AN OBJECT-ORIENTED CELL LIBRARY MANAGER

Naresh K. Sehgal[†], **C. Y. Roger Chen**[‡], **John M. Acken**[+]

Intel Corporation[†] Santa Clara, CA

ECE Department, Syracuse University[‡] Syracuse, NY

CrossCheck Technology, Inc.[+] San Jose, CA

New techniques are proposed to obtain better estimates and optimizations at higher levels of design abstractions, which are then used for library cell selection. A single object-oriented database repository is used during all phases of VLSI design to enhance the early design estimates. As compared to a relational database using sorted tables of attribute values, the proposed object-oriented cell library manager reduces search time for an appropriate cell, with m constraints among n cells, from O(nm) to O(m log n). The proposed method also reduces design cycle time by reducing the number of iterations due to mismatched performance estimates done in the earlier phases of a design.

1.0 Introduction

Existing VLSI design methodologies for a custom design often do not meet the conflicting need for high design productivity, low power consumption, and higher performance goals. The current VLSI product design methodology is to develop an RTL model [36] after high-level architectural details are known. This model is tested for functionality and power estimates are obtained on the basis of preliminary mapping to a logic library. After the designer determines individual tasks to be performed in each clock cycle, any necessary buffers and multiplexors are inserted, and driving strengths of individual cells and devices are determined [37] to complete the circuit design. During the layout phase, place and route algorithms use the circuit netlist information to place individual components and interconnect them. Power adds yet another dimension to the design space.

Performance and power estimations at RTL and circuit level are difficult because of the lack of layout capacitance data, e.g., two gates which may be in close proximity to each other at a higher level of design representation could be placed far apart in the actual layout. This results in a mismatch in the estimated area, power and performance at the logic level to the actual characteristics of the finished layout design, as shown in Fig. 1.

One possible solution is to the overlap the design steps so that estimates done at the higher levels are more realistic. The trade-off is in the increased complexity of estimation models and an effort to complete part of the layout design before the logic mapping is fully done. Our approach is to use a library of cells with views at every design abstraction level for aiding the decision making process. The cell information is stored in a database, which is queried at each design phase. At the lowest level of hierarchy, various cells are stored as objects with attributes and at higher levels they are grouped by functionality. This information is difficult, if not impossible, to represent in the tabular format of a relational database. If *n* cells, each with *m* attributes, were to be arranged in a table, then *O(nm)* searches would be needed to satisfy design constraints on all the *m* attributes. This stems from the fact that after a subset of cells in each ordered attribute value table has been identified to satisfy a design constraint on that attribute, all the *m* tables of *O(n)* cells need to be merged together. A cell library many contain thousands of cells, and an efficient method to represent and search cells in the library is needed.

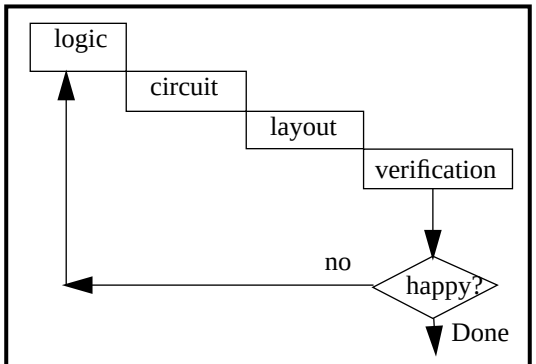

Fig. 1: A classical model of VLSI design cycle.

1.1 An Object-oriented Paradigm

Existing relational databases may be suitable for a financial or a spread-sheet like application, but do not well represent the complex relationship between different cells and the variable placement of a cell in the 3-dimensional search space consisting of area, timing performance, and power. On the other hand, object-oriented paradigm uses the following properties [1] to mimic the way we form performance and power consumption data models of cells:

1) Encapsulation: The fact that the layout area of a cell is a fixed value whereas its power consumption is a function of the actual load being driven, is completely oblivious to the user of the cell. At search time, depending on the usage environmental values such as voltage, temperature, loading requirements etc., the variable attribute values of a cell are computed and tested against the design constraints to

judge the suitability of that cell.

2) Inheritance: This property is very useful to represent cells, pins and their attributes as individual objects. No matter if a cell is a NAND gate or an ADDER, they both share several common properties (of course with different values) such as a list of input-output pins, layout area cost value and a boolean function. Furthermore, a 2-input and a 3-input OR gate have a common function representation and the later can be constructed by a repeated use of the former. It means that the library need not have each type of cell pre-stored and methods can be introduced to derive new cells on demand. This is very useful to build an 8-bit, 16-bit or 32-bit comparator from the basic building block of a 1-bit comparator.

