The Symbolics Ivory
Design and Verification Strategy

Neil Weste, Chris Terman, Howard Shrobe,
David Sarrazin, David Tan, Kalman Reti, Eric Nestler,
Henry Minsky, Alan Corry, Jim Cherry, Clark Baker

VLSI Systems Group
Symbolics Cambridge Research Center
Cambridge, MA 02142

Abstract

This paper describes the design and verification strategies used to create a third generation symbolic processor optimized for the Lisp language. The approach is unique in that it is probably the first large scale application of object oriented programming techniques to a production IC design problem. These techniques have yielded not only high productivity in the construction of the design tools but also clearly demonstrate the designer productivity available through the application of a well integrated set of VLSI design tools.

1. Introduction

Design and verification of complex architectures must occur at several widely separated levels of abstraction including the virtual machine, instruction set, architectural, gate and circuit levels. During the design, congruent descriptions must exist in the behavioral, schematic and layout domains. Many proposals have been made for dealing with the diverse levels of detail evident in these various design domains. Our common denominator is the Lisp language and the uniform virtual address space provided by the Lisp machine operating system combined with an object-oriented electronic design system called NS. The result is no HDL but Lisp, no data files but Lisp data objects and no raw test vectors but Lisp code. We believe this approach is clean, easy to customize and extend, and requires little maintenance as Lisp is a commercially supported language with excellent compilers, editors, debuggers and advanced run-time environments.

NS [1] is a design system written in New Flavors, an object-oriented extension of Common-Lisp. Diagram objects and electrical network objects are the two basic object types upon which a wide variety of tools are based. A diagram object, for instance, might be a schematic which is comprised of lines, named terminals and hierarchical instances. Extraction is performed by traversing the diagram object. The result of extraction is an electrical network which consists of nodes, transistors and other primitive circuit elements. A switch-level simulator works by traversing the network object. Message passing between diagrams and networks yields an extensible design system with an effective user interface.

The implementation of the Ivory single-chip Lisp microprocessor provided some interesting VLSI design tradeoffs. It was undertaken by a small team of people, had to be completed in a short time and had to proceed in parallel with the refinement of the architecture. This paper summarizes the set of tools and design approaches used in the development of the chip. Where possible we will show how Lisp was used as a specification or verification language.

2. Architectural Design

2.1 Architectural Simulation

The most abstract design level models the I-machine (the virtual machine formed by the Ivory chip) by reflecting the state of the architecturally-defined stacks and registers. This level of design is used to verify the virtual machine architecture and gather architectural statistics.

Two virtual machine simulators were written to model this level of the machine. The first was written by the VLSI group, while the second was written later in the design cycle by the software development group. Both use a fairly straightforward implementation of an instruction set emulator. This consists of a set of data structures modeling the relevant machine state (in this case, the system's stacks and stack pointers) and a Lisp routine corresponding to each instruction which modifies the core data structures to reflect the effect of the instruction. The first simulator is 2500 lines of Lisp and runs at 2000 instructions per second.

The following represents the specification of the ADD instruction at this level of design.

```
(defemulator add operand
  (multiple-value-bind
        (first-operand second-operand)
        (fetch-two-operands operand)
    (stack-push (+ first-operand second-operand))))
```

2.2 Behavioral Simulation

The behavioral simulator was built using the object-oriented programming facilities provided by the Lisp Machine environment. The system is decomposed into approximately 20 modules, each of which is modeled as an instance of some class of objects. Each instance maintains private state. Attached to each class was a set of methods for implementing generic behavior in a manner suitable for objects of that class. Each module defines a set of I/O signals that constitute a module's communication ports. These specify the ports to be used in the circuit design. The following specifies an adder with inputs OP1 and OP2 and outputs EXTERNAL-BUS.

```
(defmodule (adder ivory)
  :local-state ()
  :local-phase-1-registers ()
  :local-phase-2-registers ()
  :phase-1-registers ()
  :phase-2-registers ()
  :phase-1-inputs ()
  :phase-1-outputs ()
  :phase-2-inputs (op1 op2)
  :phase-2-outputs (external-bus)
  )
```

Each class has a method for simulating the behavior of a module during the first phase of the clock, another for simulating the second phase, and others for updating the window-oriented display after each phase. The following is the specification of the adder behavior during the phase-2 clock with the method that executes to display the adder state in the simulator window.

```
(defaction (adder :execute-phase-2)
   (let* ((unsigned-op1 (32-bit op1))
          (unsigned-op2 (32-bit op2))
          (unsigned-result (+ unsigned-op1 unsigned-op2)))
      (setq external-bus unsigned-result)))

(defaction (adder :display-phase-2)
  (format window "~O ~& op1 ~O ~& op2 ~O" op1 op2)
  (format window "~& external-bus ~O" external-bus))
```

Figure 1 shows the window interface used to interact with the behavioral simulator.

