A STRUCTURAL REPRESENTATION FOR VLSI DESIGN

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

Abstract: This paper presents a data structure for representing the structure of VLSI circuits and basic operations for manipulating this data structure. Its features include conceptual integrity, rich expressive power, and high extensibility. It forms the nucleus of a design analysis and synthesis system which has been used to design several major chips.

1. Introduction

The representation and management of VLSI design data has been the focus of intense scrutiny over the last few years. These efforts range from traditional database issues such as access synchronization and version management [6] to data structures which represent specific views of a design [7]. Models of design representation which break design data into specific views, such as behavioral, structural, and physical, in which each view is further decomposed into multiple levels of abstraction, have also been proposed [4, 5, 17].

This paper describes a data structure for the structural representation of a design with an extensible set of abstractions. This data structure does not deal with traditional database issues nor is it a hardware description language. Rather it is an intermediate form which resides between design capture and design tools. Design capture and an architecture for synthesizing layout utilizing this intermediate form are discussed in companion papers [1, 2].

In many respects, this data structure is a parsimonious implementation of previous ideas [8, 12]. A small number of data types are used to represent the fundamental structure of a hierarchical net list. These data types are liberally sprinkled with property lists to allow arbitrary annotations to be placed upon the data structure. Instead of a fixed set of device types, a universal device interface with a class mechanism facilitates new abstractions. Procedural translation from one abstraction to another allows new abstractions to be added to the system without requiring modification of all the tools.

A partial design automation system already existed at Xerox PARC when the design of this data structure began. It contained a high-quality layout editor, as well as plotting and mask-making software that are still in use today. In addition an enhanced MOSSIM II [3] switch level simulator, a transient analysis circuit simulator, and a combined DRC and extraction program were available, each with its own notion of net list specifications.

This software is implemented in the Cedar programming environment [11, 15, 16] running on Dorado computers [10]. The automatic storage management facilities of Cedar, notably an incremental garbage collector, and Cedar support for procedure variables, allowing widespread use of the object-oriented paradigm [13], were influential in the design of our data structures and of the programs for manipulating them.

The major users of this software are the designers of the Dragon computer system [9]. The additional tool requirements to support the Dragon design included integration of the simulators with a better design capture and synthesis system. These additional requirements needed to be met quickly and so we took a fairly conservative approach to the design and implementation. The data structure presented here provided the nucleus of the additional tools needed to satisfy those requirements.

The following sections describe the basic data types of, and operations upon, the data structure and some abstract classes of design representation. The paper concludes with results and plans for future work.

2. Basic Data Types

Only four data types are needed to define the annotated, hierarchical, net list: property, wire, cell type, and cell class. Net lists are composed hierarchically to form a direct acyclic graph (DAG) in which nodes are cell types and edges are instances of these cell types. We show the corresponding declarations for the data types in a Pascal-like syntax.

2.1 Property

```
Properties: TYPE = LIST OF RECORD [key, value: POINTER];
```

A *property* is comprised of a key and a value. The value is a pointer to an arbitrary Cedar object. A key must be an identifier which is unique across the entire design automation system. There is no automatic detection of key conflicts. Generally each property is associated with a distinct portion of the system. A convention which requires that a property key have a prefix unique to that portion of the system has been sufficient to avoid trouble.

All of the basic data types have a property list, including properties themselves. A most useful *property property* is a procedure to pretty print the property value. A global property key is defined to allow systematic naming of all of the objects represented by data types.

Properties are used throughout the system for many purposes. They capture tool-specific information such as names or transistor sizes. Properties can be used to store computed data, such as wire capacitance. It is common for a tool to cache its result in the property list of its argument. Properties can record the methods applicable to an object. Finally properties are convenient places for passing information between programs.

