Comparing structurally different views on a VLSI design

Spreitzer, Michael Joseph, Ph.D.

Stanford University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

COMPARING STRUCTURALLY DIFFERENT VIEWS OF A VLSI DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Michael Joseph Spreitzer
June 1989

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

Mark Horowitz
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Forest Baskett

Forest Baskett

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

John L Hennessy

John Hennessy

Approved for the University Committee on Graduate Studies:

Elizabeth Closs Traugott

Dean of Graduate Studies

Abstract

One of the major problems of VLSI design is coping with the quantity and complexity of the design data. The leading solutions use 'divide-and-conquer' techniques. Two different ways of dividing are popular: division by a structural hierarchy, and division into various levels of abstraction (a *view* is a description at a particular level of abstraction). VLSI designs are so large and complex that both divisions are needed, which raises a question: should all the views of a design use the same hierarchy? This question is currently controversial. This dissertation, while not presuming to settle that question, argues in favor of allowing the views to have different hierarchies, and addresses a problem that is complicated by differences in hierarchy. That is the comparison problem, which has two parts: (1) verify consistency between alternate views, and (2) determine the correspondence between the design entities of those views. Previously existing techniques either work on flat views (that is, ones not divided into a hierarchical structure), or can only compare views that have essentially identical hierarchies. Of course any hierarchical description can be flattened, but flattening is disadvantageous for a number of reasons. The most important reason is that flattening can exponentially increase the size of the description. Many comparison techniques require an amount of time that grows exponentially with the size of the circuit descriptions. Flat comparison techniques are thus impractical for VLSI designs.

This dissertation introduces a new comparison method, *Informed Comparison*, which neither requires the views to have essentially identical hierarchies nor flattens the views. Informed Comparison requires the designers to maintain a *key*, which is a description of the intended relation between the hierarchies of the views. Informed

Comparison first *reconciles* copies of the views by applying hierarchy transformations, under the guidance of the key, until the copies have essentially identical hierarchies. Informed Comparison then finishes with a *base comparison*, which can use any existing (or new) hierarchical technique that assumes essentially identical hierarchies. Informed Comparison thus has many of the good features, including good asymptotic performance, of other hierarchical methods.

Several characteristics of Informed Comparison depend on the repertoire of transformations available to the reconciliation step and on the base comparison technique. This dissertation illustrates those dependencies with two examples of Informed Comparison.

Acknowledgments

I have been aided and abetted by a four advisors. Forest Baskett got me started, with a look at logic extraction and an introduction to Xerox PARC. He helped me through the difficult task of choosing a thesis project. Manolis Katevenis and John Hennessy helped me along the way. Mark Horowitz deserves much credit for helping me to bring my project to a conclusion.

I wish to thank the Fannie and John Hertz Foundation for the generous support they give to their fellows. The Computer Science Laboratory of the Xerox Palo Alto Research Center also deserves credit for providing both financial support and a highly stimulating and productive environment.

My writing was much improved, thanks to the reviews of several people. Mark Horowitz and Jay Showalter told me of the importance of helping my readers to keep their bearings. Rich Horvath tried to show me how to eliminate dead wood. Stefan Demetrescu, Dan Swinehart, Rick Barth, Bertrand Serlet, and my wife Chris were also helpful.

I am indebted to many people, especially Chris, for patience, understanding, and encouragement during this long project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An *Integrated Circuit* is an electronic circuit fabricated in a small 'chip' of some semiconducting material. As this technology has developed, the sizes of circuit elements have continually shrunk. This has increased the speed of circuit operation, as well as the amount of circuitry that can fit on a given area. At the same time, the size of the chip that can be practically fabricated has increased. These trends have greatly increased the complexity of the circuits being designed and fabricated. The acronym *VLSI*, for *V*ery *L*arge *S*cale *I*ntegration, refers specifically to these complex circuits. Because of their complexity, it is quite difficult to quickly produce working VLSI designs. This dissertation introduces an idea that makes it easier to design complex circuits.

Designing a large complex IC is an enormous task. Designers cope with this complexity by using divide-and-conquer strategies. In fact, a single IC design is often divided up in several different ways. The central idea of this dissertation concerns keeping some coherence across those different ways of dividing.

