PATCHWORK: LAYOUT FROM SCHEMATIC ANNOTATIONS

Richard Barth, Louis Monier and Bertrand Serlet
Xerox PARC Computer Science Laboratory
3333 Coyote Hill Rd., Palo Alto, CA 94304

Abstract: We present PatchWork, a silicon assembler which describes layout generators graphically with annotated schematics. The system is highly extensible and provides a unified framework in which both basic and powerful layout generators coexist. Several major chips have been built using this system.

1. Introduction

Layout generation systems solve two problems. The first is the elimination of tedious, error prone hand layout assembly. This is especially important during the assembly of a large chip when the resolution of the editing display is not sufficient to accurately position subblocks [9]. The second is the capture of design knowledge in a parameterized form so that it may be reused [3, 7].

Some layout generation systems use textual programming languages and have fixed operator sets. Others use menus or tables for capturing the parameters of blocks. Only tool developers can really extend these systems and generators cover only the very common cases because of the large development effort. At best, a tenuous connection to the logical representation of a design is formed. Some generators synthesize rectangles and focus on detailed leaf cell synthesis rather than whole chip assembly. For many full custom design systems the truth is in the layout. Schematics are an independent description used for comparison late in the design process.

PatchWork is sufficiently easy to use so that every chip design can be a parameterized generator. This expands the base of generators with more and more abstract blocks, lifting the level of design description so that designers are more productive. PatchWork describes layout generators through a mixture of annotated schematics and code, thus the designer can pick the medium which best expresses his/her intent. Some of the layout information, such as relative position, is captured by drawing the schematic with a particular *style*. Entire chips can be completely described with schematics which do not require programming skills from hardware designers. PatchWork is sufficiently powerful so that libraries of common generators and macros written by experts can be quickly copied and customized by users. The resulting circuit descriptions can be simulated in addition to producing layout. New layout primitives are easily added.

Included in these proceedings are two companion papers covering the *data structure* which is the foundation of our synthesis and analysis tools [2], and the details of capturing designs as a mixture of annotated schematics and code [1].

In this paper we present the basic schematic annotations for synthesizing and assembling layout, the overall layout generation framework, and our experience in using it for several designs. Finally, we discuss the extension of our *silicon assembler* towards the mythical *silicon compiler*.

2. Basic annotations

Let's define some terminology. At any level the hierarchical description of a circuit may consist of a schematic, an icon, and a layout. The *schematic* is the hierarchical description of the structure of the cell, expressed in terms of subcells. Leaf cells belong to atomic classes such as transistors. An *icon* summarizes the *interface* of the cell in a denser form. It simply describes the ports of the cell. Finally, the *layout* for the cell is assembled recursively from the layout of its components. In our system, schematics and layouts can be mixed in the same design since the same editor is used to draw both. This has the advantage that designers only need to be familiar with a single graphical editor.

The designer *annotates* his schematic to specify the corresponding layout. This is done by including in the schematic a visible property *Layout* whose value is a preregistered key corresponding to a layout generator. Other properties might provide extra parameters for the layout generation. An interactive command allows the designer to select the schematic and ask for its layout. At this point the layout is generated and displayed. Further commands allow the extraction of the net list and highlighting of all the rectangles belonging to a single electrical node.

The rest of this section presents examples of increasing complexity illustrating the use of layout annotations.

2.1 Hand Drawn layout

The hierarchical description of a circuit starts at the level of a leaf cell, i.e., a library cell or a critical cell. A designer using our tools draws the *schematic* of a leaf cell with primitive objects, usually transistors or gates, and a *layout* for the same cell (figure 1).

He annotates his schematic with the visible property *Layout: Get*. Another property indicates the design from which the layout should be fetched. In the example in figure 1, a private design is used, which is indicated by the property: *Design: "LouisPrivateLibrary"*. Producing the layout in this case is especially simple; the layout cell is simply fetched, extracted and checked for conformance to the schematic.

