THOMSON REUTERS
ENDNOTE

[Vadde., 3(5): May, 2016] **ISSN 2349-4506**
Impact Factor: 2.265

Global Journal of Engineering Science and Research Management

APPRAISAL OF CAD ESTIMATIONS FOR CMOS DESIGN FOR TWO FOUR TRANSISTOR STATIC MEMORY CELLS

Vadde ArunaRai*, Vemulapalli Venkataramana, Merugu Chandramouli
*Department of Electrical and Electronics Engineering, Eritrea Institute of Technology, Mai nefhi, Asmara, Eritrea
Department of Computer Engineering, Eritrea Institute of Technology, Mai nefhi, Asmara, Eritrea
Department of Computer Engineering, Eritrea Institute of Technology, Mai nefhi, Asmara, Eritrea

DOI: 10.5281/zenodo.51765

KEYWORDS: CAD, Electric, circumstances and interfaces.

ABSTRACT
Electric is an investigation of CAD calculations. Since of its representational workplaces, it can keep an extensive variety of information much is starting now available. All the basic CAD system limits exist and are easily used. In this way center point extraction and cleaning should not to be executed after the data are there. Data and yield workplaces are moreover open, as are competent practical appear and changing. The CAD instrument programming build needs to create only the internal circle of new code, and it can be adequately associated with the Electric structure. This paper structure's competent blend allows the extension of instruments, circumstances, basic systems, sensible interfaces, and significantly more.

INTRODUCTION
Electric was built as an alternative to traditional VLSI design systems that did not combine graphics, connectivity, and accurate geometry [1]. Many textual systems existed for VLSI design and there were also graphics systems with no notion of connectivity. Some systems attempted to merge graphics and connectivity, but they always abstracted the graphics somewhat. The goal of Electric was to combine these into one highly flexible system [2]. To implement connectivity, the Electric design system has an extendible database, built on a network structure [3]. Nodes of the network correspond to components in the circuit, and arcs of the network are connecting wires. In addition to this network information, there are geometric data associated with every component and wire so that correct layout can be represented. Since the database is extendible, additional structures can be stored to describe behavior, power consumption, design rules, and so on [4, 5].

The most interesting aspect of Electric is its programmability: Users can specify layout relationships that will be continuously enforced during design. This is achieved with a constraint system that hierarchically propagates layout relationships through the design [6]. The constraint system strengthens the ability to do hierarchical design because different levels of the circuit always remain connected regardless of changes to the layout [7]. Although these constraints are all spatial in nature, there are also many interpretive programming systems available that allow more flexible programmability [8]. Electric accepts the fact that there is an unlimited number of environments for doing design, and the system integrates them in a uniform manner. A design environment is a single module describing primitive components and wires that can be composed into circuits [9]. Included in the module is all the environmental information such as graphic attributes and simulation behavior. To create a module, one simply expresses the environment in network form so that it can be manipulated as a collection of nodes and arcs [10].

In addition to providing for any design environment, Electric also allows an unlimited number of synthesis and analysis tools [11]. The database oversees their execution in the fashion of an operating system so that each tool can operate in turn and keep up with the activities of the others [12]. This flexibility allows tools to operate incrementally or in a batch style, and also keeps all activity consistent [13]. Change control is managed by the database so that any incorrect tool activity can be undone. In this context of tools as processes, even the user interface is a tool, separate from the database [14, 15].

ISSN 2349-4506
Impact Factor: 2.265
Global Journal of Engineering Science and Research Management

REPRESENTATION

In Electric, circuits are composed of nodes and arcs that represent components and connecting wires. For example, a transistor component connected to a contact component will be represented by two nodes and an arc (see Fig. 1). In addition, nodes have ports on them that are the sites of arc connections. There are two ports in this figure, one on each node at the connection site of the arc. Nodes, arcs, and ports are dynamically allocated objects in Electric, with pointers being used to link them together into networks. Each object has a basic set of attributes that hold vital information such as physical position and network neighbors. This set is extendible and can admit any attribute with any value. For example, a new attribute called "delay-time" could be created on the arc that holds a floating-point number. Also, a new attribute called "gate-contact" could be created on the transistor to hold a direct pointer to the contact node.

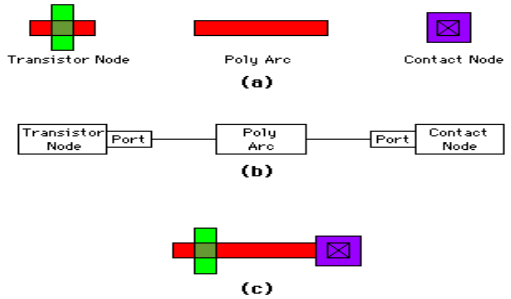

Figure 1 Electric's node and arc representation: (a) Objects (b) Representation (c) Actual layout.

To make the representation more flexible, Electric uses the distinction between instance and prototype to aggregate common information about classes of objects. For example, the transistor node in Fig. 1 is a node instance object, one of which exists for every different node placed in a circuit. There is also a transistor node prototype, which exists only once to describe all its instances. Node prototypes have port prototypes on them, which are templates for possible port instances in a circuit. These port prototypes define the physical locations on the node that can be used for connection, and also the allowable arcs that may be connected. To complete the instance-prototype representation, there are arc prototypes that describe defaults for every different kind of arc. Figure 2 illustrates how instances and prototypes relate. Notice that port prototypes may have multiple instances on the same node.