There are two major ways of dividing VLSI designs. One way is to divide a circuit into interacting sub-circuits, and then divide each sub-circuit into interacting sub-sub-circuits, and so on. These divisions define the *hierarchy*, or the *structure*, of the design. The other way to divide the design is by issues. A VLSI design must address several different kinds of issue: solid state, electrical, geometrical, and logical. There are various *levels of abstraction* at which IC designs are described;

each level of abstraction focuses on one or a few kinds of issues. Examples of these levels of abstraction are the solid-state device, the masks used for fabrication, the electrical circuit, and the logical circuit. Let us refer to a description, at some level of abstraction, of a circuit as a *view* of that circuit.

Most designs describe the whole circuit at more than one, but not every possible, level of abstraction: often there is both a mask and a logical view of the whole circuit, but rarely is there a solid-state view of the whole circuit. Each view of a VLSI circuit is itself so large that it is hierarchically divided. Should all the views of a circuit use the same hierarchy? This is controversial.

There is a strong reason to allow each view to use a different hierarchy: it improves the clarity of the views. Recall that the essence of a hierarchy is the division of a part into interacting sub-parts. The fewer and simpler those interactions, the greater the clarity of the view. But the interactions depend on the abstraction employed; thus, a hierarchy that is good for one view many not be good for another.

But a design has more than one view, and if the relationship between those views is sufficiently tortuous, the overall clarity of the design suffers. A part in one view might not correspond to any one part in another view. This makes it difficult to test or maintain consistency between views. The reason for forcing all views of a circuit to use the same hierarchy is that it avoids these difficulties.

The consistency problem arises because, although the various levels of abstraction focus on different issues, the information content of the views is not completely disjoint. For example, both an electrical view and a logical view describe the behavior of the circuit. Those behaviors will be in different terms: electrical circuit behavior is cast in terms of continuous functions of time for voltages and currents, and logical circuit behavior is cast in terms of Boolean functions of discrete points in time for logical variables. Even though they are in different terms, the two behaviors can be compared. Logical TRUE and FALSE can be associated with ranges of voltage, and the discrete time points of the logic behavior can be associated with moments of time in the electrical behavior. Thus, we can ask whether an electrical circuit and a logical circuit have consistent behavior.

One way to answer the behavioral consistency question between an electrical and a logical circuit is to simulate both, and check whether the voltage functions and logic variables are consistent, as outlined above. To do this requires knowing which voltage function is supposed to correspond with which logical variable. This is an example of why it is useful to know the correspondence between the entities of the two views. Another example is a simple cross-referencing function. It would be nice if a designer could point at an entity in one view, and get back an indication of what that corresponds to in a different view. There are many other examples. Any task that needs to use more than one view needs to know the correspondence between the entities of the views.

Most existing techniques for maintaining the correspondence between entities in different views simply insist that all the views of a design use the same hierarchy, which makes for a direct correspondence between design entities. While the other techniques are more sophisticated, they are not able to accurately represent the correspondence across general hierarchy transformations. Most existing techniques for assuring consistency either (1) insist that all the views use the same hierarchy (or allow only certain minor differences in the hierarchies), and take advantage of that sameness, or (2) can only compare flat views. A flat view is one that does not employ hierarchy: it simply describes one large set of interacting indivisible parts. The hierarchical techniques have significantly greater performance than the flat ones. The hierarchical techniques also have greater precision and flexibility, except for that one requirement of sameness of hierarchy.

This dissertation introduces a new hierarchical comparison method, called *Informed Comparison*, that does not require the views to use essentially identical hierarchies, yet has many of the benefits of other hierarchical techniques. In preparation for an Informed Comparison, the designers must document the intended relation between the hierarchies of the views. This information is called the *key*. An Informed Comparison of two views is done in two stages: first the *reconciliation* of the views' hierarchies, and then the *base comparison* of the reconciled views. The base comparison can be any existing (or new) hierarchical technique that requires essentially identical hierarchies. The reconciliation consists of applying hierarchy transformations to copies of the views, under the guidance of the key, until their hierarchies are similar enough for the base comparison. An Informed Comparison can both verify consistency and determine the correspondence between the entities of two views. The power of Informed Comparison depends on both the base comparison and the repertoire of transformations available for the reconciliation.

Informed Comparison cannot remove the tension between the clarity of the individual views and the clarity of the relationships between the views. Informed Comparison's contribution consists of enabling designers to use whatever hierarchies maximize overall clarity without paying the penalties associated with flattening. While there are other difficulties in the comparison problem (such as coping with the differences in level of abstraction), and not every design benefits greatly from hierarchical division, Informed Comparison is useful because it is an efficient technique for coping with hierarchy differences.