3) Polymorphism: This property is used to mix cells with different VLSI implementation techniques, e.g., an adder at behavioral level can be implemented with a Carry Look-ahead circuit or a simple Ripple Carry structure. The dynamic delay computation of input to output pins will be modelled by different characteristic equations in both circuits and will be stored as a method in each cell. The search routine simply needs to specify the interfacing pins and the external conditions to the adder cell and functions with same name but different bodies will compute the delay result for both the cells. This property is very useful when making trade-offs at a higher level without considering the internal implementation details of each cell.

Consider the inverter cell, INV1, shown in Fig. 2. At the root of the hierarchy is the list of various cell groups, each group typically contains cells with common functionality. Directly below in the hierarchy is the list of attributes on the cell: area, power etc. and a list of output pins. Each output pin further has more attributes such as a maximum capacitive load that can be driven within a clock period. Each output pin further has a list of input pins that affect its logic function value. This is very useful in case of combinational cells with multiple output pins, where not all primary input pins can influence each of the output pins. Each of these input pins has a list of inter-pin attributes, e.g., pin-to-pin delay.

Some of the attributes have fixed values which depend on the design of the cell, i.e., area of the layout view of a cell or its maximum load driving strength. However, other attribute values depend on the design environment in which the cell is to be used, e.g., pin-to-pin delay depends on the actual load being driven and the slope of the input signal transition. For the latter case of variable value attributes, characteristic equations are formulated and their coefficients are obtained with extensive simulations with device models of the circuit and layout parasitics. These simulations are done over a range of possible loads expected to be driven and then load-delay curves are drawn. Each curve is represented by a characteristic equation and the same equation with different values of coefficients is used to represent different cells in a group. These coefficients are stored in the database in order to select an appropriate cell later. Since pin attributes can have an integer, a floating point or a string data type of value, an internal tag is used to identify the data type of value. Also an index, attrIndex, in a global table of all the attributes in the database helps identify the name of the containing attribute. This technique helps to save memory and improves efficiency. The output pin class definition is derived from the input pin class by adding an object store collection data member to store the list of input pins driving the output pin. Collections provide a convenient means of storing and manipulating groups of objects. They are used to model many-valued attributes, and they have member functions for inserting, removing, and retrieving elements as well as a function that returns the collections's cardinality. They also have member functions analogous to set-theoretic operations such as intersection and comparisons. In addition, queries can be performed over collections, to look up particular collection elements by name or ID number. Using the inheritance property, the output pin definition is derived from the input pin data class.

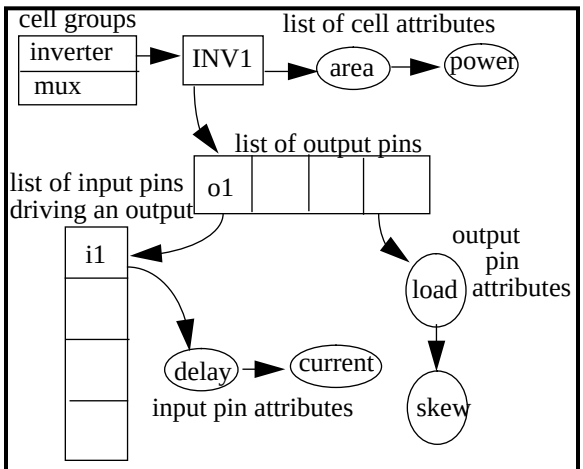

Fig.2: Schema information for an inverter cell.

A higher class of cell is further derived with the collection of input and output pins, and more attributes of its own. A template is a mechanism in the object-oriented programming paradigm which enables a generic function to operate on a family of classes without knowing the specific details of each class. Templates in the cell library manager are used to manipulate the data inherited from a common base class, e.g., a simple search by name can be performed on either the input or output pin types. In a library database, the individual attributes are classified as follows:

1) Cell attributes: these are the properties on the cell, i.e., the area of a cell, its boolean or sequential function. This category can also include additional information to manage the library, such as a version number,

name of the designer, and a time-date stamp.

2) Pin attributes: these are the properties associated with the individual input or output pins, e.g., the maximum load that can be driven a particular output pin or any restriction on the drive type for the input pins such as no domino logic in a noise susceptible circuit.

3) Inter-pin attributes: these are the properties expressing a relationship between input-output pins or the input-input pins. The former category is useful for pin-to-pin delays and the later is used to express special design conditions such as that reset and set input pins on a cell should always be mutually exclusive.