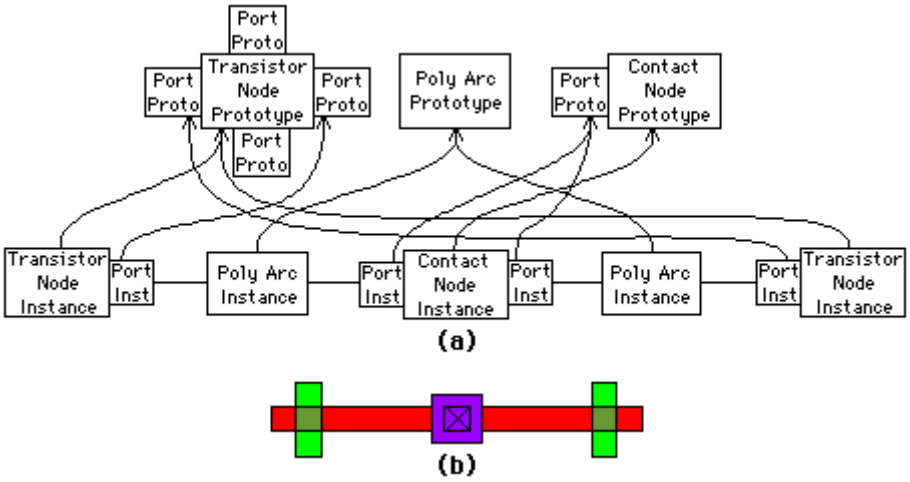

Figure 2 Instance and prototype representation: (a) Representation (b) Layout

A technology in Electric is an object that aggregates node, port, and arc prototypes to describe a particular design environment. For example, the nMOS technology object has three arc prototypes for metal, polysilicon, and diffusion wires. It has node prototypes for enhancement and depletion transistors in addition to a series of node prototypes for interwire connections (butting, buried, and cut contacts) and intrawire connections, or pins.

Instances of these prototypes can be assembled into an nMOS circuit .Since technologies are objects, they can also hold additional attributes. Initially, the added information includes design-rule tables and shape descriptions, but more could be handled; for example, one could add an attribute called "toxicity-level," which contains a paragraph of text describing the environmental ramifications of this semiconductor fabrication process.

The representation of hierarchy is done through a twist of the instance-prototype scheme. Rather than have a separate class of objects to represent cells of circuitry, the node-instance object can point to either primitive or complex prototypes. Whereas primitive node prototypes are defined in the environments as described, complex node prototypes are actually cells that contain other node, arc, and port instances (see Fig3). Port prototypes on complex node prototypes (that is, ports on cells) are actually references to ports on nodes inside of the cell (see the heavy line in the figure). This notion of "exporting" ports combines with the complex node-prototype scheme to provide a uniform representation of components in a hierarchical circuit. Although the prototype objects vary slightly in their information content, the node- and port-instance objects are the same regardless of whether they describe primitive or complex prototypes.

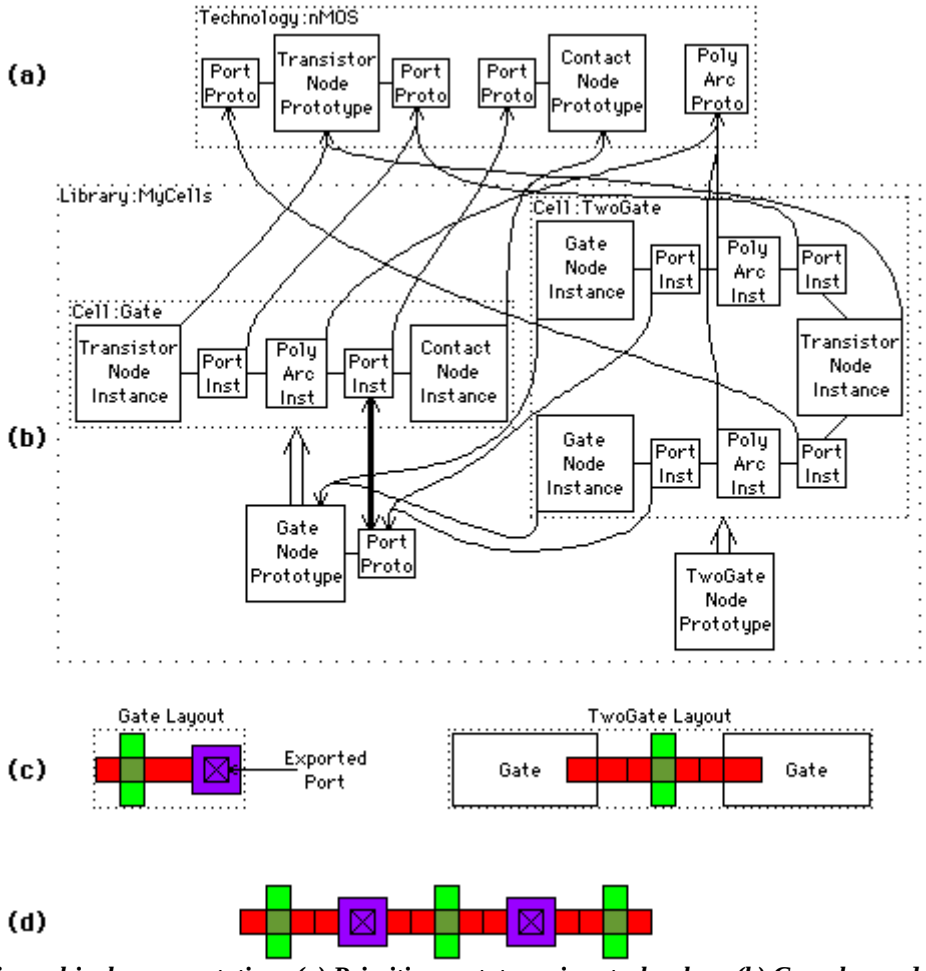

Figure 3 Hierarchical representation: (a) Primitive prototypes in a technology (b) Complex node prototypes, or cells (c) Cell layouts (d) Exploded layout.