2.2 Wire

```
Wire: TYPE = POINTER TO RECORD [
    properties: Properties ← NIL,
    elements: ARRAY size: INTEGER OF Wire];
```

Wires represent electrical nodes. In order to capture the

notion of a bus and to allow arbitrary grouping of nodes, a wire must be an arborescent data structure. A tree representation for wires would not be sufficient to indicate wire permutations, as shown in figure 1, or to capture multiple groupings of the same nodes. Therefore wires are directed acyclic graphs. A wire representing a single electrical node is called *atomic*. Nonatomic wires are called *structured wires*, and capture an ordered sequence of sub-wires. The wires without parents in a wire DAG are often grouped for convenience in a single wire called a *root wire*.

The need for wire types, i.e., a data type that describes the structure of a wire independent of its instantiation, never arose.

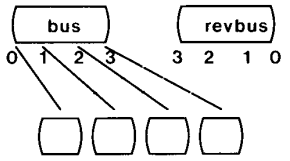

Figure 1. Wire permutation

2.3 Cell Type

```
CellType: TYPE = POINTER TO RECORD [
    class: CellClass,
    public: Wire,
    data: POINTER ← NIL,
    properties: Properties ← NIL];
```

A *cell type* represents a device or a collection of devices. In addition to a property list, every cell type has a public root wire which represents the electrical nodes that must be bound when the cell type is instantiated. A cell type belongs to a cell class and contains a data field specific to the class. A cell type which belongs to cell class *C* is referred to as a *C cell*.

2.4 Cell Class

```
CellClass: TYPE = POINTER TO RECORD [
    name: STRING,
    recast: RecastProc ← NIL,
    properties: Properties ← NIL];

RecastProc: TYPE = PROCEDURE [CellType] RETURNS
        [CellType];
```

A *cell class* defines the internal implementation of a cell type. A cell class can be primitive or composite. Primitive cell classes are technology and abstraction specific. In a switch level abstraction of a MOS technology, the only primitive is the transistor. In a lumped element circuit abstraction of a technology, the primitives include resistors, capacitors, inductances, voltage sources, etc. The *unspecified cell class* is a cell class independent of technology or abstraction. It indicates that the internal composition of a cell type is unknown. Usually, unspecified cells have properties that define their behavior for tools, e.g., the simulator. This allows tools to be applied to partially specified designs.

Composite cell classes are generally technology independent and are used to compose more complex objects out of simpler objects. A composite cell class, the *record cell class*, is the most fundamental cell composition mechanism. The record cell class contains a literal list of instances. Alternatively this cell class could specify a literal list of nets where each net specifies a literal list of instances to which it is connected. The choice of representation is empirical because many tools enumerate the instances and the wires bound to the instance rather than enumerating wires and their related instances. The cell type specific class data for the record cell class is stored in the "data" field of every record cell:

```
RecordCellType: TYPE = POINTER TO RECORD [
    internal: Wire,
    instances: ARRAY SIZE: INTEGER OF CellInstance];

CellInstance: TYPE = POINTER TO RECORD [
    actual: Wire,
    type: CellType,
    properties: Properties ← NIL];
```

Each instance specifies the cell type of the instance and a root wire, called *actual*, to which the public wire of the instantiated cell type is connected. The public subwires are bound to the actual subwires in a manner analogous to positional parameter notation in programming languages but applied in a recursive manner. Instances have a property list, as do wires, cell types, and cell classes.

All of the wires in a record cell, including public and all actual wires, can be enumerated from a root wire called the *internal*.

Every tool which understands composite cell types must understand the record cell class. Every composite cell class except the record cell class must have a procedure, the *recast procedure*, which defines a translation from the original cell type to a cell type whose cell class is simpler. A finite series of recasts must result in a cell type of record or primitive cell class. In a traditional design aid system architecture, each tool knows how to deal with each class. With the recast mechanism, each tool can define new classes as appropriate, and as long as a recast procedure is supplied for these new classes, no changes need to be made to existing tools. This reduces a problem of size $n \times p$, where n is the number of cell classes and p is the number of tools, to a problem of size $n + p$.

