CHIP ASSEMBLERS: CONCEPTS AND CAPABILITIES

Randy H. Katz
Shlomo Weiss

Computer Sciences Technical Report #486

November 1982

Chip Assemblers: Concepts and Capabilities

Randy H. Katz and Shlomo Weiss
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

Abstract: A chip assembler is a tool for managing design information. It encourages a structured design methodology, wherein a design is described by a collection of hierarchical design decompositions, one for each of its representations. It assists in the enforcement of consistency constraints up, down, and across the different hierarchies. We argue that a chip assembler, as an integrated design environment, requires an integrated approach for the management of design data. We describe what a chip assembler is, how it is used for design management, and what are its desirable features.

1. Introduction

It is widely recognized that the greatest impediment to the successful use of VLSI technology is the lack of adequate tools to support VLSI circuit design [LOSL80]. A special need exists for tools that organize the information describing a design, including its many representations, equivalences existing between these, and the constraints that need to be satisfied, thus helping to maintain the consistency and correctness of a design. Once the design data is made machine readable, its self-consistency can be improved dramatically if verification aids can be applied in a controlled and economical way. This can be achieved if the interface specifications between subsystems within a representation, and the equivalences of design subparts across representations, are made explicit. An important objective is to structure the design data so ramifications of a design change can be identified. A tool providing such facilities is a *chip assembler*.

A chip assembler automates the bookkeeping associated with design activities. Traditionally, the designer has coordinated the vast quantity of design data himself. Information regarding the function, interfaces, and implementation details of the system's components are kept only in the designer's head (or perhaps in a not quite up-to-date design notebook). Systems of the complexity of a VLSI design can only

be designed by teams. Design teams must confront the added problems of communication within the group, undocumented design assumptions, and interference among designers. A chip assembler manages the documentation and implementation data associated with the design of systems undertaken by a team of designers. It also assists in the control of the configuration of a design. It coordinates the update of design data, maintains system releases, supports subsystem alternatives, and notifies designers when important design changes have occurred.

"Chip assembler" is a somewhat deceptive term; we are not referring to a low-level variant of a silicon compiler (e.g., [JOHA79]). A silicon compiler begins with a high-level description of a digital system, perhaps expressed in a register-transfer language, and transforms it into a circuit layout. On the other hand, a chip assembler is primarily a data management tool. It provides an environment within which traditional synthesis and verification tools can be integrated, including layout editors, placement and routing tools, design verification aids, and simulators. By maintaining design data across hierarchical decompositions of a design in different representations, it also supports the development of multi-level and mixed-mode tools. Tools that can conveniently access data across more than one design representations do not exist today.

An early attempt at an integrated environment for circuit design is the SCALD system [MCWI78a, MCWI78b]. It supports the hierarchical construction of systems by allowing designers to define subsystems as macros, which can be included in higher level subsystems. The logical representation of the design (e.g., gates and signal wiring) is closely linked to the physical representation (e.g., the placement of gates and assignment of signals to pins on packages). SCALD produces reports used in constructing, debugging, and documenting the system (such as wire-wrap lists). A variety of consistency checks are enforced by the system. While SCALD supports the design of TTL SSI/MSI level systems, many of the ideas can be extended to VLSI systems. A major shortcoming is its lack of technology and representation

independence.

Mudge first introduced the "chip assembler" concept in his CHAS system [MUDG80, MUDG81]. CHAS is a data manager for the manual composition of three different circuit representations: layout (mask geometries), circuit (transistor networks), and logic (gates). The floorplan (structural hierarchy) is the "gateway" to the design, i.e., data in the three representations is interrelated by being associated with the same cell in the floorplan. Cell interfaces are explicitly specified, and tools for routing and stretching of layout cells are integrated into CHAS. Mudge's experience is that even such a simple tool is an enormous aid in (1) making rapid changes to a design, (2) exploiting hierarchy for verification tools, and (3) managing a design project. CHAS is a simple tool (which may well be one of its virtues!) whose primary objective is to aid in composing the elements of a design. The next step is to develop chip assemblers that address the issues of design data management and design consistency maintenance.

