CHIP ASSEMBLY TOOLS

Stephen Trimberger and Chris Kingsley
California Institute of Technology
Pasadena, California 91125

TM #5005

TECHNICAL MEMORANDUM

Department of

Computer Science

CALIFORNIA INSTITUTE OF TECHNOLOGY · THE TRUTH SHALL MAKE YOU FREE ·

CHIP ASSEMBLY TOOLS

Stephen Trimberger and Chris Kingsley
California Institute of Technology
Pasadena, California 91125

TM #5005

CHIP ASSEMBLY TOOLS

Stephen Trimberger and Chris Kingsley

California Institute of Technology, Computer Science Department
Pasadena, California, USA

ABSTRACT

In large-scale integrated circuit design, chip assembly is more difficult, more time consuming, and more error prone than the design of the low-level cells. Assembly errors tend to persist until late in the design cycle requiring extensive rework Unfortunately, the tools traditionally provided for custom integrated circuit design address the problems of cell design well, but do not properly address the problems of chip assembly. A great deal of emphasis at Caltech has been placed on tools that do address chip assembly. This paper reports on some of these tools.

INTRODUCTION

Custom integrated circuit layout can be split into two major parts: *cell design*, the development of the low level cells making up the "leaves" of the hierarchical tree; and *composition*, assembly of those cells into larger cells and systems [Rowson 1980]. Although composition is at least as time consuming as cell layout [Wedig 1981], current design systems address low-level cell design, and rely on physical positioning and orientation to compose chips. These operations are not necessarily helpful in composition, since the kinds of operations required to perform the two kinds of operations are different. Composition systems do not preclude custom design, since they can be built to give the designer control of the connection mechanism.

COMPOSITION TOOL OVERVIEW

To avoid errors in composition, a system must have primitive operations which attach connectors on instances as well as position them. Proper connection operations free the designer from a whole class of mistakes, eliminating much of the post-design checking, like geometrical design rule checking. The primitive operations used in Caltech's composition tools are abutment, routing and stretching. If the connectors on the instances to be connected are perfectly aligned, the instances may be *abutted* to make the connection (figure 1). Unfortunately, this is rarely the case. When the connectors do not match, connection may be made by creating a piece of *routing* wiring between the cells

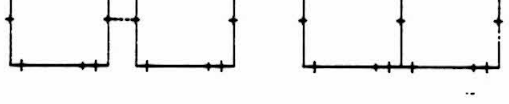

Figure 1. Connection by Abutment.

(figure 2). Composition in industry typically uses general routing to connect instances -- a wasteful practice. Composition tools at Caltech generally restrict the complexity of the routing giving the global routing task to the user explicitly.

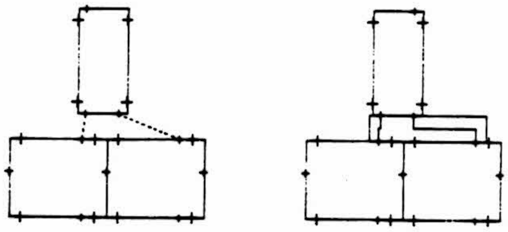

Figure 2. Connection by Routing.

In many cases, global chip area can be reduced and global chip performance increased by *stretching* the cells to connect by abutment instead of placing the cells and routing between them (figure 3) [Johannsen 1981]. Stretching does not require any wiring channels, but does require that the positions of components in the instances be changed. This is not possible if the cells are defined as layout, but is possible if the cells are defined *algorithmically* or *symbolically*. Both methods of cell definition are used at Caltech for defining cells to be stretched in composition tools.

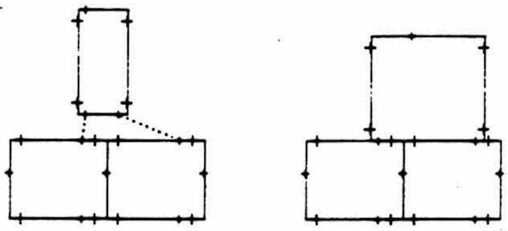

Figure 3. Connection by Stretching.