The cell class is the basic abstraction extension mechanism. Within this single structural framework, it allows a cell type to be as complex as a state machine specification or as simple as a single transistor. The recast mechanism is invoked only when some tool is applied to an abstract cell type which it cannot understand. This allows lazy evaluation of the translation from abstract specifications to detailed implementation without requiring explicit sequencing control by the tool user. Cell classes extend the set of abstractions available to capture design intent. In addition, they also allow abstractions with different space/time tradeoffs.

The series of recasts from the original cell type to a cell type understood by the tool form a chain of cell types, each with its own property list. It is legal for a recast to compute a new property which did not exist on the original cell type. Thus a tool which is looking for a specific public property must continue searching the recast chain until it finds the property it wants or no more recasting is possible. The property search mechanism is further complicated by an inheritance scan of the property list on each cell class of each cell type in a recast chain and by a performance optimization which avoids high cost recasts.

The split between evaluation during net list construction

and evaluation during recast depends upon the abstractions which the tool set utilizing the net list supports. Typically a new cell class is introduced when some synthesis or analysis tool needs a different abstraction to be captured. If the abstraction is merely structural, then the structure is usually built during net list construction.

3. Operations

This section presents operations available as Cedar modules for manipulating the net list.

3.1 Printing

Printing procedures display properties, wires or cell types of arbitrary classes. Printing is object-oriented, i.e., a specialized printing function can be defined for a new class or property. Printing utilities are helpful for debugging, but they are not intended to be used in designer interactions, where graphical feedback is preferred.

3.2 Wire Enumerations

There are various procedures for enumerating wire DAGs or wire bindings. Some of these utilities only enumerate atomic wires, so that the implementation of tools which only need to know about atomic wires, such as layout tools, is as simple as if all wires were atomic.

Throughout the system, enumerations are the preferred way to traverse data structures because they do not require object allocation other than on the program stack.

3.3 Wire Naming

From the beginning of the system design names have not been mandatory. This forces the user interface to handle graphical abstractions as well as textual. Since names are not mandatory, they are expressed as properties and not as fields in the data structure. This improves memory utilization, especially for representing layout where most objects are unnamed.

Any node in a wire DAG can be named. Sibling wires must have unique names. A root wire provides a naming context. Any wire reachable from a root wire has a name which is the concatenation of the names along the path from the root to the wire. If a wire is not explicitly named, then its name is the index in its parents sequence of wires, as shown in figure 2. Functions are provided to construct all the path names for a given wire, reachable from a root wire, and to find a wire given its path name.