The rest of this dissertation is organized as follows.

- Chapter 2 presents the comparison problem in detail: why it is desirable to use alternate views at different levels of abstraction with different hierarchies, and why it is desirable to test consistency and determine the correspondence between the entities of two views. Chapter 2 also presents existing comparison techniques.

- Chapter 3 introduces the method of Informed Comparison. Some of the characteristics of Informed Comparison depend on the base comparison technique and the repertoire of reconciliation transformations; these dependencies are discussed in general in this chapter.

- Chapter 4 presents one particular version of Informed Comparison, called PWCoreLichen. It is a program in a design aids suite created and used at Xerox PARC. PWCoreLichen has a particularly simple base comparison and limited repertoire of reconciliation transformations.

- Chapter 5 presents a more general Informed Comparison, the Lichen and MIPS-X study. Lichen is an experimental program for applying reconciliation transformations, with a general repertoire. MIPS-X is a microprocessor designed at

Stanford. This chapter presents an Informed Comparison of two views of part of MIPS-X. This comparison requires a new method, called *Comparison Modulo Boring Components*, for the base comparison.

- Chapter 6 presents a review of Informed Comparison, a discussion of its problems and limitations, and suggestions for further research.

- Appendix A presents Lichen's repertoire of transformations and the key used in the MIPS-X comparison.

Chapter 2

Background

Informed Comparison is a method for comparing alternate views of a VLSI design, where those views use different levels of abstraction and different hierarchies. This chapter provides the background necessary to understand and evaluate Informed Comparison. The chapter begins by presenting the reasons for using alternate views at different levels of abstraction and with different hierarchies, and then describes the need for comparing those views. Comparison involves both checking consistency between the views and finding the correspondence between the parts of the views. Finally, some existing techniques for comparison are presented, because Informed Comparison makes use of them and will be evaluated against them.

2.1 The Problem

2.1.1 Ways to Divide

One of the problems in VLSI design is coping with the great quantity and complexity of the data involved. With current technology, chips containing a million transistors are regularly made. Some chips, such as memories, can have relatively simple structure, because they consist mainly of a large repetition of a small pattern; other chips, such a microprocessors, have much more complex structure. To cope with the large amount of information, designers and programs use divide-and-conquer strategies. In fact, two different ways of dividing, by abstractions and by hierarchy, are popular.

Dividing by Abstractions

In this way of dividing up design information, a number of different abstractions are employed, each of which focuses on only some of the concerns of the VLSI design problem. Examples of such abstractions are solid state devices, fabrication masks, electrical circuits, Boolean equations, and register-transfer machines.

Dividing design information by abstractions is good practice because it allows the designer attacking only some aspects of the problem to focus on just those aspects. For example, the designer can develop a set of Boolean equations before designing the electrical circuits that will implement them. It is helpful to be able to think about the logic of the problem separately from the details of the electrical implementation.

In this dissertation, the word *view* is used to mean a description in a particular abstraction.

Hierarchies

In this way of dividing up design information, the chip is divided into interacting sub-parts, and then those sub-parts are divided into interacting sub-sub-parts, and so on. The value of this is also that it lets the designer focus on only some of the design problem at a time. Dividing by hierarchy differs from dividing by abstraction in the way the locus of attention is defined: hierarchies organize by structural relations, abstractions organize by conceptual relations. Hierarchies also support a use/definition dichotomy, which makes hierarchical descriptions more succinct than flat ones.

A number of different formulations of hierarchy have been used; this dissertation uses the following one. The basic division into parts and sub-parts is described with *cell types* and *cell instances*; the interactions between parts are described with *wires*, *ports*, and *connections*. The type/instance dichotomy for cells makes it possible to describe, design, and analyze multiple occurrences of the same pattern succinctly. A cell type is either *atomic* (has no internal structure) or *composite*. Each composite cell type *contains* a number of cell instances and wires; each cell instance represents

