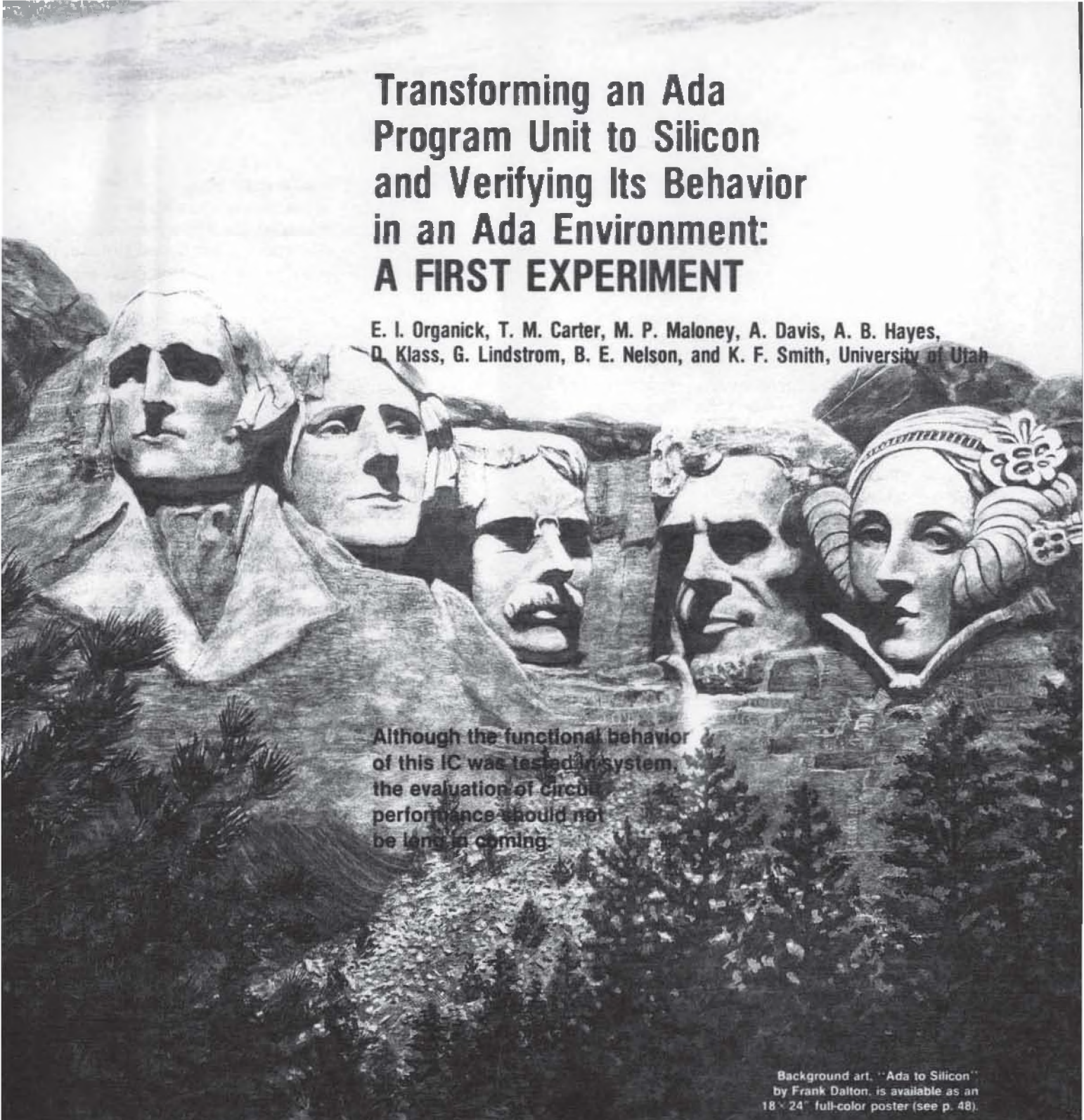
Transforming an Ada Program Unit to Silicon and Verifying Its Behavior in an Ada Environment: A FIRST EXPERIMENT

E. I. Organick, T. M. Carter, M. P. Maloney, A. Davis, A. B. Hayes, D. Klass, G. Lindstrom, B. E. Nelson, and K. F. Smith, University of Utah



Although the functional behavior of this IC was tested in system, the evaluation of circuit performance should not be long in coming.

Background art, "Ada to Silicon" by Frank Dalton, is available as an 18 × 24" full-color poster (see p. 48).

Microelectronics technology has advanced so rapidly and been so successful that we are now having to build large systems with a multitude of diverse, interacting components. Some components of these systems exhibit distinct architectures and may, in fact, be implemented following different choices of data abstraction realized in a variety of logic *and* circuit technologies. When we as designers understand how to build such systems, we are no longer *just* software engineers or *just* hardware engineers—we

0740-7459/84/0100/0031$01.00

become "heterosystems" engineers, a more accomplished breed of engineering professional concerned with building systems that are truly heterogeneous in the fullest sense.

Fitting the diverse components of large systems together has always been a troublesome design and implementation challenge. Although at times doable, the cost for success is usually very high, often prohibitive. We now know that a major reason for this high cost is the lack of a suitable system modeling language—one that is formal enough in its syntax and semantics to be used both for specifying system components at required levels of abstraction *and* for running simulations at chosen abstraction levels. With a satisfactory modeling language, we would be able to build components from their formal descriptions in the modeling language, rather than merely simulate them. Software engineers have been developing such approaches to system building for some time, and hardware engineers are now beginning to recognize their usefulness.[1,2]

It is intriguing to consider what the required relationship between a conventional compiler and a very high-level silicon compiler must be. On the one hand, a conventional compiler transforms a high-order language specification of a system component into a program/data structure for a host machine, but a sufficiently advanced silicon compiler could compile the same source-language specification into a semantically equivalent hardware component. An examination of this relationship has led us to the following observation (and principle): if we are going to build heterogeneous systems with the aid of compilers, then *all buildable parts of the system should be produced by compilers that are driven by the same language syntax and semantics.*

This principle implies another, namely that *conventional and silicon compilers must recognize the same compilation units of the system modeling language and that only these compilation units should be converted to silicon.* By adhering to these two principles, we can be assured that system parts built using different implementation media (and exhibiting different levels of data abstraction) will consistently fit at their interfaces. We also have the assurance that system parts can be replaced with semantically equivalent ones when implemented in different technologies. For example, if standard Ada were used as the modeling language, then naked Ada tasks, which are not legal Ada compilation units, could not be "extracted" from a program, converted into silicon, and then "hooked up" with the rest of the program executing, say, on a general-purpose host.

We were obliged to seek an existing language whose characteristics came close to what we needed and, for this reason, chose Ada.

Having an appropriate system modeling language in hand permits the engineer to view an entire system under design as a single "program" whose static components correspond to compilation units and whose associated interfaces have clearly specified semantics. Thus, when choosing a particular implementation medium for a selected component *C*, a guiding principle to follow is the preservation of *C*'s behavior as specified in the system modeling language. This also implies the preservation of *C*'s interface (semantics) with the other system components, which we'll call "Others"—regardless of what medium of implementation has been chosen for the Others. Thus, we have the derived principle that alternative interfaces between a pair of components can differ physically, but cannot differ semantically. If a modeling language is rich enough and has adequate expressive power, it should be possible to specify a sufficiently wide range of implementations for any selected system component or group of components, thus maintaining the designer's control of the design responsibility.

We are aware that system modeling languages fully meeting the above criteria may not currently exist, let alone be widely available, even for use in designing a relatively limited class of systems. On the other hand, since we wanted to proceed as far as possible with the development of a system-building methodology and style based on the use of a system modeling language, we were obliged to seek an existing language whose characteristics came close to what we needed. For this reason, we chose Ada—both to learn how to use it as a system modeling language for building systems with heterogenous components and, in the course of doing so, to learn what crucial features, if any, Ada now lacks. In fact, we began reporting our initial study of Ada as a system modeling language candidate some time ago.[3]

Mapping specific program units to silicon. Any high-order language program module can, in principle, serve as a specification for an architecture and corresponding circuit realization tailored to the algorithms and data structures of that module. In particular, an Ada package can play a number of useful roles—for example, it can function as an abstract state machine consisting of a set of operations on objects of private data types, or as a server/requester task, which can be either purely functional or also own some private data objects.[4,5] The circuit equivalent of a package can be logically embedded in (and physically appended to) an environment whose components are also specified in Ada.[3]

All program modules in such an environment are executable as a compiled program on a conventional host computer. The resulting system is physically heterogeneous, but semantically homogeneous since the semantics are Ada based. The physical interfaces between disparate media of a system and the specific

logic used to implement these interfaces have to be transparent with respect to the semantics of the high-level specification language. The homogeneity of the semantic medium provides the opportunity to design Ada-level in-system evaluation testbeds for studying the behavior of circuits that are equivalent to software *packages*. Indeed, the importance of developing effective testbeds like these cannot be stressed enough[6]—they can even help realize the full potential of *existing* silicon compilers.

Our approach to assembling such heterogeneous systems presupposes a continued decline in cost and time for reliable VLSI design and fabrication. If this comes to pass, the advantages to be gained for such systems are impressive: the achieved logical and physical tailoring of components; the isolation and replication of function, with the resulting reduction of resource management overhead (time and space) and complexity; and increased speeds (achieved through concurrency).