In an *algorithmic definition*, a cell is designed as a program rather than as wires and boxes. When executed, the program generates the required wires and boxes. Cells can be made to change shape to fit more easily into the chip being designed, and they can modify driver sizes and interconnection wire width to improve the performance of the circuit.

In *symbolic definition*, the circuit features are defined in an abstract manner with components such as transistors and contacts along with wires connecting them. In a symbolic, or *Sticks* form, the components can be positioned anywhere and the wires change length to maintain the connection. This form can be used to ease cell design, since the absolute positions of components need not be specified. The cells can all be compacted into minimum area by a single *compaction* program, or *stretched* to match connectors with the environment. The symbolic form is easier to generate than the algorithmic form, but the kind of manipulations that can be performed on a cell is determined by the manipulating program, not the designer. The Sticks Standard [Trimberger 1980] is used at Caltech for all symbolic data.

Cells may also be defined as mask geometry. In geometry form, no stretching can be done, so connections must be performed by routing. CIF [Sproull 1980] is used at Caltech to represent geometry in cells. Extensions to CIF provide cell names and connector locations.

BRISTLE BLOCKS

Bristle Blocks [Johannsen 1981] is a composition tool that specializes in the construction of *datapath* chips. A datapath chip consists of data processing elements connected by and communicating across data busses (figure 4). Typical data processing elements include register files, arithmetic/logic units, and shifters. The chip is controlled by microcode which is decoded on-chip to drive the individual control lines of the processing elements. Bristle Blocks imposes this generic floorplan in return for ease in automating the layout. While not all designs can fit into the high-level datapath floorplan, those that can fit are implemented as efficiently as hand layout.

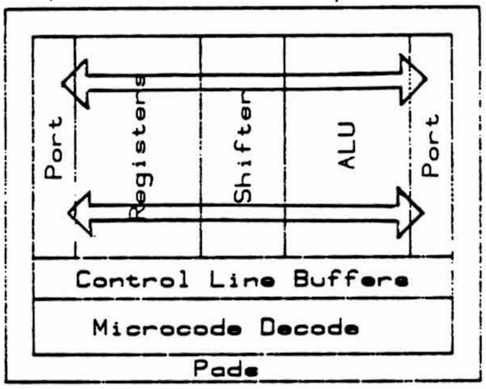

Figure 4. Datapath Floorplan Block Diagram.

The input to Bristle Blocks is a textual specification in two parts: the algorithmic definition of the cells, and a high level description of the chip. The high level description of a fairly large chip is approximately three pages long, and consists of calls to the cell programs with parameters appropriate to the desired chip. Typically, the parameters are behavioral information, such as the conditions required for the register cell to load information from a bus. Internally, Bristle Blocks generates positional information from the datapath floorplan and passes that information to the cell programs also.

Bristle Blocks executes the cell programs to create the datapath portion of the chip. Bristle Blocks datapath cells are different sizes, so the height of a bit slice is determined by the tallest cell needed. The cells are algorithmically defined with the means for stretching built into them, and they stretch to match connectors with other cells in the datapath. The datapath timing and control information is used to add control line buffers and an instruction decoder to drive the datapath. Finally, Bristle Blocks adds bonding pads and wiring to complete the chip and provides documentation about the locations of pads and signals.

Bristle Blocks has been operational since 1978, and has been used to lay out several chips. In addition, Bristle Blocks data paths have found their way into many other Caltech chips. Bristle Blocks designs have circuit densities comparable to hand design, and the datapath floorplan has been found to be a powerful implementation structure.

EARL

Earl is a low level cell design and composition system, based on an interpreted language. The primitive data types of the language include points and cells, and cell connection is a primitive operation. Unlike Bristle Blocks, Earl handles generalized layout; the floorplan of the design is totally in the hands of the designer.