Everything in Electric is an object with extendible attribute-value pairs. This section has already mentioned node, arc, and port instances; node, arc, and port prototypes; and technologies. There are also library objects that aggregate complex node prototypes, and tool objects that operate on the circuitry (see Fig. 4). Tool objects contain many attributes that are actually code routines for performing the essential synthesis and analysis functions.

ISSN 2349-4506
Impact Factor: 2.265

Global Journal of Engineering Science and Research Management

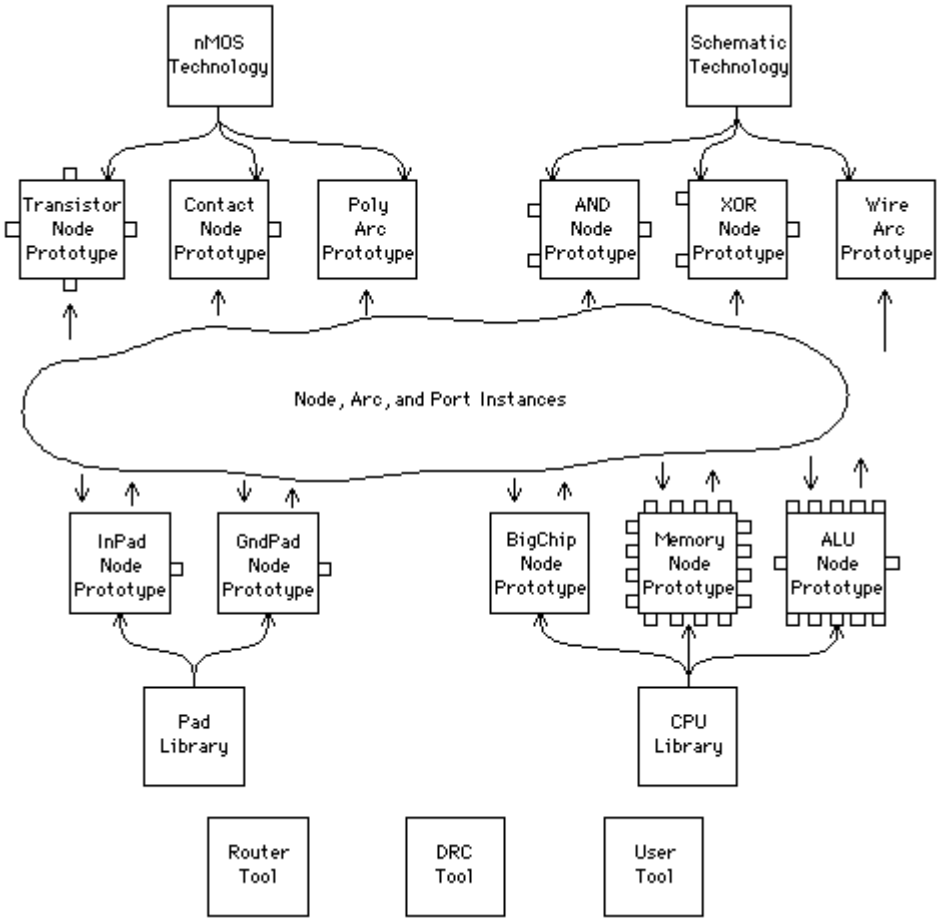

Figure 4 All objects of the Electric database.

PROGRAMMABILITY

The most interesting aspect of Electric is its powerful facility for programming designs. Besides providing built-in interpreters for LISP, TCL, and Mathematical, Electric has a few constraint systems available in the database. One constraint system is based on linear inequalities of wire lengths, as Chapter 8 described in detail. The primary constraint system, unique to Electric, works both within the cell and incrementally, propagating changes upward in the hierarchy. This hierarchical-layout constraint system enables designers to work top-down in a graphical editor.

Although the hierarchical-layout constraint system was initially an integral part of the database, the need for multiple constraint solvers fostered a more modular scheme. Each constraint solver provides interface routines for the possible database changes: creation, deletion, and modification of nodes, ports, arcs, and cells. Additional interface routines must exist for initialization and special control directives, and to inform the constraint system that a batch of changes has been described and can be solved. With such a scheme, any constraint system can be incorporated in the database.

HIERARCHICAL-LAYOUT CONSTRAINTS

Hierarchical-layout constraints are always placed on wires in Electric. They set properties so that a change to a component on either end of the wire affects the component on the other end. In the absence of constraints, wires rotate and stretch as necessary to connect their two components whenever either moves. Thus the essential premise of Electric is that wires always remain properly connected through all changes to the database.