Experiment summary. Our experiment was intended to further the development of design methodologies and procedures for the system-building approach described above—something that we consider a modest but nontrivial first step. As a demonstration of our ideas, a minor component of the Department of Defense Internet Protocol[7] was specified as a server/requester task embedded in an Ada package. This package was transformed to a speed-independent NMOS circuit composite using various design aids, many of which were developed locally. It was simulated at various levels, fabricated through the MOSIS (MOS implementation system) as a single chip, and then tested at three levels: electrical, logical (gate), and in-system Ada.

This circuit, mapped from approximately 100 lines of Ada code, is the first, large NMOS circuit we have designed with cell-based PPL methodology (see box at right).[8,9] The circuit's active area measures only 3.8 × 3.0 mm, but represents the equivalent of 1928 two-input NAND gates. It is also the first circuit demonstrating the effectiveness of the Assassin (assembly, specification, and analysis system for speed-independent control unit design)[13,14] silicon compiler and was produced by adhering to a completely asynchronous (speed-independent) design discipline.

The package chosen for transformation has three favorable characteristics that significantly increase the chance for a successful first experiment:

(1) Only simple arithmetic, corresponding to nested for-loop control and array addressing, is needed.

(2) A relatively small amount of on-chip RAM-like memory is required. This RAM requirement is so small, in fact, that it is feasible to im-

Our approach to assembling such heterogeneous systems presupposes a continued decline in cost and time for reliable VLSI design and fabrication.

Some useful information about VLSI

The VLSI Group at the University of Utah has developed a methodology known as "path programmable logic" for the design of integrated circuits.[10-12] PPL uses predefined cells for an NMOS silicon gate process and has proven useful as the foundation upon which CAD tools for structured logic can be built.

With PPL methodology, IC design is performed by placing small circuit modules (cells), which can be represented by logic symbols, on a grid representing the IC. When the grid is completely populated, it is both the logical representation *and* the topological layout of the circuit. The circuit modules have predefined schematic and composite representations. They are custom-designed to optimize performance and size for any specific IC process. The conversion from high-level descriptions to the PPL cells is easily accomplished because of the tight coupling between the topology and the logic.

PPL methodology yields circuits similar in structure to that found in a programmed logic array. In PPL, however, the AND and OR planes of the PLA are folded into one plane. The AND conditions of the input signals are formed on the rows of the PPL and the OR conditions are formed on the columns. Complex cells—other than those needed to perform the AND/OR products of a function—are provided and can be inserted into the grid at arbitrary locations. These cells can be flip flops, inverters, loads, row and column connections, pass transistors, etc. AND/OR cells are of unit size—one row high and one column wide—but the complex cells are composed for multiple rows and columns.

The row and column wires that connect the cells can be interrupted at any cell boundary. In contrast to a PLA, the columns and rows in a PPL can be divided into any desired number of segments, allowing greater design flexibility. Discrete modules, which can include memory as well as combinational logic, can be placed anywhere on the grid and segmented from the other modules by row and column breaks. Special ohmic contact cells are used to directly connect rows with columns, allowing the path of a signal to be routed between different modules.

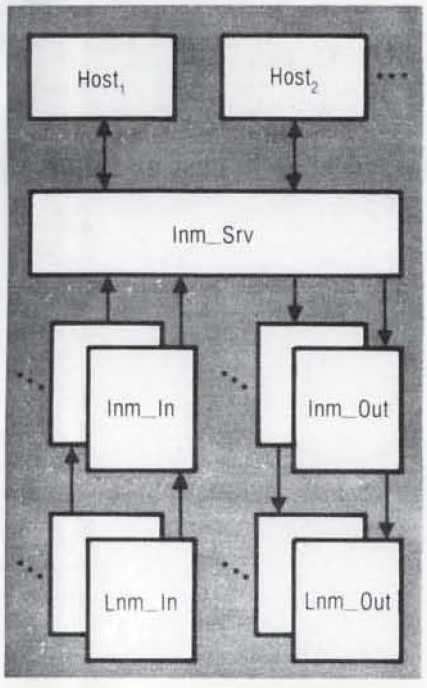


Figure 1. Logical structure of the internet module.

plement it as a linear array of registers; at the Ada level, the array represents a two-dimensional table.

(3) The embedded server/requester task never has more than one external task at a time requesting service of it; consequently, no queueing circuitry is needed either on- or off-chip. This makes implementation of the Ada rendezvous especially simple.

Choosing a package exhibiting these simplifying characteristics allowed us to concentrate on a number of important practical design and system-building-and-testing issues that might otherwise have been overlooked had we been too ambitious in selecting our first package. Some of our follow-up research should give us the opportunity to choose Ada packages whose characteristics are not so favorable. On the other hand, we are also conducting research aimed at applying the discipline developed thus far to building interesting signal processing subsystems. For this class of system-building applications, the kinds of Ada packages needed are (happily) characterized by somewhat simpler communication protocols than those reported in this article.

The context

Here we review the original context from which the candidate software specification was selected for implementation in hardware. Our sample problem involves the specification and implementation of the Department of Defense Internet Protocol.[7] This example, termed the "internet module," performs datagram formation, fragmentation, addressing, and reassembly. Our top-level formulation of the internet module is roughed out in Figure 1. The internet module can be decomposed into three logically distinct modules.[3] Inm_Out, which processes outbound datagrams, Inm_In, for inbound datagrams, and a host gateway module called Inm_Srv, which interfaces the host computer to the Inm_In and Inm_Out modules.

For our experiment, we concentrated on the Inm_Out module and constructed a complete Ada implementation of it.[15] Hardware design time limitations required further decomposition of the Inm_Out module, so the module was separated into two submodules (see Figure 2). The RIP_Manager package with its embedded Read_Init_Parameters (RIP) task was extracted from the original Inm_Out module. Of these two submodules, the reduced Inm_Out module and RIP_Manager, the RIP_Manager was implemented as an NMOS circuit.

The original function of the RIP task was to accept initialization parameters for the Inm_Out module. These parameters were then stored in

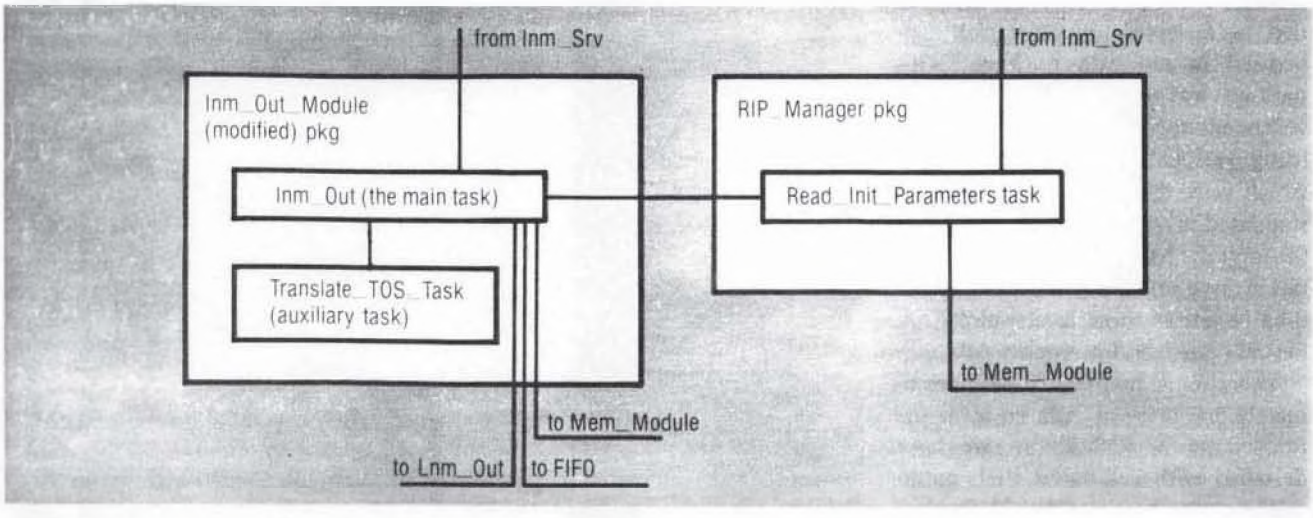


Figure 2. Decompositon of Inm_Out.

the Inm_Out_Module package. The separation of the RIP task from the Inm_Out_Module required storing the initialization parameters in the RIP task, meaning that the RIP task functioned as a small, smart store for the parameters.

The resulting Ada subsystem consists of four packages: the two modules making up the original Inm_Out module and its immediately surrounding modules (Figure 3). Note that each package contains one embedded task[15] and that the RIP task communicates with surrounding tasks by accepting and issuing entry calls.

The transformation of RIP into silicon

The transformation of the Ada code for RIP_Manager and its embedded Read_Init_Parameters task into a silicon implementation required two manual steps—the extraction of the data path and associated control from the Ada program and the rendering of these two elements as a path-programmable logic program unit.

The extraction of the RIP data path was relatively straightforward since it contained only the simplest arithmetic operations. Each declared variable was implemented as a self-timed register and as a register/counter combination. The memory array for the Type_of_Service (TOS) table was also implemented as a bank of eight, eight-bit self-timed registers. The only data operations besides reading to and writing from registers were increment-by-one and an equality comparison. Increment-by-one was implemented as a register and an associated up-counter, and equality comparison as a bank of exclusive-NOR gates, whose output indicated when equality occurred. Figure 4 shows the block-level diagram of the RIP chip, which was created from the Ada code. Note that all registers and counters are self-timed and return a DONE signal upon completion of the requested operation.

