VLSI design.

1.3 Personnel changes

On 1 October 1993, Charles L. Seitz started a one-year leave of absence (without salary) in order to start a company aimed at transferring certain of the project's research results in the areas of multicomputers and VLSI routing devices to commercial practice. Dr. Seitz's affiliation with Caltech and his appointment as a Professor of Computer Science are unchanged, and he continues to be actively involved in the research project.

1.4 Highlights

- Scaling up the Mosaic multicomputer to more nodes has been delayed by our subcontractor, but the development of its programming system has made major strides in this reporting period (chapter 2).
- Interesting, practical results have been achieved for advanced multi-precision arithmetic algorithms and routines, with precise, proven error bounds (section 3.4).
- An improved communication protocol was developed for multicomputer communication channels, and a VLSI chip that implements this protocol was designed (section 4.1).
- We have a fully functional GaAs asynchronous microprocessor running at 100MIPS (section 4.4).
- We have designed a cubic algorithm to limit the lengths of transistor chains in the CMOS implementation of free-logic asynchronous circuits (section 4.7).

2. The Mosaic Project

The Mosaic is an experimental, fine-grain multicomputer based on single-chip nodes. Our previous (April 1993) semiannual technical report included a detailed description of the Mosaic hardware and programming system.

A 256-node Mosaic was assembled in September 1992. After replacing several chips that failed during the first several-hundred hours of operation, this system has been operating reliably for the past year; there have been no failures, and no memory or communication errors. Production of the 8×8-board assemblies for the Mosaic is subcontracted to Hewlett-Packard. Scaling this successful design to a larger system should have been routine, but Hewlett-Packard has repeatedly delayed this production.

Research efforts that are using the prototype Mosaic multicomputers for programming experiments are described in sections 2.2 and 3.1.

2.1 Mosaic C 8×8-board Production

Chuck Seitz, Wen-King Su, Arlene DesJardins

Although the prototype 8×8 boards operate flawlessly in a 256-node Mosaic system that is being used for programming-system and application experiments, a series of production delays by our subcontractor, Hewlett-Packard (Corvallis), has prevented us from scaling the Mosaics to larger sizes. Previous problems with Hewlett-Packard had already delayed the project. The problems have included erratic chip-fabrication yield, infant-mortality failures from chips fabricated on one or two incorrectly processed wafers, chip damage caused by the inner-lead bonding, and chip damage caused by inappropriate wafer-test parameters.

A wafer lot that came out of fabrication in December 1992 provided enough good chips to build 10 more 8×8 boards. Many of these chips were damaged in the inner-lead bonding, but we finally received 5 8×8 boards in May 1993. All of these boards tested as good, were monitored during a burn-in period, and exhibited no failures. One 8×8 board was sent to USC/ISI for use in the ATOMIC project, and another to Aerospace Corporation for use in an experimental satellite network.

After previously reported delays in starting production wafer lots, Hewlett-Packard agreed to one wafer-lot (24 wafers per lot) start per week beginning in March 1993. The yield was very good on the early lots, so we stopped production in May 1993 after 12 lots. According to wafer testing, these 12 wafer lots produced ≈17,000 working Mosaic chips.

Hewlett-Packard people told us that they expected board production to begin again in June 1993 from the chips produced on wafers started in March 1993; however, no board production has started even yet. The reported delay this time is in retesting chips in the TAB frames after they have been inner-lead bonded.

Ordinarily, we would have responded to these excessive delays by cancelling our contract with Hewlett-Packard; however, there would be little cost saving in cancelling the contract at this time. Hewlett-Packard has circuit boards and other materials for building 150 8×8 boards, and we have enough chips to build twice this many boards. Our project is responsible for the costs of the circuit boards and other materials, and the outer-lead bonding and testing is a very small part of the overall cost of an 8×8 board.

2.2 The C+− Programming System

Nan Boden, Jakov Seizovic, Chuck Seitz

A version of C+− is currently operational and is being used for general-purpose multicomputer programming by members of our research group. This version supports all the essential aspects of C+−, but does not include all the planned constructs. (See [Caltech-CS-TR-93-18] for a full exposition, and [Caltech-CS-TR-93-32] for an Introduction to C+−. Both reports are available by anonymous ftp.) The unsupported constructs are of interest primarily for advanced programming projects, such as operating systems and simulations.