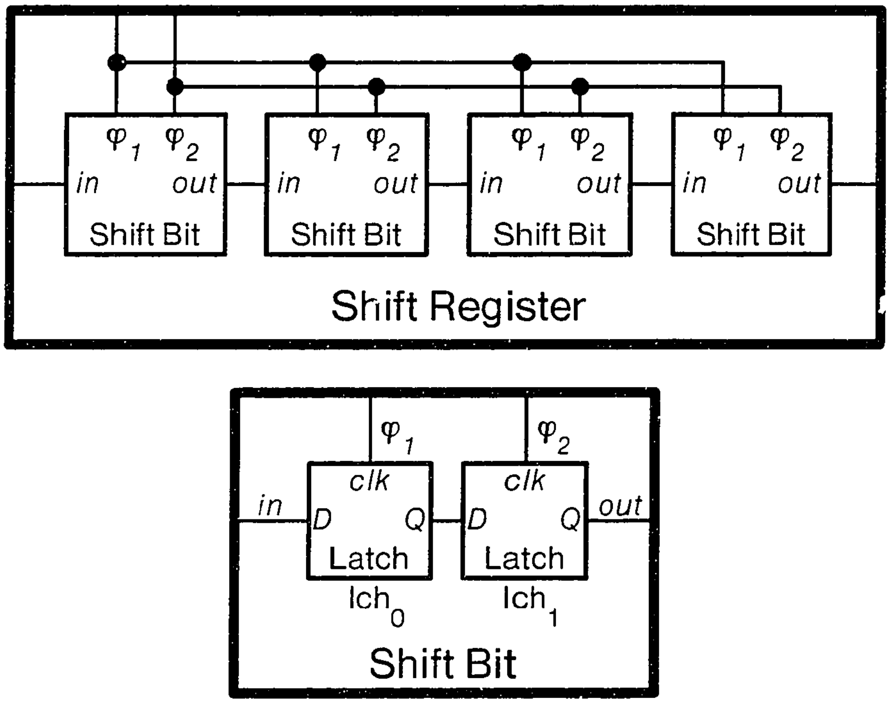


Figure 2.1: A Fragment of a Hierarchy

a use of the cell type it *instantiates.* A view has a distinguished composite cell type, called its *root*, that stands for the whole chip; the other cell types describe parts of the chip. Every cell type has an explicit interface, which is a set of ports. Each connection is between a wire and a port at a *site*, where a site may be either a cell type or cell instance. The cell instances at which a wire may be connected are only those contained in the same cell type that contains the wire—that is, a connection cannot 'skip' levels of hierarchy; this is what it means to have explicit interfaces on the cells. The only cell type at which a wire may be connected is the cell type that contains the wire; such a connection indicates the fact that the port to which the wire is connected *exports* the wire.

See Figure 2.1 for an example of part of a hierarchy. It shows the definitions of two cell types, *Shift Register* and *Shift Bit.* The *Shift Bit* cell type has two cell

Wire	Port	Site
in	*in*	*Shift Bit*
in	D	lch_0
mid	Q	lch_0
mid	D	lch_1
out	Q	lch_1
out	*out*	*Shift Bit*
φ_1	φ_1	*Shift Bit*
φ_1	*clk*	lch_0
φ_2	φ_2	*Shift Bit*
φ_2	*clk*	lch_1

Table 2.1: Connections in the *Bit Shift* Cell Type

instances (both of the same type *Latch*), five wires, and four ports. The wires of *Bit Shift* participate in 10 connections, which are listed in Table 2.1.

A few more terms concerning hierarchical structure will prove useful. The cell instances contained in a cell type are called its *subcells*. A genealogical cast is sometimes applied to cell structure. The subcells of a cell type are also called its *children*, and a cell type is considered to be a child (the only child, in fact) of each of its instances. Cell type or instance (hereafter, simply *cell*) A is considered to be an *ancestor* of cell B iff either $A = B$ or A is a parent of some cell C that is an ancestor of B. Two cells that share a parent are called *siblings*. *Clipping a hierarchy* is an operation that discards details. It changes some composite cell types into atomic ones, forgetting their decompositions into interacting sub-parts. Cell types that are no longer used are forgotten entirely. Clipping can be done to various degrees. A particular clipping is specified by a *frontier*, which is a set F of cell types such that every cell type in the view has at least one ancestor or descendant in F. A frontier conceptually divides the cell types of a view into three disjoint sets: (1) the frontier F itself, (2) the cell types *above* the frontier, and (3) the cell types *below* the frontier. Since a cell type can have both ancestors and descendants in F, the concepts of *above* and *below* need to be defined carefully. A cell (type or instance) not in F is above F either if it is the root cell type of the view or if that cell has a parent that is above F. Cells neither above F nor in it are below it. The cell types above F are unaffected by the clipping;