In the remainder of this paper we describe the features that should be incorporated into the next stage of sophistication in a chip assembly system. In the next section, we describe our model of the design process. Then we state the requirements for a chip assembly system, describe the design representations and interfaces we must be able to support, and discuss the consistency constraints that can be supported. We describe the structure of the design files themselves and ways to implement design versions and alternatives. We end the paper with our conclusions and plans for future work.

2. Model of the Design Process

In this section, we describe our view of VLSI system design. Any reasonable approach will adopt a hierarchical design method. A design is specified as a forest of cell hierarchies, one hierarchy per representation domain. While we consider trees for simplicity, rooted directed acyclic graphs are easily accommodated (we will have

more to say about this when we describe the structure of the design files).

The root in each representation hierarchy ("design hierarchy") describes the same system. For example, if the object being designed is a microprocessor, then the root of the geometry hierarchy represents the layout for the complete processor, while the root of the electrical hierarchy represents its electrical description, i.e., the distribution of control signals, data, power and ground. The children of the root are the system's decomposition into major subsystems. Examples include the data path and control circuitry in the geometry tree, and the power, clock, and signal distribution in the electrical tree. These subsystems are further decomposed into more detailed subsystems at the next lower level, and so on. Internal nodes of the design hierarchy are *composition cells.* The most primitive cells are *leaf cells*, i.e., those that cannot be further decomposed. Leaf cells are convenient collections of primitives (e.g., an inverter), rather than primitive objects of the representation (e.g., a transistor). The choice of what constitutes a leaf cell is determined by the designers.

A designer begins by either describing his system with a block diagram or by providing a procedural description of its functions. Either description is usually in terms of subsystems, which themselves need to be designed or taken from a library. He then proceeds to describe each subsystem, and its interaction and interconnection with other subsystems, as he further refines the design. Normally, a design does not proceed strictly top-down or within a single representation. A designer may piece together a design from a library of building blocks, proceeding in a bottom-up fashion. He will probably need to span representations, giving more breadth to the design as he proceeds.

If he has provided enough data to describe the function and interfaces of all cells, he may proceed with a simulation of the entire system, one of its subsystems, or perhaps a single leaf cell. Most verification tools require that the complete details of a design are specified before they can be used. It should be possible to verify and

simulate a design without providing all the details in any particular representation. Perhaps the behavioral description has been specified for major subsystems, but geometries are only associated with leaf cells. For example, the function of the system has been specified, but its floorplan has not been attempted, even though its primitive cells have been layed out (e.g., consider a gate array design before placement and routing). In another scenario, stringent size constraints may require that top-level floorplans be defined, but layouts of primitive subsystems are still to be specified. The designer should be able to concern himself with the strategic issues of design without needing to include details he is not yet prepared (or able) to finalize.

Design hierarchies support a more conceptual partitioning of the design, which is crucial for supporting design teams. A designer is given responsibility for a subtree of the design. He can proceed independently as long as his implementation does not alter the specified interface and function. Alternative implementations can be represented as subtrees in parallel with the root of the subsystem.

To summarize, a design is initially sketched out at a high level. Designers provide interface and behavioral descriptions of their part of the design. They proceed independently of each other as long as they stay within the interface and behavior specified. Alternative implementations for a design part can be explored without choosing among them until implementation time.

3. Requirements for Chip Assembly

A chip assembler is not just a composition tool, but a data management and communication tool. Furthermore, it aids the designer without constraining him. It is unacceptable to only support a prescribed set of design representations, or to impose the constraint that only completely parallel and conformable design decompositions across all representation hierarchies be permitted. There will be many benefits in having a common structure across representations (in fact, we know of no other reasonable way to tackle the complexity of a VLSI design), but it need not be imposed

on them by the chip assembler. Structure, where it exists, can be exploited. A chip assembler should not force designers into any particular design method, other than that the structure of the overall design be hierarchical. As was shown above, rarely is a design carried forward completely in a top-down or bottom-up fashion. Designers have a good idea of what they want to build, and what primitives are available to build their system. A design is completed by alternating between a top-down approach of refining a subsystem into more detailed subsystems and a bottom-up approach of composing primitives to form more useful building blocks.