Once the data path had been defined, the extraction of control was equally straightforward. In the absence of contention, a task entry call is implemented as a request signal generated by the caller and an acknowledge signal generated by the called task, thereby maintaining the self-timed design discipline.

There are two buses in this system. The first is a four-bit bus connecting RIP, Inm_Out, and the Memory_Module. Since RIP simply stores the first value present on this bus (INITNUM) and forwards the rest to the Memory_Module as actual parameters, we decided not to latch these values in order to forward them. This optimization is acceptable only because of the self-timed design discipline, which requires the requester to maintain the data being sent until the acknowledgment is raised. Thus, RIP simply does not respond to Inm_Out until the INITNUM bus value has been received by the Memory_Module.

The second bus is bidirectional and is used to receive and transmit data to and from the Memory_Module. Since this is not a contention bus (even though it is bidirectional), there is no violation of the self-timed design discipline.

The RIP control-unit must handle interfaces with three other modules and control two banks of eight registers and three counters. As a result, it required 12 states, 17 transitions, 20 inputs, and 21 outputs. In the case of loops, which include entry calls to other tasks, much of the (rendezvous) control was achieved without the addition of extra states. We did this through the judicious use of the conditional output generation feature provided by Assassin. The control-unit was expressed in Asssassin's input language and was functionally simulated in the compiler; thereafter, a PPL module implementing this control logic was automatically generated.[13] The control units produced by Assassin were physically implemented using a one-hot style encoding.[16]

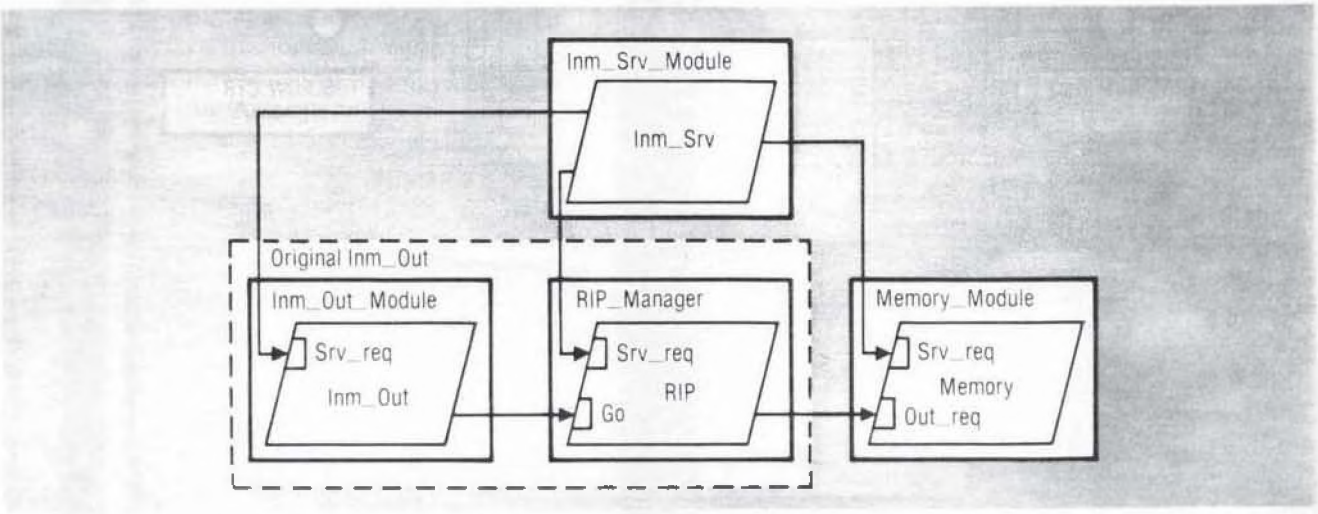


Figure 3. The Read_Init_Parameters software environment.

The PPL RIP program and the RIP chip

The actual implementation of the RIP chip required the addition of one cell (with four different layout configurations) to the PPL library. It also required the implementation of three new pads that could be used for self-timed communication. Because a large portion of the chip was going to be taken up by self-timed memory (registers), a single cell was designed that required only one-third the area of a functionally equivalent implementation in standard PPL. This cell includes the circuitry for a normal PPL latch as well as bus drivers and sensors and the request/acknowledge circuitry required by the self-timed architecture. It covers an area of three PPL columns by seven PPL rows. Four different layouts of this cell (mirror images) were produced so that the

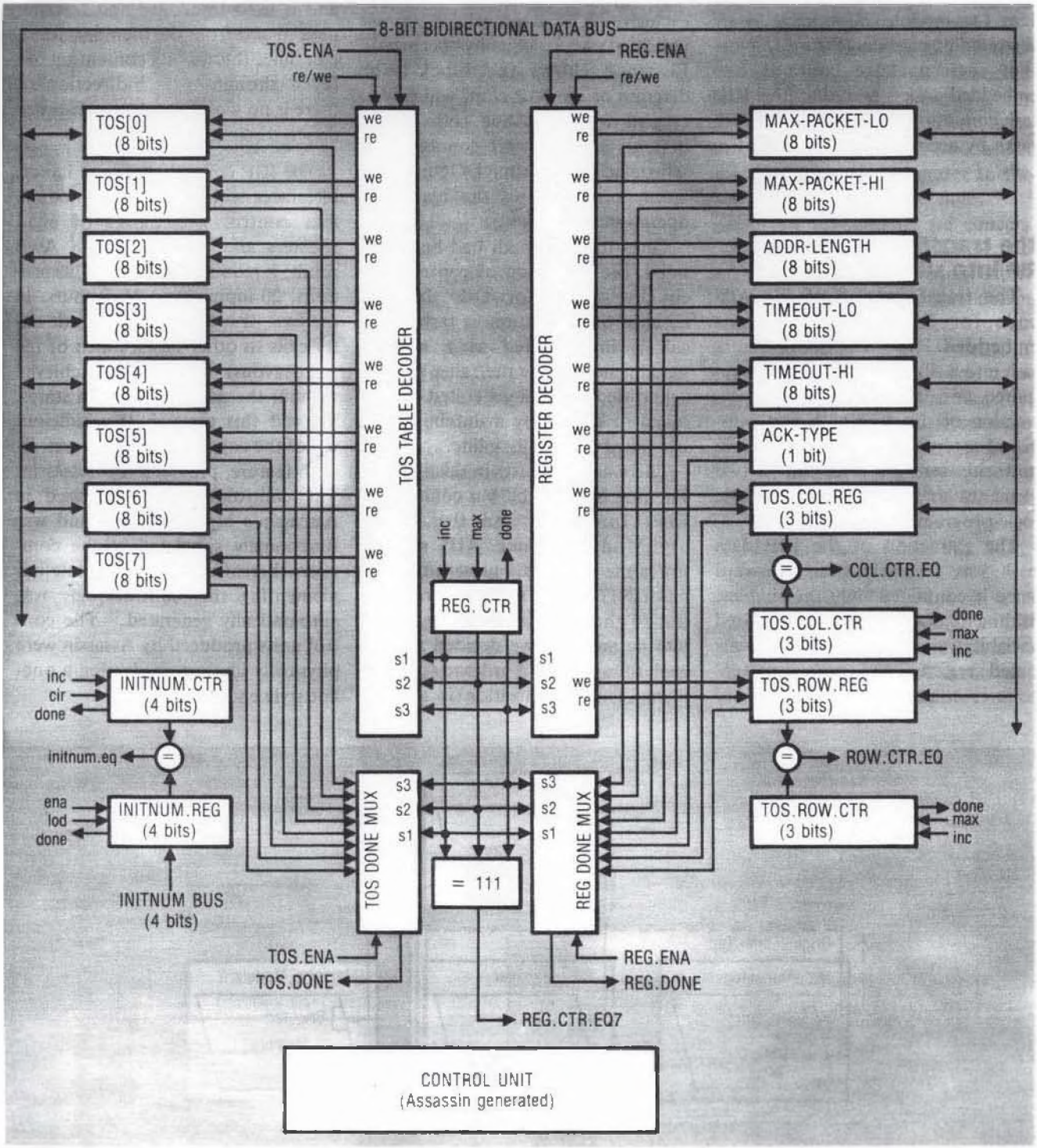


Figure 4. Block diagram of the RIP chip.

registers could be packed as tightly as possible. A composite layout of this cell is shown in Figure 5.

The electrical environment inside a single chip is relatively "friendly" as far as noise and electrical levels are concerned; but, as we know, the outside world is not nearly so well-behaved. Here, noise becomes a problem, and electrical levels cannot be easily assured. Additionally, in order for the RIP chip to send data to another device, it must know when the data is stable in the outside world before it issues a request to the Memory_Module to receive the data. Noise and electrical level problems are solved by using hysteresis-inverting drivers on all signals coming onto the chip. The stability of data being transmitted is estimated by sampling the output pad with a hysteresis inverter and assuming the outside world is stable when the hysteresis sample is stable.

Once these additional cells were available, the layout of the RIP chip advanced rapidly, taking about one week to complete. The only processing elements that needed to be constructed by hand were the counters and the comparators, the PPL programs for which are illustrated in Figure 6. The chip's multiplexers and decoders were easily implemented using PPL's inherent PLA implementation capabilities. The difficult part of this assembly task was the manual PPL interconnection of the modules. In fact, the control module was altered somewhat from the layout produced by Assassin to better conform to the rest of the circuit. The changes, however, were merely cosmetic; the inputs and outputs were placed at ports different from those assigned by Assassin. A floor plan of the entire RIP chip is shown in Figure 7. Once the PPL layout of the chip was completed, some simulation was performed using Asylim[17] (a PPL logic simulator) and Mossim[18] (a switch-level simulator). Exhaustive simulations were not performed because of fabrication deadlines.