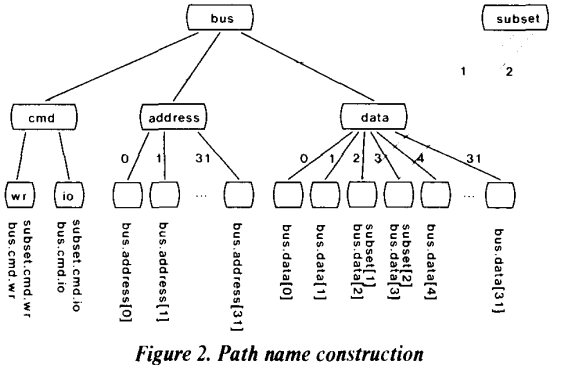

Figure 2. Path name construction

3.4 Creation Conveniences

A rich set of utilities is provided to ease the creation of properties lists, wires, or cell types. But even with these abbreviations, code description of record cells is awkward and not very compatible with the concept of minimizing designer input. Therefore, the system supplies a higher level set of functions for creating record cells, based on the assumption that public wires are usually named. These functions permit the creation of instances in which only bindings of wires with different names are specified. Arguments of these functions can be either names or wire objects, thus maintaining complete compatibility with the abbreviation utilities. Finally exhaustive error checking is made at the time the record cell is created. This approach offers many advantages compared to the traditional introduction of a hardware description language. It blends perfectly with the rest of the system, does not require yet another language design, and is easy to implement (1 module, 10 functions, about 200 lines of Cedar). Of most importance, the designer model is simple. Figure 3 illustrates the use of code to construct a decoder with *n* inputs.

```
Decoder: PROC [n: INTEGER] RETURNS [cell: CellType]
  size: INTEGER ← 2**n; array: TileArray ← NEW [TileArrayRec[size]];
  FOR i: INTEGER IN [0..size) DO {
    array[i] ← NEW[TileRowRec[n]];
    FOR j: INTEGER IN [0..n) DO {
      array[i][j] ← NEW[TilingClass.TileRec];
      array[i][j].type ← IF XthBitOfN[j, i] THEN One[] ELSE Zero[];
      array[i][j].renaming ← ["In", Index["In", j]], ["nIn", Index["nIn", j]];
      };
    };
  cell ← CreateTiling[Wires[Seq[n, "In"], Seq[n, "nIn"],
    Seq[size, "Out"], "Gnd", "Vdd"], array];
};
```

Figure 3. Decoder expressed by program

3.5 Flattening

It is common for analysis or synthesis tools to recursively explore the cell type hierarchy while maintaining a binding context. Some tools, such as simulators, flatten the hierarchical net list into a new data structure prior to beginning their operation. Others, such as static checkers, operate on the hierarchical data structure but keep context-dependent auxiliary information. Using the model of virtually flattened net lists, the system provides a set of functions useful in the implementation of any of these tools.

These functions operate on two new data types: a *flat cell* expresses a path from a root cell in the cell hierarchy and a *flat wire* combines a flat cell and a path from a root wire in a wire DAG. Currently, the only instances found in the path of a flat cell are record cell instances, designated by their index within the sequence of instances in the record cell.

The most basic operation provided is the enumeration of a hierarchical net list data structure. This operation can perform an in-order scan of the net list DAG or it can be directed to scan from a root cell to a target flat cell. The directed scan is useful when a flat cell is given in the context of a root cell type and when some tool needs to directly access the data structure of the specified cell type. The enumerator can keep track of wire bindings, i.e., the mapping of the wires of the current cell into the highest level in the cell hierarchy where they were defined.

A tool-independent method for controlling enumerations is needed. Cell instances, types, or classes or specific flat cells

can be marked to define a *cut set*. During an enumeration, the cut set defines where the enumeration cuts off. Portions of the cell DAG below a cut point are not enumerated.

Other functions are provided to print and parse flat cells and flat wires. They are very useful for implementing rudimentary textual user interfaces and for debugging tools. Tools maintaining a flattened representation of the net list generally store flat cells and flat wires as part of their data structures, in order to keep the relation with the hierarchical net list. In this manner, all communication with the user can be in terms of his original source. For example, the schematic editor has a command for adding a selected wire to the oscilloscope-like output of the simulator.

4. Abstract Cell Classes

In this section, we present some of the composition classes introduced to capture higher level concepts than the standard record cell class. All these classes eventually recast into record cells. This is not an exhaustive list. Some styles of layout, e.g., Alps [14], are better represented with special purpose abstractions which will not be presented here.

4.1 Sequence

Repetitive structures are very common in custom silicon. The simplest construct in the system for repetitive structures is the *sequence cell class*. This cell class abstracts one dimensional arrays of a single base cell type. The manner in which the base cells are wired together to form the sequence cell is somewhat restricted. The common cases are covered but not all unidimensional arrays can be described with this abstraction.

Each public wire in the base cell type may belong to one of three different types of connections in the sequence cell type. The wire may be global to the array so that every cell has this wire connected to the same net, the wire may be used to form a bus which has as many elements as the number of cell types in the array, or the wire can be stitched from one cell type to the next. An example of this last type of connection is connecting carry-out to carry-in in a ripple adder.