The first hierarchical-layout constraint is rigidity, which, when applied to a wire, causes the length to remain constant and fixes the orientation with respect to both components. If either component connected to a rigid wire moves, then both the wire and the other component move similarly. If a component rotates, then the wire rotates and the component on the other end spins about the end of the wire. It can be seen that a rigid wire essentially creates a single object out of the two components that it connects (see Fig. 5). To allow for unusual constraint situations, rigidity may be set or unset temporarily, which means that the property overrides the current constraint for the next change only.

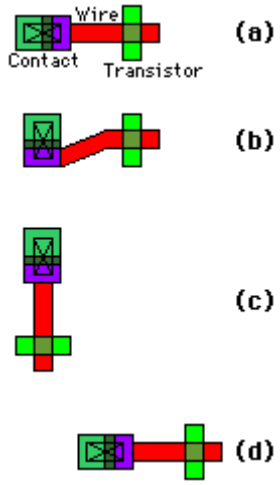

Figure 5 The effect of rigid wires: (a) Original (b) Contact component rotated, unconstrained wire (c) Contact component rotated, rigid wire (d) Contact component moved right, rigid wire.

When a wire is not rigid, there are two other constraints that may apply: fixed-angle and slid able. The fixed-angle constraint forces a wire to remain at its current angle. If one component on a horizontal fixed-angle wire moves up, then the other will also. If that component moves to the left, however, the wire will simply stretch without affecting the other component. The rotation or mirroring of a component does not affect a fixed-angle wire unless the wire is attached off-center, in which case a slight translation occurs (see Fig. 6).

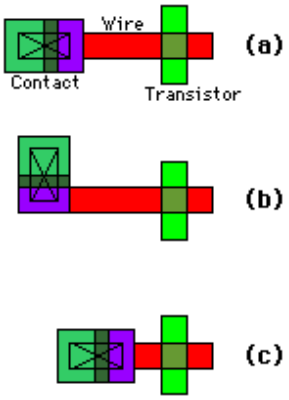

Figure 6 The effect of fixed-angle wires: (a) Original (b) Contact component rotated, fixed-angle wire (c) Contact component moved right, fixed-angle wire.

The other constraint that can be applied to nonrigid wires is the slidable factor, which affects components with nonpoint connection sites. When a connection has a positive area, the wire connecting to it can slide within that area. This means that small changes to the component may not affect the wire at all because the connection area still properly contains the end of the wire. The slidable constraint allows the wire to make these independent movements, whereas without this constraint both component and wire must move together. In Fig. 11.7, the 6 × 4 contact component has a port area that begins 1 unit in from its edge. The wire is thus connected in the middle

of a 2-wide port and can slide by 1 in either direction. The figure also illustrates an AND gate the port of which is the entire left side (a 0 × 5 port). The wires remain connected at the same place because they are not slidable.

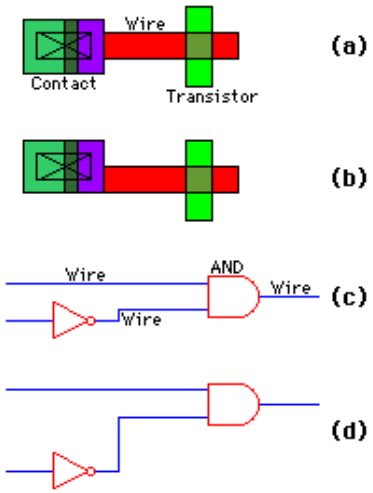

Figure 7 The effect of slid able wires: (a) Original (contact has connection area that is inset 1 from the top and bottom) (b) Contact component moved up by 1, slid able wire. (c) Original (d) AND moved up, fixed-angle non slid able wires.

Most design environments allow slidability so that the wires can adjust to detailed changes of connection configuration. An example of the need to turn off this constraint is in schematic gates such as AND and OR, in which the inputs are all attached to one large connection on the side. If slidability is allowed, then small motions of the gate component will bunch the inputs together.

The most powerful of the hierarchical-layout constraints is one that relates hierarchical levels of a layout. All exported connection sites in a cell definition are bound to their actual connections on instances of the cell, higher up in the hierarchy. This means that, if a component in a cell has an exported connection and the component moves, then that connection moves and any wires attached on instances of the cell also move (see Fig. 8). This is a natural extension of the Electric philosophy that everything must remain connected after a change.

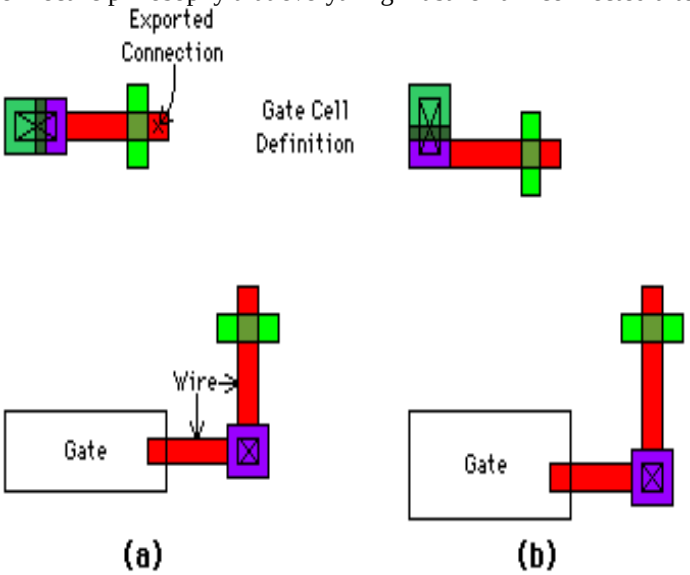

Figure 8 Propagation of hierarchical-layout constraints: (a) Original (b) Contact component in gate cell rotated all three wires fixed-angle.