We have defined and implemented the interface between C+− programs and the runtime systems for two different hardware platforms:

1. *Networks of workstations running the Cosmic Environment.* This programming environment includes a G++-based compiler and linker and a GDB-based debugger. Message-passing in C+− (*i.e.,* atomic functions) is implemented using the "x" primitives of the Cosmic Environment. The C+− runtime system is quite simple, approximately 500 lines of C+- code, and has been operating reliably for the past year.

2. *Mosaic ensembles.* C+− code is generated for Mosaic ensembles using a G++-based compiler. Because of the small amount of memory available on each node, the entire user program is not loaded onto each node. Instead, a code-splitting linker was developed to split the user program into code pieces, each of which contains an atomic function. Each of the code pieces is then loaded onto a subset of nodes of the ensemble. The runtime system for the Mosaic ensemble automatically manages access to the code pieces. The report [Caltech-CS-TR-92-10] reports on a prototype version of this runtime system. A version of that Mosaic runtime system that will be released to users is currently under development.

Our agenda for C+− includes cleaning up the user interface so that we can release an alpha version to interested users for use on networks of workstations. The subsequent release of C+−, which will be for use on workstations and Mosaic ensembles, will include support for all planned programming constructs.

3. Concurrent Computation

3.1 The Modula-3D Programming System

K. Rustan M. Leino, Jan L.A. van de Snepscheut

Our work on Modula-3D is an on-going effort to explore the possibilities and limitations of safe, high-level programming notations on fine-grained multicomputers. Most programming of multicomputers has been done in languages to which send and receive operations were added, forcing the programmer to deal with details of each communication. By extending the language instead of providing a library of routines, we are able to provide a more efficient implementation, and provide more support for detecting program errors.

The Modula-3D report was completed in May 1993. It contains the definition of Modula-3D – a distributed extension of Modula-3 – , a flavor of the programming style of Modula-3D, and some performance numbers from our implementation of the language on the Mosaic multicomputer.

The previous Modula-3D compiler, like the DEC SRC Modula-3 compiler from which it stems, translates to C code. We have now written a new back-end for the compiler to produce assembly code directly. The new Modula-3D compiler produces an abstract syntax tree for each module. These are seamed together at link time, at which time code is produced. The code generator can thus make use of more information than is available at the time each abstract syntax tree is produced. We hope to use the extra information to produce better code and to detect more program errors. We have not yet compared large pieces of compiled code by this compiler with the code produced by other compilers, nor have we measured the compilation and link speed of the new compiler.

3.2 Program transformation

Jan L.A. van de Snepscheut

Developing programs through transformation is a programming method that has been widely advocated but not so widely practiced. The idea is to start with an "obviously correct" but probably inefficient program and then transform it by stepwise application of correctness-preserving transformation rules. Although it is attractive in principle, it has not been attractive in practice for two reasons. First, it is a lot of work because of the large number of steps required. Second, most transformation rules are valid only under certain conditions that are easily forgotten.

We are developing an editor that maintains the current program text. It keeps a history of the program's development and carries out the transformation steps. A theorem prover is used to check the validity of transformation steps. Interaction with the prover is made as invisible as possible. It shows only in counter examples produced by the prover if a step is found to be invalid, or in a warning message when the prover is unable to verify or disprove the validity. Although the latter case is unavoidable, it hardly ever occurs in practice once the proper library of rules has been developed.

We have developed libraries for equational reasoning about functional programs and for stepwise refinement of imperative programs. Nontrivial examples have demonstrated both the strong and the weak characteristics of the editor, and an improvement in the user-interface has been one of the beneficial side-effects thereof. We have gained new insights in the mathematical properties of quantifiers that have led to an immediate simplification of the editor. We are experimenting with a mechanism for defining new notations that were not built in.