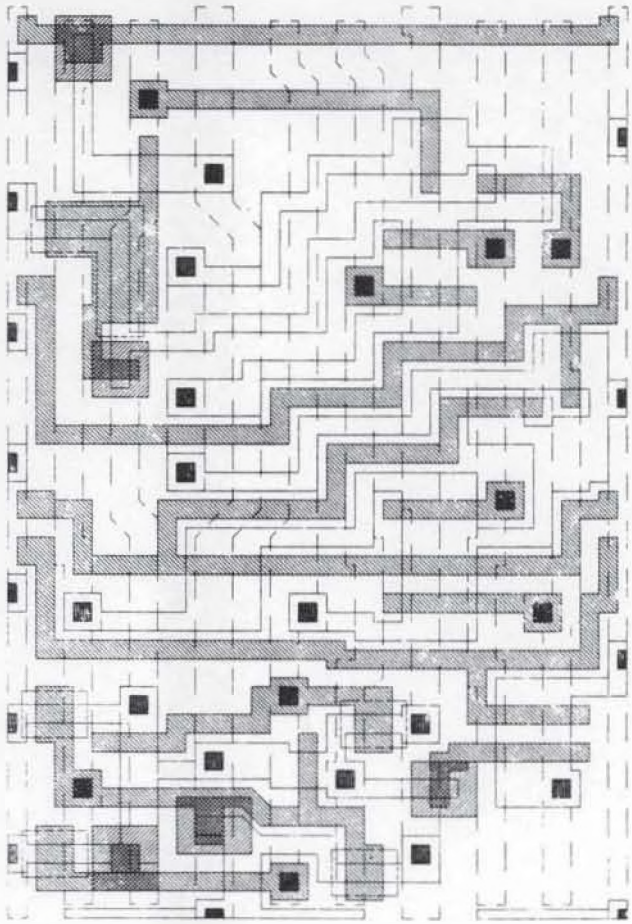


Figure 5. Composite layout of the self-timed register cell.

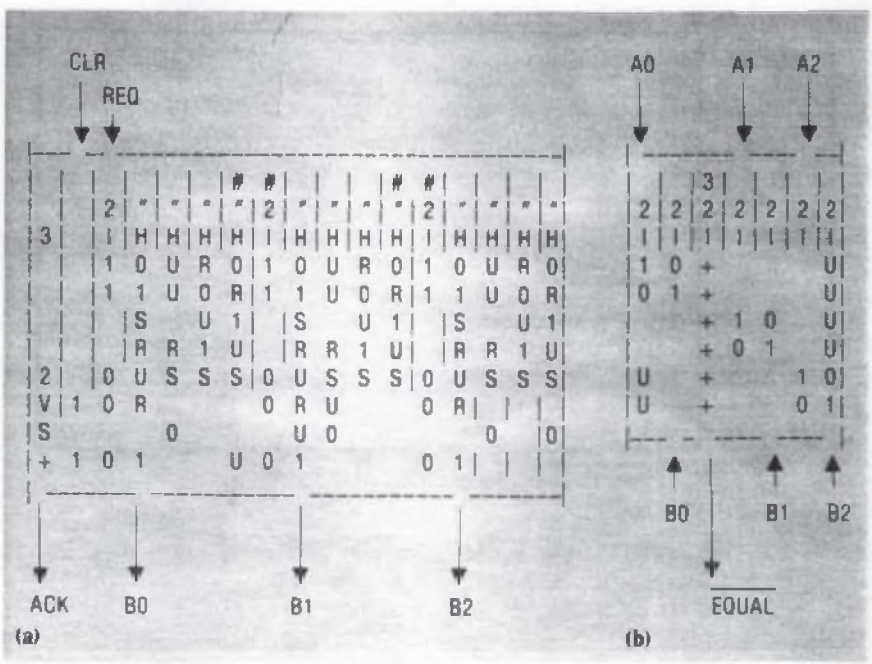


Figure 6. A self-timed PPL counter (a) and a PPL comparator (b).

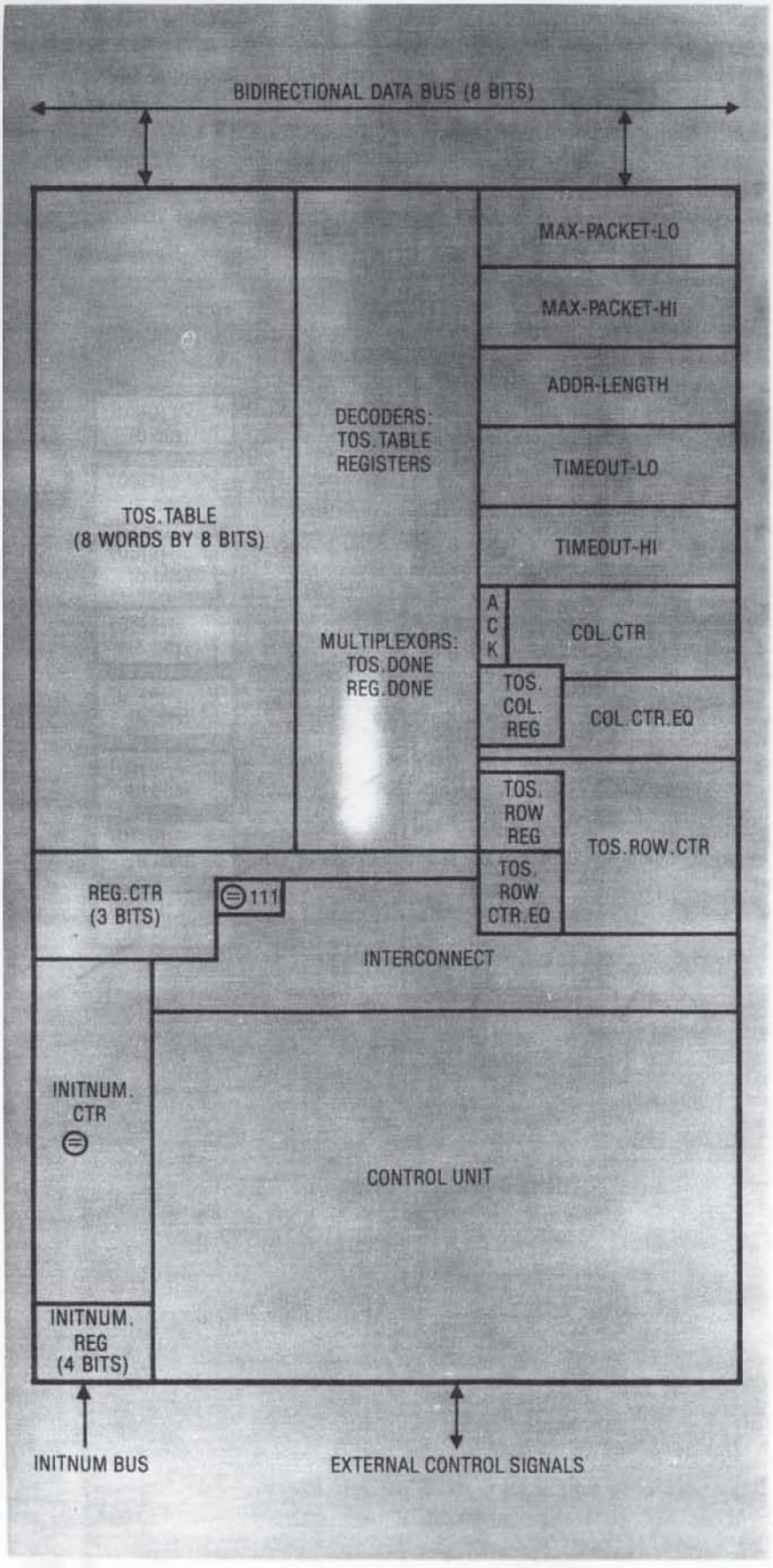


Figure 7. Floor plan of the RIP chip.

In closing this section, please take note of the following statistics concerning the PPL layout of the RIP chip:

(1) The chip took one week to lay out.

(2) Total size of the chip was 119 × 149 mils (no special effort was made to minimize the layout).

(3) Only 18 percent of the total PPL cell placements were completely unused—that is, not used for logic or interconnect.

(4) The chip contains 1928 transistors (an equivalent two-input NAND-gate implementation would require about 2000 gates).

(5) The most random portion of the chip layout (the control unit) was automatically generated in less than five minutes using the Assassin compiler.

Ada-level test strategy applied to the RIP_Manager chip

To transform our package to an equivalent silicon composite, we first had to replace all rendezvous communication with port-based communication to/from the Read_Init_Parameter task. (The semantics of such ports are defined by Cox et al.[19]) We then enclosed RIP_Manager with an interface package, RIP_Interface (Figure 8), which owns the communication ports and provides public procedures in place of entry calls to the RIP task. Request and reply ports were also provided for each RIP_Interface procedure (corresponding to each entry of the Read_Init_Parameters task). To assure that no change took place in the sourrounding Ada packages, we added an Entry_Call_Forwarder task. In short, the RIP_Interface package preserves the integrity of the surrounding Ada software by providing an interface to the hardware version of the RIP_Manager package—one that is transparent to the packages remaining in software.