ISSN 2349-4506
Impact Factor: 2.265

The order of constraint execution is critical to ensuring a proper solution and correct layout. Electric solves all constraints inside a cell before moving up the hierarchy to other cells. When the hierarchy is more complex than a simple tree organization, care must be taken to ensure proper traversal so that lower-level cells are solved before the higher-level ones. For example, if cell C in Fig. 9 is changed, then cell B must be reevaluated before cell A, even though both contain instances of cell C

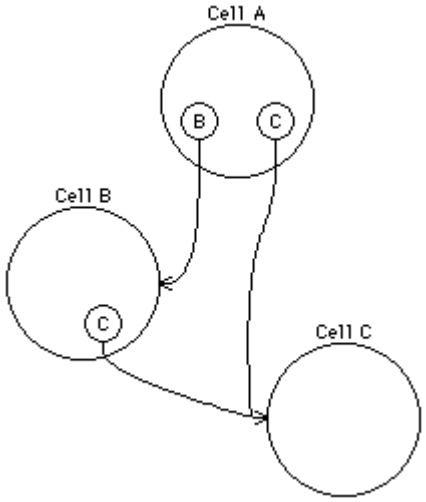

Figure 9 Non tree hierarchy must be traversed carefully.

Within a cell, the constraints are solved in two passes: first the rigid wires and then the nonrigid wires. This gives rigidity the priority it needs to work correctly. A time-stamp mechanism prevents constraint loops from running amok, by detecting reapplication of constraints and forcing a quiescing action. If a reapplied constraint is consistent with the layout, then no change is necessary and the loop terminates. Otherwise, there is an over constrained situation and Electric jogs the wire, replacing it with three that properly connect. This scheme for propagation of constraints proceeds quickly and effectively in the domain of VLSI design.

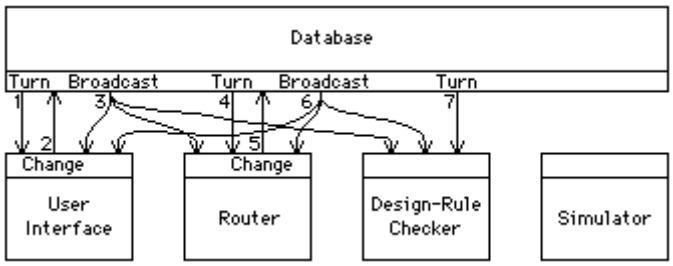

Figure 10 Sample Electric control sequence.

INTERPRETIVE LANGUAGES

When more powerful programmability is needed, the user can write a piece of LISP, TCL, or Mathematica to generate any layout. There was even a Prolog interpreter many years ago, but it fell into disuse. These language systems do not function incrementally, reacting to changes as they occur, but rather act like a standard imperative language that manipulates the database when executed. Interpretive code can create circuitry by invoking the Electric database to define new cells and place objects in cells. It is also possible for interpretive code to invoke the constraint system by setting wire properties and changing component locations. These capabilities are identical to those available in C programs linked with Electric, except that these languages run interpretively and thus provides a better development environment.

Simulation

Electric currently has many simulator interfaces that generate netlists for various simulators. ESIM, RSIM, and RNL simulate MOS at a switch level as does MOSSIM [Bryant]. CADAT [HHB] is a functional simulator and MARS is an experimental hierarchical simulator that can handle arbitrary behavioral specification.

Besides simulator interfaces, Electric has a gate-level simulator built-in. This 12-state simulator can accept input values and display simulation results in a separate waveform window or on the actual circuit.

It is interesting to examine the SPICE simulation environment in Electric, which can graphically specify all parameters. Connecting meter and source components to a circuit, and parameter zing them appropriately, allows a complete input and output to be determined (see Fig. 11). When the simulation runs, the output values are used to generate a waveform plot with artwork primitives. Since this plot is a cell like any other, it can be saved as documentation, zoomed into for closer examination, and even altered with the normal editing commands to fudge the data.

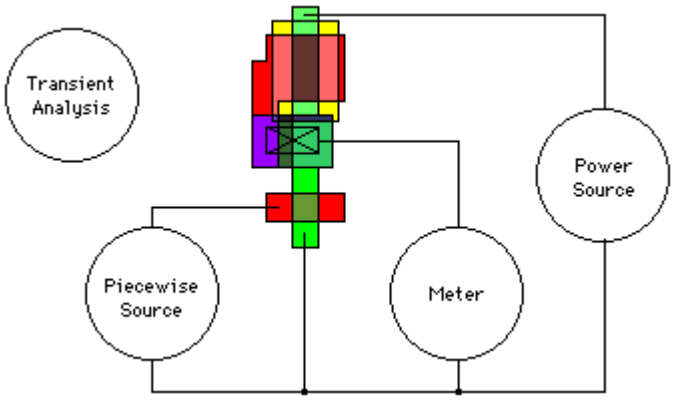

Figure 11 Graphical specification of SPICE simulation. Source components indicate inputs and meter components produce output plots. The transient-analysis component indicates the nature of the simulation.