3.3 Refinement of Concurrent Programs

H. Peter Hofstee, Jan L.A. van de Snepscheut

Work continues on a workable semantics for a concurrent programming language with synchronizing communication. In the past half year we have obtained the following results:

- Within our model for stateless communicating processes we now make a distinction between processes which we consider to be implementable, and those which are not.
- A class of implementable stateless communicating processes has been found which is closed under sequential, parallel, and choice composition and hiding of internal communications.
- We have found a model which allows for the treatment of processes with state. The class is somewhat richer than CSP, as it allows for processes to be composed from sequential *and* parallel components. However, processes composed by parallel composition can exchange information by communication only; variables may only be shared among processes composed sequentially.
- A relation has been established between the refinement ordering proposed for the latter class, and the standard refinement ordering for sequential programs. This opens up the possibility of refining a sequential program by a concurrent one.

3.4 Multi-precision arithmetic

Robert Harley

Many current packages for multi-precision arithmetic provide high precision without guaranteeing high accuracy in any precise sense. In some cases authors claim accuracy that is not justified by theoretical arguments. Furthermore, although advanced algorithms have been known for many years, few packages use them.

These two problems may be closely related: theoretical descriptions of algorithms typically describe time behavior and error bounds only up to asymptotic order. An in-depth understanding of the algorithms and their mathematical framework is required before precise error bounds can be determined and before non-trivial optimizations can be applied to reduce constants hidden in order notation.

The difficulty of performing these tasks for advanced algorithms leads, in our opinion, to the widely held misconception that algorithms which are difficult to prove and understand do not surpass naive algorithms for problems which are of useful size.

We are currently calculating precise error bounds and discovering machine-independent optimizations for arithmetic operations including algebraic, trigonometric and inverse trigonometric, exponential, logarithmic and hyperbolic functions.

We have implemented several advanced algorithms to date, and our ultimate goal is to provide a library of very efficient routines with precise, proven error bounds. Our code chooses an algorithm adapted to the input operands' sizes. As a result it is at least as fast as other packages tested, with the exception of one in whose results we place little confidence.

4. VLSI Design

4.1 Dialog Channels and the Dialog Chip

Alan Kulawik, Chuck Seitz, Wen-King Su

Background. A Mosaic or mesh-routing-chip channel consists of 11 wires:

- 8 data bits,
- 1 bit to mark the tail of a packet,
- a self-timed request signal, and
- an opposite-going self-timed acknowledge signal.

Each 8-bit flow-control unit (flit) is conveyed on the channel by the sender driving the data and tail bits, and generating a transition of the request signal. Another flit can be sent only after the receiver generates a transition of the acknowledge signal.

This zero-slack (non-interference) request/acknowledge protocol provides both timing and flow control. Between Mosaic C nodes or between Elko mesh-routing chips that are physically close, the period for transferring a flit is ≈16ns, corresponding to a bandwidth somewhat in excess of 60MB/s. However, when this signalling protocol is used over long distances, such as through cables, the period for transferring a flit increases with the round-trip delay through the cable. In addition, a fault in the request or acknowledge signals of a communication channel, or a fault in a node, blocks packets into the network. The blocking of a single channel will generally lead to additional blocking and eventual deadlock of the entire network. The only way to restore the network to an operating condition is to apply a full-network reset.

The Caltech Slack chips described in previous reports translate between this zero-slack protocol and a slack protocol that is similar to other streaming or transmission-line-write-ahead protocols. A slack channel is a relatively slight recoding of a Mosaic channel: It consists of exactly the same signals, but, for a slack k, corresponding to k flits of buffering in the slack receiver, the slack sender may get k flits ahead of the acknowledges from the slack receiver. Although slack chips allow full-bandwidth operation over long distances, they do not accommodate continued operation in faulty networks.

The USC/ISI ATOMIC LAN project started using Caltech slack chips this summer, and demonstrated full-speed operation over 100-foot cables. We learned of two practical difficulties from the ATOMIC project's use of these slack chips: (1) Depending on its state, the slack receiver produces acknowledges at different rates, which requires more careful tuning of the cable-driver and -receiver circuits than if the slack circuits operated at a fixed frequency. (2) When the error of a lost acknowledge occurs, the slack sender has incorrect information about the state of the slack receiver. It would be better to employ a protocol that was self-synchronizing.