The sequence cell class abstraction is used by the layout synthesis software to generate layout by abutting the layout of the base cell type with itself as many times as the cardinality of the sequence cell. Other uses of this abstraction, e.g., speedup of a hierarchical DRC, can be imagined but are not currently implemented.

4.2 Tiling

Custom silicon often comprises areas where a small number of tile types are abutted in rows and columns to form the layout. Common examples of tilings are PLAs, ROMs, decoders, and mask generators. A *tiling cell type* is specified by an array of cell types and two functions, one for each dimension, that enumerate the stitches between two neighbors (i.e., the list of pairs of public wires bound together).

The conceptual information conveyed by tiling cells is different from sequence cells. They allow for several different base cell types and the binding of the subcells is performed by pairwise enumeration.

Without a special representation, tiling areas are usually the major memory consumer in the representation of a custom chip. A tiling cell is much more compact than its equivalent record cell, and even more compact than two levels of record cell (the first level expressing either rows or columns). Table 1 summarizes the space consumed for these three representations in the case of the AND plane of a PLA.

	10 inputs 50 outputs	20 inputs 50 outputs	20 inputs 100 outputs
Tiling	20 Kbytes	40 Kbytes	80 Kbytes
Record	25 Kbytes	50 Kbytes	100 Kbytes
Record of Rows	130 Kbytes	260 Kbytes	520 Kbytes

Table 1. Space consumed for tiling areas

4.3 Import

The recast mechanism is useful for dealing with pragmatic system issues as well as abstraction levels. Memory space and computational requirements sometimes require storing intermediate results in a file. The *import cell class* allows the internal structure of an arbitrary cell type to be represented by a file name. When the cell type is first recast the file is opened and the internal structure is read into memory.

4.4 Finite State Machine

A prototypical abstract representation of a digital network is a finite state machine. In this system a finite state machine is just another cell class which has a very heavy weight recast procedure. Properties of a finite state machine cell parameterize the translation. Different heuristics, for state assignment and combinatorial network synthesis, and different layout styles, such as standard cell or PLA, are selected. Typically designers simulate their machines at a level of abstraction in which the states, transitions, and outputs are the primitives. The translation to layout proceeds after the machine is debugged at this abstract specification level.

5. Results

5.1 Designs

Several chips have been or are being designed with tools based on this structural description. Among working circuits is a two-chip processor, totaling about 100,000 transistors. A recently completed design is a small custom interface chip, almost 11,000 transistors, entirely designed in 3 man-months.

Various tools have been designed specifically for a given chip before being made available as general tools. A more complete review of the chips designed with this system can be found in [2].

5.2 Observations

Approximately 12 man-years have been invested in building the design automation system which uses this data structure as one of its foundations. This system contains about 200K lines of source, exclusive of the graphics editor.

The system has evolved, rather than following a grand design. The form of the data structures and algorithms for relating flattened representations to the hierarchical representation presented in section 3.5 became apparent during development of the simulator. An attempt was made to design this functionality during the initial development of the structural representation. This attempt was unsuccessful because the client program needs were not sufficiently well understood.

No source information is stored by this data structure or any of the tools which depend on it. This eliminated stringent

testing to ensure no loss of source information and enabled rapid evolution of the system to meet user requirements.

All the data structures necessary for capture, synthesis, and analysis were assumed to fit inside of the 32 megabyte virtual address space of the Dorado. It has been a battle to fit the largest chips into this space. This battle was temporarily lost and a checkpoint facility was introduced to write partial results on the disk. Through improvements of data structures and algorithms, the need for checkpoints due to space constraints has been eliminated. They are still used as caches to avoid extracting library net lists each time a library element is used. In retrospect, avoiding consideration of secondary storage management was a good decision. It allowed focus to be placed upon clean design without all the confusion that such storage management would have added.