Routing

The routers in Electric can handle many functions. A maze router is available to make point-to-point connections and a river router connects multiple sets of points on opposite sides of a rectangular area.

When array-based layout is being done, implicit connections exist where two cells abut. These connections must be explicit for Electric to maintain its topological view of the circuit. To help do this, two stitching functions exist for the explicit connection of implicitly joined circuitry. The auto-stitcher looks for connection sites that adjoin or overlap geometrically and connects them explicitly. The mimic-stitcher watches user activity as wires are placed, and creates other wires in similar situations throughout the layout. These two stitchers work incrementally, adding wires as they are needed.

Compaction

Electric has a one-dimensional compacter that can work in a batch style or can function incrementally. When turned on, it adjusts the layout to its minimal design-rule separation. If the compacter remains on, it will close up the circuit when objects are deleted and will spread open the layout to make room for new components.

PLA Generation

There are two Programmable-Logic-Array generators in Electric: one specifically tailored for nMOS layout and another that is a general-purpose tiling engine. Both work from personality tables that describe an array of elements. The nMOS PLA generator produces all the surrounding circuitry, including drivers and power connections, so that a complete functioning module is available. The general-purpose PLA generator produces

only those arrays of logic without any interconnect. However, it works from a user-specified library of array elements and orientation examples, so this library can include any peripheral circuitry.

Silicon Compiler
The QUISC silicon compiler accepts a structural VHDL description of a circuit and a standard cell library. It compiles the VHDL to a net list, places the standard cells, and routes them to create layout for the VHDL.

The VHDL compiler is particularly interesting because it is able to produce a number of different netlist formats, and it can therefore drive the simulator as well as the silicon compiler. Also, the VHDL compiler can function in reverse, generating VHDL from a schematic. Therefore, users who wish to simulate or build layout from a schematic can do so easily.

Compensation
The compensation system adjusts geometry to account for process-specific manufacturing effects. It can understand actual process features and adjusts geometry properly, regardless of the direction of compensation. The system even understands interior versus exterior geometry and knows to move edges differently if they are defining a hole.

More Tools
There is a network-maintainer tool that merely propagates node information throughout the circuit whenever a change is made. It keeps lists of net names and can highlight any connected path.

Many other tools could fit easily in Electric: Analysis tools such as power estimation, timing verification, and test-vector generation would all be useful. More synthesis tools such as floor-planners and better routers will someday be added. Not only is the list of different tools unbounded, but the number of tools in a given class can grow to accommodate experimental versions of any CAD algorithm. Electric's integration scheme is designed to be able to incorporate modularly any VLSI CAD tool.

Designing a Chip
To illustrate the use of Electric, this section will describe the construction of a static-memory chip. This chip makes use of a novel four-transistor bit of static memory. The fundamental memory cell is logically designed using the schematics environment (see Fig. 12). An equivalent CMOS layout is then produced without concern for compact spacing (see Fig. 13). For proper layout efficiency, alternate bits of memory are different, so two bits define the leaf cell of the design. These bits can be compacted by the one-dimensional compacter (see Fig. 14) and then compacted further by rearranging components and recomposing (see Fig. 15).

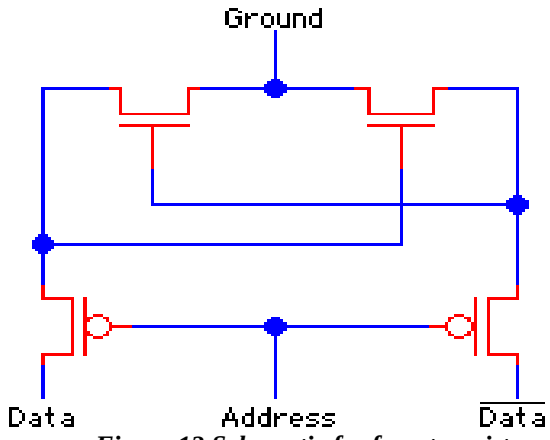

Figure 12 Schematic for four-transistor static-memory cell.

ISSN 2349-4506
Impact Factor: 2.265

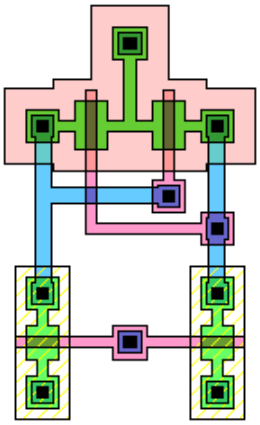

Figure 13 CMOS layout for four-transistor static-memory cell.

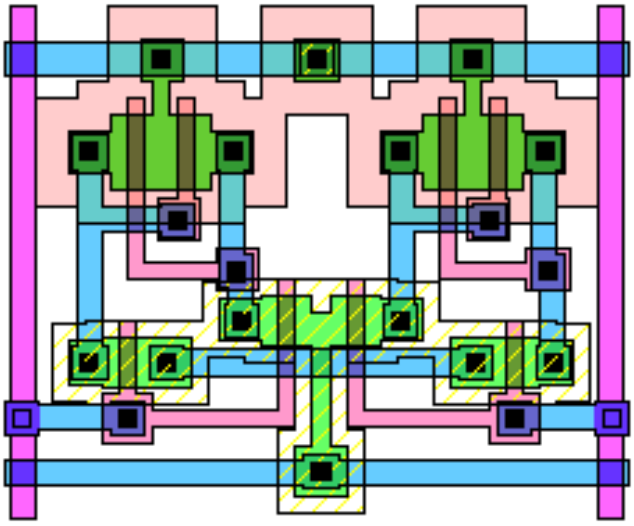

Figure 14 CMOS layout for two four-transistor static-memory cells.