While a design is in progress, there are many subsystems that are incomplete or only partially specified, while others are described completely. The chip assembler must facilitate this mode of use. Nodes at higher levels of the design hierarchy need not have any leaf cells as their descendents. In the current state-of-the-art, one frequently finds that to begin the simulation and characterization of an entire system, it must be wholly described in some representation. The designer is forced to describe his design in complete detail, either in a hardware description language or perhaps as a geometrical layout, before the verification process can begin. There is virtually no mixing of levels of detail in a system description. Multi-level simulation tools are easier to build in the framework provided by a chip assembler.

Similarly, a design need not proceed in parallel across all of its representations. The chip assembler should provide facilities to interconnect pieces of a design across representations, but these do not have to be specified with the same level of detail. For example, an ALU of a microprocessor design can be fully specified in its electrical specification (for simulation purposes), but only partially specified in its layout. The correspondence between the electrical view and layout view of the ALU must be made known to the chip assembler by the designers.

One can proceed to describe components at whatever level is deemed appropriate and still be able to perform a complete system simulation. There is no limit on the

kinds of representations supported, and no representation is the preferred gateway to the design. The representations of the same design need not exhibit the same process of refinement.

A chip assembler supporting these features is also a powerful tool for documenting a design. The design process can be reconstructed by simply descending the design hierarchies. English-language descriptions of design components, with links to the actual implementation details, can be made available to a browser. He may choose which level of detail he wishes to view the design at a particular moment. An important consequence is that completing a design also completes its documentation.

4. Representations and Interfaces

The most important facility of a chip assembler is its management of interfaces. It stores specifications on how a subsystem interfaces to the rest of the design, and provides mechanisms by which these can be checked for conformance. Since the specification of an interface is highly dependent on the design representation, we first describe some of the representations that are likely to be found in a chip design database.

Each representation hierarchy of a design is refined functionally, but the decomposition criteria can vary across representations. For example, while it is appropriate to decompose a microprocessor into a data path and control section, a similar decomposition may not be appropriate for the electrical view. It may be more convenient to decompose the microprocessor into clock, power, control, and data signal distributions, especially if a signal delay analysis is desired. Undoubtedly, an electrical representation that closely mirrors the layout decomposition will also be required. In the following, we use "behavioral" instead of "functional" to describe the representation of how a system is to function.

Several representations have been proposed to describe various aspects of a VLSI design. Verification tools exist for most representations to check their

correctness. We expect to find the following, although the list is by no means exhaustive:

(1) *Block Diagram*

Circuit subsystems are represented as named boxes, with input/output signals to denote control and data flow. Signal busses, i.e., wires that connect more than two subsystems together, are also specified. This representation is used primarily as a documentation aid.

(2) *Behavioral/Functional*

The behavioral representation is similar to the block diagram representation, except that the output signals are defined as functions of the input signals. This description is most frequently used to partition the design among subsystem designers. Functional simulators are used to verify the correctness of a behavioral representation.

(3) *Geometrical/Physical*

The physical representation describes how pieces of a design are physically placed. For custom chip designs, it is often called the floorplan (at the root). Tools include those for placement and routing (used in composition cells), cell stretchers and compactors (used to modify leaf cells), and synthesis tools that generate this design representation from the behavioral, geometrical, or sticks representations. Design rule checkers check the validity of the geometries.

(4) *Electrical/Transistor*

The electrical representation describes a design as an electrical circuit, usually as a circuit schematic (i.e., interconnected transistors with associated resistances and capacitances). The verification tools may be at the analog level (transistors modeled as analog devices), or at the switch level (transistors

modeled as perfect switches). The representation is appropriate for timing simulations. Electrical rules checkers validate the electrical representation.

(5) *Sticks*

This representation combines the topological properties of geometries with transistor switches. A sticks description is easy to stretch and compact, making it a higher level description from which geometries or physical layout can be synthesized. It is validated by switch level simulation.

(6) *Clocked Primitive Switches/Logical*

Since custom designed circuits rarely can be described completely in terms of gates at the logical level, [BELL81, STEF82] proposes a new intermediate level called Clocked Primitive Switches (CPSW). The representation combines concepts from the electrical level and logical level to insure that circuits behave correctly as logic devices. Electrical/logical checkers can verify the correctness of this representation.