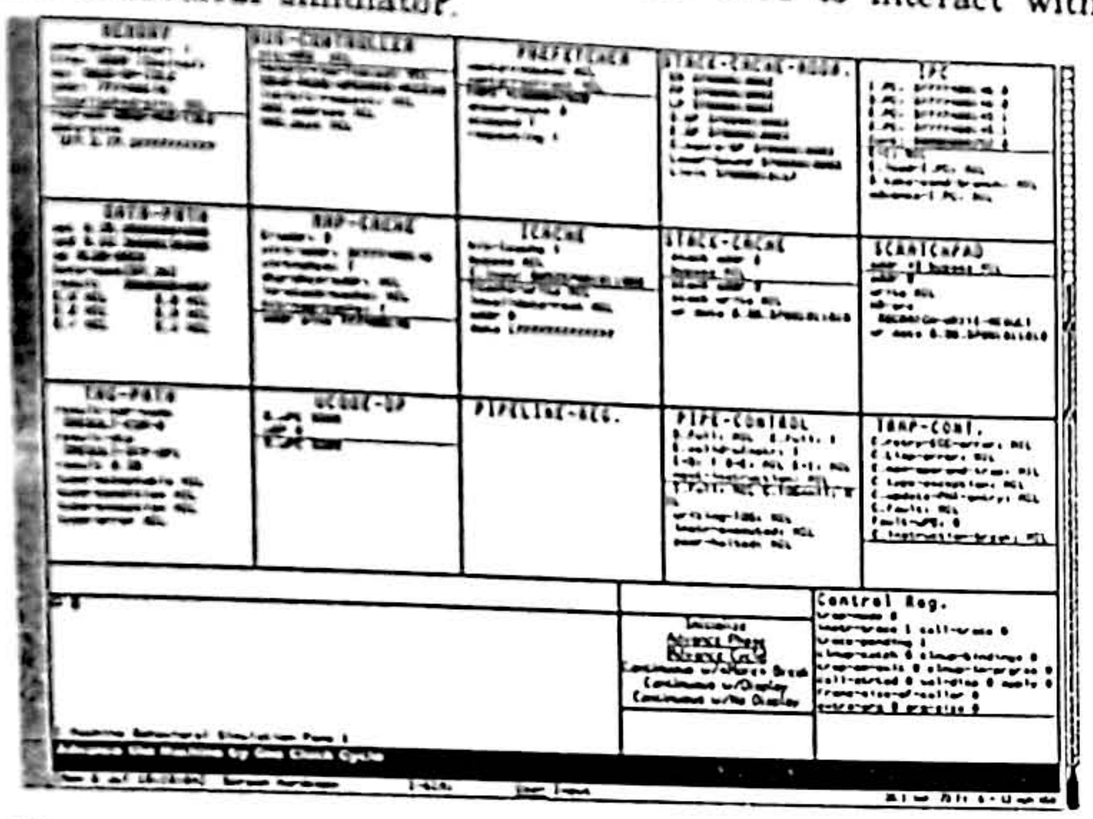

Figure 1. Functional Simulator Window Interface

An assembler supports the development of the microcode module. The following represents the microcode specification for the ADD instruction:

```
(definstruction add
  (parallel
     (check-arithmetic-operands operand-1 operand-2)
     (pop2push (+ operand-1 operand-2))
     (enable-overflow-exception)
     (next-instruction)))
```

The simulator and Lisp description of Ivory total 9000 lines of code including comments. The simulator runs at about 30 instructions per second.

What is novel about this simulator is its implementation technology and its relationship to its surrounding environment. Since the simulator is part of a Lisp environment, it can write and execute other Lisp programs which execute within the same environment. Furthermore, these procedures can be individually modified and dynamically linked into the environment without interrupting normal execution. Therefore, there is no need to create a new HDL and implement the standard variety of control structures required within it. Instead, the simulation routines are normal Lisp procedures with access to the full richness of the Lisp programming environment.

3. Structural Design

The NS interactive editor provides for the capture of schematics and schematic-icons. A menu-driven generator builds most common icons automatically. Figure 2 shows a view of the schematic editor with two views of a schematic.

