Vds: An Interactive VLSI Design System

Andrew S. Cromarty and Steven P. Levitan
Department of Computer and Information Science
University of Massachusetts at Amherst
Amherst, Massachusetts

COINS Technical Report 81-36
November 1981

Abstract

Vds is an interactive system for VLSI design, written in a graphics-extended version of CoinsLisp. All interaction with the Vds user is performed through Gus, a device-independent graphics kernel supporting a wide variety of raster-graphics, text display, and menu selection functions. Gus presents the user with a consistent interaction protocol independent of the hardware in use, and supports such features such as zoom, pan, and multiple viewports on all graphical output devices, allowing the VLSI designer to use graphics devices interchangably as their availability dictates. Multiple hierarchically-related graphical views of objects (currently full-detail, bounding-box, and floorplan views) are supported. All input-output relations are user-extendable and reconfigurable by means of "keyboard macros" and user-written functions which can be bound to keyboard keys.

A semantic net graph structure is employed for representation of circuits, allowing them to be archived, extended, and merged with other graphs or standard VLSI circuit libraries. The semantic net is hierarchically organized so as to allow composition of circuit subassemblies to form larger circuits, and supports built-in design rule checking based upon high-level semantic information about the design. Vds interfaces directly with circuit simulation software to provide additional design checking and optionally produces a translation of the semantic net into graphics commands compatible with a commercially-available Core Graphics Standard implementation. Because of its modularity, Vds is easily extended to support new graphical functions, user-coded VLSI composition functions, additional design rules and expert knowledge, and new or alternative VLSI subcircuit definitions.

[Implementation of the system described in this document is currently under way; this is a report of work in progress.]

1. Introduction

The last decade has seen a revolution in the way integrated circuits are designed. Prior to the mid-Seventies, the dominant design approach for Very Large Scale Integrated (VLSI) devices was the same as for small-scale and medium-scale integration: boolean gates (nand, nor, not, etc.) were interconnected to achieve some "random-logic" function. It soon became clear, however, that these increasingly complex devices could not be efficiently designed and debugged using the discrete-gate approach; integrated circuits so implemented were likely to waste the majority of the silicon in connections between irregular patchworks of logic, and efforts at further miniaturization were often hampered by the great complexity of chips designed according to this unstructured random-logic approach. In response to this problem, a structured design methodology was developed, pioneered largely by Mead and Conway [10]. This methodology presents the task of the designer of VLSI circuits as one analogous to that of the software designer: the top-down design of clean, regular, structured implementations of algorithms.

Concomitant with the growth of this structured design methodology was the appearance of computer-aided VLSI design systems, which assist the designer in developing a VLSI device much as screen-oriented text editors and text formatters assist the software designer in development of a program. The more primitive systems simply provided a picture of the chip while various "mask" layers were laid down and fleshed out, maintaining some minimal data base to record the location of specific polygonal objects on the chip [6]. More sophisticated systems [15, 1] performed additional functions such as compaction of subcircuits and some checking of the input to detect design flaws such as two polygons being placed too close together. (By our analogy

with software, such design-rule checking (DRC) corresponds to syntax checking in a compiler, and subcircuit compaction corresponds roughly to code motion in optimizing compilers.)

These developments signalled not the discovery of the solution, however, but merely the statement of the problem. VLSI devices can have high semantic complexity, just as large software systems do; however, current VLSI design systems leave largely unaddressed the problem of specifying, organizing, and taking advantage of such semantic information. Furthermore, it is typical that there will be a great degree of concurrent processing occurring in such a device, the description and specification of which is as unsolved a problem for the hardware domain as for software. Additionally, we still lack adequate formalisms to decribe the tradeoffs between such issues as size, speed, and complexity of these devices. Attractive graphical displays, as useful as they can be, are in and of themselves insufficient to organize the enormous amount of information involved in the VLSI design process.

It is therefore becoming clear that what is needed is not simply a computer-aided graphics system, but rather a system in which the graphics are integrally tied to successively higher levels of abstract description of the function and the structure of the device under development. Such a system should be cleanly and carefully designed so as to be amenable to extension, in order to incorporate both expert semantic knowledge about VLSI design and additional graphical models of the device. Furthermore, it should be as comfortable to use as is a good contemporary screen editor. This paper describes V_{ds}, our first steps towards the development of such a system.