The input to Earl is a set of leaf cell and composition definitions in the Earl language. Each cell has a list of connectors and constraints on connectors that define the interface to the cell, the only way the outside world may interact with the cell's instances. This interface specification is similar to modular programming techniques commonly used in software design. By explicitly declaring the interface, interactions with the cell can be controlled, localizing the behavior of the cell and simplifying the task of verifying correct usage.

Earl's composition functions allow a user to connect individual cells edge to edge, or connect a list of cells horizontally or vertically. The designer can optionally specify that wires are to be shared between two cells, causing a controlled overlap.

Earl performs composition by stretching, therefore all cells must be able to stretch between any two adjacent connectors. This use of stretchable cells is a two-dimensional generalization of the identical notion in Bristle Blocks. The composition must satisfy all the constraints specified explicitly by the user as well as those introduced by the interconnections. If the stretched connection is impossible or unsatisfactory to the designer, the designer must make a leaf cell to do the routing between them.

By restricting cells to be rectangular with all connectors lying on the boundary, the basically two-dimensional constraint problem is split into two one-dimensional problems. Each dimension is translated into a directed graph (figure 5). The nodes of the layout solution graph represent *equivalent* sets of connectors. A group of connectors constrained to move as a unit is represented by one node in the graph. The weighted, directed arcs represent minimum separation constraints between connectors resulting from constraints from the leaf cells. Notice that the graph can have cycles, which are used to allow limited stretching where this is desired. The graph is solved with an algorithm that is similar to the algorithms used in stick diagram compaction [Hsueh 1979].

Earl is currently being used to make class projects. Approximately thirty chips have been made with Earl, averaging about a thousand transistors in size. Larger projects are planned for Earl in the near future.

RIOT

Bristle Blocks and Earl are language based. The composition of instances is specified by textual invocation and

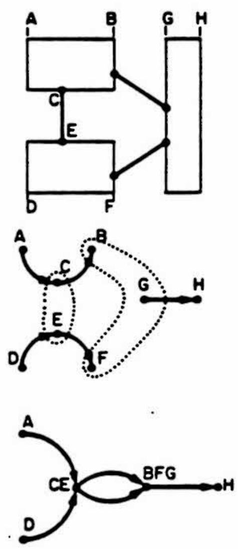

Figure 5. Solution Graph for Horizontal Direction.

connection. Although the language form is very powerful, the textual specification is tedious and error prone in general systems, since much of composition is a graphical positioning task. Highly automated language-based systems like Bristle Blocks allow designs to be expressed simply in a very abstract manner, but place severe restrictions on the floorplan of the resulting chip.

Riot [Trimberger 1982] is a simple interactive graphical composition tool which provides primitives necessary for composition. In Riot, the user may connect instances to one another by abutment, simple routing, or stretching. Riot cannot make new mask layout, except as a side-effect of a routing operation. Other tools exist which create the leaf cells which Riot manipulates. The Riot display organization is shown in figure 6.

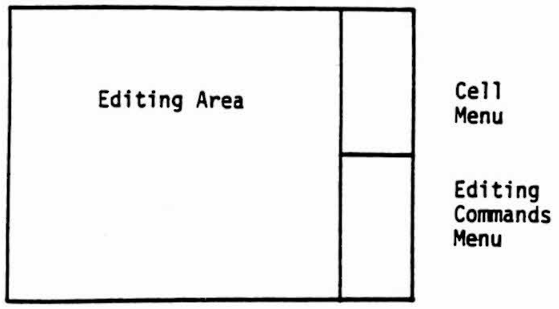

Figure 6. Riot Display Organization.

Riot can read leaf cells defined in CIF, Sticks, and its own composition format. Riot's output is in composition format which is translated to CIF for mask generation or to Sticks for simulation.