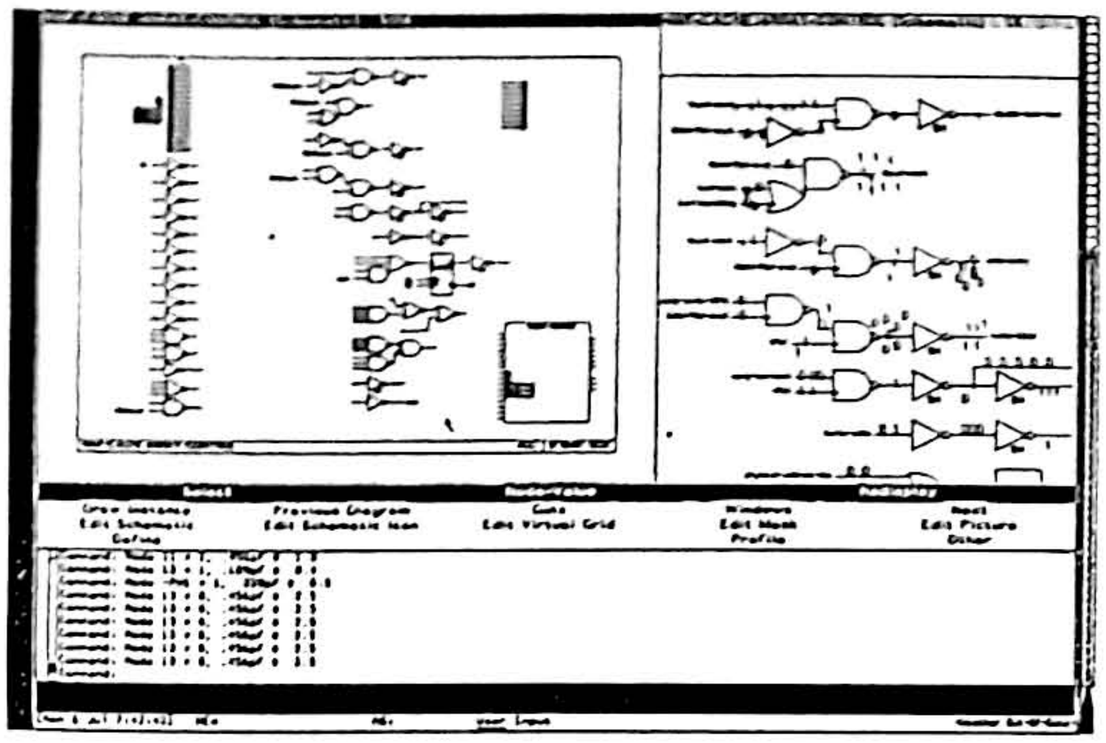

Figure 2. Schematic Editor Interface

A designer may enter schematics by the normal approach of graphical editing or may use a Lisp HDL to allow rapid and accurate entry of control logic. In the latter case, code can be for the most part taken directly from the behavioral simulator. The following description is representative:

```
(def-std-cell-schematic ("simple-example"
                         :inputs(p q a b)
                         :outputs(z w))
   (setq z (if (and p q)
               a
               b))
   (setq w (not (and a b) (or p q))))
```

When compiled, the logic is simplified and a rule-based technology selector optimizes the circuit used by choosing gates, merging gates and eliminating unnecessary inverters. A simple pattern matching language is used to apply the rules. The rules are augmented as different situations arise during the course of the design. The following represent rules for recognizing AOI's.

```
(define-eqn-transformer AOI (NOR (AND b c d) a)
        (AND3-NOR2 a b c d))

(define-eqn-transformer AOI (NOR (NOR a b)
                                 (NOR c d))
        (OR2-OR2-AND2 a b c d))

(define-eqn-transformer AOI (AND (NOT (NOR a b))
                                 (NOT (NOR c d)))
        (OR2-OR2-AND2 a b c d))
```

These generally have a higher likelihood of being accepted because they make the most efficient use of area. Different, but equivalent logic expressions can map onto the same AOI. In all cases, the canonical expression represented by the nested ANDs, ORs and NOTs is converted to a specific gate (i.e. an OR2-OR2-AND2) which is in the standard cell library.

Some rules implement demorganizing that is specific to special gates (i.e. XORs), while others recognize circuit speed-up rules (i.e. it is advantageous to reduce the setup time to a latch).

```
(define-eqn-transformer INVERSE (XOR a (NOT b))
        (XNOR a b))
```

Finally, less desirable rules are attempted which try to reduce the placement problem by assigning gate-inverter chains to a single standard-cell, thus reducing channel routing occupancy.

```
(define-eqn-transformer IMPLODE (NOT (NAND a b))
        (AND a b))
```

The designer can specify named gates (ie. NAND5, NOR2) thereby forcing a specific implementation.

