DECEMBER 1992

WRL
Technical Note TN-32

Design Tools for BIPS-0

Jeremy Dion
Louis Monier

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Design Tools for BIPS-0

Jeremy Dion

Louis Monier

December, 1992

Abstract

The design of a 1 ns cycle-time microprocessor by a small team poses unique problems in computer-aided design. In this paper, we describe a complete set of tools which combines the obsession with performance of full custom design with the ease of use of semi-custom design.

Table of Contents

1. Introduction

Researchers at Digital Equipment Corporation’s Western Research Laboratory are currently designing BIPS-1, a single-chip ECL microprocessor with a 1 nanosecond cycle time. Using a sub-micron BICMOS technology, this processor will be about 15mm on a side and will contain 4 million active devices. A major part of this project is the design of a comprehensive set of computer-aided design tools. Early in the project we quickly determined that available commercial and academic design tools would be inadequate. None scale to the sizes and performance goals of a 1ns cycle time, 4M device processor, and none adequately exploit the advantages of BICMOS technology.

The tools we have designed attempt to bridge a gulf between semi-custom and full custom design. Most effort in commercial design tools concentrates on logic and layout synthesis for semi-custom gate arrays, standard cell arrays and programmable arrays. Designs done this way are highly modifiable, and can be done by small teams. But they suffer performance penalties due to the use of restricted forms of circuits and the inevitable compromises of synthesized layout. In a design style where wire delay must be accounted for in every block of logic and memory, these tools remove too much control from the designer and are fundamentally inadequate.

At the other extreme of the design spectrum are the full custom tools used by the ever smaller number of microprocessor design teams. These tools are built around the layout as a master representation, and the layout editor as a main design tool. Such designs allow complete control over all aspects of performance. Unfortunately, layout is a very rigid representation that is difficult to modify. It is also very vulnerable to design rule changes in the underlying technology. Custom design teams need dozens of layout specialists for years.

Our tools attempt to bridge this gap in order to allow a small team of a dozen designers to design a very high-performance microprocessor in a year. We have two main requirements. The first is that there be no compromise on performance. Our tools allow us to control all aspects of the processor’s circuits and layout. All tools work at the device level, not the gate level, and there is no fixed gate library of circuits from which designers or synthesis tools must choose. Of the billions of possible ECL gates, only those actually used by the design are synthesized, simulated and laid out. The second requirement is that the design be modifiable. Large complicated designs are created once, but modified forever. We needed a system permitting quick prototyping followed by incremental refinement. This would allow us to do early and continual floorplanning, global performance tuning, and to track changes in technology. Clearly, this design style requires extensive assistance from automatic tools.

We made several implementation choices that directed much of our efforts. These choices were mainly of what *not* to do, since our end result was a working chip, not a perfect design environment. Perhaps the most important decision was to represent circuits as programs. Programs are simultaneously the most complicated and most modifiable descriptions we knew of. We cast as much as possible of the design process as a problem in software development, and use all the standard programming tools to change and debug our design. We decided to develop many of our CAD tools as libraries which could be linked and run with the circuit design. This precluded the use of almost all commercial tools, for which source code is not available. A second decision was not to develop any new graphical editors. We relied on an existing drawing editor, **udraw**, for circuit schematics, and an existing layout editor **magic** [7] for examining layout and creating the small number of hand-drawn cells. We decided instead to focus on two core problems; the representation of the circuit netlist as a data structure in the program's memory, and a set of custom-quality placement and routing tools.

The full set of tools developed for this project also includes a switch-level bipolar timing verifier, written by Ramsey Haddad following ideas in [6] adapted for ECL design, a switch-level bipolar simulator, written by Russell Kao and other Stanford University students [2], electrical rules and noise margin checkers, written by Don Stark [1], and extensions to the **magic** layout editor [3]. These tools are beyond the scope of this paper.

