

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A220 740

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NW-LIS-89-12-07	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) WireLisp: Combining Graphics and Procedures in a Circuit Specification Language		5. TYPE OF REPORT & PERIOD COVERED Technical
AUTHOR(s) Carl Ebeling, Zhanbing Wu		6. PERFORMING ORG. REPORT NUMBER
PERFORMING ORGANIZATION NAME AND ADDRESS Northwest Laboratory for Integrated Systems University of Washington Dept. of Comp. Science, FR-35 Seattle, WA 98195		6. CONTRACT OR GRANT NUMBER(s) N00014-88-K-0453
1. CONTROLLING OFFICE NAME AND ADDRESS DARPA-ISTO 1400 Wilson Boulevard Arlington, VA 22209		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research - ONR Information Systems Program - Code 1513: CAF 800 North Quincy Street Arlington, VA 22217		12. REPORT DATE December 1989
		13. NUMBER OF PAGES 4
		15. SECURITY CLASS. (of this report) Unclassified
		18a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this report is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Lisp, TLisp, Graphical description, procedural description, VLSI circuits, schematic, parametrized schematic.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) WireLisp is a language that incorporates both procedural and graphical constructs for describing the structure of complex circuits. This combination provides both the clarity of graphical representation and the expressiveness of a procedural description. This paper describes how this is done in a conceptually simple way by representing procedural information graphically. WireLisp is built on Lisp which Continued on back page....		

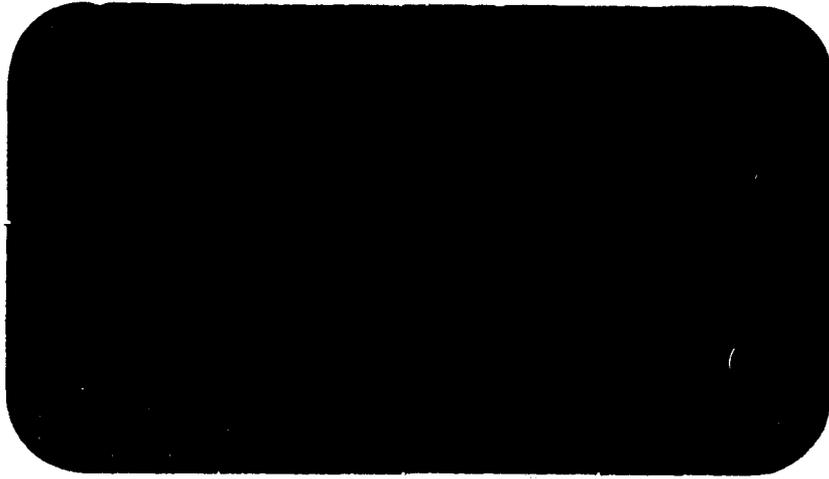
DTIC
ELECTE
APR 20 1990
S E D

(Continued from front page)

allows the designer to extend the language with arbitrary functions. WireLisp can be used to generate a variety of different target output descriptions, and allows the incorporation of other kinds of descriptions such as behavioral and physical descriptions. WireLisp is implemented in TLisp and has been used to describe a complex (130,000 transistor) VLSI chip design.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	





DEPARTMENT OF COMPUTER SCIENCE

University of Washington

Seattle 98195

90 04 18 069

**WireLisp: Combining Graphics and Procedures
in a Circuit Specification Language**

Carl Ebeling and Zhanbing Wu

Technical Report #89-12-07

**Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA**

WireLisp: Combining Graphics and Procedures in a Circuit Specification Language

Carl Ebeling

Zhanbing Wu

Department of Computer Science and Engineering
University of Washington
Seattle, Washington

ABSTRACT

WireLisp is a language that incorporates both procedural and graphical constructs for describing the structure of complex circuits. This combination provides both the clarity of a graphical representation and the expressiveness of a procedural description. This paper describes how this is done in a conceptually simple way by representing procedural information graphically. WireLisp is built on Lisp which allows the designer to extend the language with arbitrary functions. WireLisp can be used to generate a variety of different target output descriptions, and allows the incorporation of other kinds of descriptions such as behavioral and physical descriptions. WireLisp is implemented in T. Lisp and has been used to describe a complex (130,000 transistor) VLSI chip design.