Leaf cell layout does not have to be produced by hand: it could be imported from another system (e.g., CIF reader), synthesized from a net list and interface constraints, or generated by a compactor from a sticks diagram. The corresponding generators would be very similar to *Get*.

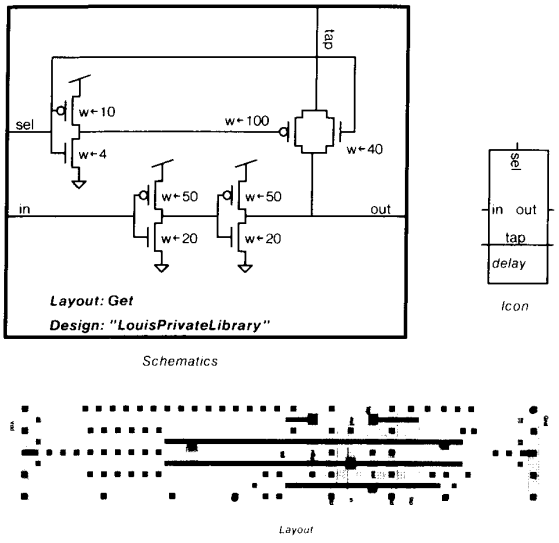


Figure 1 Schematic, icon and layout

2.2 Abutment

The simplest composition operator we use is the *abut* operator. Assuming that a cell is made of a list of subcells whose *abutment rectangles* share a common dimension, its layout is the abutment of the layout of the subcells in the appropriate direction. We build almost all regular structures with this operator.

Figure 2 shows an example of abutment. The direction of abutment and the order of the subcells is implied by the schematic drawing. Ports must match so that after abutment the resulting net list is identical to that of the schematic.

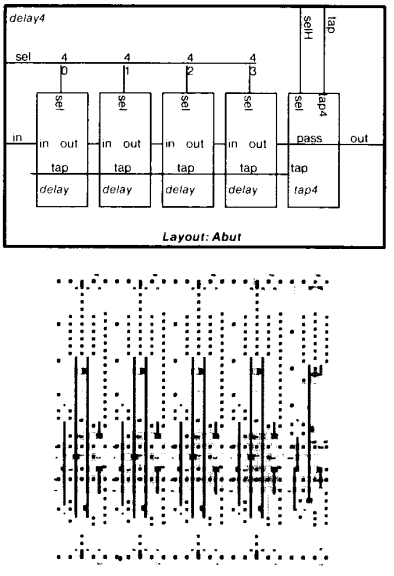


Figure 2 Abutment

2.3 Routing

A higher-level operator invokes a channel router. Given two cells, the layout generator returns the abutment of the first cell, the routed channel, and the second cell. The channel is defined by *parsing* the schematic. All wires which need to be connected and which appear on the appropriate side of the subcells' layout will be connected in the channel. The actual ordering of the subcells' wires may differ between the layout and schematic. Wires appearing on the exits of the channel are by convention found on the corresponding sides of the schematic. All router options, such as channel width or minimum net width, are explicit annotations in the schematic.

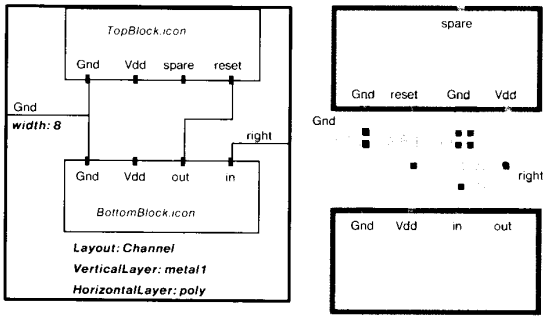


Figure 3 Toy Channel example

2.4 Standard Cell

A more powerful operator assembles standard cell blocks. The schematic is flattened so that the original hierarchy is ignored. The positions of the wires of the schematic define the sides from which the layout of those wires must exit. The relative ordering of wires in the schematic is used as a hint for the placer.