A library database is created before the target cell based VLSI design begins. Since all the data access is read-only, multiple readers accessing the same information have no locking conflicts. Each reader is actually a design tool using inter-tool communication protocol to invoke the cell library manager. The invoking tool has the client part of the library manager process which knows how to interpret the data and validate the cells. The actual library manager process is running on a database server machine, which is aware of the various methods to traverse the library data. Design constraints are provided to the library manager client which orders them according to the hierarchy, e.g., the function is the highest level of a constraint, as all cells with a common function are grouped together on one server segment. After the server library manager receives the request, an entire segment of data is transferred over the network to the client machine where it is virtually mapped to the local disk of the client. Remaining constraints are used on this mapped data to find matching cells. In case no cell satisfies all the given constraints, some constraints are relaxed according to a design objective function to identify the closest matching cell, without requiring another transfer of data over the network. This reduces the network communication overhead during the search and improves the throughput to the design tool. All computation of variable value attributes is done on the client end, and the selected cell is presented to the user. Hence adding more client readers to the library has little performance degrading effect on the overall system. In case more servers are needed, the read-only database is simply duplicated to many server machines which work independently serving their clients.

1.2 The Library Data Structures

The cell library is expected to grow beyond 10,000 cells, and it is important that cells be arranged properly to optimize searching, as well as to guarantee a result whenever a desired cell exists in the library. This problem gets more complicated when the sizes of the stored cells range from a simple inverter to complete 64-bit adder circuits. Our approach is to use multiple levels of hierarchy, as shown in Fig. 3, to store these cells. The library may have information regarding performance of the cells in various design environmental conditions, i.e. supply voltage, operating temperature, frequency, type of driver etc. The first level of hierarchy is created by specifying each set of environmental conditions. These sets are mutually exclusive and are termed process-corners. Each process-corner is stored on a unique segment of the database. Information of a particular cell is split among various process-corners, and a particular designer may choose to work in any available corner. Only data on the chosen corner is transferred from the database server to the client machine. The second level of hierarchy is a functional representation of cells, followed by cell-specific grouping, e.g., a group of NAND gates is categorized in 2-bit, 3-bit NAND gates, etc. On each cell, pin information and inter-pin performance information in terms of coefficients of characteristic equations is stored as shown in Fig. 2. During a search, the value of these equations is computed at run-time and compared with the given constraints. Special algorithms using set theory to optimize the multiple constraints based query time have been developed. A description of the search algorithm is provided later.

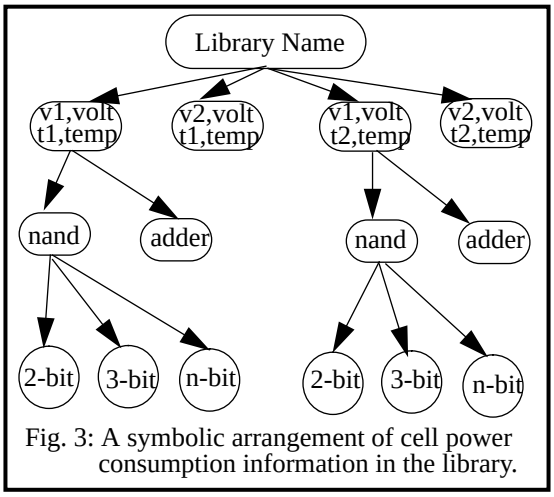

Fig. 3: A symbolic arrangement of cell power consumption information in the library.

Such an arrangement helps to model the behavior of a circuit when the operating conditions change, e.g., to find the potential loss of external load driving strength if the operating temperature is increased. Another advantage is the ability to simultaneously design a circuit for various market segments, e.g., 5 volts and 3.3 volts. All that needs to be changed is the actual cell mapping with similar functional and performance specifications. The range of operational parameters of the library cells can be extended or modified by adding or deleting a process-corner without affecting rest of the database. Similarly individual cells can be added to specified process-corners incrementally.

1.3 Search Algorithms

We have successfully implemented a prototype with more than 5,000 cells in the library. The cell library manager is actually capable of providing more cells than are stored in the database using the methods of decom-

posing the given function to existing library cells. Each cell group (e.g., 3-input NAND gates) has another collection of attributes, and this collection hierarchically contains common attribute values for all the cells in that group. This 2-dimensional value representation is shown in Fig. 4 and facilitates cell search based on desired functionality and performance constraints. Let a group of *n* cells be represented as $G=\{c_1, c_2,...,c_n\}$ and the set of *m* attributes be $A=\{a_1,a_2,...,a_m\}$, such that $c_i(a_j)$ represents the value of *j*th attribute on *i*th cell. Now set $G(a_j)=\{c_1(a_j),c_2(a_j),....,c_n(a_j)\}$ represent values of a_j attributes on all cells of *G*.