Introduction

WireLisp is a language for describing the *structure* of a digital system. Structure refers to the relationship between the elements of a system and is easiest to represent and understand graphically. The common representation of circuits is thus a *schematic* representation corresponding directly to the structure of the circuit. The concept of hierarchy is very useful in describing circuits and can be incorporated directly into schematic descriptions by allowing subcircuits comprising related modules to be combined into a single *abstract* component. This abstraction mechanism reduces the amount of information required to understand a circuit description at any one level by hiding information within abstract components. Abstraction extends the designer's grasp by removing unnecessary detail and reduces the interaction between parts of a system by isolating that detail. Combining hierarchy with graphical constructs provides a very intuitive and structured way to describe circuits and most modern schematic drawing systems incorporate hierarchy.

Although graphical descriptions are clear, they are not very expressive. That is, there are many conceptually simple circuits for which graphical descriptions are very inefficient. For example, describing a 32-bit register as a collection of 32 flip-flops requires a large, repetitive schematic. The procedural concept of iteration is a much more expressive way to describe this circuit. The graphical description of a large decoder is also unwieldy as the circuitry for each output is slightly different and cannot be simply replicated within the drawing. However, this circuitry can be succinctly described using iteration, conditionals and the ability to perform computation on the iteration variable. Graphical descriptions are also "brittle"; simple changes in the parameters of a system can require very time-consuming changes to the schematic.

By contrast, procedural descriptions can be written in terms of system parameters to adjust to new values without modification. Thus procedurally based descriptions can efficiently specify the large, regular circuits from which large digital systems are constructed.

Unfortunately, procedural descriptions, while expressive, are obscure. That is, the structure of a circuit is not immediately obvious from a procedural description. Both writing and understanding procedural descriptions can be very difficult. The approach of some schematic systems is to allow limited procedural constructs in circuit drawings. For example, arrays might be used as a limited form of iteration to generate several copies of a device or a signal. Parameters may be allowed in this context to allow differently sized components. While this increases the expressiveness of schematic drawings, it is only in a limited form that corresponds to syntactic textual substitution. Without variables, conditionals and general iteration many circuits can be described only inefficiently.

The goal of WireLisp is to incorporate full procedural power into graphical descriptions. Since it is difficult to predict all the constructs a user will need, WireLisp imposes no restrictions and includes the full range available in Lisp. Indeed, the user is free to define and use new functions. Whether a circuit is described graphically or procedurally is a decision left to the designer who can choose the appropriate approach based on the type of circuit being described. Moreover, procedures and graphics can be freely intermixed. That is, Lisp expressions can be embedded in graphic descriptions and graphic descriptions embedded in Lisp expressions. This allows descriptions that are highly expressive, yet easily specified and understood.

The underlying model of WireLisp uses procedures to describe how devices are constructed. A device procedure is invoked to create an instance of the device, which may vary according to the parameters passed to the procedure. An example of the definition of a device called *treecomp* is given in Figure 1. This device compares two N-bit numbers using a recursive divide and conquer approach. While this description relies heavily on procedural constructs, the resulting circuit structure is quite obvious from the circuit drawing.

Device procedures are typically specified using a drawing editor, but they can also be produced by other CAD tools such as logic synthesis programs. Thus one device may be described graphically while another described using a behavioral description which is converted into a WireLisp procedure by a synthesis tool such as a PLA or multi-level logic generator. The output produced by a WireLisp program is also flexible since it is produced by the device procedures themselves. Although device libraries can be provided for common devices and technologies, the user can easily produce whatever output is desired. WireLisp provides a structural framework for specifying a system as a collection of components described in different ways.

¹This research was supported by the Defense Advanced Research Projects Agency, DARPA Contract #N00014-88-K-0453, by NSF Grant #CCR-8657589, and IBM Contract #87-653490.