Riot has simple positioning and orientation commands similar to those found in traditional graphics systems. These commands are used in Riot to prepare instances for connection but are not used to make the actual connection of instances.

Connection of instances is done in two parts: first, the connections to be made are specified graphically, by pointing to pairs of connectors on the screen. Riot remembers the connections to be made, and makes all specified connections simultaneously when a *route*, *stretch* or *abut* command is given. Then the connections are made and the logical connection information is thrown out.

A *connection* is made by Riot by the placement of instances so that the connectors on the instances touch. Riot handles connection in the positional sense, not in the logical sense: a connection is the result of appropriate positioning. Therefore, once a connection is made, it can be easily (perhaps accidentally) destroyed. Riot does nothing to guarantee that connections will be maintained. This simplified handling of connection has limited the usefulness of Riot.

Riot's route is a simple multiple layer *river route*, in which no two signals cross. This limitation has not been severe, due to the interactive nature of the tool. Riot's stretching uses Rest [Mosteller 1981] to stretch one cell at a time to make a set of connections. Options on Riot's connection operations move instances to make minimum routing spacing, route to current positions, and control overlapping to share common power and ground lines in adjacent cells.

Riot has been used in the design of several custom chips. The interactive nature of the tool and the powerful composition primitives enable the designer to generate designs very quickly. Riot gives a simple way to express stretching of cells, and has thus improved the usefulness of our symbolic layout systems.

Riot's greatest drawback is with its geometrical view of connection. Because connections are forgotten, the user may accidentally destroy some connections after they have been made. Usually, though, this has not been a problem, since designers can see what they are doing interactively, and repair the mistake quickly. A more insidious problem has been with modification of leaf cells. Since the instances of the cells are positioned geometrically, there is no guarantee that new leaf cells will still maintain old connections. Thus, the designer may be forced to re-edit the entire chip.

COMPED

Comped is an interactive composition editor which attempts to preserve the connections specified by the user, thereby avoiding the problems with Riot. Comped cannot make primitive geometry, and all leaf cells for Comped must be defined in Sticks. A Comped design is composed of rectangular instances placed next to one another, with all connectors on each instance connected to the four neighbors of the instance. Comped provides a connection command to do exactly that connection (figure 7). The user of Comped indicates desired connections by placing instances to be connected next to one another. The logical connection is made by abutment, and the instances are stretched to make the connection.

Comped's connection mechanism connects all connectors on the adjacent sides unless the user explicitly *omits* some of the connections. In Caltech designs, cells are designed to fit next to one another, therefore, fewer connections have to be omitted from a design than have to be explicitly stated, saving some work. The adjacency of the placed instances together with the list of connector omissions uniquely specifies the interconnection of the instances in the cell. This information is remembered so that changes in the cell do not destroy the previously-specified connections.

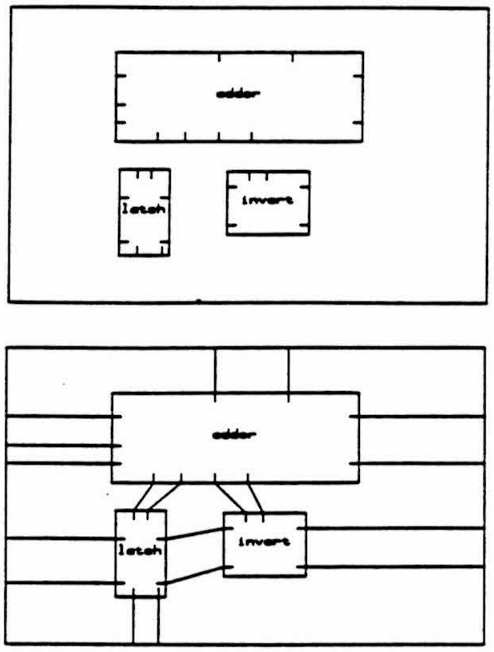

Figure 7. Comped Cell Before and After Connection.