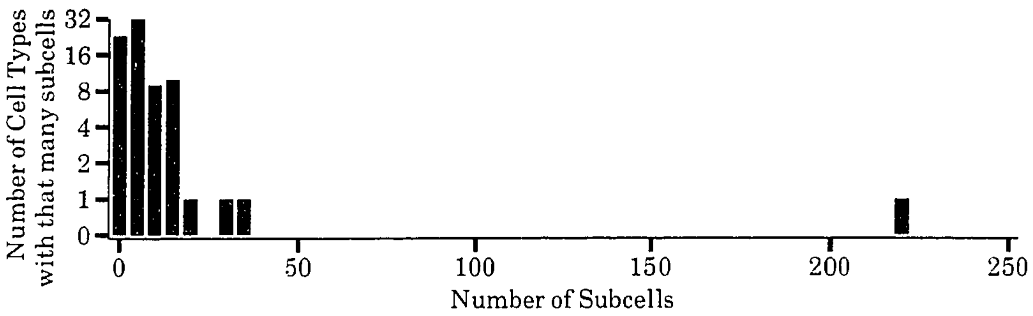


Figure 2.2: Distribution of MIPS-X pc Layout Cell Types by Number of Subcells

the cell types below F are the ones forgotten. There is no point in including in F a cell type that is below F.

A flat description can be considered a degenerate hierarchical one, wherein the root cell type is the only composite cell type. *Flattening* is the process of removing hierarchical structure, and can be done to various degrees. Complete flattening removes as much hierarchy as possible, leaving the root as the only composite cell type. One partial degree of flattening is replacing every instance of a cell type with its first level of decomposition, suitably interconnected of course; this is called *flattening out* that cell type. A lesser degree of flattening is replacing only one cell instance with its type's first level of decomposition; this is called *flattening out* that cell instance.

The type/instance dichotomy makes hierarchical descriptions compact. For example, Figure 2.2 shows a histogram of the cell types in the layout of the program counter unit of MIPS-X (a 32-bit microprocessor that will be introduced in more detail later), according to number of subcells. More than 95% of the composite cell types have fewer than two dozen subcells; only one, a PLA, has more than about three dozen. The whole hierarchical description uses 705 cell instances. When completely flattened, the description uses about 6740 cell instances.

The distinguishing features of this formulation of hierarchy are two: the use of a cell type/instance dichotomy, and the use of explicit interfaces on the cells. The type/instance dichotomy makes the descriptions compact, and the explicit interfaces on cells facilitate the hiding of information.

2.1.2 Methodological Choices

The previous section presents two different ways to divide up the information in a VLSI design. The quantity and complexity of the information in a VLSI design are so large that both techniques must be used together in order to make the design task manageable. How should these two different techniques be coordinated? This is an open question of design methodology. This dissertation does not purport to settle this question—only to address a problem that arises in the context of one popular answer. This section sets forth, and argues for, two methodological principles that form that context.

Use Multiple, Independent Views

One methodological principle is that each design should use multiple, independent views. Two views are independent when a designer can work on (i.e., edit, analyze, and so forth) one without having to work on the other simultaneously (note that this definition is not directly concerned with how the design information is stored in files, databases, or whatever). The motivation for this principle is that it allows the abstractions to perform their intended function—focusing attention on only some of the design information.

Use Different Hierarchies in Different Views

The other methodological principle is that the views of a design need not all use the same hierarchy. This is a controversial rule, and both sides talk about clarity. The argument against this rule is that when the views use different hierarchies, the overall clarity of the design suffers. The argument for this rule is that forcing all the views to use the same hierarchy reduces the clarity of some or all of the views, which in turn reduces the overall clarity of the design. Recall that hierarchy involves the division of a part into interacting sub-parts. The best division minimizes the number and complexity of the interactions—and the interactions of interest vary from one view to another. The freedom granted by this rule allows the designers to maximize the clarity of the individual views just enough to maximize the overall clarity of the