Dialog Channels. Dialog channels retain the flow-control and packet-framing characteristics of Mosaic channels, but employ a different signalling protocol in order to provide high-bandwidth communication over long distances, continued operation in the presence of faults, and a number of other features. In particular:

- Dialog channels provide flow control, but with sufficient slack to allow high-bandwidth communication over long cables. The flow-control mechanism is analogous to the simple X-off/X-on (control-S/control-Q) scheme used in RS-232 telegraphy. The dialog receiver determines the amount of slack; the operation of the sender is simple and independent of the amount of slack.

- Dialog channels do not block with faulty channels or nodes. If a dialog channel is disconnected or contains stuck-at faults, or if the destination node is powered off or faulty, packets are dropped instead of blocked.
- Dialog channels can operate in either an asynchronous or a fixed-frequency mode. The fixed-frequency mode of operation regulates the flit rate, can be used to optimize signal recovery at the receiving end of long electrical cables, and simplifies interfacing with synchronous media such as optical fiber.
- Dialog channels provide a means of reseting individual channels, and an open-ended set of possibilities for conveying control information through a multicomputer message-passing network or a local-area network.

Dialog-Channel Signals and Control Symbols. The nominal implementation of a byte-wide dialog channel consists of 9 wires: 8 data bits (0-7), and a signal to distinguish between data flits and control symbols (d). All signals propagate in the direction of the channel. The data bit and d are transition-encoded (NRZ encoding); a transition indicates a binary 1, and the lack of a transition indicates a binary 0. A character with d=1 conveys a packet-data byte on the data wires, whereas a character with d=0 conveys a control symbol on the data wires. The control symbol encoded with the data bits all being 0 is the NULL control symbol, which is, in effect, discarded by the dialog sender.

There is no separate wire for the tail bit. Instead, the tail flit is identified by being followed by a GAP control symbol.

There is no acknowledge signal. Flow control is, instead, accomplished by the sender responding to STOP and GO control symbols that the receiver injects on the opposite-going channel. (Mosaic channels are always used in bidirectional (full-duplex) pairs.)

Characters may be sent either only when required, as with a self-timed request, or the characters may be sent at a fixed frequency. These two modes of operation are equivalent at a channel input. Operation of a dialog channel in fixed-frequency mode has advantages for signal recovery at a channel input, and can be used to simplify interfacing to synchronous media such as optical-fiber channels. The NULL control symbol is used to fill unused cycles.

Control symbols are not part of the flow-control discipline for flits, and have priority over flits. Control symbols are injected immediately (in the next time slot) on the sending end of a dialog channel, and are processed immediately on the receiving end. Between two modules connected by a pair of dialog channels, the control symbols allow an arbitrary "dialog" of control information to be superimposed on the packet traffic. (One may, equivalently, regard a physical dialog channel as consisting of two virtual channels in which flow control is not required for control symbols, but is required for data flits.)

The ATOMIC-Dialog Chip. Our first implementation of a Dialog channel is a translator between Mosaic and Dialog protocols. This chip will find immediate application and a test platform in the USC/ISI ATOMIC project; this chip is, accordingly, called the ATOMIC-Dialog chip.

Each 132-pin, 1.2μm SCMOS chip contains two independent, fixed-frequency transmitter–receiver pairs. Each transmitter converts a Mosaic channel input into a Dialog channel output, and each receiver converts a Dialog channel input into a Mosaic channel output. The two units in each pair are coupled and parameterized to operate at the 60MByte/s rate at which 30MHz Mosaic nodes can source and sink packets, and for a cable of 40m maximum length.

The ATOMIC-dialog chip has been a challenging design that has required unusual circuit and organizational approaches. In order to minimize the fallthrough time and the flow-control overhead, the slack buffer is implemented as a register-based FIFO. In order to satisfy speed requirements, this FIFO is double-banked. The input circuitry and FIFO operate asynchronously; it is only at the transmitter that the pipeline-synchronized flow of packet data and control symbols becomes synchronous.

Most of the area of the dialog interface is devoted to the slack buffer. At 40m and 60MBytes/s, a cable with a typical propagation velocity of 0.6c has a round trip delay of 27 character periods. The Dialog flow-control protocol is less efficient than the pure slack protocol; more than twice this many registers are required to provide slack to start, slack to stop, and hysteresis between these limits.