Next, an entry call to Read_Init_Parameters was represented as a call to the appropriate RIP_Interface

procedure, which sends inbound parameters to a request port and receives outbound parameters from a response port. The calling task is suspended pending service from Read_Init_Parameters because the "entry procedure" does not return until the outbound messages are actually received from the response port.

An entry call from Read_Init_Parameters to the Memory task was then performed by the Entry_Call_Forwarder task; this task waits for a command (containing inbound parameters) at the Mem_Req port to perform a rendezvous with memory. When the rendezvous is terminated, the Entry_Call_Forwarder task will send the outbound parameters to the Mem_Resp port. (The Ada code for this RIP_Interface package is shown in the sidebar on pp. 43-47.)

Replacing the software RIP_Manager package with the RIP chip causes the RIP_Interface package to function as an Ada-level "communication interface" to this chip. The RIP chip interfaces to parallel ports of a peripheral subsystem of an Intel 432. The peripheral subsystem consists of an Intel 432 interface processor, an 8086-based attached processor, and parallel ports, all of which are connected to a Multibus. Software executing in this peripheral subsystem provides port-based communication facilities to the RIP chip.

The testing subsystem for the RIP chip is summarized as follows:

(1) To provide a hardware link between the RIP chip and a peripheral subsystem, we used one hardware parallel interface for each of the communication paths between Read_Init_Parameters and the software tasks. The parallel interfaces consisted of three programmable peripheral interface chips (Intel 8255s).

(2) The peripheral subsystem software supplied by Intel was augmented to (a) utilize the parallel interfaces and (b) provide the RIP chip with message-based communication facilities. This additional software was written in PL/M using Intel's RMX-88 real-time multitasking executive.

(3) As previously mentioned, the software RIP_Manager was removed from the RIP_Interface package. The port-based communication procedures provided by our peripheral subsystem software (and associated hardware) allowed the direct replacement of the software RIP_Manager package by the RIP chip.

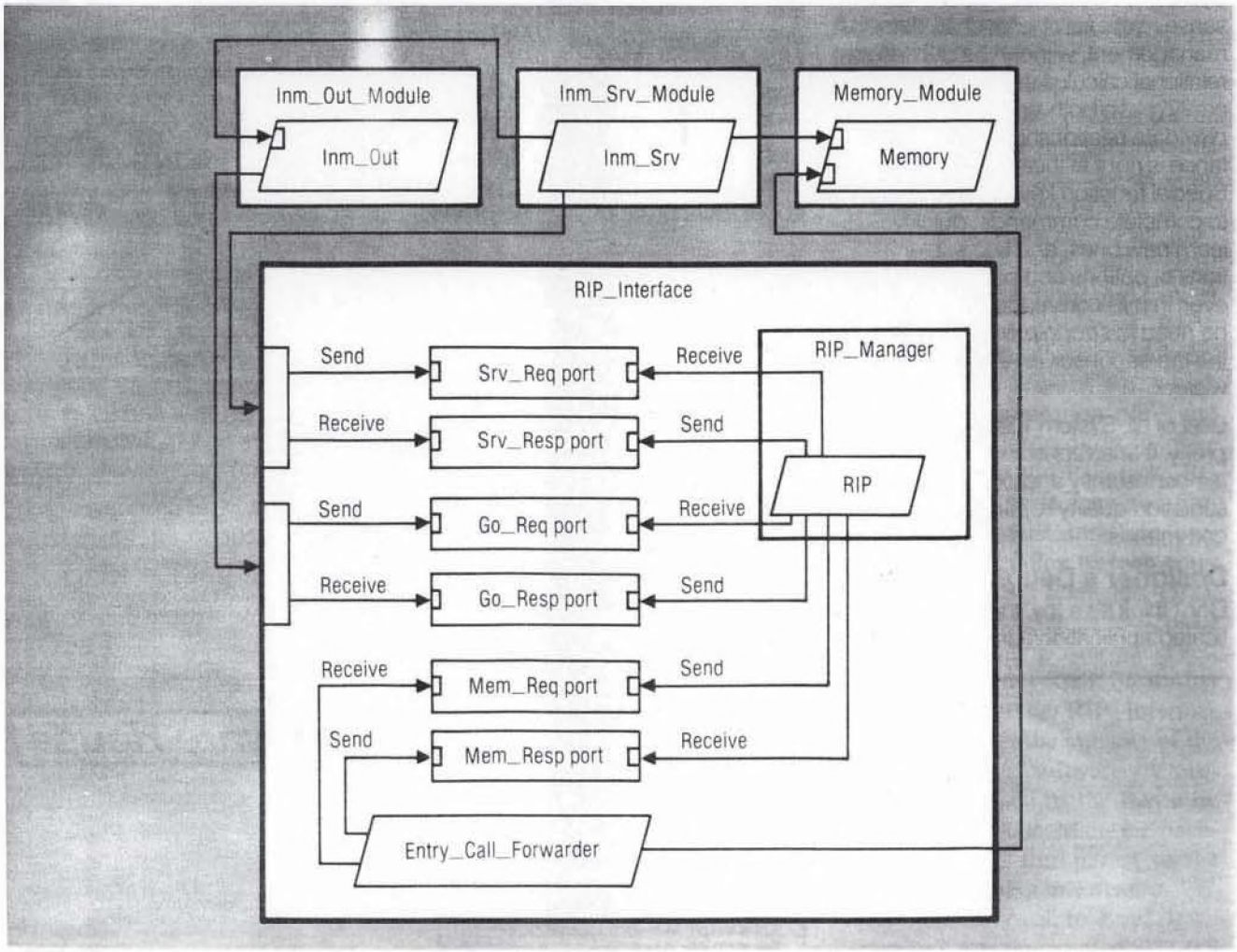


Figure 8. The RIP_Interface package.

This Ada-level test strategy makes it possible to specify in Ada any desired (and possibly redundant) testing procedures whose execution by the chip under test can shed light on the functional correctness of the chip's operations. In our case, we specified certain operations that, when invoked, will indicate that the RIP chip correctly loads the initialization parameters—in other words, that it behaves as a smart store.

If we were to redesign RIP as a stand-alone smart store (something we did not originally anticipate the need for), the Ada specification for the chip and its Ada test environment would have been different. The top-level test strategy, however, would have remained the same. Namely, the internal behavior of the chip would still have been verified by Ada-level probing actions from its testbed environment.

Figure 9 is a schematic of our "high-level testbed." Its principal components are

(1) *The Ada package (chip) under test.* In this instance, it is connected to the interface through three two-way communication channels—one for each of three other Ada tasks that constitute the chip's Ada environment.

(2) *The transparent interface implemented using off-the-shelf components (referred to as the I/O processor).* This interface converts each two-way hardware channel into corresponding software representations for sending and receiving messages. The same interface is also instrumented, using software written in PL/M, to include a pin-level viewing window. Using this window, one can monitor the chip by viewing its behavior at the register transfer level.

(3) *The 432 system host, which executes the packages that constitute the chip's Ada environment.* These program units are modified to include steps to permit the user to interactively control the course of the test via terminal I/O. Responses to input commands generated by the chip under test are viewed on the terminal screen, which therefore serves as a functional-level window.

With this type of testbed, the experimenter is able, during the course of the experiment, to view the chip under test at two levels of abstraction: the Ada level or the register transfer level. Viewing at both levels is also possible.

Testing the RIP chip

Functional and electrical testing. The fabricated RIP chips were easily tested with a switch and light panel. This simple test setup was made possible by the self-timed nature of the part. RIP chip state variables were brought to pads as well as to the off-chip interfaces. In this way, we could observe the behavior of the chip without microprobing. We soon discovered, however, that there was a problem with the circuit when more than one row was required for the TOS table. The chip behaved as expected when the table had a single row, but caused failure when tested for cases involving more than one row.

To find the cause of this problem, we had to use a Tektronix DAS-9000 digital analysis system and microprobing. This digital analysis system was used as a pattern generator, causing the circuit to loop quickly through its complete functional cycle and allowing us to watch internal circuit nodes on an oscilloscope. Microprobing was facilitated because we had included one-mil probe pads on almost every internal node (any node not brought to a pad) in the circuit. These pads were implemented as a PPL OCB (ohmic contact of both column wires to a row wire) cell to which a glass cut was added over a one-mil by one-mil piece of metal.

Microprobing the circuit showed us that the comparator for the TOS row counter was not functioning correctly. The problem, it turned out, was that one set of comparator inputs was incorrectly connected to the TOS row register. The result was that a correct comparison occurred only when the register was equal to zero. Correcting this error (which took several days to locate) involved placing three additional PPL cells on the chip, a redesign step requiring less than two minutes.

Our experience here confirms one important advantage of using PPL to design a circuit, but also shows that the manual layout portion of the work is still a source of potential error. Contrast this to the fact that no errors were found in the automatically generated portions of the circuit. As our CAD tool capability for PPL design increases, we expect that errors, such as the one accidentally introduced into the data path, will be completely eliminated.