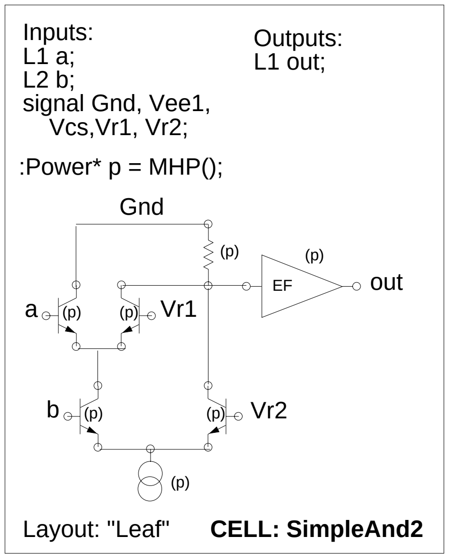
2. Design Capture

There is no single best way to describe circuits and logic. For analog circuits such as RAMs, schematic drawings of interconnected transistors are the most concise specification. For control logic, Boolean equations allow easiest debugging. For cell generators, such as a parameterized n-bit adder, a program is the most flexible representation. Rather than attempting to mix several different forms of circuit description, we chose to use their greatest common divisor, the program, and to translate schematics and Boolean equations into programs.

A circuit in our system is a C++ program. A procedure in this program is a cell generator. It can take arbitrary parameters, and returns the netlist for the requested cell. Many such generators take simple parameters, such as the amount of current drive to provide in the outputs, but some are quite complicated. For example, as we realized that the design would require many different adders, we implemented the adder generator to minimize the design effort. Our carry look-ahead adder generator takes as parameters the number of bits to add or subtract, and the number of bits in each look-ahead group, and returns an adder with the optimal-delay carry chain. Similarly, instead of having a library of OR-gates, we have an OR-gate generator, to which we pass the number of inputs, and a description of the outputs required. At this level, our form of description is quite like other hardware description languages.

Another library of C++ functions provides the syntax of Boolean equations, which are extensively used for control logic and for prototyping new blocks of logic. The library maps these into valid ECL gates (such as n-input OR gates) and makes use of two-level series gating, free inversion, and wired-OR. The result of calling a cell generator defined by Boolean equations is a netlist identical to that which would be obtained by explicitly interconnecting a collection of gates, flip-flops and multiplexors; we trust the equation mapper to make this translation on parts of the circuit where precise selection of the gates used is not critical.

Two cell schematics are shown in figure 1. Schematics are translated into a program which is the equivalent structural description of interconnected devices. This is done by analogy with programming. Our "source code" is a drawing produced by a conventional drawing editor which has no specialized knowledge of schematics, just as text editors have no knowledge of programs or text documents. We use **udraw**, but any conventional drawing editor would suffice. We then "compile" this drawing into C++ code using a drawing interpreter **drip** written by Ramsey Haddad, which interprets lines as wires, and names as labels of wires and devices. It uses only visible cues in the drawing to parse it into devices and wires, and can put arbitrary code, such as



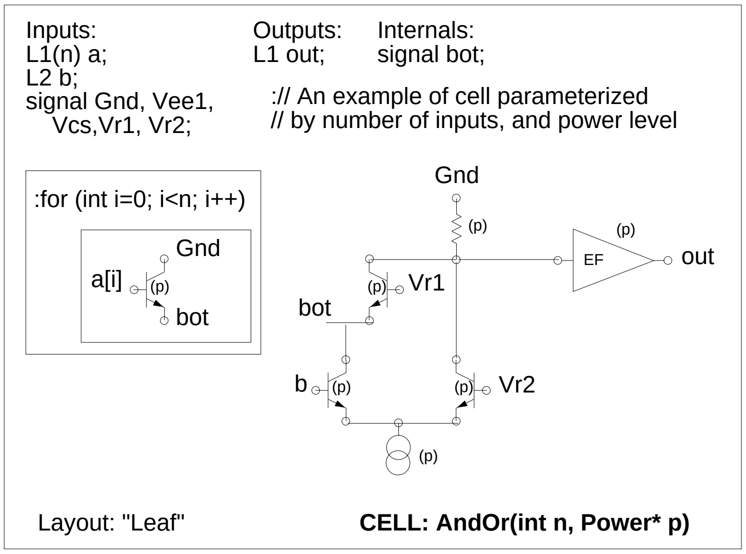


Figure 1: Cell schematics

loops and tests, into the generated program. The example on the right in figure 1 shows a cell generator containing a *for* loop. The output of **drip** is a program identical (except for verbosity) with one written by hand.

To generate a netlist for the entire chip, we translate all schematics into C++. The C++ source code is then compiled, as are the files containing Boolean equations and the hand-written cell generators, and all are linked with the CAD libraries. We also include a main program which calls the generator for the top-level cell of the chip.