The standard cell operator accepts as a leaf any cell whose layout follows a number of rules such as standard height and standard position of the power rails. It may be hand drawn, the result of an abutment, or produced by a program from a symbolic description. As with the router, many standard cell options are controlled through visible properties, e.g., the number of rows, defaulting to a square aspect ratio.

A similar generator, *SCRemote*, utilizes a different placer on a remote Unix server. The interface of the two generators is exactly the same.

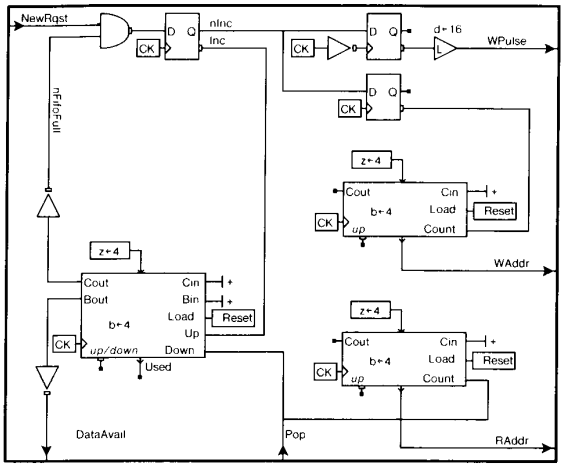


Figure 4 Standard cell block

3. Model

In order to understand how we integrate generators in our environment, it is necessary to introduce some of the data structures of our implementation.

Our most basic data structure is purely geometric and is the physical representation of layout. Primitive physical *classes* are rectangles, MOS transistors, contacts, and polygons. Composite classes include structured cells and abutments. In order to allow the abutment operator to deal with slightly overlapping cells, there is an *abutment rectangle* defined for each physical object. This rectangle may be set by the layout editor and defaults to the bounding box of the entire geometry.

We extend this physical data structure with other atomic classes, such as text, in order to represent *schematics*. A side benefit of using the same data structure for both purposes - layout and schematics - is uniformity of user and programming interfaces.

We also have a data structure for structural representation [2]. In it, a cell is either primitive, such as a transistor or behavioral cell, or composite, such as a cell made up of a literal list of arbitrary subcells. Every cell has an interface consisting of the list of its *public wires*. This data structure describes a structural abstraction of communicating black boxes and cannot be displayed without further information. Cells are created by applying the schematic extractor on schematics, or directly by executing programs [1].

All these data structures are extensible in different ways. They heavily use the object-class paradigm; also they have generic *properties* fields. We use *annotation* as a synonym for property.

In the *decorated* structural representation, a cell is annotated with layout or schematic geometry and its public wires are annotated with *pins* which are public geometry. Decorated structural representations link the physical representation to the structural representation and can be displayed. For example, one can display a layout view or a schematic view of a decorated structural representation, depending on which decoration is present. Our layout and schematic extractors generate the structural representation decorated with the layout or schematic geometry and not just a net list.

Frequently, design aids operate with the structural representation as a basic abstraction. The decoration of the structural representation with geometric information allows these design aids to easily communicate with the user in terms of geometry. For example, when a program is checking that a structural representation derived from the layout matches that derived from the schematic, it can easily highlight the rectangles in either representation.

3.1 PatchWork Engine

The job of PatchWork is to take the structural representation annotated with *layout keys* such as *Get* or *Abut* and to decorate it with the layout which is the result of interpreting those keys. Layout keys have layout, decorate, and attribute procedures associated with them. The layout procedure, *LayoutProc*, generates the layout data structure; the decorate procedure, *DecorateProc*, links the layout to the net list expressed by the structural representation; the attribute procedure, *AttributeProc*, gathers parameters from the graphical information in the schematic. The function *PatchWork.Layout* takes as input the annotated structural representation, searches in a table of registered keys, and calls the corresponding LayoutProc followed by the DecorateProc. The AttributeProc is optional and, when present, is called before the LayoutProc. For efficiency, a cache is maintained so that successive calls to PatchWork.Layout with the same cell will return the same layout.