The ATOMIC-Dialog chip, which includes two Dialog translators, is a 7.5mm$\times$5.3mm chip in 1.2μm SCMOS. This pair of dialog translators will be inserted into the ends of each existing Mosaic-channel link, converting it to a Dialog channel. More highly integrated components will be developed after this approach is tested.

4.2 The "Dot-Box" Experiment

Chuck Seitz, Wen-King Su, Jakov Seizovic, Nan Boden, Alan Kulawik, Charles Grosjean

In order to compare the consequences of different styles of self-timed design, we formed two teams from students in the second and the third terms of our VLSI-design class. Both groups produced designs for a pipelined dot-product device. The chip accepts as inputs two streams of 16-bit, two's-complement-integer vectors and produces as an output a 24-bit, rounded, scalar product of the two vectors. The basic architecture of the chip was intended to be a pipelined 16$\times$16 array multiplier followed by an accumulator and a rounder. The task was made more interesting for the students by posing the experiment as a competition, in which the metric to be minimized was the product of chip area and cycle time. The chips designed could also be compared with other designs reported in the literature, both synchronous designs and asynchronous designs produced by other design styles.

One group designed a "dot box" using dual-rail self-timed signalling, in which each bit is carried by two forward signals and one backward acknowledge signal. Each bit contains its own request, and each bit is separately acknowledged. The other group designed the same device using single-rail signalling, in which one request and one acknowledgment is provided for each data word. Since the control of the dual-rail design is more local, and may consequently employ shorter wires, and since there are fewer elements involved in each synchronization event, we expected that it should be able to operate at a higher speed but would require more area than the single-rail design. However, another difference between the designs emerged in the design process. The single-rail design was able to use two-cycle (transition) signalling, which meant that each control signal had to make only one transition for each transaction; whereas an early cycle-time–area comparison of the elementary cells showed that the dual-rail design had to use four-cycle signalling, which requires two transitions for each transaction.

By the end of the third term, very high quality, full-custom designs and layouts for both versions of the "dot box" were completed and simulated. Our expectation was completely wrong. The cycle time of the dual-rail design was, surprisingly, about 3 times larger than that of the single-rail design, whereas the areas were comparable. The single-rail design turned out to have a cycle time of 3.5ns versus 10ns for the dual-rail design. A major part of the reason for this result was the use of two-cycle *versus* four-cycle signalling. The advantage of the dual-rail design over the single-rail design is delay insensitivity, but not speed, and not area. Both designs had substantially lower cycle-time–area products (normalized to technology) than self-timed designs described previously in the literature.

An on-chip pattern generator was added to the dual-rail design, and a test chip was sent to MOSIS for fabrication in 1.2μm SCMOS. This chip, which is now being tested, is expected to have a cycle time of 12ns limited by the pattern generators. Although the single-rail design was superior, we saw no reason to fabricate it because the internal circuits are quite similar to those used in the Caltech routers.

4.3 Timing Model For Performance Analysis and Optimization of Data-Dependent Circuits

Tony Lee, Alain Martin

Designing a highly optimized VLSI chip requires searching through a possibly large state-space of solutions. That is what CAD tools should be about, but are not. In order to perform this search successfully, three major algorithmic components are required. First, a method must exist to *generate* all possible solutions. Secondly, an algorithm must be available to *optimize and compare* the solutions against some figure of merit—speed, power, power–delay product, *etc.* Thirdly, an algorithm must be available to *traverse* the (potentially exponential) tree of solutions. The problem of global optimization of a VLSI circuit is very poorly understood today.

CAST, the suite of CAD tools developed at Caltech for asynchronous circuit design consists of two sets: synthesis tools and tools for global performance analysis and optimization. At different levels of the synthesis, the performance of alternative solutions can be evaluated and the best solution can then be selected for further refinement. We are refining the performance-analysis program to handle circuits with data dependencies. So far, the data dependencies had to be removed prior to the analysis by choosing *a priori* a given scenario for the computation.

Compilation of a data-dependent program into a production rule set yields rules with disjunctive guards. Occasionally, several disjuncts in a disjunctive guard may be true concurrently. Consequently, in the corresponding circuit, the conducting path between the output node and the power rail (VDD or GND) is no longer a simple chain of transistors in series, but, instead, may contain parallel chains. Thus, in order to analyze the performance of data-dependent asynchronous circuits, a timing model that can estimate delays caused by general networks of transistors is needed.