(7) *Clocked Registers and Logic*

This is another hybrid representation suggested in [BELL81, STEF82] that is similar to the behavioral level, but explicitly includes knowledge about the clocking of circuit elements. A variant of a functional simulator can be used to validate this representation.

The kinds of information found in the interface specifications for several different representations are shown in figure 1. Some of these parameters have been used to describe cells in the Stanford Cell Library [NEWK81].

Interfaces not only describe how a cell interacts with its neighbors, but are also constraints on the cell. They should be specified even before the cell design begins. When interfaces are explicitly supported, new kinds of tools are possible. For example, tools to extract the interface specification from a fully specified cell, and tools

Representation Type	Interface Specifications
Block Diagram	Signal Names Busses
Behavioral	Signal Names and Types (Input, Output) Output Signal = function(Input Signals) Inter-block Connection list of outputs to inputs
Electrical	Signal Names and Types: restored input, restored output switch input, switch output Loading: input capacitance, resistance output currents Power Consumption
Geometrical	Signal Names and Positions connection coordinates connection layers connection width Cell Area, Symmetry, Stackability
CPSW	Signal Names and Types: restored input, restored output switched input, switched output clocked input, clocked output control input data input qualified clock input

Figure 1 -- Some Representation Types and Interfaces

to verify that the cell meets its interface specification can be built. These can be one in the same; such a tool could fill in unspecified or defaulted parameters of the interface and check the specified ones.

A cell's interface also describes (1) the cells that interface to it within a given representation, and (2) their designers. A modified cell may no longer meet its interface specification. It is then inconsistent, and the design cannot be released in its current state. This is likely to closely follow the design's behavioral/functional

decomposition. A chip assembler can support a process of negotiation between the designer of the changed cell and the designers of cells that interface to it. The conflict must be resolved, which may involve abdicating the responsibility to a higher authority for arbitration, i.e., the designers' managers. The "administration" hierarchy of who is responsible for parts of a design, including their managers, is yet another representation of a design. Either the interfaces are changed or the offending cell is redesigned.

5. Integrity Across Hierarchies

We have already argued that it is unacceptable to constrain a design to reflect the same structure across all of its representations. However, the job of integrity enforcement is much simpler if this common structure exists.

Suppose that there is no common structure across the electrical and layout hierarchies of a microprocessor design. At least the roots of the two hierarchies are constrained to describe the same object. The only way to enforce this is to reverify the entire design after each design change! This takes place as follows. An electrical description is extracted from the layout. Equivalent test vectors are simulated with the extracted description and the description from the electrical representation. These are "equivalent" if they respond with the same behavior. Integrity is maintained, but at a high price, since verification over the completely instantiated design is extremely expensive.

If it is possible to identify those parts of the layout and electrical views that describe the same design subpart, e.g., the ALU, then a change only causes a reverification over the extracted description of its layout, not the entire design (see figure 2). A smaller part of the design is reverified if there are common ancestors in both representations known to be equivalent and which cover the design change. In the worst case, this is the root of both hierarchies. It is in the designers' interest to insure that there is much common structure in the hierarchies, to keep manageable