2. VLSI Design: Decomposition and Synthesis

The design process for VLSI is similar in many ways to other logical design tasks. The initial problem is decomposed, in a top-down fashion, into subproblems and sub-subproblems until the problems are trivial to solve. These low-level solutions are then composed, bottom-up, to solve the bigger and bigger problems, until the original problem is solved. This technique leads to a modular structure for the final system.

There are, however, two differences beteween a purely logical design process, like that used for software, and the design of systems in VLSI: for VLSI (at this point in time) the "trivial" solution of the bottom level problems is not simple; and unlike programming tasks, the composition of solutions to the subproblems is itself a difficult task. For these reasons VLSI design is often seen, incorrectly, as strictly a bottom-up process.

Although the logical representation of bottom level problems such as logic gate design is simple, their electrical characteristics are poorly parameterized, and their geometrical representations are complex. Unfortunatly, design decisions during this part of the design process have a large effect on overall circuit behavior and size. This is largely because the final circuit will typically have many (often thousands) of replications of a simple structure.

The composition of simple subcircuits into more complex structures is also a difficult and poorly understood task. Unlike programming, where composition is done textually, the composition of circuits is a geometric problem involving three-dimensional size and placement problems. Even worse, the process of interconnecting the component parts, which in programming simply involves concatenation of lines of text, is an enormous problem in

itself for VLSI design [8, 12, 13].

2.1 The tasks of the designer

With this background we can look at the specific tasks of the VLSI designer and consider the ways in which a design system can help perform these tasks. Hon and Sequin [7] list the following tasks for the IC designer:

1. Entering design geometry
2. Outputting design geometry (plotting, printing)
3. Documenting the design
4. Checking for design rule violations
5. Checking for logical errors
6. Simulating the behavior of a design

All of the above tasks are necessary but by no means sufficient to characterize the design process. These are the bottom-up part of the design process and do not capture the initial top-down part of the process. Furthermore, conceiving of these tasks as independent, discrete events thwarts attempts to capture the full potential of a symbolic interactive design aid system.

2.2 The tasks of the design aid system

Instead of considering VLSI design from the perspective of design tasks, we can look at the information that the designer must keep track of and manipulate during the design process. This yields three coherent views of the system as it evolves from a problem to a solution. These alternative views of the device are called domains by Weste [15]:

1. Structural: A view in which the design is conceived of as a graph. Here each node of the graph represents a component of the circuit and each edge represents connections between components. Subcircuits are modeled as subgraphs.

2. Physical: The actual device construction is based on geometric areas on several photo-masks. This view directly represents the circuit as a composition of two-dimensional objects in different layers. The bottom level design, or layout, is performed almost exclusively with this view of the device.

3. Behavioral: The device is viewed as a collection of electrical components, each characterized parametrically. This view is concerned with timing, power and loading issues of the design. At a higher level of abstraction, this view captures the logical behavior of the device for incorporation into a "system" view.

We take the position that a design aid system must help the designer manage and manipulate alternatives views of the design [2]. As such the design system must perform the tasks of editing and managing a database containing both graphical data and text. The system should keep track of the different views of each part of the system and maintain "crosslinks" between different views where appropiate.

2.2.1 Support in the structural domain

The design system must support a hierarchical data base which reflects the structure of the design. As circuits are built by composition or modification of other circuits, the data base must keep track of the "parentage" of the circuits. This allows for automatic inheritance of design changes. In addition, connectivity, a relation that is lost in the physical domain, must be maintained in the stuctural domain. This allows the system to distiguish between circuits, which often have fixed geometries, and connections between circuits, which have more flexibility.

2.2.2 Support in the physical domain

There must be a graphics/text editor to allow the designer to manipulate the physical objects of the design [11]. These are both the geometric objects which represent the photo-masks and text objects necessary to comment, annotate, and document the design. The system should perform such chores as justification of text and compaction of graphics objects. In addition the system must help the designer with (or, to be more ambitious, solve) the problem of interconnecting subcircuits together.