In the remainder of this section we treat each of these procedures in turn.

3.2 Generating Layout

LayoutProcs can recursively use the PatchWork.Layout function. For example, when the annotation is *Abut* the layout method consists of applying PatchWork.Layout to each subcell and abutting the resulting layouts.

Each annotation may restrict the structure, cell class, and properties of the cells using it. For example, the *ChannelRoute* LayoutProc assumes that the cell is composite and made of two subcells while the *StandardCell* LayoutProc assumes that the cell is an arbitrary composition of library leaf cells.

The structure of the layout and the structure of the cell may be unrelated. For example, standard cell layouts are structured in rows abutted together to form an array, while the corresponding cell is structured according to the functional specification. Generally, however, flattening both structures should lead to isomorphic descriptions.

3.3 Linking Layout and Net List

LayoutProcs are sufficient for describing Abuts, but not for routing, since we need to know not only the sublayouts, obtained by calls to PatchWork.Layout, but also how the geometry relates to the structural representation. Pins define where in the sublayout a wire appears. Therefore, pins have to be built and propagated up as we compose layout with PatchWork.Layout. This is the purpose of the DecorateProc.

The DecorateProc corresponding to *Abut* decorates the cell with pins obtained by taking the union of all the pins of the subcells, properly translated, and clipped by the abutment rectangle.

The semantics of PatchWork.Layout can now be seen as a

process of decorating a cell with layout. Checking that decorations are consistent, i.e., that every public wire has pins, has proved to be a very efficient way to detect mismatches between the schematic and its layout implementation.

LayoutProcs and DecorateProcs use only interface information. This model is similar to the block-structure model in programming languages. For example we do not allow a LayoutProc to decipher the inside of a subcell in order to do its own layout generation. Along the same lines, once a cell is generated it is never modified.

3.4 Gathering Schematic Parameters

Any layout generator is a procedure which can be called directly with all of its parameters provided explicitly. For example, *Abut* needs a list of subcells and a direction. However, when calling a generator from a schematic, a lot of information can be derived from the schematic itself with the help of a few conventions. A special procedure, the AttributeProc, uses the schematic decorations and the properties of the schematic to find such things as the relative placement of subcells or the side where a wire becomes public. The AttributeProc defines the user interface of each generator.

A problem we had to solve was a lack of uniformity amongst generators. We introduced a set of standard functions that derive annotations from schematics. These functions perform syntactic analysis of graphical descriptions, thereby defining a graphical language.

4. Results and Problems

4.1 Engineering Tradeoffs

Ideally, one would like the layout of a cell to be checked as PatchWork generates it. Many checkers, such as design rule, connectivity, and electrical checkers, or schematic comparators can work hierarchically. They are compatible with PatchWork if the checking information is propagated through the cell interface.

Unfortunately, these checks are time consuming, since they require an extraction of the layout, which in turn necessitates solving a rectangle intersection problem - a typical $n \times \log(n)$ algorithm. We decided to trade early detection of errors for speed of interaction, and, except for the basic interface check mentioned earlier, checks are done as batch jobs later in the design process.

The decoration of a cell is computed from the decorations of the subcells and knowledge of the layout operation being performed. In some cases, such as routing, the layout operation naturally produces the decorations. In other cases, such as abutment, the decorations can be produced at low computational cost by assuming invariants or the invariants can be checked and the decorations produced simultaneously.