Although the designer does not have to be aware of the underlying object-oriented data-base to complete designs, a knowledge of these details can aid productivity. Most of our designers can write simple data-base query programs such as the following Lisp program which calculates the total internal capacitance of a network.

```
(defmethod (internal-capacitance rsim-network-mixin) ()
  (loop for node in nodes
        unless (or (eq node gnd)
                   (eq node vdd))
          sum (node-capacitance node)))
```

A electrical rules checker is used to check port names, port directions, bus widths and other structural rules.

4. Physical Design

All of the chip cell designs (including pads) are specified in the virtual grid symbolic layout style either with the NS graphics editor (color or monochrome) or via Lisp procedures. This is a process independent layout style in which the designer does not have to deal with geometric design rules. Rather transistors and wires are placed on a coarse grid in a relative manner to each other. Cell designs are compacted or spaced according to a target technology description. The design style can deal with large pitch-matched cells of the order of 50K transistors. Our approaches to this are summarized in [2].

A standard cell layout system automatically generates symbolic layout from control schematic diagrams with the option of using port locations specified by the mask-outline. Both min-cut and thermal-annealing [3] approaches have been used. Min-cut is much faster but thermal annealing has a slight density edge. We expect the min-cut density will improve and will become the automatic layout style of choice. The aspect ratio of the module may be specified to allow area tuning. Figure 3 shows a typical virtual grid standard cell layout with the contents of some cells displayed.

Data paths are constructed manually. Basic cells such as registers, muxes and adders are provided in a data-path standard cell library. To improve productivity, liberal use is made of generators [4]. The following generator horizontally abuts three cells and raises the instance ports to this level of the hierarchy.

```
(defaspect-generator (data-path :virtual-grid) (flag)
  (HORIZONTALLY-ABUT 'module-a
                     (if flag
                         'module-b
                         'module-d)
                     'module-c)
  (import-ports-on-edges))
```