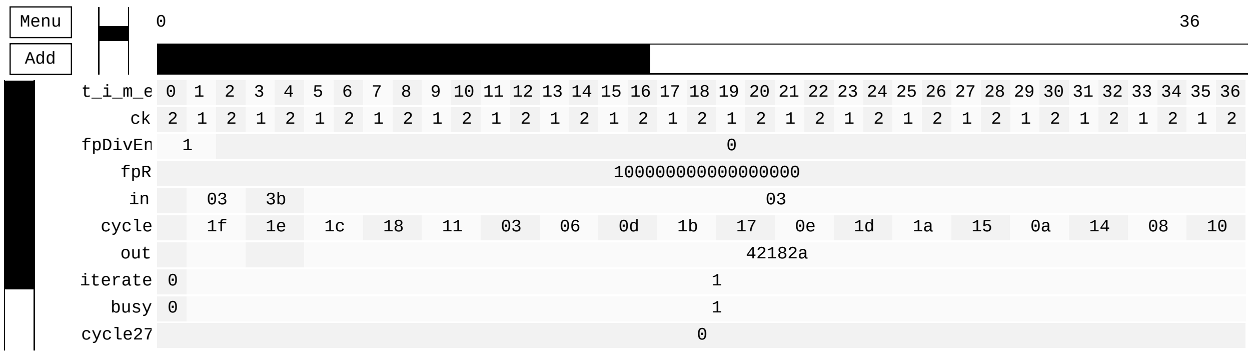
To describe a complete microprocessor having 4 million devices, we use 25K lines of C++; 15K lines of CAD libraries are linked with the design, resulting in a 10MB executable. The chip netlist is generated in a couple of minutes. An important point is that this netlist is hierarchical, with shared cells. This means that the netlist distinguishes between a cell and an instance of the cell. Cell sharing is provided by a programming convention in all cell generators. Each cell generator stores its computed netlist in a cell cache before returning it to the caller. On subsequent calls with identical parameters, the generator returns the cached netlist instead of creating a new copy. The advantages of a hierarchical netlist with shared cells are enormous, since all aspects of a cell which are common between its many instances are shared. For instance, there is exactly one RAM cell in the netlist, but thousands of instances of that cell. The RAM cell layout is generated only once, but is then copied to many places in the chip layout. Cell sharing speeds up layout generation by orders of magnitude.

3. Interface to Simulators

A netlist data structure in the virtual memory of a running program is a precise but useless representation of a circuit. The CAD libraries contain procedures to enumerate this netlist and produce input files for various simulators. We use **spice** for circuit simulation of analog circuits, and **bisim** [2] for switch-level simulation of digital circuits. Each simulator requires its own input file format, so there is a different enumerator of the circuit netlist for each simulator. Because both enumerations work from the same in-memory netlist, both are by definition consistent. The **spice** enumerator is simple, and produces a hierarchical **spice** deck with the same cell hierarchy as the original netlist. The only subtlety is in obeying the peculiar rules for legal **spice** names of cells and wires, a common problem when interfacing with commercial tools.

Bisim is a switch-level simulator for BICMOS circuits written in conjunction with Stanford University. It performs transistor-level simulation of ECL and CMOS circuits, and allows mixed-mode simulation with behavioral models. Cells in the in-memory netlist can carry functional models provided as C++ routines by the designer. In particular, a cell can both carry a functional model, and be defined as an interconnection of instances of subcells. As part of the enumeration which generates the **bisim** input files, the user selects the level of detail for simulation by specifying which of the functional models should be used, and which ignored in favor of lower-level models on subcells. The result of the enumeration is a flat **bisim** netlist at the level of detail requested by the user, and a file of C++ functional models which need to be linked with the simulator. Most of these functional models are at the gate level, because the library of generators for ECL gates automatically attaches a functional model to each gate. Large memory blocks carry hand-written functional models.

Results of simulations are examined with **krono**. This is a simple interactive graphical browser which can be used to examine the hierarchical netlist as well as log files produced by **bisim**. An example of a debugging trace for a floating-point divider is shown in figure 2. Here time is along the horizontal axis, and wire names from the original C++ source code are along the vertical axis.



```
  t_i_m_e 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
       ck 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
 fpDivEn   1                              0
     fpR                      100000000000000000
       in    03   3b                           03
    cycle    1f   1e   1c   18   11   03   06   0d   1b   17   0e   1d   1a   15   0a   14   08   10
      out                                   42182a
  iterate 0                               1
     busy 0                               1
  cycle27                                0
```

Figure 2: Circuit and simulation browser

4. Generation of Layout

Layout generation is best seen as just another enumeration of the in-memory netlist. In the same way that cells can carry functional models for simulation, cells can also carry **layout recipes** which define how their layout is to be generated. A layout recipe is simply the name of a C++ procedure, so this mechanism is general and extensible. As new styles of layout are developed, they are described as new layout recipes which can be attached to cells. This recipe is an integral part of the definition of the cell, and is specified by the designer just like the wires defining the cell’s interface. If the same circuit needs to be laid out in two different ways, it is described by a cell generator accepting the layout recipe as a parameter. Two different cells will result from calls supplying different layout recipe arguments. They will have identical netlists, but different layouts.