The design and implementation of intermediate file formats has been completely eliminated. Tools in the old system, which used net lists for their primary input description, had over half the source code devoted to printing and parsing. Since all data structures, from the original source representation to the representations used by the tools, are all in memory at the same time, it has been fairly easy to link them together. For example, the schematic capture program and the simulator, which are two very independent programs, are linked tightly enough that a designer can select a rectangle in his schematic and invoke a command which displays the waveform of the wire represented by that rectangle.

The extensibility of the structural representation worked out very well. Very few substantial changes to the semantics of the structural representation were made after its initial definition. Wires were initially defined as trees but, during the design of the schematic capture program, it became clear that the generality of a DAG representation is needed.

The assumption that cells and wires are immutable, with the exception of properties, significantly simplified the system design. Relying on the Cedar garbage collector to recover unused objects greatly reduced the complexity of tools, as well as communication between tools.

6. Future Work

This section describes improvements to design representation, the computational model for programs which create the representation, and the underlying programming environment. These changes improve either designer or tool efficiency, or improve support for a wider range of design methodologies.

6.1 Representation

The current instance binding mechanism is costly and inflexible. A class mechanism which discriminates on cell type or cell instance would allow experimentation with better methods. Wires are by far the most space consuming data in this representation. Frequently this is because a designer has described the physical cell structure as slices across a data path and the many wires which run orthogonal to the slicing structure must be repeated in every cell. Wires are often carried deep into a hierarchy and the wires must be represented at each level.

The recast mechanism currently returns a whole new cell type. It could compute a new representation of only the internal structure of the cell. This would eliminate the complex and inefficient property inheritance scheme mentioned in section 2.4.

Many of our tools, especially the simulators, flatten the hierarchy prior to doing their work. The current mechanism is adequate for all tools, but a standard flattened representation could maintain the links to the original hierarchical representation. This would further reduce the amount of tool-specific code by eliminating both the flattening code and much of the user interface code.

The physical and logical descriptions of a circuit must be isomorphic in order to back-annotate the logical description with physical properties such as wire capacitance. Property lists which are context-dependent would allow removal of the isomorphism restriction. A context-dependent property carries the path to the parent cell which forms the context. Any tool searching for a property which might be context-dependent must supply the current path from the root cell in order to find the correct value of a property.

Breaking the structural to physical isomorphism must be more general than just context-dependent properties. The schematics which designers currently draw do not reflect the functional view of a design but rather the physical design. Two structural descriptions are needed: one functional, which terminates at fairly high level behavioral components, and another physical, much the same as what exists now. These descriptions must be able to share common pieces, so a single data structure must represent both. The physical description should not be required to specify connections already specified by the functional description. A physical cell decomposition, which parallels the logical description and partially specifies interconnections to guide layout generation, is needed. In such a scheme, checking procedures to ensure that the logical and physical representations match the functional description's leaf cell behavior are also needed.

The current specification of behavior uses Cedar procedures. These procedures are not easily manipulated. Automatic synthesis of structure from behavior, and tools to check that the implementation of behavior is correct, are difficult to implement because of this. Temporal information also needs better representation.

The issue of persistent data storage has been completely sidestepped by this representation. When designs are captured directly by the structural representation data structure, then the issue of persistent storage of captured and computed data will have to be dealt with.

6.2 Computation Model

Currently, each source change causes the structural description to be rebuilt before any tools are applied. Some caching occurs at the cell type level, but the model of computation is essentially a batch procedural one where objects are immutable once built. All of the tools follow this batch model. No results are saved from the analysis of one structure to the next when only minor changes are made. For large chips, this model leads to source change and simulate loops which last 30 minutes or longer. Changing the model to one in which the objects are mutable and computed incrementally may require fundamental changes to the structural representation. Layout and simulation are both

areas where significant performance improvements could be realized from this change in model.