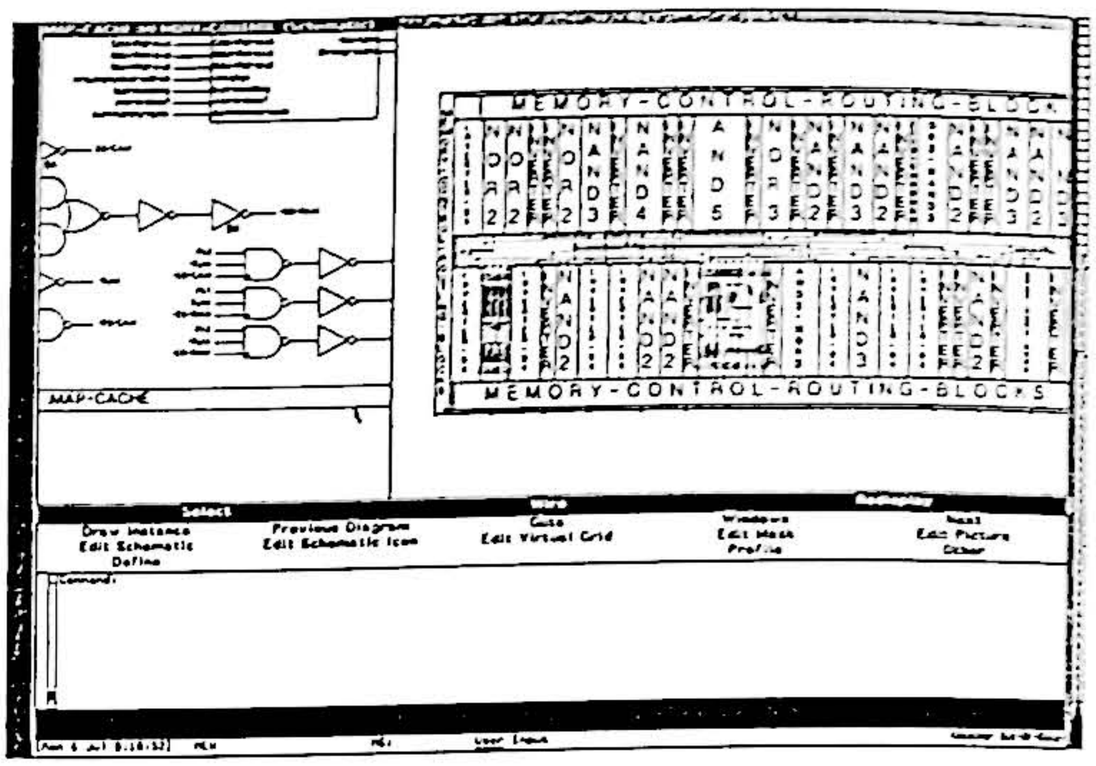

Figure 3. A Symbolic Standard Cell Layout

When all modules have been designed symbolically and compacted, the NS interactive editor is used to specify a slicing style floorplan [5][6]. Using the connectivity of the corresponding schematic, this floorplan is used as the basis to automatically place and route the entire chip. A global router first assigns nets to the routing channels. When this is complete, modules are composed according to the floorplan composition ordering. As they are connected, power, ground and clocks are also routed. This chip composition takes 2 hours to run for the complete Ivory chip. At early stages of design, partial floorplans can be constructed using estimates of block sizes. The following specifies the "example" module which has an estimated size of 250u by 300u and has the inputs entering on the top and the outputs exiting on the bottom.

```
(def-mask-outline example (250 300)
   (:top "s<3:0>"
         memory-write-pending
         bus-master-pin)
   (:bottom mcw mcr mcrw))
```

Figure 4 shows the floor-plan of Ivory.

A network comparison program is able to compare any two extracted networks (i.e. from virtual-grid layouts, schematics, mask layouts and vendor net-list files)[7]. Interactive feedback is provided to identify suspicious nodes. No node names are necessary.

A fast interactive DRC is provided for final mask artwork checks.

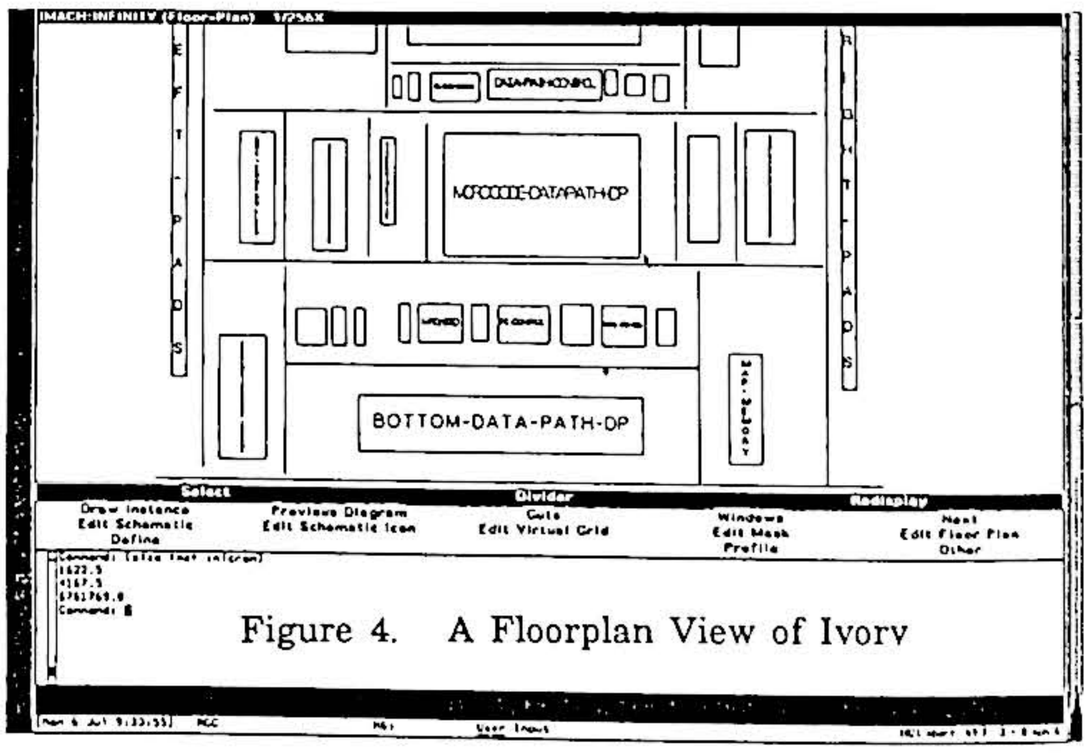

Figure 4. A Floorplan View of Ivory

5. Simulation

5.1 Circuit Simulation

A switch-level simulator with timing (RSIM)[8] was used to bridge both the gate and switch level circuit simulation requirements. Apart from being optimized for fast simulation, our version of RSIM has the ability to specify functional models in the following manner (a RAM):

```
(deffunctional-model cache-memory
  :inputs ("addr<6:0>"
           -row-enable
           write
           "write-data<39:0>")
  :outputs (("data<39:0>" :pd-size 8/1 :pu-size 16/1))
  :local-state ((cache-array
                 :initform
                 (make-array 128 :initial-element 'x)))
  :delays ((row-enable↓ → data :delay 25))
  :timing-constraints
          ((addr → -row-enable↓ :setup 16)
           (write-data → write↓ :setup 15))
  :model (cond
           ((eql addr 'x)
            (setq data 'x))
           ((eql write 1)
            (setf (aref cache-array addr)
                  write-data)
            (setq data write-data))
           ((eql write 0)
            (setq data
             (aref cache-array addr)))))
```

Access to the RSIM simulator is available in parallel either via Lisp code that can set, read and compare values on a circuit node, or via mouse clicks on a schematic displayed in the graphics editor window. A hierarchical schematic can be traversed using PUSH/POP commands. The right window in Figure 2 displays the result of clicking near nodes. Test programs written in Lisp use a protocol consisting of three *generic functions*:

- VALUE, which returns the value of a node,
- SET-VALUE, which sets the value of a node, and
- SIM-STEP, which propagates all changes through the network until no further changes occur. Optional arguments to SIM-STEP can specify the length of the simulation period.