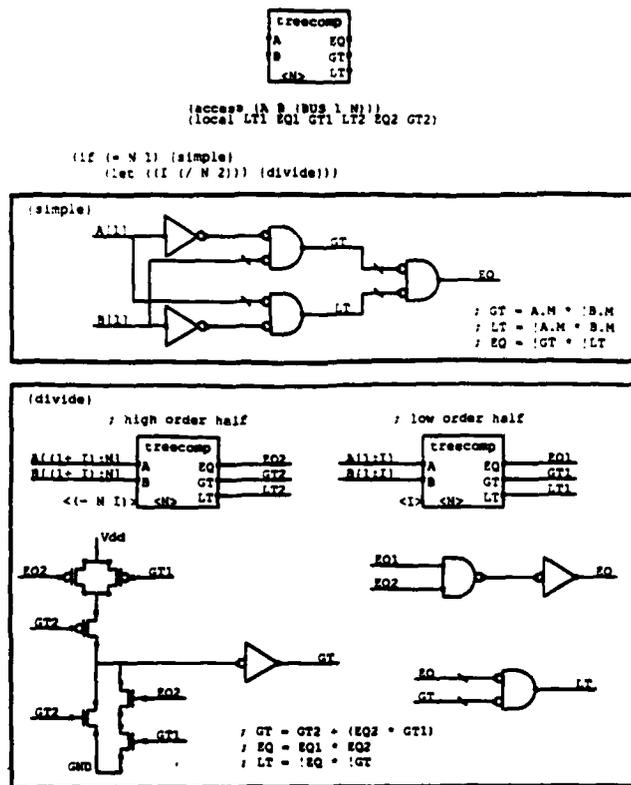


Figure 1: A WireLisp Drawing

Related Work

Programming languages have been used before to describe circuits, notably the use of Lisp in netlist [12] and DPL[4]. WireLisp is similar to netlist in its use of procedure hierarchy and its definition of device connections via parameters. However, netlist is purely procedural and is unwieldy for representing complex systems. Moreover, netlist has only a very primitive notion of signal structure that prevents information hiding.

Both graphically and procedurally based VLSI layout systems have been described, but little has been done to incorporate both types of description in the same specification. Some systems such as Mocha Chip[10] and WIN[1] use both types of description in the specification of chip layout, but they cannot be intermixed at a fine-grained level.

Xerox PARC has recently produced a circuit specification system that incorporates both graphics and programs[2, 3] that is similar in philosophy to WireLisp. The mechanism in WireLisp is conceptually simpler which allows extensions to be defined more easily. WireLisp also intertwines the graphical and procedural constructs more closely which yields specifications that are more expressive and self-apparent.

The WireLisp Model

In the model that WireLisp uses, each device in a circuit description is a procedure that describes how it is constructed. When a module is used, this procedure is executed to generate an instance of the module. A device procedure may be parameterized and therefore define an entire family of devices.

Devices are either composite or leaf devices. Composite devices are comprised of other devices while leaf devices are primitive devices without components. An instance of a composite device is constructed by first invoking the device procedures of its components and then connecting the resulting instances together. Leaf devices have no component device procedures to invoke, but instead perform actions such as writing the appropriate description of the device to an output file.

Devices are connected together via named signals which correspond to wires in the actual circuit. These signals are connected to devices via signal parameters which correspond to the terminals of the physical device. When a device procedure is called, each signal parameter is bound to the signal in the environment of the device to which the terminal is connected. Thus devices are interconnected by connecting them to common signals.

Devices may also have general parameters which describe which instance of a family of devices is to be constructed. Optional parameters with default values are also provided to reduce the amount of information required in the circuit description. In Figure 1, N is a general parameter indicating the size of the numbers being compared.

A complete hierarchical WireLisp description becomes a program with nested procedure calls corresponding to the circuit hierarchy. This program is executed to produce an output description which depends on the side effects of the procedures during program execution. For example, if each leaf device procedure writes a description of itself to an output file, the result will be a flat netlist of the circuit. On the other hand, the procedures might construct an in-core data structure corresponding to the circuit, or make entries into a design database. The result of execution is thus defined by the user, although libraries can be provided for primitive devices in common target technologies and output description formats.