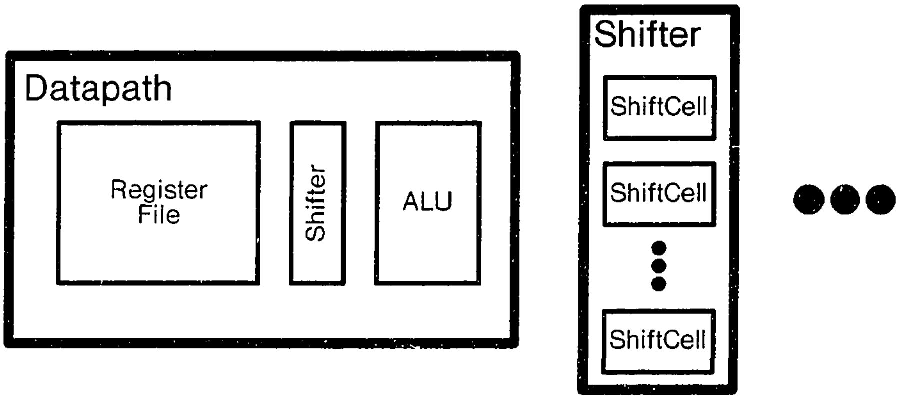


Figure 2.3: Function Slicing

design. Some examples of desirable differences in hierarchy between views follow.

A familiar example arises in the design of the canonical microprocessor datapath: the schematic view divides the datapath into a register file, a shifter, and an ALU, each of which is, say, 32 bits wide; but the layout view uses a 32-fold replication of a bit-slice cell, which contains one bit's worth of each of the three major parts. Figure 2.3 illustrates the function-sliced hierarchy, while Figure 2.4 illustrates the bit-sliced one. For the schematics and other high-level views, the function-sliced hierarchy is best, because it most clearly exposes the function of the circuit; for the layout, the bit-sliced hierarchy is best, because it focuses on the most interesting layout problems—the interaction of adjacent bit cells from the different major parts.

A number of cases of desirable differences in hierarchy arise because there are special considerations to be made in layout that should not disturb the clean organization of more abstract views. A very important layout consideration is minimizing area. One way to do this is to share features between adjacent cells; sometimes it is best to use mirroring and intermediate levels of structure to accomplish this. An example is shown in Figure 2.5. This example concerns a horizontal array of 32 inverter cells. When mirrored in the horizontal dimension and overlapping properly, adjacent inverter cells share the power lines and contacts, saving a significant amount of area. To accomplish this in the layout hierarchy, there is an extra intermediate level of