The normal Lisp-machine operating system features such as breakpointing, incremental compilation and function restart, along with the scope-probe capabilities of the NS design system, create an extremely powerful verification capability with a fast edit/compile/debug loop. This interactive circuit debugging capability is available on either schematics, symbolic layouts, or mask layouts.

At the detailed circuit level, SPICE [9] was used to verify critical parts of the design, such as adder speed and memory timing. An interactive interface is provided to SPICE which can be run locally on the Lisp machine or remotely over the network. Figure 5 shows a typical SPICE interaction session.

To give some idea of the extensibility of NS, an experimental timing simulator mode based on backward Euler integration [10][11] was added to NS in a matter of a morning by a designer. We intend to incorporate parallel fault simulation into RSIM in the future, but it will probably take more than a morning.

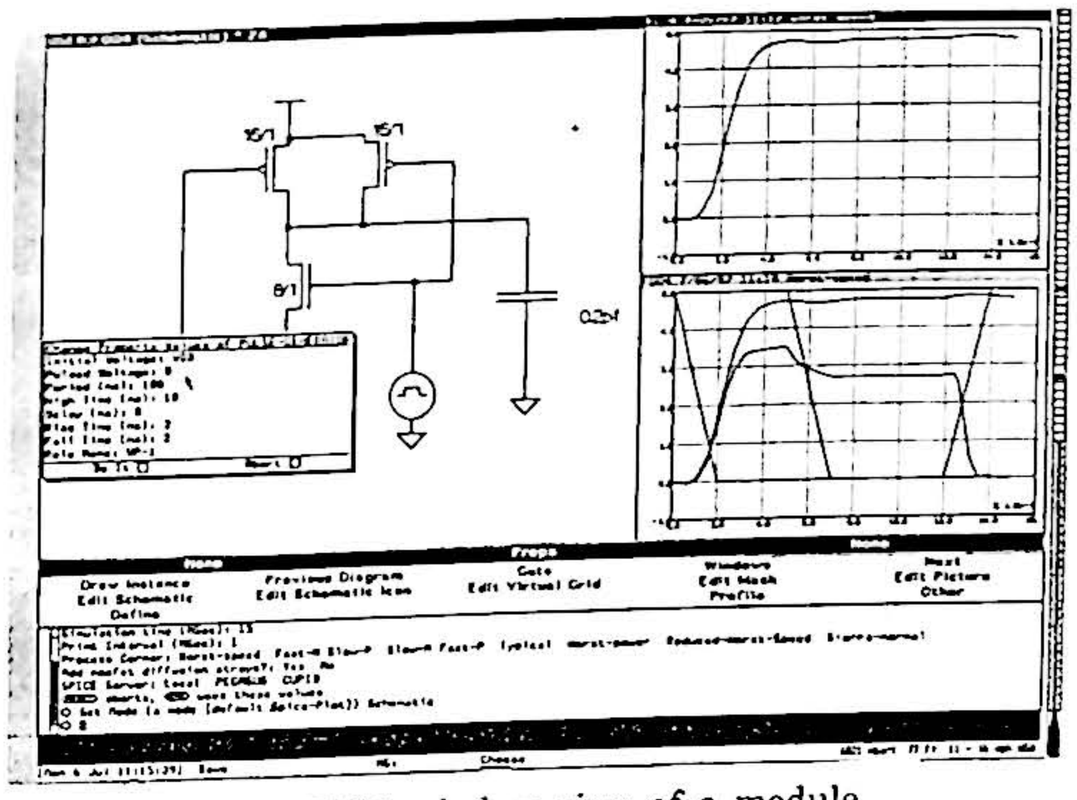

Figure 5. A SPICE window view of a module

5.2 Hardware Simulation Acceleration

Although RSIM is relatively fast (of the same order of speed as current software logic simulators), the size of some modules (and certainly the whole chip) results in simulation times beyond realistic limits. A number of methods are available to solve this problem, including operating one module at a time at the switch level while the others operate at the functional level. The solution we adopted was to invest in a hardware simulation accelerator. With a moderate software effort, we were able to create an interactive environment for the engine that is identical to the interactive RSIM environment. The accelerator is currently the only method of simulating the entire chip at a sufficiently low level to give the chip designers confidence in the transistor-level design. The simulation speed is roughly 10 instructions a minute.