Drawing Device Procedures

Although in principle the designer can write device procedures directly, the intent of WireLisp is for these procedures to be described in circuit drawings similar to schematics. Typically, most of a circuit can be described graphically, with conditionals and iteration used as required. However, even procedural constructs can often be represented graphically as shown in Figure 2. Here the if form is represented by a pseudo-device which acts as a compile-time multiplexor conditioned by the variable i: The first section of the shift register is connected to an input SI instead of the output of the previous section.

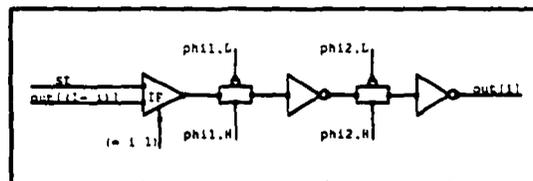


Figure 2: A graphical form of the Lisp if conditional.

Each device definition is given by a single drawing as in Figure 1. The symbol for the device being defined is placed at the top of the page and corresponds to the procedure head. The interconnected component devices are placed below it and form the body of the procedure. The device symbol contains named connection points for the input/output signals of the device: These

are the signal parameters of the device. Pins are used to mark the connection points in the device symbol, and the device definition provides the parameter name for each pin. General parameters, both mandatory and optional, are simply listed next to the device symbol.

The device symbols appearing in the procedure body each corresponds to a procedure call. The names of the wires connecting these symbols via connection points are passed as the corresponding actual parameters to these procedure calls. Device instances may specify optional parameters that do not appear in the device definition, in which case they are ignored. This allows the designer to attach arbitrary information to a device instance which may or may not be used in the current context. For example, the designer can include layout information which is ignored by a definition that generates a netlist.

Signal wires are represented as simple lines in the drawing and are named by closely placed text. The signal wires appearing in the procedure body must either be a formal parameter, i.e. an input or output signal, or be declared locally. Defining signals locally allows irrelevant information to be hidden from the surrounding context. Unnamed wires are automatically declared as a local signals and assigned uniquely derived names. Different wires with the same name are considered to be the same wire, and naming a wire more than once creates aliases for that signal.

Lisp expressions may be used anywhere in a drawing and are included in the device procedure as they appear in the drawing. The most commonly used Lisp expressions are conditionals, loops and arithmetic operations; however, any Lisp function can be used, including those defined by the user. General Lisp functions can appear either as text or as graphics symbols that operate over variables and signal values. Conversely, circuit drawings may be used within Lisp expressions. This is done by enclosing the circuit drawing with a named rectangle and referencing this name in a Lisp expression. Figure 1 contains an example of this: The if conditional is used to generate one of two circuits depending on the size of the input values.

Branch points can occur in the hierarchy if there are alternative definitions of a device. Design alternatives are easily implemented in WireLisp by using a conditional statement in the device definition. For example, a global style variable could be used to choose among a number of different types of output. For simulation, a composite device may be implemented as a functional model while for a netlist, its details would be completely generated.

The result of executing a WireLisp program can be defined by the user. However, the user will typically use library procedures for a particular technology and a standard output representation of the circuit. The most common use of WireLisp produces a flat

representation of the circuit containing only primitive devices connected as specified in the WireLisp program. For example, at the University of Washington we use WireLisp both to design CMOS circuits in standard SIM or COSMOS format, and to perform board-level design. However, a WireLisp program might just as well build an in-core representation of the circuit or make entries directly into a design database like Berkeley OCT[8]. The flexibility of a procedurally based description allows the general WireLisp framework to be used for a wide variety of applications with only a small amount of additional work.