Another implementation difficulty is to reduce the memory size used for decorations. If we consider the case of a RAM made of 100 rows of 32 cells each, each cell having 10 wires, and each wire 10 pins, a naive encoding of the decorations uses hundreds of thousands of objects. We solve this problem with several techniques. Sharing cells as much as possible is of course the most effective. Using a lazy representation for pins and trading time for memory, has reduced the memory consumption by one to two orders of magnitude. Finally, using optimized representations for the two common cases of irregularity in custom chips, routing areas and semiregular tilings such as PLAs, ROMs or decoders, proved invaluable.

4.2 Wide Spectrum of Generators

Our collection of generators is quite large at the present time, and demonstrates that the framework is applicable to a large spectrum of generators. It includes geometrical transformations, PLA and Alps [11] generators from boolean equations or truth tables, finite state machines, generalized arrays, a sophisticated pad frame generator with power routing, and a data path generator which is a hybrid of abutment and channel routing.

The modularity of our implementation language and the rich programming environment available at Xerox PARC [2] help significantly by providing a set of interfaces which are like a tool box for building new generators. Most of the recent generators have been written by non-wizard users, other than the authors.

All of our generators are either technology-independent, e.g., *Abut* or *Get*, or parameterized by the design rules, e.g., *ChannelRoute*, or technology-independent but parameterized by technology-dependent cells, e.g., PLA and *StandardCell*.

4.3 Flexibility of Generated Layouts

In addition to the technology-independence of the generators, the specific chip layouts built using this system are relatively easy to modify. The assembly process is normally independent of detailed design rule or pitch changes. Changes in the layout assembly or local modifications can usually be implemented by minor editing of the source schematics. For example, the pitch of the data path of a microprocessor was increased in a few hours of editing, without changing the annotations.

The underlying system is only partially incremental, and changes in the schematics often result in regeneration of most of the layout.

4.4 Physical vs. Structural Description

For a whole range of VLSI circuits, the structural and physical descriptions are extremely close. In these cases, the schematic description of the physical implementation is concise and readable. However, structural and physical sometimes have little in common. PatchWork forces the description to be annotated schematics and some generators impose constraints on the structure of the description. Therefore, when our designers turn their functional schematics into layout, they are forced to make their schematics evolve from a functional to a physical description. For several chips, this has lead to a final description that is easy to relate to the layout, but far from a readable functional specification. This ambiguous usage of schematics has been the main design methodology impact caused by PatchWork.

4.5 Designs

Several chips have been, or are being, designed using PatchWork. One of the largest chips designed at this point, the control unit of a processor, contains over 60,000 transistors. For this chip, layout generation was done in less than 3 hours on a Dorado [10], most of that time being routing time. Its layout description is entirely annotated schematics and includes a data path layout generator designed specifically for that chip but now used in other designs. Appendix A contains

the schematic and layout for an 11,000 transistor custom interface chip. Everything needed to generate this chip is visible on the schematics. A more complete list of the chips designed with PatchWork can be found in [1].

5. Future work

5.1 Break Physical to Structural Isomorphism

We believe that more advanced generators which take high-level descriptions and heavily modify the structure to produce layout will loosen the logical-physical tie. Examples of such generators currently available include a Standard Cell system; a datapath compiler; a pad frame router; and a finite-state machine synthesizer driven by transition diagrams. We are also working on a global place and route system, which uses schematic hints.

Another strategy acknowledges the difference between functional and physical descriptions, and allows several related descriptions of the design. Although this is the traditional approach, it seems difficult to implement in its full generality, and too restrictive when implemented simply. It seems necessary to prove that the structural compositions of different behavioral primitives results in the same behavior. Promising results in the theorem-proving area may make this problem tractable.

5.2 Enhance Generators

The most obvious extension of the system adds more layout generators. We lack a cell compiler producing leaf cells in a way similar to *Get*, but starting from sticks or directly from a structural or functional representation. Such a generator would give us *design rule* independence. Full *technology* independence, for example, addition of a new metal layer, seems much harder to reach.