Apart from a UNIX™ server and a Lisp-machine server required to operate a network command protocol, the most significant piece of software is a "gate-recognizer." To maximize modeling element use in the accelerator, all groups of transistors are converted to logic gates, unidirectional switches, or (if unavoidable) bidirectional switches.

6. Design Verification

6.1 Functional Comparison

The consistency of the architectural and behavioral simulators is verified by applying a large set of test programs to each simulator and checking for consistency according to an instruction-set architecture specification. Comparing the behavioral simulator and the circuit data-base can be performed on a signal by signal basis as the behavioral simulator is used to specify the modularity and communication between modules of the chip.

As both the behavioral simulator and RSIM were implemented using the object-oriented programming facilities of the Lisp machine, one can pass common messages to both simulators to achieve the required behavior.

During verification, SET-VALUE messages are passed to both the behavioral simulator and RSIM. Each particular simulator takes the appropriate action to set internal nodes to a particular value. This is achieved using a "forwarding network" which takes a list of two networks, one the RSIM

network and the other the behavioral simulator 'network' and the node under question and successively applies the SET-VALUE procedure to both nodes in each simulator. Thus the SETV function is used to set values in both networks:

```
(defmethod (setv forwarding-network) (node value)
   (set-value rsim-network node value)
   (set-value behavioral-model node value))
```

As the functional simulator works with a two-phase clock, a function is written to emulate the clocks for the RSIM simulation. A simple version of this would look as follows:

```
(defmethod (simulate-phase-1 rsim-network-mixin)()
    (set-value self 'ph1 1)
    (sim-step self)
    (set-value self 'ph1 0)
    (sim-step self))
```

This message is applied to both networks to advance through a phase1 clock cycle. The following forwarding network function calls both phase1 execution functions:

```
(defmethod (phase1 forwarding network) ()
   (simulate-phase-1 rsim-network)
   (simulate-phase-1 behavioral-model))
```

The VERIFY command operates by asking the two simulators for values and then comparing the results. If the results disagree, a debugging session with the user is initiated.

The following represents a simple test program.

```
(declare-bus 'op1 32)
(declare-bus 'op2 32)
(declare-bus 'external-bus 32)

(defun compare-adder-ops (op1 op2)
  (phase1)
  (setv 'op1 op1 )
  (setv 'op2 op2 )
  (phase2)
  (verify 'external-bus))
```

Such programs are written by designers to test the functionality of individual modules.

To provide higher level tests, a *spy strategy* was developed. With this facility, the complete behavioral simulator could be exercised by Lisp test programs. Arbitrary collections of modules can be grouped together (to model physical layout groupings) and their collective inputs and outputs monitored to provide a trace history of the boundary signals. The storage of the history allows both interactive and batch simulation. RSIM simulation code consisting of SETV and VERIFY statements, is generated from the history. This code was then applied directly to the extracted network with any discrepancies detected by VERIFY errors.

The SPY code was extended to allow interfacing to an engineering tester. This allows interactive debugging of tests in an engineering environment that was closely linked to the program development environment of the Lisp machine.

6.2 Timing Analysis

A timing analyzer based on finding critical paths through transistor networks was implemented based on Crystal[12]. It was subsequently rewritten to incorporate ideas from Leadout [13]. The timing analyzer heuristically determines directions on bidirectional devices that cause loops in the circuit [14]. Support for functional models (using RSIM functional models) is also provided.

The timing analyzer works on schematics, virtual grid symbolic layouts and mask layouts. It provides copious feedback, including highlighting critical paths in the interactive editor and a provision for plotting the actual clock waveforms that may be used with the circuit. Provision is made for blocking false paths via the RSIM interface. Transistor sizes and stray capacitance may be changed and the timing results calculated incrementally. On Ivory (113K transistors and functional models for ROM and RAM) it takes 2 hours (on a Symbolics 3640 Lisp machine) to find all clocking constraints.

7. Version Control

With any complex design involving a team of people, measures must be incorporated into a system to prevent the inadvertent inclusion of erroneous design data. Apart from NS support for the update of stale data libraries, we also implemented a change control system called the Management System.