Signals

Simple signals in WireLisp correspond to wires in a circuit communicating values between devices. At higher levels of abstraction, it is convenient to group related signals together and refer to them collectively as a unit. Examples include busses, control signals and dual-rail values. WireLisp allows signals to be grouped together into complex signals called *cables*. A cable comprises an ordered set of signals which can be referenced collectively by the name of the cable, or individually via an *access mechanism*. Cables are created via a local declaration which declares the cable and component names. Alternatively, cables can be created dynamically by bundling together a set of already declared signals using the *bundle* operator. Cables are allowed to have hierarchical structure so that components may be either simple signals or cables.

WireLisp provides two different ways to access the components of structured signals: *records* and *busses*, corresponding to records and arrays in conventional languages. Records use named selectors to access elements in a cable while busses are indexed. WireLisp separates the physical structure of a cable from the access mechanism used to access its components. Thus a structured signal may be referenced in different ways in different devices. For example, one device may number the components of a bus from 0 to 15 while another numbers them 1 to 16.

Cables may be defined either textually or graphically as shown in Figure 3. Cables are accessed in a straightforward way so that any component in the structure tree can be referenced. For example, `sysBus.control` refers to the three signals R/W, STROBE, and ACK and `sysBus.ADDRESS[23]` refers to the high-order address bit.

When a cable is passed as a parameter to a device, an access structure must be defined in order to access its components. Devices only need to declare access to components used within the device, leaving the rest hidden. WireLisp dynamically checks to make sure that the structure used to access a cable matches the actual physical structure of that cable. In some cases a high-level device may connect two subdevices together with a cable without knowing how large the cable is. In this case, the definition of the size of the cable can be delayed until a subdevice is called and explicitly defines it. This delayed binding provides a convenient way to specify generic modules.

Cable declarations and expressions often contain long lists of similar names. WireLisp provides a simple mechanism, called the *iterator*, for describing such lists of names succinctly. Iterators are merely syntactic shorthand used to reduce the amount of repetitive information in a drawing. An iterator occurring in a name causes that name to be expanded into a list of names. An iterator has the form `n1:n2`, where `n1` and `n2` are maximal substrings of digits appearing in the string. For example, `data16:31H` contains the iterator `16:31` which expands to the list `data16H data17H data18H ... data31H`.

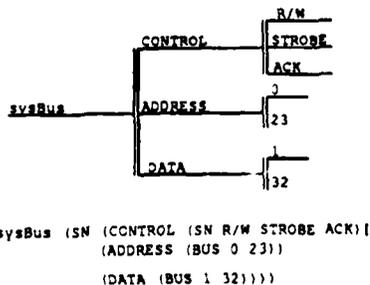


Figure 3: Graphical and textual declarations of structured signals.

Behavioral Information in WireLisp

While much of a digital system is best described structurally, there are parts of complex systems which have no obvious structure but whose input-output behavior is easy to describe. Control logic is an example of a module whose specification is better given behaviorally, for example as a finite state machine.

Behavioral modules are incorporated into WireLisp descriptions by invoking synthesis tools to generate an implementation in the form of a WireLisp procedure. By having the behavioral part of the system integrated with the structural description, the system can be designed, evaluated and implemented as a whole. For example, as an architectural description is being evaluated using simulation, the description of the behavioral modules can remain at the abstract level and simulated functionally. As the implementation is being detailed, the behavioral description can be mapped into a gate or transistor implementation.

Including behavioral modules in a WireLisp description is straightforward. A module is created with the appropriate input-output signals and an implementation that refers to the behavioral description external to WireLisp. For example, the behavior may be given as a set of logic equations, a finite state machine description in a high-level language, or a state diagram. WireLisp depends on other tools to translate this description into an appropriate WireLisp procedure. For example, a PLA description might be translated into a functional model for simulation or into a WireLisp description of a PLA circuit implementation. This mechanism of incorporating externally described modules allows WireLisp to take advantage of the wide range of synthesis tools available with very little additional effort.

WireLisp Implementation

There are a range of possible implementations of WireLisp depending on how closely the drawings and their interpretations are bound. We describe here the current implementation of WireLisp which uses a generic drawing program and an interpreter written in the T dialect of Lisp[11].