2.2.3 Support in the behavioral domain

In this domain the system must maintain a logical description of each circuit and extract and maintain each circuit's electrical parameters for simulation purposes. Simulation must be done on both the electrical and logical levels of abstraction.

2.3 A coherent view of the design

For designers to do their work efficently and correctly they must have a coherent and detailed conception of all aspects of the design. Of course, it is not possible to keep all the trivial details in mind at all times; the system must help. It must have a structural, physical and behavioral representation of all the components of the system. And it must be able to present them to the designer in a uniform, easy to manipulate way.

3. The design of a design system

A driving force behind the design of V_{ds} is the commitment to provide the user with a consistent high-level model of the design process. In order to achieve this effect, several tools are brought to bear:

1. Device-independent graphical display
2. Simple efficient interaction with the design system
3. User extension and reconfiguration of the user interface
4. Hierarchical organization of both display and design process

3.1 Interactive design and device-independent graphics

An important means of attaining a consistent model of user interaction in V_{ds} is the use of device-independent input/output functions.[1] All input and output operations in V_{ds} are performed using primitives provided by the Gus device-independent graphics package [5, 14]. For example, Gus provides the capability needed for multiple viewports in which different graphical views of the object can be presented (e.g. a "floorplan" view and a more detailed view); additional viewports contain text, error messages, and menus describing the current status of the system and various keyboard bindings. These viewports can of course be dynamically created, reconfigured, and destroyed as is appropriate.

---

1. It is presently in vogue to consider dedicated VLSI Design Workstations to be the state of the art in VLSI design systems. Nonetheless, it is the feeling of the current authors that much or most of the VLSI design performed in the next five years will continue to occur on timesharing systems using shared-access graphics devices, because they are already in widespread use and have a substantial software base; in such environments, device independence of graphical display is indispensible.

In addition, Gus features like zoom and pan allow the user to "fly" across the silicon surface, installing and relocating named subcircuits or creating new ones. Such tasks as clipping, coordinate transformations, antialiasing, and polygon fill are all handled by Gus in a uniform, predictable fashion for all graphics devices.

Most important about the use of Gus, however, is its complete device-independence: because all input and output occur through a single device-independent subroutine library, the user may switch from one type of display device to another without any change in style of interaction. For example, if a high-resolution color monitor is currently unavailable, VLSI design can be performed at a standard low-resolution terminal. Checkplots no longer require special handling by a separate program; they are obtained by simply requesting redisplay of the desired chip region, with a plotter or printer as the output device. All graphics and text requests are handled on a "best try" basis, using with maximal efficiency whatever capabilities the display and input hardware support. This has the effect of presenting the user with an extremely uniform model of device interaction without concern for device availability, reliability, or capability.

3.2 Style of user interaction

The interface between the user and the system is through a graphical VLSI editor. This approach is taken for two reasons. First, most of the user's time is spent editing graphic or text objects. Second, we want the user to have a single mode of interacting with the system. There is no edit-mode versus run-mode dichotomy; even in the editor there is only one mode. Since the editor accepts command strings from the user, all functionality inherent in the V_{ds} standard design function library is available to the user at all

times.

To facilitate easy and efficient communication with the system we have incorporated the following principles into the design of the editor:

1. Single-keystroke commands: There is no "enter text" or "enter command" mode. All keys either are commands or are inserted as text or graphics; keys have labile definitions which can be changed by system or user (described below).

2. Work spaces: Editing of objects is done in a workspace. There are many workspaces current at any given time. Work spaces can be named and then composed together into new workspaces. This can be done as a merging process or as a hierarchical process (see next item).

3. "Feltboard" approach to composition of primitive objects: Creating primitive objects is done by creating a "felt" object which can be applied to the "feltboard" workspace currently in use. Another workspace or feltboard can be treated as a felt object itself; feltboard "circuits" can thus be composed hierarchically into more complex objects.[1]

4. User-defined commands: In addition to the design functions provided in the base system, new functions may be defined by the user. Such functions can act on any level of the database; for example, the user can add new design rules, new graphics functions, or new circuit-defining routines.

5. User control over the user interface: The assignment of various V_{ds} functions to keyboard keys can be customized according to the user's tastes, both to allow redefinition of the default key bindings and to incorporate functions newly defined by the user.