During the design, specific verification steps have to be run on each module. For instance, a standard cell logic block has to be electrical rule checked, port checked, simulated, auto-placed, compacted, net-compared and the mask description saved. For each different kind of module a particular "step" was written to describe the necessary construction and verification tasks. When the step was run, all files and their version numbers used in the execution of the step were recorded. A data-base was kept which could be queried for the steps that had been run and the steps that had to be run if any sub-module (and hence archive file) was changed. The management system provides an up-to-date view of the design progress and can remotely execute jobs on vacant Lisp machines. Using a history of previous job times, it can predict the time to completion of a given tapeout run.

To create and check the entire Ivory physical data-base takes 30 Lisp CPU days. By using machines in parallel over the network, this can be done in 3-4 days. While this might seem a long time, this is the time to create a totally new physical data-base. The incremental time to produce a new chip after a typical ECO is on the order of hours.

8. Summary

Currently, the NS design system is also being used in Symbolics for the board level design of products based on Ivory, gate array design and other internal product designs.

The tools developed to design Ivory comprised approximately one third of the total design effort. Without such tools, it is unlikely that the design of a chip of the complexity of Ivory would have been successful at first silicon. Lisp is the design language and implementation language for the entire project. By designing in Lisp, no intermediate languages have to be designed and no extraneous parsers written. The full richness of the Lisp software development available to the Lisp system programmer is also available to the VLSI designer.

All of the tools mentioned in this paper were written in New Flavors Lisp from scratch under the Symbolics Genera 7 operating system by various of the authors of this paper. The NS system totals roughly 50K lines of code. The brevity of the code is derived from the object-oriented approach which provides for modularity, reusability and extensibility.

On the basis of the wide range of design activities supported, the intimate integration of the tools, the small programming team size and the complexity of chip designed,

we believe our success strongly supports the high productivity claims made for symbolic programming environments. The success of this Lisp object-oriented approach to design appear to herald a consistent, modular and manageable approach to complex VLSI design tasks.

9. Acknowledgments

We would like to thank our other designers Mark Matson and David Chan, who through their activities have helped direct the development of NS.

10. References

[1] J. Cherry, "**CAD Programming in an Object Oriented Programming Environment,**" in *VLSI CAD Tools and Applications*, ed. W. Fichtner and M. Morf, Kluwer Academic Publishers, 1987.

[2] D. Tan and N. Weste, "**Virtual Grid Symbolic Layout 1987,**" see this proceedings.

[3] C. Sechen and A. Sangiovanni-Vinceltelli, "**The Timber-Wolf Placement and Routing Package,**" *Proc. CICC*, May 1984, pp. 522-527.

[4] J. Batali, N. Mayle, H. Shrobe, G. Sussman and D. Weise, "**The DPL/Daedalus Design Environment,**" *VLSI 81*, Edinburgh, 1981, pp. 183-192.

[5] U. Lauther, "**Channel Routing in a General Cell Environment,**" *VLSI '85*, E. Hoerbst, ed., Elsevier Science Publishers B. V., North-Holland, Amsterdam, 1986, pp. 389-399.

[6] C. Wardle, C. R. Watson, C. A. Wilson, J. C. Mudge and B. Nelson, "**A Declarative Approach for Combining Macrocells by Directed Placement and Constructive Routing,**" *Proc. 21st Design Automation Conference*, June 1984, pp. 594-601.

[7] C. Ebeling and O. Zajicek, "**Validating VLSI Circuit Layout by Wirelist Comparision,**" *Proc. IEEE International Conference on CAD*, Sept. 1983, pp. 172-173.

[8]: C. J. Terman, "**Timing Simulation for Large Digital MOS Circuits**", *Advances in Computer-Aided Engineering Design*, Vol. 1, JAI Press, 1985, pp. 1-92.

[9]: L. Nagel, "**SPICE2: A computer program to simulate semiconductor circuits,**" *ERL Memo No. ERL-M520*, University of California, Berkeley, 1975.

[10] B.R. Chawla, H.K. Gummel and P. Kozak, "**MOTIS - An MOS Timing Simulator,**" *IEEE Transactions on Circuits and Systems*, Vol. 22, No. 12, Dec. 1975, pp. 901-910.

[11] B. Ackland and N. Weste, "Functional Verification in an Interactive Symbolic IC Design Environment," Proc. of 2nd Caltech Conference on VLSI, Jan. 1981, pp. 285-298.

[12]: John K.Ousterhout, "**A Switch Level Timing Verifier for Digital MOS VLSI,**" *IEEE Transactions of Computer-Aided Design*, vol. CAD-4, No. 3, July 1985.

[13] T. G. Syzmanski, "**LEADOUT: A Static Timing Analyzer for MOS Circuits**" *Proc. ICCAD*, 1986.

[14] N. Jouppi, "**TV: An nMOS Timing Analyzer,**" *Proc. 3rd Caltech VLSI Conference*, 1983.