We arrange each of $G(a_j)$ as an ordered binary tree, such that the search time for a particular value is $O(\log n)$. There are back pointers from each of the attribute value to the cell on which this particular attribute instance is defined. For a given set of constraints, with cardinality less than or equal to *n*, on the attributes of the desired cell, we can reduce population of each of the $G(a_j)$ sets, such that their values meet the given constraint on a_j. Then an intersection of all $G(a_j)$ sets is taken to find the common cells, that meet all the given constraints. Actual implementation of query iteratively applies the remaining constraints on the cells of the reduced $G(a_j)$ set. Each reduction of a set of $O(n)$ cells takes $O(\log n)$ time, and there are *m* such reduction operations taking a total of $O(m \log n)$ time. This is possible due to a traversal ability from an attribute value, which meets the given constraint, to its container cell class, and then to all the remaining attributes on that cell. As compared to a relational database using sorted tables of attribute values, the proposed object-oriented cell library manager reduces the search time for an appropriate cell, with *m* constraints among *n* cells, from $O(nm)$ to $O(m \log n)$.

1.4 Experimental Results

We have applied the proposed technique to previously custom designed layouts with fixed pin library cells. Table 1 shows the size of the database variation in number of cells and number of attributes. Note that these attributes represent information at all design levels of a cell, and the variation of search time for one or more cells matching a given set of constraints has been shown.

1.5 Conclusions

Special techniques have been proposed to obtain better estimates to guide cell selection at higher levels of design abstractions. It is possible to take advantage of the layout parasitics information of mapped cells at logic and circuit levels. This allows a designer to explore the design space much early in the design cycle and improves the accuracy of the estimates. A single database repository is used during all phases of VLSI design to aid the early design estimates. As compared to a relational database, the proposed object-oriented cell library manager reduces the search time for an appropriate cell, with m constraints among *n* cells, from $O(nm)$ to $O(m \log n)$. Using The proposed method also reduces the design cycle time by reducing the number of iterations due to mismatched performance estimates done in earlier phases of a design.

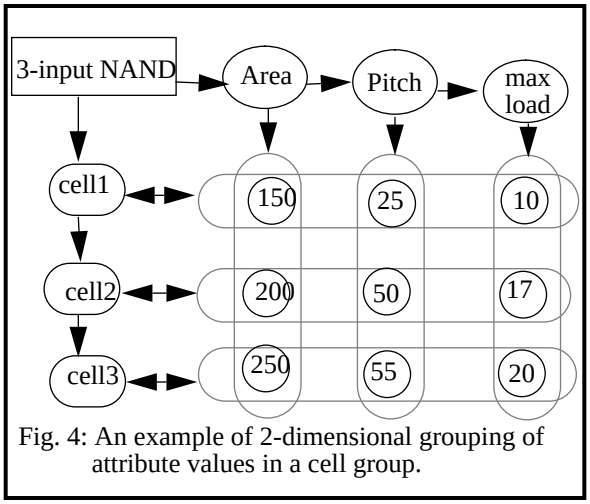

Fig. 4: An example of 2-dimensional grouping of attribute values in a cell group.

Table 1: Use of a Cell Library for VLSI Design

Test case	# of Cells	DB size (bytes)	Search time per cell (secs)
Lib1	100	4 Meg.	1
Lib2	500	18 Meg.	3
Lib3	1000	29 Meg.	3.8
Lib4	2000	60 Meg.	5.1
Lib5	3000	89 Meg.	6.2
Lib6	4000	120 Meg.	7
Lib7	5000	146 Meg.	8.6

2.0 Acknowledgments

The authors gratefully acknowledge the conceptual contributions and working tool implementation efforts by Qinghong Wu and Alan Jen.

3.0 References

[1] Object Store 2.0.1 reference manual, 1992. ODI Inc.

[2] Turbo C++ Users Guide, pp. 127-128, Borland Corp., 1992.

[3] N. K. Sehgal, C. Y. Chen, and J. M. Acken, “Datapath Cell Design Strategy for Channelless Routing,” ASIC’94.

[4] N. K. Sehgal, C. Y. Chen, and J. M. Acken, “A Cell Library Paradigm for the Channelless Datapath Layout Design,” IEEE International Conference on Microelectronics’94.