In-system testing. For the in-system evaluation of the RIP chip, the Inm_Srv task was coded to

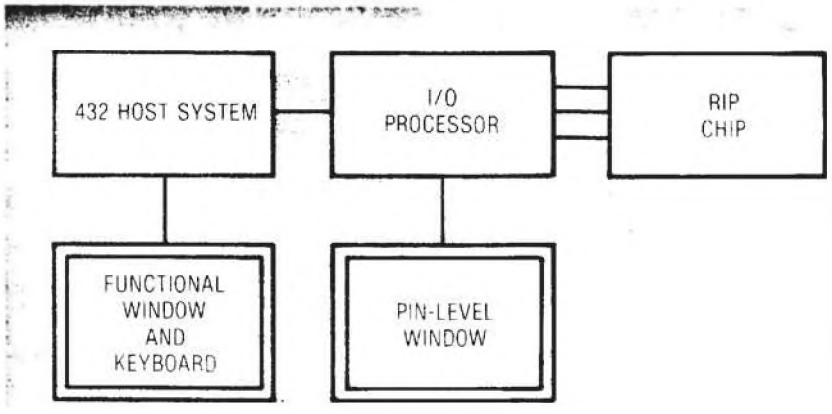


Figure 9. High-level testbed schematic.

behave as the driver for the whole Ada system. This task passes parameters supplied by the user at the terminal to the memory task. This same driver then communicates the proper parameters to the Inm_Out and RIP task. The memory task is also enhanced to print the parameters as it receives them from RIP.

First-level testing of the Ada environment involved the use of the software version of RIP. We were able to test the port-based communication paths between RIP and the rest of the Ada environment, confirming the fact that our message-based design properly implemented the original Ada environment, where rendezvous was used to communicate to and from RIP.

Second-level testing involved verifying the message paths between the Intel 432 and our I/O controller (8086) subsystem. To do this, we used simple Ada tasks on the 432 to send and receive messages to and from the 8086.

At this point, we could have performed one more level of testing by writing a software version of RIP for residence within the peripheral subsystem, but the arrival of the fabricated chip enabled us to skip this step and move on to testing the chip itself in the Ada environment. The chip was connected to the I/O controller through three off-the-shelf programmable peripheral interface chips. The Ada parameters contained in the messages transferred to and from the RIP task were passed through these three chips to and

The RIP code

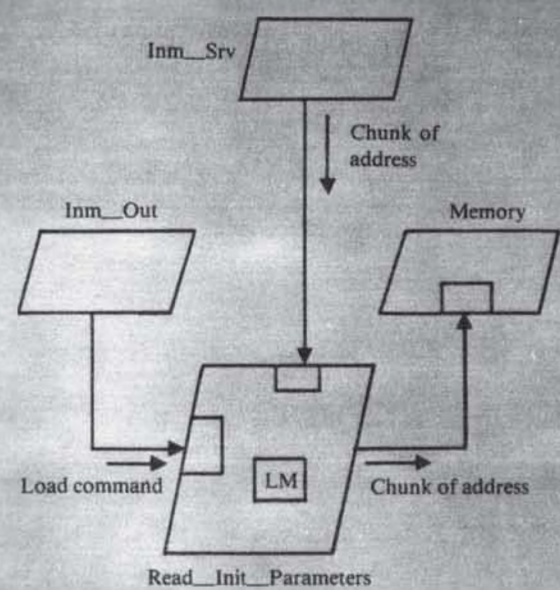
```
with Inm_Out_Defs, In_Out_Srv_Defs;
use  Inm_Out_Defs, In_Out_Srv_Defs;

package RIP_Manger is
 --
 -- Function:
 --    Described in the body of this article.

 task Read_Init_Parameters is

   entry Go(
        init_num_formal:      bit4;
        response            : out out_response);
     --
     -- Function:
     --    This is the principal entry. The task operates in either of
     --    two modes. These modes are referred to below as:
     --    NORMAL or TEST, according to the value of
     --    init_num_formal.
     --
     --    In NORMAL mode, i.e., when init_num_formal > 0, a
     --    call on GO causes the task to
     --    (a) accept init_num address chunks from INM_SRV
     --        and forward them to the associated memory module,
     --        forming the base address of the storage block contain-
     --        ing the initialization parameters;
     --    (b) get values of initialization parameters from the mem-
     --        ory module;
     --    (c) set out_response to either send_ok if successful or to
     --        bad_srv_command if unsucessful. (Can be unsuc-
     --        cessful if required TOS table size exceeds available
     --        local storage space.)
     --
     --    In TEST mode, i.e., when init_num_formal = 0, a call
     --    on GO causes the task to "dump" its local memory to the
     --    memory module.
     --
     --    Control flow diagrams, showing the ensuing entry calls
     --    that are then followed in both the NORMAL and TEST
     --    modes, are given below. Data flow patterns implied by
     --    these entry calls are also shown. The first two diagrams
     --    illustrate the NORMAL mode operation of GO, and the
     --    third diagram depicts behavior for the TEST mode of GO.

 -- NORMAL MODE: Load command (phase I)
```



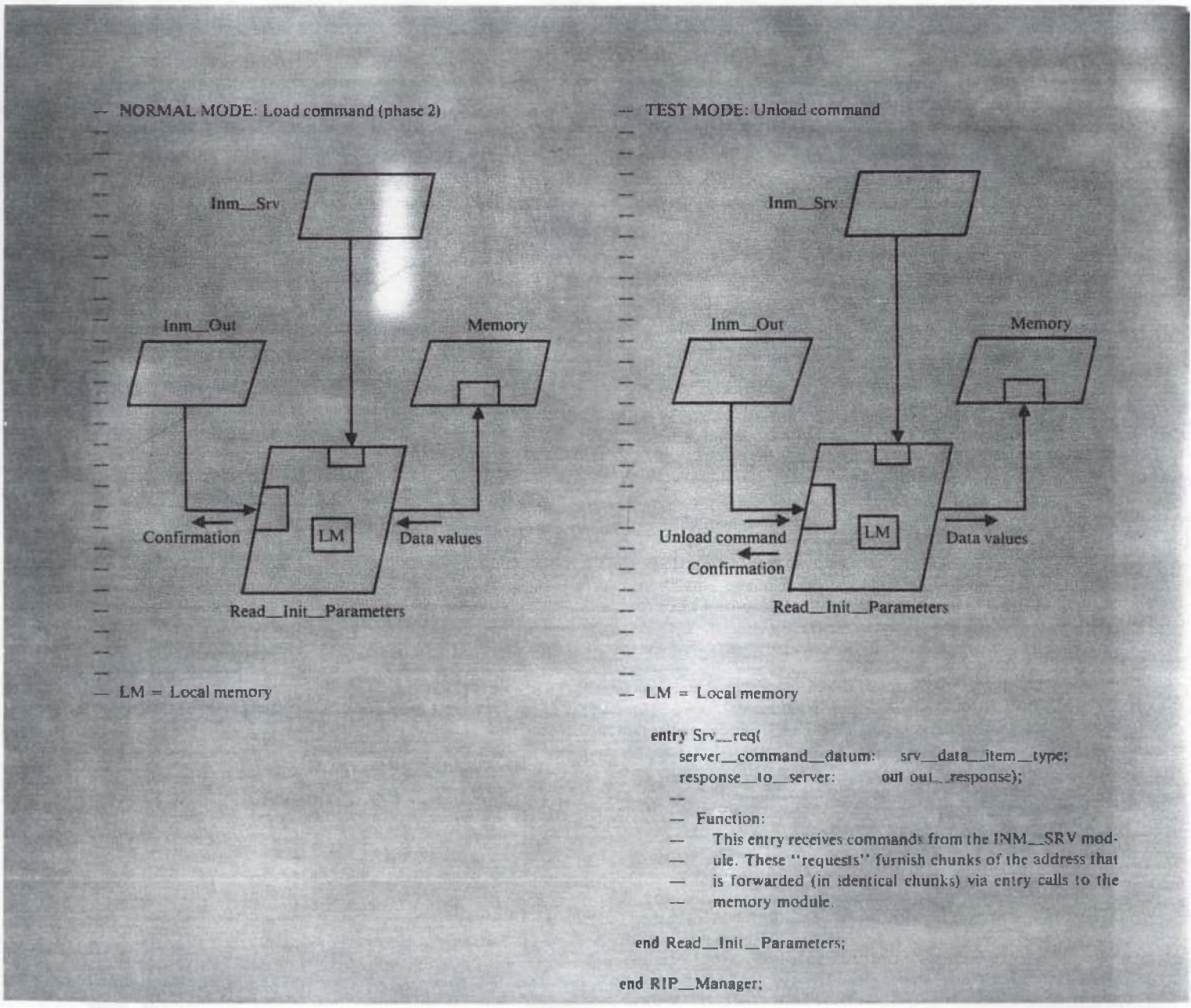

```
 -- LM = Local memory
```

from the RIP chip. This final level of testing confirmed that the behavior of the RIP chip was identical to the software RIP, with a minor exception explained below.

A logic bug in the chip design, discussed earlier, was discovered by the PPL circuit designers during their testing of the first fabricated RIP chip. This bug was then deliberately modeled in the software version of RIP to ensure that both the software and first hardware versions of RIP would exhibit the same functional behavior. The bug will, of course, be eliminated in the next fabrication of the RIP task.

This mode of testing a chip in a high-level environment was a new experience for us. Several problems slowed down work on this end of the project. One was that the Ada compiler we were using did not fully support the Ada language. This limitation forced us to simulate features of the Ada language on our test system. We should not have this handicap in the future. Also, failure to enforce maintenance of one master copy of the code used by both the hardware and software designers caused another source of confusion, which, again, can be eliminated in future work. Additionally, the interaction between the software and hardware designers was limited and did not improve until near the end of the experiment.

Recognition of these problems (and their elimination) will undoubtedly allow for a quicker development of test environments and smoother interactions between the hardware and software designers in later projects.



Schemes for future experiments. In our in-system evaluation of the RIP chip, only the functional behavior of the integrated circuit was tested. A logical next step is to augment the testbed to determine the circuit's performance, and ways to obtain timing information about the circuit in our high-level testing environment are now being considered.