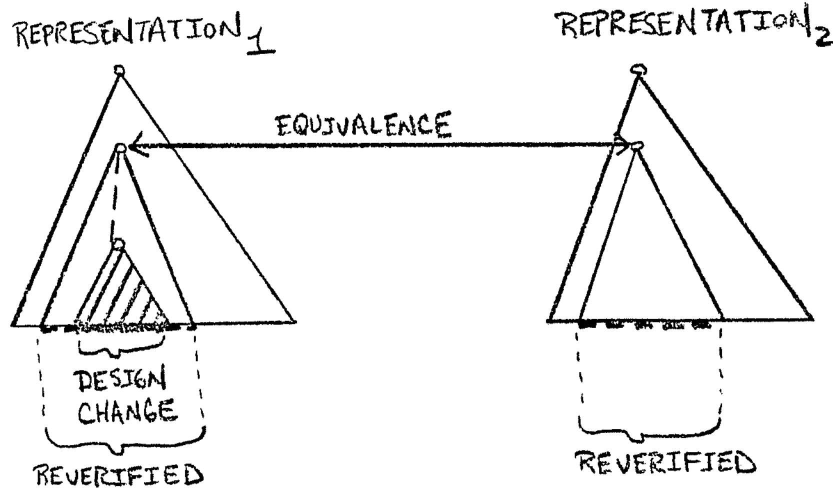

Figure 2 -- Equivalence Across Representations

the job of reverification after a design change. Also, from the viewpoint of overall project management, there is likely to be much common structure. It is too difficult to control a project when a design's different representations are completed with total independence.

In the example in section 2, the ALU was contained within the data path. Unfortunately parts of the ALU involve both power and signal distribution. Thus, a change in the ALU layout forces reverification to take place at the root level of the electrical representation. It is necessary to maintain an alternative decomposition of the electrical description that closely mirrors the layout hierarchy. This is an alternative created by splitting the root. Thus we have one electrical description that mirrors the behavioral/functional decomposition of the layout, and another that is based on a more natural decomposition of the electrical representation. Obviously the verification problem also exists for alternatives of a subtree within the same representation.

6. Structure of Design Files

Design files are highly interrelated collections of design data. At least five different kinds of data must be represented: 1) the design hierarchy for each design representation, 2) equivalences across design hierarchies, 3) composition information, i.e., how a subsystem is constructed from its constituent subsystems, 4) representational data, i.e., instances of representational primitives (e.g., mask geometries, transistors, gates, etc.) associated with individual design cells, and 5) interface specifications of how a cell interacts with surrounding cells in its representation. Representation of alternatives will be described in the next section.

The design hierarchy is represented as a tree of instances of design cells of a particular representation. Data that is identical across all instances of the cell are stored once, while dynamic data, such as state, are stored separately for each instance (see figure 3). Whether to choose a tree or a more space efficient directed acyclic graph depends on the representation and the tools available for its verifica-

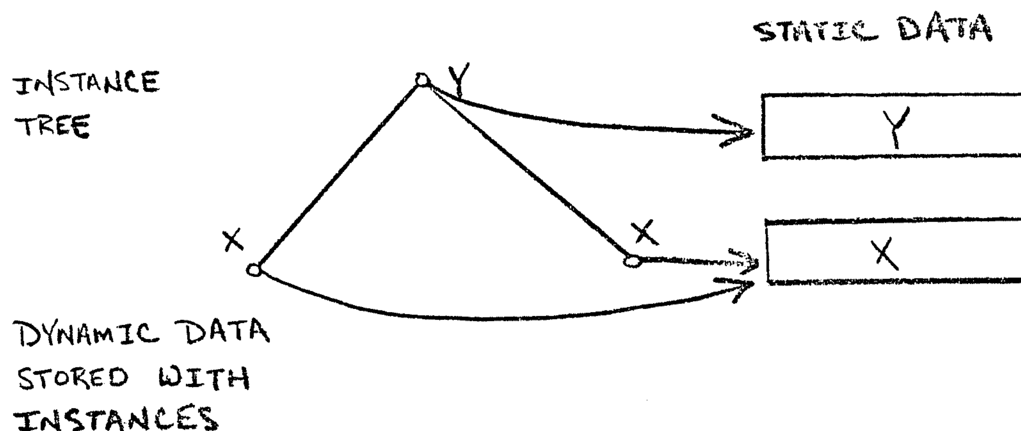

Figure 3 -- Static vs. Dynamic Data Representation

tion. For example, the validation aid for geometrical layout is a design rule checker, and since each placement of a cell contains identical geometries, a DAG representation can be used effectively. An instance tree must be used if a representation is verified through simulation, since cells with state must be represented individually. The design hierarchy has a node for each cell, represented by a record with links to the records of its immediately subordinate subsystems. It contains pointers to the static data files and stores (or points to) dynamic data.