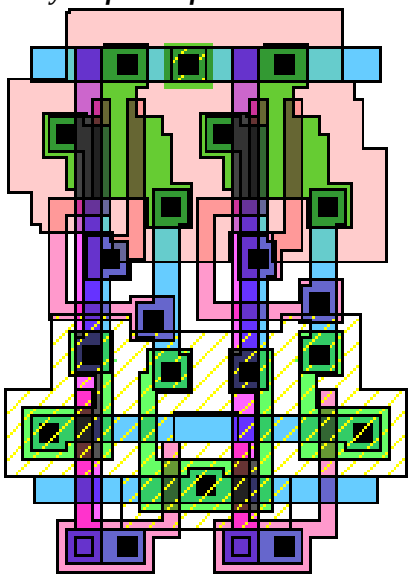

Figure 15 Compacted CMOS layout for two four-transistor static-memory cells.

ISSN 2349-4506
Impact Factor: 2.265

To create a 128 × 32-bit array of memory, six levels of hierarchy are employed to build a 4 × 2 array, a 4 × 4 array, a 16 × 8 array, a 64 × 8 array, a 64 × 32 array, and a 128 × 32 array (see Fig. 11.16). Note that each level of hierarchy actually connects its subcells with little stitches so that the overall connectivity is maintained. These stitches are automatically created by the router. Also, each level of hierarchy must export all unstitched ports to the next level. This is done automatically by the array-based layout commands.

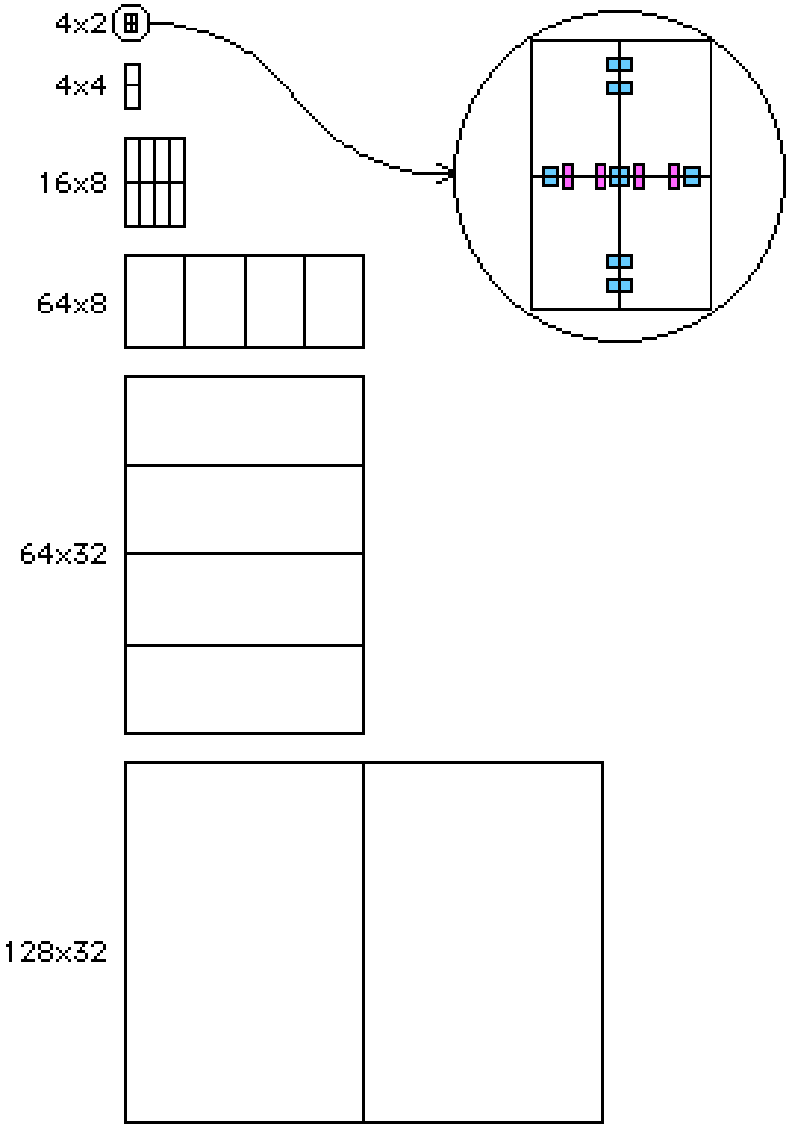

Figure 16 Hierarchical organization for 128 × 32 array of static-memory cells.