The layout process is a bottom-up, batch process. No interaction from the designer is required, since all the information needed to generate the layout is in the netlist and in the layout recipes. Layout happens in the same way for all cells. First, the subcells are laid out recursively. Then the layout recipe is followed to place the subcells and to route any connections which are not made by abutment. Finally, connectivity checks are made to detect shorts and opens.

The recursion ends at **leaf cells**, which do not depend on layout of the subcells. Leaf cells can either be hand-drawn or synthesized. Hand-drawn layouts are created with our layout editor. They are used for tricky analog cells such as pads and voltage references, or for memory cells, where the gain in density is compelling. Very few of the cells in our design are hand-drawn, but they constitute well over half of the 4 million devices in the final layout. Figure 3 shows the layout of a pair of register file cells.

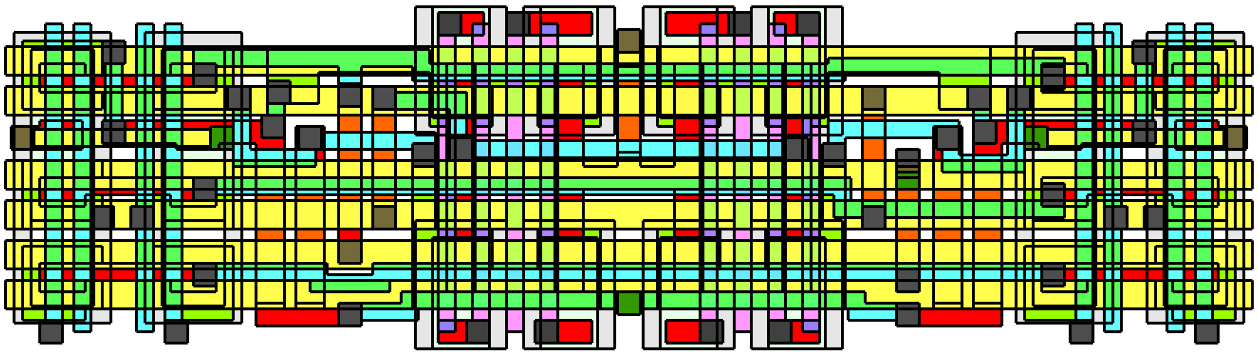


Figure 3: A typical hand-drawn cell

All other leaf cells are synthesized. Typically, these cells are at the gate level, containing up to fifty transistors. A general cell synthesizer produces finished layout given the netlist of devices, a set of templates for devices, and a vertical pitch. The placement uses no hints from the designer, but selects positions of the transistors and resistors which maximizes the use of polysilicon interconnect. The resulting placement is very close to the density of hand designs. In part this is due to the regularity of current trees in ECL logic and the similar sizes of bipolar devices. Routing of leaf cells is done first in polysilicon. The connections which cannot be made in the planar polysilicon routing are given contacts to first-level metal, and then the router is called again to finish the routing using the metal layers. Placement and routing for a typical leaf cell takes under a minute on a DECStation 5000/200.

This style of leaf cell synthesis may be contrasted with semi-custom design. In our system there is no cell library, and we cannot predict in advance which 300 of the enormous number of legal ECL gates will actually be used. The particular gate selection is determined by the parameters passed to the gate generator procedures during creation of the netlist. Each distinct gate is made exactly once - because of the sharing enforced by the cell cache during netlist generation - and is instantiated one or more times. During layout generation therefore, each unique ECL gate is placed and routed exactly once, and that layout is shared among the many instances. Figure 5 shows a flip-flop with a 10-input integrated multiplexor after placement and routing. Figure 4 shows the same cell without the wiring added by the router.