Each record in the design hierarchy also has pointers to equivalent records in parallel hierarchies. If the same hierarchical decomposition is used across all representations, then a single hierarchy of records can be used, which is a composition of the individual representations. The description of how a subsystem is constructed from lower level subsystems is associated with the non-leaf design cells of the hierarchy. The details of the composition data differ according to representation. With geometries, the compositions are described by geometric transformations, i.e., translations, rotations, and mirrorings. This description is not appropriate for other representations, where interconnection among cells is the predominant form of composition. Input/output connection points are defined for each cell. The composition of cells is described by specifying instances and the explicit connection between them. Composition data is stored in a file associated with a cell's record in the design hierarchy.

We assume that representational data can be associated with any cell of a design (separated hierarchies may be desirable from a methodological standpoint, but are immaterial from the viewpoint of data management). A design cell's representational data is placed in a file and associated with its record in the design hierarchy. Its format is uninterpreted by the design data manager, but is chosen for conformance with the kinds of design tools expected to manipulate the representation. Typically, these files contain variable length records, with different representational primitives mixed in the same file. For example, boxes, flashes, and wires are stored in

the same file for the geometrical representation of a cell. The internal structure of the cell is chosen to be the most economical for the design tools which use it. For example, different geometries may be clustered within the file (e.g., all boxes, followed by all flashes, followed by all wires), or may be clustered by another criterion (e.g., all geometries on the polysilicon layer, followed by all the geometries on the diffusion layer, followed by all the geometries in the metal layer, etc.). Since most design tools accessing these files load the data into in-core data structures, more complicated structuring on secondary storage is probably not necessary. In keeping with a well-structured design approach, most cells will have few primitives associated with them. Thus, representation files are likely to be small.

The final piece of data stored with a node is the interface specification of its cell. The design considerations related to choosing the form of this data have already been described. The interface is an integral part of the representation of a cell, and could be associated with the cell's representation file (this is the approach used in CIF files). However, once the interfaces are supported explicitly, there will be new tools to manipulate the interface data. For example, it may be possible to derive the interface of a system from the interfaces of its subsystems in conjunction with the composition data. It will be more convenient to store the interface specification in its own data file. The complete data structure is shown in figure 4.

The implications of the above are that a design's files are highly interconnected, that there are a large number of separate files, and that individual files are small. This could cause problems for the design data manager. However, because of the functional decomposition of the design, designers will be working on subtrees that represent functional subsystems, and not a random collection of files. This is exploited by the data management component to limit the amount of control information it has to manipulate while design activity is in progress. A subtree is identified by its root, and accessing a root grants simultaneous access to all of its descendents. Certain interference problems among designers can be avoided if each follows a