5.3 Improve Decorations

So far, only the connectivity information is preserved on decorations. More detailed information could be exported in the interface of a cell. The resistance between pins of a net could be of use to the global place and route for sizing power nets, and for using internal interconnect to complete the route. Capacitance of wires could be used by sophisticated generators for sizing drivers. All the geometry within a certain range of the abutment rectangle could be exported in order to construct DRC-correct cells hierarchically as in the Pooh system [13].

5.4 Change Generation Framework

Currently, the generation process has no notion of a global context. Only local parameters for each generator are available. A global context would be useful to specify libraries, design rules, degree of optimization required, etc.

A great deal of time is spent synthesizing and verifying the layout. Some of our problems are due to the memory size limitations of our processors, but to achieve the multiple orders of magnitude improvement required to allow this portion of the design process to be interactive will require fundamental changes. An incremental computation model could provide the framework necessary.

Instead of being strictly hierarchical, generation could be directed by constraints, or directed by an intelligent system such as in [5].

6. Summary

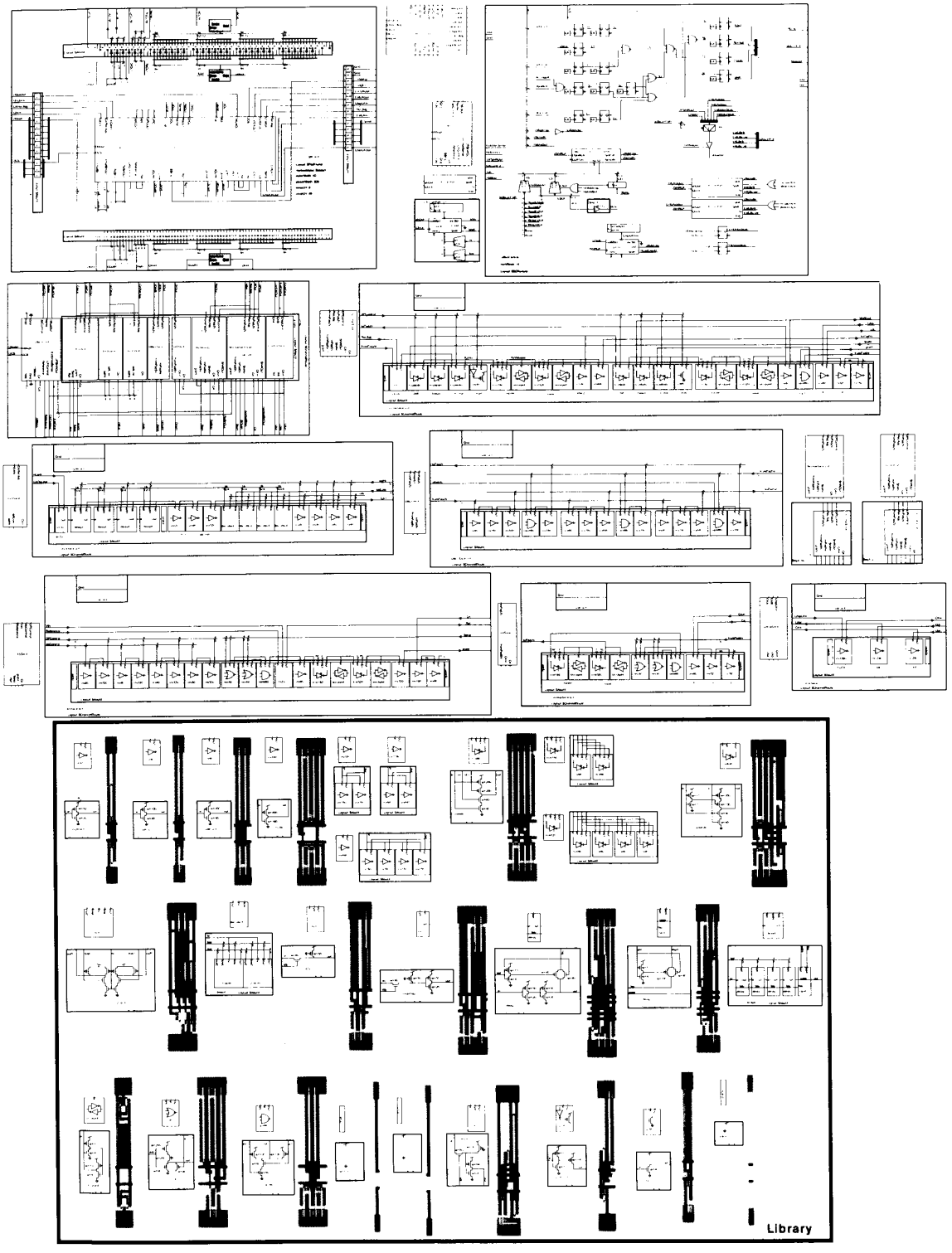
We have presented PatchWork, a unified framework for generating layout from annotated parameterized schematics. In less than two years, PatchWork has matured from a prototype to a real system, complete with a large collection of generators. Designers at PARC have produced a variety of circuits proving the validity of this concept. Generation directed by annotated schematics is the next logical step in layout synthesis.