6.3 Environment

A lot of programming effort was expended because of lack of sufficient virtual memory. Many optimizations had to be created in a fire-fighting fashion so that larger chips, 50,000 to 100,000 transistors, could be completed quickly. A larger virtual memory would allow space to be traded for ill thoughtout hacks because some design ran into a brick wall.

The programming environment currently has a single language. The capability of running languages which are compatible with other design automation efforts is critical for building a complete system which remains at the state of the art. Some tools have been recently integrated through the use of remote procedure calls but there is a need to tightly integrate interactive programs written elsewhere into our design aid system.

7. Summary

An extensible data structure for the representation of the structure of VLSI devices has been presented. The utility of this data structure has been demonstrated by the design automation system quickly constructed on top of it and the integrated circuits built using that system. Finally some directions for improving the representation of designs, the computational paradigm used by the tools, and the software development environment were sketched.

8. Acknowledgments

In addition to the authors, Michael Spreitzer is responsible for the ideas expressed in this paper. We are indebted to the many people who worked on the tools which utilize the work presented here. We are grateful to the Dragon designers for their willingness to bear with us during the development of the system. Rapid development was made possible by the Cedar system. Xerox provided a stimulating environment that made this work enjoyable as well as fruitful.

9. Bibliography

1. R. Barth, L. Monier, and B. Serlet, "PatchWork: Layout From Schematic Annotations," 25th ACM/IEEE Design Automation Conference, June 1988.
2. R. Barth, B. Serlet, and P. Sindhu, "Parameterized Schematics," 25th ACM/IEEE Design Automation Conference, June 1988.
3. R. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," California Institute of Technology, July 1983.
4. D. Gajski, N. Dutt, and B. Pangrle, "Silicon Compilation: A Tutorial," *Journal of Semicustom ICs*, pp. 5-21, December 1986.
5. D. Harrison, P. Moore, R. Spickelmier, and A. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment," IEEE International Conference on Computer-Aided Design, pp. 24-27, November 1986.
6. R. Katz, M. Anwarrudin, and E. Chang, "A Version Server for Computer-Aided Design Data," 23rd ACM/IEEE Design Automation Conference, June 1986.
7. D. Knapp and A. Parker, "A Data Structure for VLSI Synthesis and Verification," Technical Report CRI-85-19, Department of EE Systems, USC, August 1985.
8. D. LaPotin, S. Nasif, J. Rajan, M. Bushnell, and J. Nestor, "DIF: A Framework for VLSI Multi-Level Representation," Semiconductor Research Corporation Preprint, November 1983.
9. L. Monier and P. Sindhu, "The Architecture of the Dragon," Proceedings of the IEEE 1985 COMPCON, pp. 118-121, Spring 1985.
10. K. Pier, "A Retrospective on the Dorado, a High-Performance Personal Computer," 10th Annual International Symposium on Computer Architecture, pp. 252-269, December 1983.
11. E. Schmidt, "Controlling Large Software Development in a Distributed Environment," Xerox Palo Alto Research Center Technical Report CSL-82-7, December 1982.
12. B. Serlet, *Description Structurelle et Simulation de Circuits Intégrés*, Thèse de troisième cycle, Faculte d'Orsay, January 1984.
13. B. Serlet, "Object Oriented Programming in Cedar," Actes des Journées Langages Orientés Objet, Bigre Globule, pp. 64-68, January 1986.
14. B. Serlet, "Fast, Small, and Static Combinatorial CMOS Circuits," 24th ACM/IEEE Design Automation Conference, pp. 451-458, June 1987.
15. D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, "A Structural View of the Cedar Programming Environment," ACM Transactions on Programming Languages and Systems 8(4), October 1986.
16. W. Teitelman, "A Tour Through Cedar," IEEE Software, 1(2), pp. 44-73, April 1984.
17. R. Walker and D. Thomas, "A Model of Design Representation and Synthesis," 22nd ACM/IEEE Design Automation Conference, pp. 453-459, June 1985.