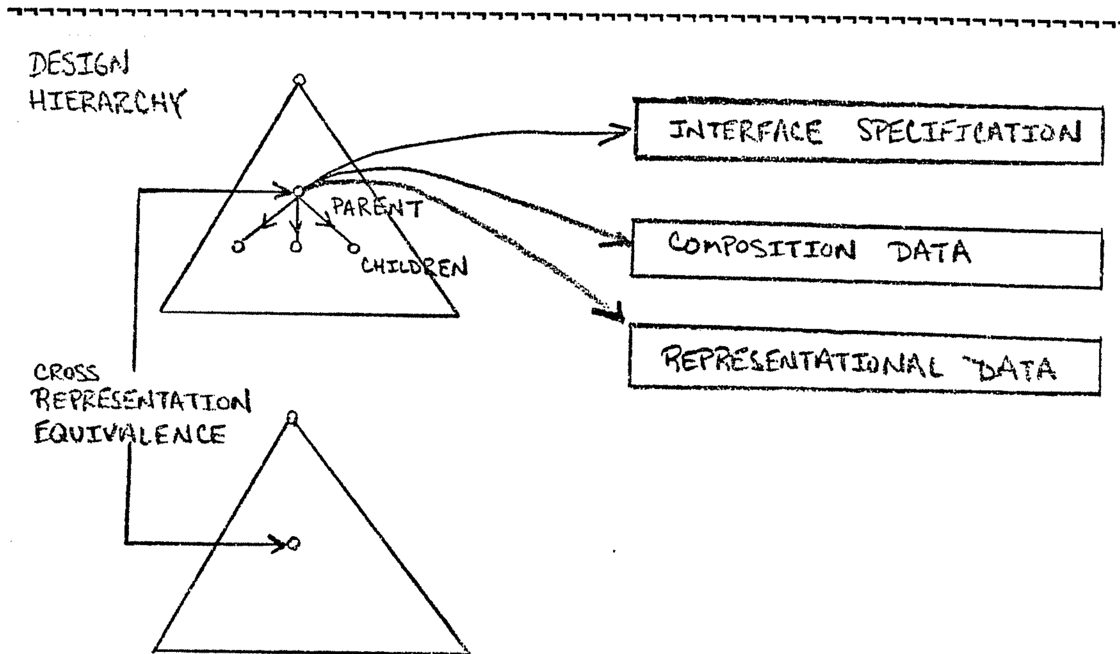

Figure 4 -- Structure of Design Files

protocol of traversing the design from its root to the root of the subtree of interested, leaving an indication of their visit at each node along the path. A consequence is that two designers can be updating the same design as long as they are in parallel and not in contained subtrees. Unfortunately, the ability to traverse between representations makes it necessary to associate this control information with every node in a subtree being updated, and not just its root (unless all representations have the same design hierarchy and accessing a subtree root implies accessing the data across all representations).

7. Design Versions and Alternatives

Design data is a precious resource that evolves over time. Old data is not discarded immediately, since it is needed for documentation and legal purposes, and because it provides valuable insights into the design procedures employed in creating the design. A *design version* is a collection of design files that represent a

consistent specification of a design at a point in time. A design typically evolves through several versions before it reaches a point where it can be released as a "finished" product. New features and other refinements are continually being incorporated into the design, resulting in new versions and releases. A *design alternative* is a hypothetical version, i.e., one that is not the official design. Alternatives are used to explore experimental solutions to a design problem. Recent old versions of a design are usually kept on-line, while very old versions are archived. A version mechanism can be made space-efficient by only recording data about design objects when they change across versions [KATZ82b, BOBR82].

We adopt the terminology of [BOBR82] to describe how design versions and alternatives can be implemented on top of design files. Although their concern is for artificial intelligence knowledge bases, the techniques apply equally well to design data files.

A design file is readable by all members of the design team. However, it is owned by a single designer (the "administrator") who is responsible for its contents and who is the only individual allowed to update it. Other designers can create private copies of the file, to which they can make changes, but these are not reflected in the public copy until explicitly incorporated by the administrator. Changes do not overwrite design data, but are appended to the end of the file instead. Private copies of public files provide a simple way to construct alternatives.

Versions and alternatives are implemented by dividing files into *layers*. The initial layer contains the original version of the file, the second layer contains new objects added to the file and new copies of old objects that have changed in creating the next version, etc. Each object is uniquely identified, making it possible to reference different copies of the same object across layers. For conventional design files, an object is a record.

Layers provide a very flexible mechanism for organizing design versions. They need not be searched from newest to oldest when looking for an object. Layers can be reordered or omitted from the search order. The first definition of an object found in the specified order is its version within that *environment.* An environment, existing for each designer on each design file of interest, is an index structure that maps unique identifiers into objects, taking account of a designer's desired layer ordering. An environment can also contain layers from the designer's private copy of the file, thus supporting multiple alternative of a design file.

For more efficient usage of storage, a design file's administrator can *summarize* a design file, replacing the file with the most up-to-date versions of all objects in the file. Since this could disrupt the views of designers who have formed alternatives over the file, they have the option of *freezing* it, whose effect is to make their own copy of the layers they need. Old copies can be discarded by *thawing* the file. Before summarization takes place, presumably at a consistent point in a design file's evolution, its administrator snapshots the file into an archive.

We have described the versions of a design file, but a design version is actually a collection of design file versions. It is specified by a set of environments over the constituent design files, including the design hierarchy file, composition files, representation files, and interface files. Thus it is possible to have a version of a microprocessor with a new ALU and an old barrel shifter.

Certain versions of a design are designated as releases, and these correspond to designs committed to implementation. A version cannot become a release unless all interface violations have been resolved.