3.3 Hierarchical organization of design and display

The V_{ds} VLSI design process is hierarchically organized, largely by virtue of the semantic network database and the functionality it provides. The virtual "feltboard" workspace seen by the user corresponds to a named space in Grasper. In each such space lives an arbitrary number of named nodes

---

1. Composition is simply a matter of installing indirect references into a currently active workspace (feltboard), so that the workspaces remain unchanged when they are used as components of other workspaces.

connected by labelled edges; each node represents an instance of either a primitive silicon object, such as rectangles occurring at a certain mask level, or else an instance of some composed circuit (that is, a pointer to another space). "Application" of a new "felt object" by the user corresponds to creation of a new node in the current space. The spaces can be named and then composed in turn to form larger named circuit objects. Instances of these circuits can be parameterized for geometric operations such as rotation, translation, and mirroring.

Grasper spaces are also used to provide multiple simultaneous orthogonal partitions of the database. For example, the silicon mask layers form one partition of the data, the circuit class (e.g. gate type, PLA style, etc.) of the circuit objects forms another, and the geographic location of objects forms a third. An important effect of this partitioning scheme is that relational information about currently available subcircuits is easily obtained or changed by using subset operations provided by Grasper; for example, the user can ask for (list or display) all objects in a specified region, all objects in a certain mask layer, and so forth, or any boolean combination of these requests.

The graphical world of V_{ds} is similarly hierarchically structured to reflect the hierarchical structure of the design. Primitive silicon polygons correspond to primitive polygonal graphical objects, and larger composed circuits can be represented graphically either as unitary graphical objects (e.g. a labelled box) or through detailed display of all their myriad component parts. Associated with each graphical object may be several names corresponding to different levels of abstraction; choice among these at display time is based upon the level of abstraction currently requested for the entire feltboard in a viewport. For example, an array of NAND gates might

be labeled "NAND" when viewed at from an intermediate level of (logical and graphical) abstraction, but when the user zooms in for a closer look at the circuit, the names may expand to "3-input NAND with limited driving capability"; zooming up to a higher level, the user might find them part of a single box named "Timing FSA".

4. Implementation of the design

V_{ds} is implemented in CoinsLisp [3, 4], a highly structured and regular dialect of LISP. There are several reasons for this choice of implementation language:

1. Graphics support: CoinsLisp supports the Gus device-independent graphics package primitives as CoinsLisp functions. This allows the software writer to capitalize on the efficient and logically coherent high-level graphics capability such a system provides and thus avoid the overhead of designing, implementing, and maintaining a complete graphics programming language or low-level graphics model of the VLSI design process.

2. Extendability: Because CoinsLisp is an interpreted language with dynamic binding, graphics and design functions can be changed or added by either the V_{ds} system designer or the user at any time. This allows composition of new graphics or VLSI design functions "on the fly" at the time that the user determines their need.

3. Performance: Attention was paid to issues of efficiency and speed in the design of the Vax/VMS implementation of CoinsLisp, so that reasonable performance could be guaranteed. In fact, preliminary informal benchmarking of the V_{ds} system indicates that its performance using interpreted code is competitive with compiled C code performing the same function at other installations (in the specific case considered, parsing VLSI layouts in Caltech Intermediate Form[1] and installing the resultant information into the database). Of course, the V_{ds} system itself can be compiled for an additional improvement in execution speed.

---

1. CalTech Intermediate Form (CIF) is the standard portable low-level representation language for VLSI devices.

4. Database support: Most languages require database structures and access functions to be built up starting essentially from scratch; CoinsLisp, however already contains embedded within it the Grasper graph-processing language [9]. This provides the software writer with coherent, optimized database support at a very high level (that of the semantic network), freeing the software designer to consider issues more germane to VLSI design.

5. Conclusions and future directions

The approach we have taken is to implement a simple consistent "toolbox" of VLSI design functions in a single coherent, extendable environment. Because the user is presented with a uniform customizable interaction protocol without regard to features or limitations of specific hardware devices, V_{ds} can comfortably serve experts and novices alike: beginners need not spend most of their time adjusting to inconsistencies among various piecemeal routines and devices, while more experienced users can alter the design environment to suit their taste.