Our current testing procedure is mainly interactive. In the future, we would like to streamline the test operation by executing generated test scenarios. The test data received by exercising the chip will automatically be compared with the test data received by exercising its software equivalent. This technique should be especially easy to implement because all the host resident software is specified at the Ada level.

Our work on system-building methodology is breaking new ground in three areas: First, we have developed a systematic approach to mapping high-order language specifications of system components (Ada packages) to VLSI equivalents. These VLSI equivalents adhere to an asynchronous (speed-independent) discipline, simplifying the job of interfacing these and similar components with a larger subsystem.

Second, this exercise has demonstrated the utility of PPL in the rapid design of integrated circuits. The use of PPL results in very short design times, the elimination of many design and layout errors, and the simplification of design and layout corrections, when required. We have demonstrated the use of the Assassin silicon compiler (for state-machine

```
with Inm_Out_Defs, In_Out_Srv_Defs,
     Inm_In_Out_Defs, Memory_Module;

use  Inm_Out_Defs, In_Out_Srv_Defs,
     Inm_In_Out_Defs, Memory_Module;

package body RIP_Manger is

  task body Read_Init_Parameter is
    --
    -- The following initialization variables were originally located in
    -- the package Inm_Out_Module and are now located in the
    -- task body of Read_Init_Parameters.
    -- Variables to hold initialization parameter values:
    Inm_max_packet:          two_octet_record;
                               -- Largest size packet for the
                               -- local net. Represented as a
                               -- pair of octets and also used
                               -- as a 16-bit integer
                               -- after applying
                               -- unchecked_conversion.

    Inm_address_length;      octet_type;
                               -- Used in Read_in_header.

    Inm_time_out:            two_octet_record;
                               -- Waiting time at LN.
                               -- Represented as a pair of
                               -- octets and also used as a
                               -- 16-bit integer after apply-
                               -- ing unchecked_conversion.

    ack_type:                octet_type;
                               -- Early/late.

    local_net_type_of_service_table_row_size: octet_type;
    number_of_local_net_types_of_service    : octet_type;

    -- TOS array:
    tos_table: octet_buffer_type(0 .. max_tos_table_size - 1);
                                 -- The size of this table
                                 -- depends on the storage
                                 -- space available in the
                                 -- hardware implementation
                                 -- of RIP
    -- init_num value used for echoing back the initialization
    parameters read_init_information: constant integer := 0;

    -- Local variable declaration:
    octet_register:              octet_type;
    dummy_memory_request:        memory_request_type;
    index:                       integer;

  begin
    loop
      accept Cio(
          init_num_formal      : bit4;
          response             : out_response)
      do
        response := sent_ok;          -- Also means init_ok.
        if init_num_formal = read_init_information
          then dummy_memory_request := send_datum_octet;
                                       -- test mode
                                       -- (echo contents of RIP
                                       -- into Memory)
          else dummy_memory_request := receive_datum_octet;
                                       -- normal mode
                                       -- (load contents of
                                       -- Memory into RIP)
        end if;

        -- accept from the server all of the addr_chunks needed to
        -- form the base address in memory that holds the
        -- initialization parameters and send these chunks to the
        -- memory module.

        index := 0;
        if not (index = init_num_formal) then -- normal mode
        loop
          index := index + 1;
```

design), which also provides a good argument for the use of the one-hot state-machine implementation techniques advocated by Hollaar.[16]

Third, we have developed and demonstrated a message-based interprocess communication strategy that operates at the level of the high-order specificaton language. This can be used for testing produced circuits and, hence, for testing subsystems to which such circuits are appended.

We have yet to fully automate our transformation methodology. Ultimately, we would like to input an Ada package to a transformation system as a means of producing a circuit composite or, at the least, producing input to the transformation system discussed in this article. (A high-level transformation system[20] which maps a subset of Ada to a form that can be input to the Assassin system is, in fact, under development; however, as yet it is not usable.) We also believe that extensive research and development must be done on the lowest level circuit layout and design methodologies in order to develop an efficient automated transformation system.

The semantically transparent interface that we have described for use wth our Intel 432 object-based testbed must be streamlined. It is currently, and admittedly, slow and baroque. One approach worthy of future research is based on the "frame" idea, recently advanced by Lynn Conway. Although Conway's approach may be considered ambitious in this context, the development of on-chip interface circuits that are easily tailored for particular Ada packages converted to silicon may be possible. Augmented by this interface circuit, the chip could then

```
    accept Srv_req(
      server_command_datum      : srv_data_item_type;
      response_to_server        : out_response)
    do
      Memory.Out_request( -- Put chunk out to the Memory
                          -- module.
            request_type_formal          => load_address,
            chunk_of_address_formal      => server_
                                            command_
                                            datum,
            octet_formal                 => dont_care_
                                            octet);
    end Srv_req;
    exit when index = init_num_formal;
end loop;
  end if;

-- Get the six individual initialization parameters (contained in
-- the next eight octets received) from the memory module,
-- or, if init_num_formal is read_init_information, send
-- them back to the memory.

index := -1;
loop
  index := index + 1;
  if init_num_formal = read_init_information then
    -- test mode case index is
      when 0 => octet_register := Inm_max_packet.lo;
      when 1 => octet_register := Inm_max_packet.hi;
      when 2 => octet_register := Inm_address_length;
      when 3 => octet_register := Inm_time_out.lo;
      when 4 => octet_register := Inm_time_out.hi;
      when 5 => octet_register := ack_type;
      when 6 => octet_register :=
            local_net_type_of_service_table_row_size;
      when 7 => octet_register :=
            number_of_local_net_types_of_service;
      when others =>
        null;
    end case;
  end if;
```

```
Memory.Out_request(
      request_type_formal          => dummy_memory_
                                      request, -- set to test
                    -- (echo) mode or normal (load) mode
      chunk_of_address_formal => dont_care_X_datum,
      octet_formal                 => octet_register);

if not (init_num_formal = read_init_information)
      then -- normal mode
  case index is
    when 0 => Inm_max_packet.lo      := octet_rgister;
    when 1 => Inm_max_packet.hi      := octet_register;
    when 2 => Inm_address_length     := octet_register;
    when 3 => Inm_time_out.lo        := octet_register;
    when 4 => Inm_time_out.hi        := octet_register;
    when 5 => ack_type               := octet_register;
    when 6 => local_net_type_of_service_table_
              row_size
                                     := octet_register;
    when 7 => number_of_local_net_types_of_service
                                     := octet_register;
    when others =>
      null;
  end case;
  end if;
  exit when index = 7;
end loop;

-- Read in type of service translation table (or write it out).

declare
  row_number: integer range -1 ..
            number_of_local_net_types_of_service
            := -1;
  col_number: integer range -1 ..
            local_net_type_of_service_table_row_size;
  index:      integer range -1 ..
            number_of_local_net_types_of_service *
            local_net_type_of_service_table_row_
            size
            := -1;
```

be connected directly to the host's system bus.

We have also yet to demonstrate that our PPL methodology can be mixed on the same substrate with other, more space-efficient tiling components for RAMs, ROMs, and other large, but important, building block elements. Such a "mixture" would permit us to extend the application areas for our silicon compiler to functionally more elaborate program units.

Finally, consider the following as our vision of the essence of the system modeling "art" as we currently understand it; in short, think of it as a tentative set of modeling principles to be adhered to when using Ada to model systems for later translation into silicon:

(1) We must somehow choose a one-to-one correspondence between the Ada compilation units that we supply to an Ada compiler and those that we want to represent as distinct physical (silicon) units.

(2) The particular computation distribution among the physical units (and also between the components to be reduced to silicon and the remaining compilation units in the specifying program) must be deducible by an *Ada silicon compiler.*

(3) When we wish to prohibit the sharing of communication channels among physical units to achieve maximum concurrency and minimize arbitration circuitry, we must avoid writing Ada specifications from which such sharing can be inferred.

(4) Memory should be distributed among the Ada units in the same way that it will be divided among their physical counterparts. This means, for example, that pointers into a central memory must not be passed within communication packets. Thus, the actual values would be passed, not the pointers to them.

(5) Lastly, we must confine recursion, if it is to occur at all, to within (and only within) single Ada compilation units. In this way, recursion within a constructed system component will imply dynamic storage management localized to the storage within that constructed (silicon circuit) component.

The list of principles above is not likely to be complete, and some of its elements may not even be necessary. Further study along these lines is currently underway. ■

```
begin
  loop                    -- Outer loop reads all rows of TOS table.
    col_number := -1;
    row_number := row_number + 1;
    loop                  -- Inner loop reads in one row of TOS table.
      col_number := col_number + 1;
      index := index + 1;
      Memory.Out_request(
          request_type_formal          => dummy_mem-
                ory_request, -- set to test (echo) mode
                                       -- or normal (load) mode
          chunk_of_address_formal   => don't_care_
                                                X_datum,
          octet_formal                 => tos_table(index));
      if (index = max_tos_table_size) and
        col_number /= local_net_type_of_service_
        table_row_size or
        row_number /= number_of_local_net_types_
        of_service) then
        response := bad_srv_command;
        return;     -- Exit the current accept statement.
      end if;
      exit when col_number =
            local_net_type_of_service_table_row_size;
    end loop;       -- End inner loop
    -- the test of row_number = 0 has been added to simu-
    -- late an error found in the first fabricated RIP chip
    exit when (row_number = 0) and
        ( row_number = number_of_local_net_types_
        of_service);
  end loop;                -- End outer loop.
 end;                      -- End declare block.
 end Go;
 end loop;                 -- End of outer-most (infinite) loop
end Read_Init_Parameters;
end RIP_Manager;
```

Acknowledgment

We wish to acknowledge the special help and advice given us by several others: in particular Lee Hollaar, Doug Fisher, P.A. Subrahmanyam, and Sanjay Rajopbadye.