For non-leaf cells, the designer must choose a layout recipe which suits his purpose. This will depend on how the cell is to fit in the floorplan of the chip. Layout recipes vary over the

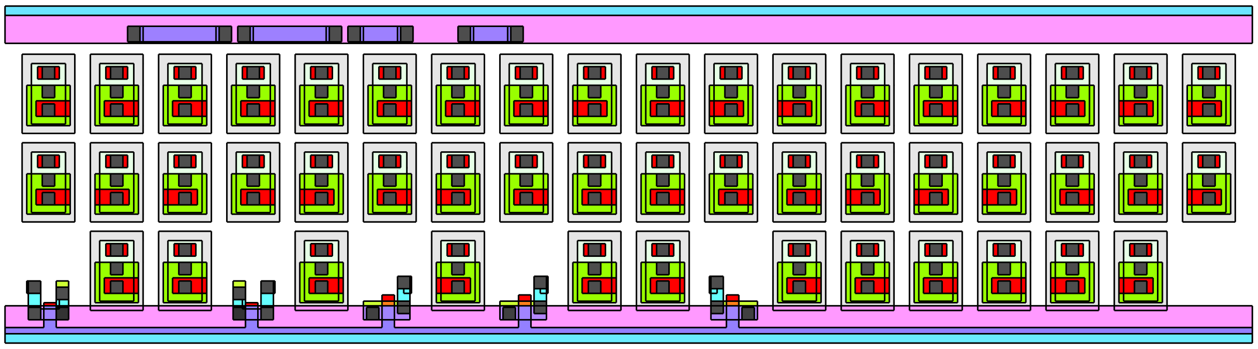


Figure 4: A leaf cell after placement

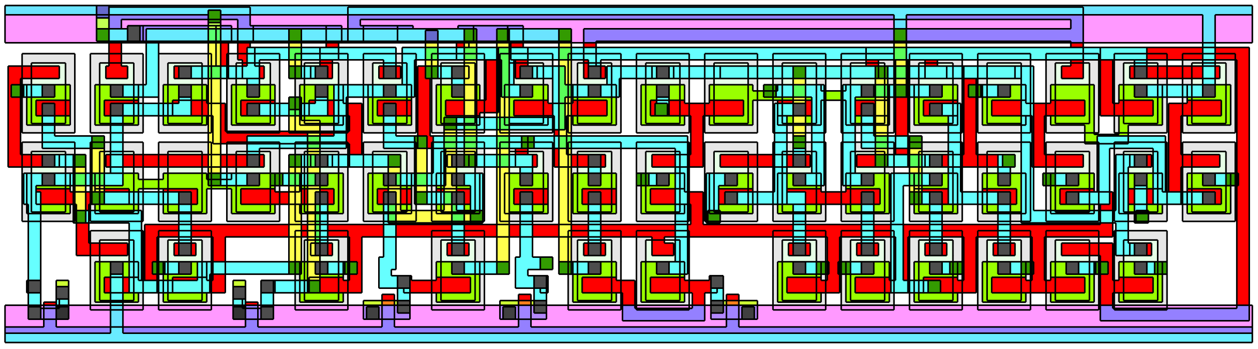


Figure 5: The same leaf cell after routing

spectrum of greater convenience to greater control. At the convenience end, a fully automatic placer based on conjugate gradient [5] is used to generate layout for blocks of control logic. At the control end, designers use a collection of recipes based on corner alignment of subcells to specify placement to the last micron (e.g. "place the lower left corner of the cache at the upper left corner of the data path").

A large fraction of the development effort was spent on a general router based on a hybrid maze/line search principles [4]. An important point about the router is that it is used in each cell of the design to complete connections not made by placement. In general therefore, the router is adding wires to cells on top of wires already routed in the subcells. Routing over the top of active logic is one of the characteristics of custom VLSI, and is largely responsible for its density. For this reason conventional channel routers - which route only over unobstructed rectangular channels - were unacceptable. The router reads design rules from the same file used by the layout editor and the design rule checker. It can generate routing with minimum dimensions and clearances from obstacles on all wiring layers simultaneously. There is only one router, and it is used repeatedly at all levels of the design, from routing polysilicon in the leaf cells to making millimeter-length connections at the chip level. Figure 6 shows the search structures used by the router to make a connection in a leaf cell.