We have chosen the RC model to represent networks of transistors. The delay incurred by a given network of transistors is taken to be the Elmore delay of the corresponding RC circuit. For a chain of transistors in series, the delay from this new model agrees with the one from the simple tau model that we have been using.

A refinement that has been made to the new model is the notion of unsaturated transistors. Two different values are used as the resistance of a transistor depending on whether it operates mostly in the saturated region or not. We have found that this small increase in complexity yields a higher degree of accuracy in the timing model.

We have incorporated this timing model into our performance analysis tools for data-dependent systems. Currently, we are working on ways to find the optimal sizes for the corresponding transistors under this model.

4.4 Gallium Arsenide Asynchronous Microprocessor

Jose Tierno, Alain Martin

The GaAs asynchronous microprocessor designed in the last reporting period was re-fabricated and tested. This version was fabricated using Vitesse's HGAAS III process. The expected performance of this design was about 200 MIPS with a dissipation of 2 Watts. In the past, power and speed predictions using the Hspice models have been very accurate for the HGAAS II process, and we were confident that the expected performance would be achieved. However, in this case the measured performance was only 100 MIPS. The same discrepancy seems to have been observed by other research groups that had projects on the same run. There is some evidence of fabrication problems and underestimation of the parasite capacitances as extracted by the MAGIC layout program.

The good news is, of course, that we now have a fully functional 100MIPS asynchronous microprocessor.

Another factor affecting performance is pad delay. So far we have used ECL levels on the outside, to be able to interface to standard parts and simplify prototyping. Pad delays are in the order of 1ns, mostly spent in level conversion. In addition, the padframe uses a considerable amount of power, close to $1A$ worst-case for the processor. This delay and power can be greatly reduced by designing matched pad drivers and receivers in a system composed exclusively of GaAs parts. This approach would certainly be a requirement in the interface with cache memory.

4.5 Low-power Asynchronous VLSI

Jose Tierno, Alain Martin

Power consumption is becoming the main design issue as highly complex VLSI chips find their way in applications remote from a conventional power source, and as large numbers of chips are packed together in computing systems for high performance. Conventional clocked designs are wasteful in power. First, a large part of the power is consumed driving the clock itself. For instance, in the DEC Alpha, half of the power is consumed by the clock. Secondly, the requirement to meet a certain clock period forces the designer towards a more aggressive design style with respect to the slew rates of signals. Thirdly, parts of the circuits consume power even when they are not doing any relevant work.

Asynchronous designs avoid all three of these problems. Indeed we have observed that all the circuits we have designed operated at low power without any special precautions.

The power advantage of asynchronous circuits, which is suggested by the Caltech Asynchronous Microprocessor experiment, is still a matter of controversy. In the absence of identical designs implemented as both asynchronous and clocked circuits, it is difficult to compare the power performances of both implementation techniques. An argument advanced *against* asynchronous design with respect to power consumption is the use of special data encoding techniques. Such techniques, like dual-rail, require that a larger number of data wires switch for each data transmission than with usual clocked datapaths. In standard four-phase dual-rail encoding, which is the data transmission scheme used in the microprocessor, for each transmission of an n-bit word, n wires change voltages twice. However, measurements performed on the Caltech asynchronous microprocessor show that only 10 percent of the power is consumed in the buses, making the above argument almost irrelevant. In any case, we need to study the current and future designs to understand where the power goes and how to improve the performance/power ratio. This will be one of the main research topics of the next phase of this project.

Power optimization at the gate level is important, but if the circuit is inherently wasteful in power, this optimization will only mitigate a bad problem. The situation is similar to the huge performance improvement of microprocessors over the last 20 years. An Intel 8080 from 1974, operating at 2.5MHz, will execute about 0.5 MIPS, while an Intel 80486 from 1989, operating at 25MHz will execute about 15 MIPS. There is a factor of 3 in circuit-performance improvement that is due only to better system-level design. In the same way, low power has to be "designed-in" at the system level.