Once the basic memory array is created, driving circuitry must be placed on the edges. A word (32-bit) driver for a single word is designed to pitch match the memory (see Fig.17) and the driver is arrayed using hierarchy. The block of 128 drivers attaches to the bottom of the memory array. Similar drivers and decoders are built on the sides. The overall floor-plan, including pads, is shown in Fig. 11.18. This layout contains 32,650 transistors, described with 110 cells

ISSN 2349-4506
Impact Factor: 2.265

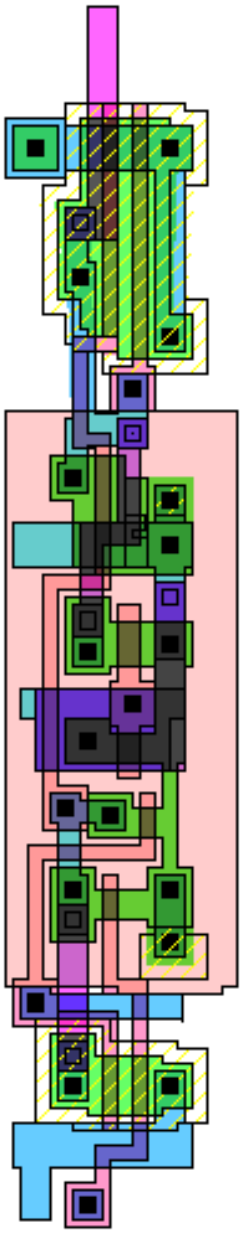

Figure 17 Word line driver for static-memory array.

As an indication of the complexity of Electric, this circuit consumes 1.4 megabytes of disk space. Although this is not small, it does contain more information than typical design databases. It can be read in 50 seconds and written in 30 seconds (on a SUN/3 workstation). Netlist generation takes only 15 seconds (these figures were compiled in 1986, when computers were significantly slower).

ISSN 2349-4506
Impact Factor: 2.265

Global Journal of Engineering Science and Research Management

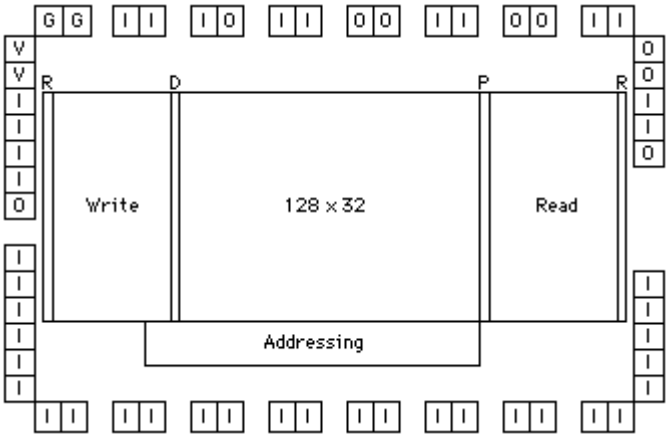

Figure 18 Floor-plan of static-memory chip.

CONCLUSION

Electric is a workbench for the exploration of CAD algorithms. Because of its representational facilities, it can keep all kinds of information much is already available. All the essential CAD system functions exist and are easily used. Thus node extraction and pasteurization do not need to be implemented since the data are there. Input and output facilities are also available, as are powerful graphic display and editing. The CAD tool programmer needs to write only the inner loop of new code, and it can be easily plugged into the Electric framework. The system's powerful integration allows the addition of tools, environments, constraint systems, graphic interfaces, and much more.

REFERENCES

1. Baker, Clark M. and Terman, Chris, "Tools for Verifying Integrated Circuit Designs," *Lambda*, 1:3, 22-30, 4th Quarter 1980.
2. Batali, J. and Hartheimer, A., "The Design Procedure Language Manual," AI Memo 598, Massachusetts Institute of Technology, 1980.
3. Bryant, Randal Everitt, *A Switch-Level Simulation Model for Integrated Logic Circuits*, PhD dissertation, Massachusetts Institute of Technology Laboratory for Computer Science, report MIT/LCS/TR-259, March 1981.
4. Calma Corporation, *GDS II Stream Format*, July 1984.
5. Electronic Design Interface Format Steering Committee, *EDIF-Electronic Design Interchange Format Version 1 0 0*, Texas Instruments, Dallas, Texas, 1985.
6. Griswold, Thomas W., "Portable Design Rules for Bulk CMOS," *VLSI Design*, III:5, 62-67, September/October 1982.
7. Guttman, Antonin, "R-Trees: A Dynamic Index Structure for Spatial Searching," *ACM SIGMOD*, 14:2, 47-57, June 1984.
8. Harrison, D. S.; Moore, P.; Spickelmier, R. L.; and Newton, A. R., "Data Management and Graphics Editing in the Berkeley Design Environment," 1986 IEEE International Conference on Computer-Aided Design, 24-27, 1986.
9. HHB, *CADAT User's Manual*, Revision 5.0, HHB-Softron, Mahwah, New Jersey, June 1985.
10. Hon, Robert W. and Sequin, Carlo H., "A Guide to LSI Implementation," 2nd Edition, Xerox Palo Alto Research Center technical memo SSL-79-7, January 1980.
11. Johnson, Stephen C., "Hierarchical Design Validation Based on Rectangles," Proceedings MIT Conference on Advanced Research in VLSI (Penfield, ed), 97-100, January 1982.
12. Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
13. Kingsley, C., *Earl: An Integrated Circuit Design Language*, Masters Thesis, California Institute of Technology, June 1982.

ISSN 2349-4506
Impact Factor: 2.265

14. Kostiuk, A. R., "QUISC: An Interactive Silicon Compiler," M.Sc. Thesis, Queen's University at Kingston, Department of Electrical Engineering, 1987.
15. Kroeker, Wallace I., *Integrated Environmental Support for Silicon Compilation of Digital Filters*, Masters Thesis, University of Calgary Computer Science Department, March 1986.