WireLisp drawings are made using xdp[7], a version of a generic drawing editor developed at Carnegie-Mellon University that runs on Sun and MicroVax workstations. xdp supports a variety of graphic objects including lines, text, circles, arcs and spline curves. xdp provides some help with circuit drawing with respect to connecting lines used as wires, but otherwise does not understand the semantics of circuit drawings. An analysis program which does understand these semantics converts WireLisp drawings into WireLisp procedures. The designer, however, only interacts with the circuit specification via the drawings.

xdp has a macro concept whereby a group of related objects can be bound into a single named object called a symbol. Symbols in WireLisp drawings correspond to devices, and connection points within devices are represented by xdp pins around the periphery of the device. Making a device definition involves defining a symbol for the device, copying in the symbols of the component devices from the drawings in which they are defined, and then connecting them using lines attached to their pins.

xdp allows the designer to move about conveniently in the hierarchy of a design by allowing devices to be "subedited". This causes the current drawing to be suspended and replaced by the drawing for the device being subedited. Editing of the suspended drawing is resumed when the subedit is completed. Subediting is allowed to any depth.

The execution of WireLisp procedures resulting from the analysis of the drawings is performed by an interpreter written in T Lisp. This is done by entering the interpreter and specifying the

name of the root device to be executed. As component device procedures are called, the corresponding drawing programs are automatically reanalyzed if they have changed and the device procedures autoloading. This automatic dependency analysis allows designs to be quickly modified and recompiled without any explicit information from the designer.

The current implementation executes WireLisp programs at the rate of about 200 devices/second for moderately complex CMOS circuits on a Sun-3/260 with 8MB of memory. This means that even a large circuit with 100,000 transistors can be processed in about 10 minutes.

Future plans for WireLisp include writing functions to make it easier to incorporate behavioral descriptions and investigating ways to incorporate physical design information for VLSI module generation. This latter information could be used to drive a chip assembler directly from a WireLisp description.

References

- [1] J.-L. Baer, M. Liem, L. McMurchie, R. Nottrott, L. Snyder, and W. Winder. A notation for describing multiple views of vlsi circuits. In *Proceedings of the 25th Design Automation Conference*. IEEE, June 1988.
- [2] R. Barth and B. Serlet. A structural representation for VLSI design. In *Proceedings of the 25th Design Automation Conference*. IEEE, June 1988.
- [3] R. Barth, B. Serlet, and P. Sindhu. Parameterized schematics. In *Proceedings of the 25th Design Automation Conference*. IEEE, June 1988.
- [4] J. Batali, N. Mayle, H. Shrobe, G. Sussman, and D. Weise. The DPL/Daedalus design environment. In *VLSI '81*, pages 183-192, 1981.
- [5] Derek Beatty, Karl Brace, Randal E. Bryant, Kyeongssoon Cho and Lawrence Huang. *User's Guide to COSMOS: a COMpiled Simulator for MOS circuits*. Carnegie-Mellon University, July 1988.
- [6] Edmund M. Clarke and Yulin Feng. Escher-A geometrical layout system for recursively defined circuits. *IEEE Transactions on Computer Aided Design*, 7(8):908-918, August 1988.
- [7] D. Giuse. DP: A drawing program. Technical report, Computer Science Dept., Carnegie-Mellon University, 1982.
- [8] P. S. Harrison, P. Moore, R. L. Spickelmier, and A. R. Newton. Data management and graphics editing in the Berkeley design environment. In *Proceedings of ICCAD*, pages 24-27. IEEE, November 1986.
- [9] C. Kingsley. *Earl: A language for integrated circuit design*. PhD thesis, California Institute of Technology, 1981.
- [10] Robert Mayo. Mocha chip: A system for the graphical design of VLSI module generators. In *Proceedings of the ICCAD*, pages 74-77. IEEE, November 1986.
- [11] Stephen Slade. *The T Programming Language: A Dialect of Lisp*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
- [12] Christopher J. Terman. *Simulation Tools for Digital LSI Design*. PhD thesis, Massachusetts Institute of Technology, October 1983.