Furthermore, an attempt has been made to support (perhaps enforce) structured design techniques, by hierarchically organizing the design database and then providing functions for graphical display, VLSI device composition, and information gathering which make efficient use of this organization.

The process of VLSI design is an excellent example of an expert knowledge domain; it is still the case that a small number of highly capable (and well-paid!) wizards perform the poorly-understood design and verification tasks. It is to be expected that techniques of artificial intelligence will be brought to bear on this problem in the near future. The extendability of V_{ds} is intended to allow rapid and reliable incorporation of this expert knowledge into the design system itself as that knowledge becomes available. In the mean time, V_{ds} allows the results of the device design process to be verified using the conventional (if somewhat brute-force) methods of low-level

circuit simulation.

Finally, the trend towards implementing VLSI design systems on personal workstation hardware merits some discussion. Problems of graphics device availability and compatibility will not simply disappear once workstations are put into use; since a variety of graphics devices can be expected to be found on workstation networks of the future, device-independent graphics will still be an asset to the VLSI designer. Similarly, local processing power and bitmapped display devices are not a substitute for well-structured, extendable design systems. Systems such as V_{ds} will thus be prime candidates for incorporation into personal VLSI design workstations, and implementation of a V_{ds}-based VLSI design station is currently under study.

References

1. Batali, J., Hartheimer, A. The Design Procedure Language Manual. Massachusetts Institute of Technology AI Lab Memo 598 / VLSI Memo 80-31, 1980.

2. Clarke, L., Graham, R., and Wileden, J. Thoughts on the design phase of an integrated software development enviroment. Proceedings of the Fourteenth Hawaii Intl. Conf. on Systems Science, Honolulu, Hawaii. January 1981.

3. Cromarty, A. The CoinsLisp Reference Manual. Department of Computer & Information Science, University of Massachusetts at Amherst, 1981.

4. Cromarty, A. CoinsLisp: A highly regular functional programming language. (In preparation for submission for publication.)

5. Cromarty, A., and Sutton, R. Gus: A portable device-independent graphics kernel. (In preparation for submission for publication.)

6. Fairbairn, D., and Rowson, J. ICARUS: An interactive integrated circuit layout program. Proceedings of the Fifteenth ACM/IEEE Design Automation Conference, Las Vegas, NV, June 1978. pp. 188-192, 1978.

7. Hon, R. W. and Sequin, C. H. A Guide to LSI Implementation (second edition), SSL-79-7 January 1980. Xerox Palo Alto Research, Palo Alto, CA, 1980.

8. Leiserson, C. E., and Pinter, R, Y. Optimal Placement for River Routing. In: VLSI Systems and Computations, Proceedings of the Carnegie-Mellon Universtiy Conference on VLSI Systems and Computations, October 19-21, 1981, pp. 126-142. Computer Science Press, Inc., Rockville, MD, 1981.

9. Lowrance, J. Grasper 1.0 Reference Manual. Technical Report 78-20, Department of Computer & Information Science, University of Massachusetts at Amherst, 1978.

10. Mead, C., and Conway, L. Introduction to VLSI Systems. Addison-Wesley Publishing Co., Reading, MA, 1980.

11. Ousterhout, J. K. Caesar: An Interactive Editor for VLSI Layouts. VLSI Design, 2:34-38, 1981.

12. Pinter, R. Y. Optimal Routing in Rectilinear Channels. In: VLSI Systems and Computations, Proceedings of the Carnegie-Mellon Universtiy Conference on VLSI Systems and Computations, October 19-21, 1981, pp. 160-177. Computer Science Press, Inc., Rockville, MD.

13. Rivest, R. L., Baratz, A. E., and Miller, G. Provably good channel routing algorithms. In: VLSI Systems and Computations, Proceedings of the Carnegie-Mellon Universtiy Conference on VLSI Systems and Computations, October 19-21, 1981, pp. 153-159. Computer Science Press, Inc., Rockville, MD.

14. Sutton, R., and Cromarty, A. The Gus Reference Manual. Technical Report, Department of Computer & Information Science, University of Massachusetts at Amherst, 1981.

15. Weste, N. MULGA — An interactive symbolic layout system for the design of integrated circuits. Bell System Tech. J. 60:823-857, 1981.