It is at this level that asynchronous circuits hold promise of greatly reduced power consumption. In principle, every transition contributes to the computation, and there are no hazards or spurious transitions. Inactive circuits consume only static power, which in CMOS technology is a small fraction of the total; it is not unusual for a clocked circuit to spend about a third of the total power just driving the clock lines, which has to be done irrespective of the level of activity of the circuit. On the down side, asynchronous operation requires extra transitions for synchronization that do not contribute directly to the computation, as well as dual-rail implementation of many logic blocks.

It is difficult at the program level to foresee the effect of a given program transformation in, for example, optimal transistor sizing, or state-variable assignment. It is possible, however, to compare the performance of two programs based on the level of concurrency that those two programs have, and expect the low-level design to preserve the performance gain.

This research tries to characterize the program transformations that we use to generate circuits by the effect they have on power consumption. Our target is low energy consumption; however, there may be restrictions on minimum performance that do not allow us to choose from some of the better energy options. It is then necessary to increase performance with the least penalty in energy.

Finally, an energy model was developed to tie high-level constructs to low-level power dissipation. This model was tested against old designs and found to be accurate. This model is used as a basis of comparison between CSP programs.

4.6 Validation of the VLSI Synthesis Procedure

Marcel van der Goot, Alain Martin

The synthesis method generates circuits that are correct by construction, and, indeed, all of this project's CMOS implementations have been found functional on first silicon. The goal of this project is to give a formal validation of our VLSI synthesis method by proving the correctness of the various transformations involved in a circuit design. As explained in the previous semiannual report, there are important practical reasons to consider formal correctness proofs in the context of VLSI design.

In order to argue about the correctness of the synthesis transformations, a formal definition of the different semantic models used for the program notations is needed. Several program notations are involved: CSP, handshaking expansions, and production rules. The semantics for the different languages must have a certain degree of uniformity, so that the meanings of programs written in different languages can be compared. A common way to describe a computation is by giving a sequence of states (a trace). Since our programs have non-deterministic choices, as well as non-determinism because of concurrency, a program can exhibit several possible computations, which can be conveniently expressed as a set of traces or as a tree of states. For a closed program, *i.e.,* a program with its environment, this is a good method to describe the program's meaning, but, in order for a semantic model to be useful to validate the transformations, we must be able to describe the meaning of program parts without giving an environment or context. Unfortunately, sets of traces do not appear to be useful for the description of such program parts.

Therefore, rather than using states to describe the steps of a program, we use state transformers, *i.e.,* mappings from states to states. The meaning of a program part is now taken to be a tree labeled with state transformers. This description achieves a separation of concerns, since the tree structure describes the control aspects of the program, whereas the state transformers describe the data dependencies. All notations we use in our design method can be described by trees, giving us the desired uniformity of description.

Currently we have the semantics for the standard sequential constructs in our languages (assignments, loops, selection, *etc.*), as well as a method to define parallelism. (Parallelism is modeled as the non-deterministic interleaving of actions, corresponding to an "interleaving" of trees.) We are in the process of choosing among several alternatives for the definition of synchronization actions, which form an essential part of the design method. Once synchronization actions have been defined, we will turn our attention to program transformations, *i.e.,* to transformations of trees.

4.7 Limiting the Lengths of Transistor Chains in the CMOS Implementation of Free-Logic Circuits

Drazen Borkovic, Alain Martin

One of the reasons the circuits generated by the synthesis method are efficient is the use of "free-logic." The designer is not limited to any particular set of operators and gates, but can generate any combination of pull-ups and pull-downs that are necessary to realize a given function. The logical representation of such arbitrary combinations of pull-ups and pull-downs is the so-called "production rule." In practice, it indeed turns out that most circuits generated make an abundant use of free-logic.

The only practical limitation to the CMOS realization of free logic is the size of the pull-up and pull-down chains—the number of transistors in series. (Please observe that, contrary to common belief, it is not the fan-in that is the important factor.)

Generating a set of production rules implementing a given function such that all production rules have a CMOS implementation with transistor chains limited by a given constant is a difficult problem. We have devised such an algorithm. The algorithm is optimal in that it always generates the minimal number of state-variable transitions necessary to limit the length of the chains and the shortest pull-ups and pull-downs that satisfy the constraint. The algorithm also avoids backtracking, and is cubic in the size of the circuit (number of transitions).