7. Acknowledgments

The initial kick for the first implementation of PatchWork was inspired by the Parquet program of Bob Mayo [8]. The concept of a unified framework for the hierarchical description of circuits comes from many places, including the VLSI group at INRIA [4, 6, 12]. Many people in the Computer Science Laboratory at PARC contributed to the current generator set. The quick development of the system was made possible by the Cedar programming environment.

8. Bibliography

1. R. Barth, B. Serlet, and P. Sindhu, "Parameterized Schematics," 25th ACM/IEEE Design Automation Conference, June 1988.
2. R. Barth, B. Serlet, "A Structural Representation For VLSI Design," 25th ACM/IEEE Design Automation Conference, June 1988.
3. M. Buric, T. Matheson, "Silicon Compilation Environment," proceedings IEEE, Custom Integrated Circuits Conference, May 1985, pp. 208-212.
4. J. Chailloux, J.M. Hullot, J.J. Levy, J. Vuillemin, *Le systeme Lucifer d'aide a la Conception de Circuits Intégrés*, INRIA report 196, March 1983.
5. D. Johannsen, S. Tsubota, K. McElvain, "An Intelligent Compiler SubSystem for a Silicon Compiler," proceedings of 24th DAC, Computer Society Press, pp. 443-450, June 1987.
6. W.K. Luk and J. Vuillemin, "Recursive implementation of Fast VLSI multipliers," Proceedings VLSI 83, August 1983.
7. T. Matheson, M. Buric, C. Christensen, "Embedding Electrical and Geometric Constraints in Hierarchical Circuit-Layout Generators," proceedings IEEE International Conference on Computer Aided Design, September 1983, pp. 3-5.
8. R. Mayo, "Mocha Chip: A System for the Graphical Design of VLSI Module Generators," proceedings of ICCAD 86, pp. 74-77, November 1986.
9. L. Monier and J. Vuillemin, "Using Silicon Assemblers," Proceedings VLSI 85, pp. 309-318, June 1985.
10. K. Pier, "A Retrospective on the Dorado, a High-Performance Personal Computer," 10th Annual International Symposium on Computer Architecture, pp. 252-269, Dec. 1983.
11. B. Serlet, "Fast, Small, and Static Combinatorial CMOS Circuits," proceedings of 24th DAC, Computer Society Press, pp. 451-458, June 1987.
12. B. Serlet, *Description Structurelle et Simulation de Circuits Intégrés*, Thèse de troisième cycle, Faculte d'Orsay, January 1984.
13. T. Whitney and C. Mead, "An Integer Based Hierarchical Representation for VLSI" Advanced Research in VLSI, proceedings of the fourth MIT conference, MIT Press, pp. 241-257, April 1986.


Library