Composition in Comped has four phases. First, the user positions instances of cells on the screen. Since the cells will be stretched, the size and shape of the rectangle representing a cell can specified by the user. This ability is essential, since the connection depends on the adjacency of the instances. The user may also specify that certain connections be omitted from the connection.

The second phase of the assembly is the connection phase in which logical connections are automatically made between pairs of non-omitted connectors on adjacent instances. The user is asked to resolve any ambiguity with the connection. At this point, the user may omit more connections or add more logic.

In the third phase, Comped stretches all instances to make the connections. Comped determines the stretches necessary to make the connection using the same graph technique used in Earl, and shows the new shapes and positions of the instances on the screen. The actual stretching of the leaf cells is delayed until the user exits Comped, so the user does not have to wait for the generation of mask geometry before viewing the effects of the stretch on the design. After the stretch, Comped provides the user with size information so that the user can determine the optimal arrangement of instances.

When an acceptable layout is produced, the user exits Comped and the real stretching of cells is done. In this final phase, the actual geometry of instances is defined and the geometry corresponding to omitted connectors is removed from the instance to ensure design rule correctness.

Comped is a new tool, and has not yet been used in any fabricated designs.

CONCLUSIONS

Work is continuing not only in the physical domain, but in the timing domain as well. A prototype system finds critical timing paths on the chip and sets device sizes to match loads. This tool will allow designers to trade off speed for power in a controlled fashion, making global decisions and relegating the detailed manipulation to the computer.

As integrated circuit densities improve, the ratio of composition tasks to leaf cell design tasks increases. Therefore, good composition tools become more essential. Traditional solutions to the chip assembly task, such as automatic place and route systems, are wasteful of chip area and performance. The composition tools at Caltech avoid the disastrous inefficiency of traditional tools by providing a less limited floorplan and more varied connection operations. These tools produce designs comparable to hand layout yet eliminate the detail usually needed to assemble custom chips. They are paving the way for much more sophisticated chip assembly tools in the future.

ACKNOWLEDGEMENTS

We would like to thank those people at Caltech who have allowed us to report on their work, notably Dave Johannsen, for Bristle Blocks; and John Tanner, the author of Comped.

The work reported in this paper was supported by the Caltech Silicon Structures Project and by the United States Department of Defense Advanced Research Projects Agency.

REFERENCES

[Hsueh 1979] M-Y Hsueh, "Symbolic Layout and Compaction of Integrated Circuits", Ph.D. Thesis, University of California at Berkeley, UCB/ERL M 79/80 Memo.

[Johannsen 1981] D. Johannsen, "Silicon Compilation", Ph.D. Thesis, California Institute of Technology, Computer Science Department.

[Kingsley 1982] C. Kingsley, "Earl: An Integrated Circuit Design Language", Technical Memorandum #4710, California Institute of Technology, Computer Science Department.

[Mosteller 1981] R. Mosteller, "REST — A Leaf Cell Design System", from J.P. Gray, ed., *VLSI 81 Very Large Scale Integration*, Academic Press.

[Rowson 1979] J.A. Rowson, "Understanding Hierarchical Design", Ph.D. Thesis, California Institute of Technology, Computer Science Department.

[Sproull 1980] R. Sproull and R. Lyon, "The Caltech Intermediate Form for LSI Layout Description", in C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison Wesley, 1980.

[Trimberger 1980] S. Trimberger, "The Proposed Sticks Standard", Technical Report #3880, California Institute of Technology, Computer Science Department.

[Trimberger 1982] S. Trimberger and J. Rowson, "Riot — A Simple Graphical Chip Assembly Tool", Proceedings of Microelectronics 1982 Conference, Adelaide, South Australia.

[Wedig 1981] R.G. Wedig, "A Phenomenal Chip for Toy Design", *Lambda Magazine*, vol II, No. 3, Third Quarter 1981.