8. Conclusions

A chip assembler is the data management system for a VLSI circuit design, providing an environment in which other design tools can be integrated. By storing information about versions, representations, design decompositions, and the constraints

across all of these, new verification aids for maintaining the integrity of a design are possible. The explicit representation of the design hierarchy simplifies the support for multi-level tools, while the integration of multiple design representations makes it possible to support mixed mode tools for the first time. A chip assembler also dramatically improves the documentation of a design.

The structure of design data files are highly interconnected. Design hierarchy, interface, representational, and composition data must all be represented in the design database. The proliferation of files and the sheer quantity of data will greatly stress the data management component of a chip assembler. Techniques are needed to limit the amount of control information that has to be handled for data management. Versions and alternatives can be supported on top of the design data files.

We are planning to build an experimental prototype chip assembler, providing many of the facilities described in this paper. We have already completed the implementation of a flexible database component for storing design data, as described in [KATZ82a].

9. Acknowledgements

This paper evolved from a series of discussions between the authors and our colleagues at Xerox PARC. Gaetano Borriello was substantially involved in all aspects of this paper, contributing key ideas and many editorial improvements. Alan Bell was involved in the discussions and made many suggestions. Danny Bobrow and Mark Stefik explained to us the mechanisms used for versions in their LOOPS system. The work was carried on within the VLSI Design Group at Xerox PARC, under the direction of Lynn Conway, where the second author spent a summer as a research intern. We were partially supported by NSF Grant MCS-8201860.

10. References

[BELL81] Bell, A., Stefik, M., Conway, L., "The Deliberate Engineering of Methodologies for Integrated System Design," Knowledge-Based VLSI Design Group Memo KB-VLSI-81-3, Xerox Palo Research Center, (Apr. 1981).

[BOBR82] Bobrow, D. G., Stefik, M., "The LOOPS Manual: A Data and Object Oriented Programming System for Interlisp," Knowledge-Based VLSI Design Group Memo KB-VLSI-81-13, Xerox Palo Alto Research Center, (Aug. 1982).

[EAST81] Eastman, C., "Database Facilities for Engineering Design," Proc. of the IEEE, V. 69, N. 10, (Oct. 1981).

[KATZ82a] Katz, R. H., "A Database Approach for Managing VLSI Design Data," Proc. 19th ACM/IEEE Design Automation Conference, Las Vegas, Nv., (June 1982).

[KATZ82b] Katz, R. H., T. J. Lehman, "Storage Structures for Versions and Alternatives," Computer Sciences Technical Report #479, University of Wisconsin-Madison, (July 1982).

[JOHA79] Johannsen, D., "Bristle Blocks: A Silicon Compiler," Proc. 16th Design Automation Conference, San Diego, CA., 1979.

[LOSL80] Losleben, P., "Computer Aided Design for VLSI," in *Very Large Scale Integration: VLSI*, D. F. Barbe, ed., Springer-Verlag, Berlin, 1980.

[MCWI78a] McWilliams, T. M., L. C. Widdoes, "SCALD: Structured Computer-Aided Logic Design," Proc. 15th ACM/IEEE Design Automation Conference, (1978).

[MCWI78b] McWilliams, T. M., L. C. Widdoes, "The SCALD Physical Design Subsystem," Proc. 15th ACM/IEEE Design Automation Conference, (1978).

[MUDG80] Mudge, J. C., Peters, C., Tarolli, G. M., "A VLSI Chip Assembler," in *Design Methodologies for Very Large Scale Integrated Circuits*, Nato Advanced Summer Institute, Belgium, 1980.

[MUDG81] Mudge, J. C., "VLSI Chip Design at the Crossroads," in *VLSI 81: Very Large Scale Integration*, J. P. Gray, ed., Academic Press, London, 1981.

[NEWK81] Newkirk, J., et. al., "The Stanford nMOS Cell Library," Stanford Information Systems Laboratory Report No. 001, (July, 1981).

[STEF82] Stefik, M., et. al., "The Partitioning of Concerns in Digital Systems Design," Proc. Conf. on Advanced Research in VLSI, Cambridge, MA, (Jan. 1982).