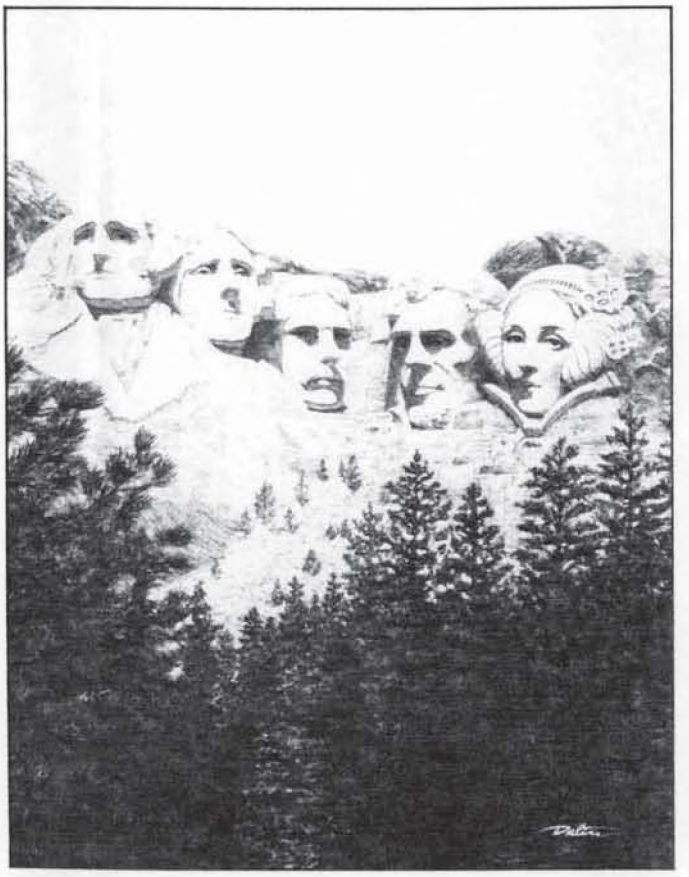
This research was supported in part by the Defense Advanced Research Projects Agency (DoD), ARPA order no. 4305, under contract no. MDA 903-81-C-0411, issued by Defense Supply Service, Washington, DC 20310.

References

1. R. Piloty et al., "Conlan Report," *Lecture Notes in Computer Science*, Goos and Hartmannis, eds., Vol. 151, Springer-Verlag, Berlin, 1983.
2. G. W. Preston, "Report of IDA Summer Study on Hardware Description Language," tech. report IDA Paper P-1595, Institute for Defense Analysis, Science and Technology Division, Oct. 1981.
3. E. I. Organick, and G. Lindstrom, "Mapping High-Order Language Units Into VLSI Structures," *Proc.*

Compcon Spring 82, Feb. 1982, pp. 15-18.

4. G. Booch, *Software Engineering with Ada*, Benjamin/Cummings, Meno Park, Calif., 1983.
5. E. I. Organick, *Programmer's View of the Intel 432 System*, McGraw-Hill, New York, N.Y., 1983.
6. R. Bisiani et al., "MISE: Machine for In-System Evaluation of Custom VLSI Chips," tech. report CMU-CS-82-132, Dept. of Computer Science, Carnegie-Mellon University, Aug. 1982.
7. "Internet Protocol: DARPA Internet Program, Protocol Specification," tech. report RFC 791, Information Sciences Institute, University of Southern California, Sept. 1981.
8. T. M. Carter et al., "Path-Programmable Logic and the Use of CADDS2/VLSI," *Proc. Fourth Ann. Computervision User Conf.*, Computervision Corp., Bedford, Mass., Sept. 1982, pp. 523-528.
9. K. F. Smith, T. M. Carter, and C. E. Hunt, "Structured Logic Design of Integrated Circuits Using the Stored Logic Array," *IEEE Trans. Electron Devices*, Vol. ED-29, No.4, Apr. 1982, pp. 765-776.
10. B. E. Nelson, K. F. Smith, and T. M. Carter, "Cost Effective VLSI Design System," *IEEE Int'l Symp. Circuits and Systems*, May, 1983, pp. 505-508.
11. K. F. Smith, T. M. Carter, and C. E. Hunt, "Structured Logic Design of Integrated Circuits Using the Stored Logic Array," *IEEE Trans. Electron Devices*, Vol. ED-29, No. 4, Apr. 1982, pp. 765-776.
12. K. F. Smith et al., "Computer-Aided Design of Integrated Circuits Using Path-Programmable Logic," *IEEE Electro 83 Professional Program Session Record*, New York, N.Y., Apr. 1983, paper no. 22/2.
13. T. M. Carter, "ASSASSIN: A CAD System for Self-Timed Control-Unit Design," tech. report UTECH-82-020, Dept. of Computer Science, University of Utah, Salt Lake City, Ut., Oct. 1982.
14. T. M. Carter, "ASSASSIN: A CAD System for Self-Timed Control-Unit Design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, to appear.
15. G. Lindstrom et al., "Ada Specifications for the DoD Internet Protocol: The INM_OUT Submodule, Report No. 1," tech. report, Dept. of Computer Science, University of Utah, Nov. 1982.
16. L. A. Hollaar, "Direct Implementation of Asynchronous Control Units," *IEEE Trans. Computers*, Vol. C-31, No. 12, Dec. 1982, pp. 183-1141.
17. Brent E. Nelson, "ASYLIM: A Simulation and Placement Checking System for Path-Programmable Logic Integrated Circuits," master's thesis, University of Utah, Oct. 1982.
18. R. E. Bryant, "MOSSIM: A Logic-Level Simulator for MOS LSI," *Users Manual Version 1*, MIT Laboratory for Computer Science, MIT, Cambridge, Mass., Sept. 17, 1979.
19. G. W. Cox et al., "Interprocess Communication and Processor Dispatching on the Intel 432," *ACM Trans. Computer Systems*, Vol. 1, No. 1, Feb. 1983, pp. 45-66.
20. S. V. Rajopadhye and P. A. Subrahmanyam, "TRACIS: Transformations on Ada for Circuit Synthesis," tech. report UTEC-83-003, University of Utah, Dept. of Computer Science, Aug. 1983.



Elliott I. Organick, a professor of computer science at the University of Utah since 1971, has worked in programming languages, operating systems, and computer architecture. His current work centers on the use of object-based programming languages for specifying subsystem components. He received BS, MS and PhD degrees in chemical engineering from the University of Michigan in 1944, 1947, and 1950, respectively, and is a member of the ACM, the IEEE Computer Society, SIAM, MAA, AIChE, and the American Chemical Society.

Tony M. Carter is currently employed as a member of the technical research staff of the VLSI Research Group at the University of Utah. His current interests include CAD for VLSI, self-timed systems, the application of database systems to engineering and computer arithmetic. He received BS, MS, and PhD degrees in computer science from the University of Utah in 1980, 1982, and 1983, respectively. Carter is a member of the IEEE and the ACM.

Mike Maloney is a member of the Ada-to-Silicon Research Group at the University of Utah. His current interests include object-based architectures, programming languages, and hardware-software system design. He completed a BS degree in physics at Washington State University in 1980 and an MS degree in computer science at the University of Utah in 1983.

Alan L. Davis is currently the manager of the AI Architecture Group at the Fairchild Research Center in Palo Alto, Calif. He received his first degree from MIT in electrical engineering in 1969 and his third degree from the University of Utah in 1972. In addition, Davis holds an instrument airplane rating from the FAA, and FCN and FCA certificates from PSIA.

Alan Hayes is a research scientist with Evans & Sutherland Computer Corporation. His areas of interest include self-timed circuits, distributed systems, and data-driven computation. Hayes received BS and MS degrees in electrical engineering from MIT and a PhD in computer science from the University of Utah. He is a member of the IEEE and an FIS technical delegate.

Dan Klass is currently working as an Ada system programmer at the Ada-to-Silicon Project at the University of Utah. He received a BS in 1978 and an ME in 1980 from the Computer Science Department at the University of Utah.

Gary Lindstrom: Current position—associate professor of computer science, University of Utah; technical interests—programming languages and systems; prior professional experience—none; awards—none; education—BS (math), MS (math), PhD (computer science), all from Carnegie Mellon University.

Brent Nelson is currently a member of the VLSI Group at the University of Utah. His current research interests include CAD tools for integrated circuit layout and verification and the development of structured IC design methodologies. Nelson received BS and MS degrees in computer science from the University of Utah in 1981 and 1983, respectively, and is currently working toward a PhD.

Kent F. Smith is currently an associate professor of computer science and electrical engineering at the University of Utah and a consultant to General Instrument Corporation. He has been active in integrated circuit design and testing for the past 17 years, holds 11 patents in these areas, and has published numerous technical papers. Smith received his BS and MS degrees in electrical engineering from Utah State University in 1957 and 1958, respectively, and his PhD degree in electrical engineering from the University of Utah in 1982.

Questions concerning this article can be addressed to Elliot I. Organick, University of Utah, Computer Science Dept., Salt Lake City, UT 84112.