As rectangles are generated by the router, they are attached to the appropriate nets in the netlist, just like all other rectangles of geometry. The geometry is thus maintained in an extracted form, distributed over the netlist, and the geometry for any net can be found quickly by an enumeration of the netlist. This permits incremental checks as each cell layout is finished, and early detection of shorts or opens. An electrical short of two nets is usually a sign of overlapping

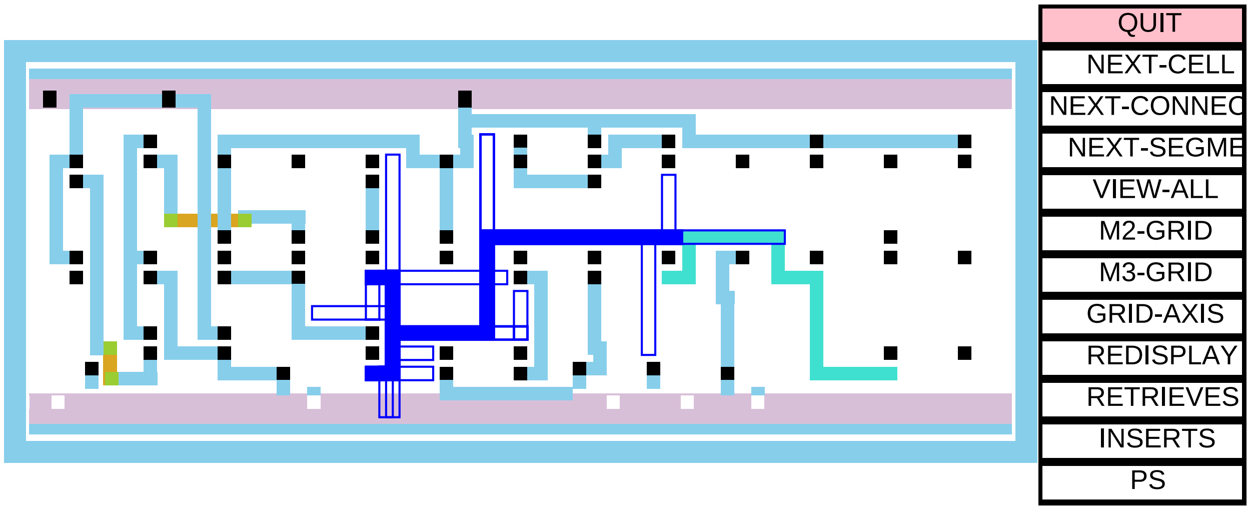


Figure 6: Router in action

subcells in an incorrect placement. Opens usually result from creating a routing problem which is too difficult. Both of these problems are solved by editing the layout recipe for the cell, and *never* by editing the layout directly. In this way we maintain the rule that all information needed to generate the complete is recorded on the netlist.

5. Conclusion

This set of tools has been under development for three years, and has been proven in the design of BIPS-0, an earlier bipolar processor. The layout generation of this complete 700,000-device circuit took 10 hours, which allowed one complete iteration per day.

On our current processor project, these tools allow a team of a dozen people to design, test, and layout a complex high-performance microprocessor in one year.

6. Acknowledgements

This work grew out of earlier work on CMOS design tools at XEROX PARC [8], and on printed circuit board tools at WRL [4]. The authors would like to thank Neil Wilhelm, Jean Vuillemin and Bertrand Serlet for convincing them that the solution to the hardest circuit design problems lies in ambitious computer-aided design tools.

References

[1] Don Stark.
Analysis of Power Supply Networks in VLSI Circuits.
WRL Research Report 91/3, Digital Equipment Western Research Laboratory, 1991.

[2] Russel Kao, Bob Alverson, Mark Horowitz and Don Stark.
Bisim: A Simulator for Custom ECL Circuits.
In *IEEE International Conference on Computer-Aided Design*, pages 62-65. Santa Clara, California, November, 1988.

[3] Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
1990 DECWRL/Livermore Magic Release.
WRL Research Report 90/7, Digital Equipment Western Research Laboratory, 1990.

[4] Jeremy Dion.
Fast Printed Circuit Board Routing.
WRL Research Report 88/1, Digital Equipment Western Research Laboratory, 1988.

[5] Jurgen M. Kleinhans, Georg Sigl, Frank M. Johannes, and Kurt J. Antreich.
GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization.
IEEE Transactions on Computer-aided Design 10(3):356-365, March, 1991.

[6] Jouppi, N. P.
Timing Analysis and Performance Improvement of MOS VLSI Designs.
IEEE Transactions on Computer Aided Design 6(4):650-665, 1987.

[7] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G.S. Taylor.
The Magic VLSI Layout System.
IEEE Design and Test of Computers 2(1):19-30, February, 1985.

[8] Louis Monier.
Layout Generation Through Parameterized Schematics.
In *7th Australian Microelectronics Conference*, pages 157-164. Sydney, May, 1988.

List of Figures