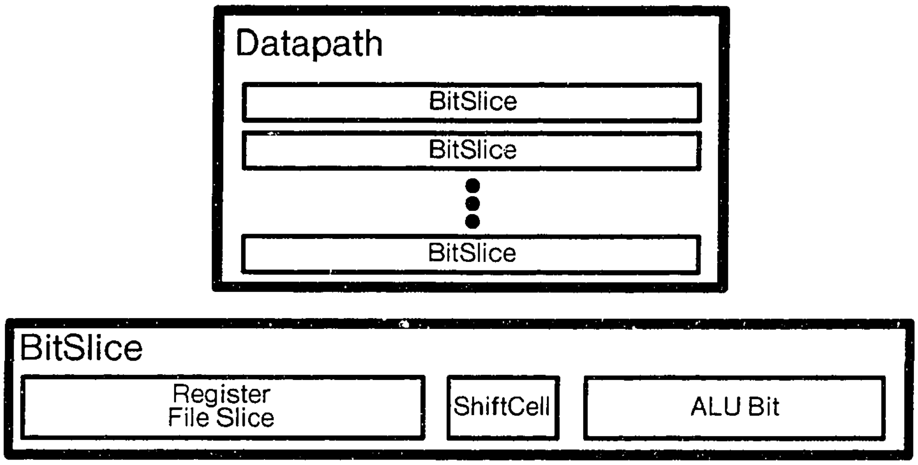
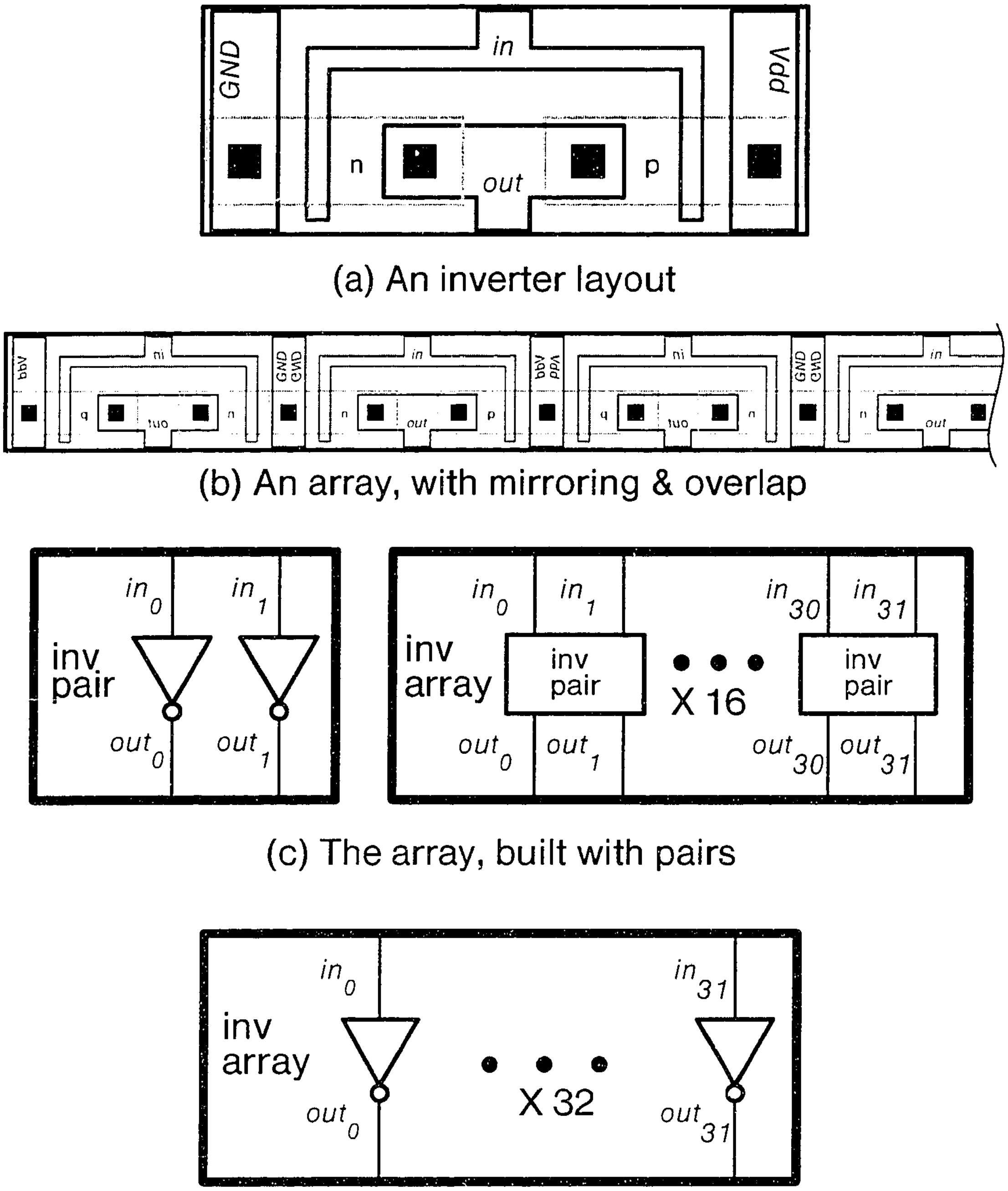


Figure 2.4: Bit Slicing

structure—the pair cell, which contains two inverters, one of which is mirrored; the pair cell is replicated 16 times to make the whole array. In the higher levels of abstraction, the mirroring of every other element cell in the array is uninteresting (and perhaps not even expressible); thus the pair cell serves no purpose, and the whole array is best described as a simple array of 32 inverters.

Another layout consideration is how to send signals down long wires. This is particularly vexing when the wire is long because it traverses a large array. In this case the array structure is disrupted by the need to do something every few elements to restore the strength of the signal traveling down the long wire. Figure 2.6 shows an example: here a control signal is routed through an array on the polysilicon layer, because it is used as the gate of some transistors; since polysilicon is not a very good conductor, the signal is also routed through the array in metal. There is a contact between the polysilicon and the metal every few elements; the contact is not put in every element because that would cost more area. The layout hierarchy would probably use an extra intermediate level of structure, as for the mirroring of alternate elements, to express the periodic appearance of a contact. At higher levels of abstraction the contact is not even expressible, and having the extra intermediate level of structure would just be a nuisance.



(a) An inverter layout

(b) An array, with mirroring & overlap

(c) The array, built with pairs

(d) A simple array

Figure 2.5: Use of Mirroring and Extra Structure to Share Features

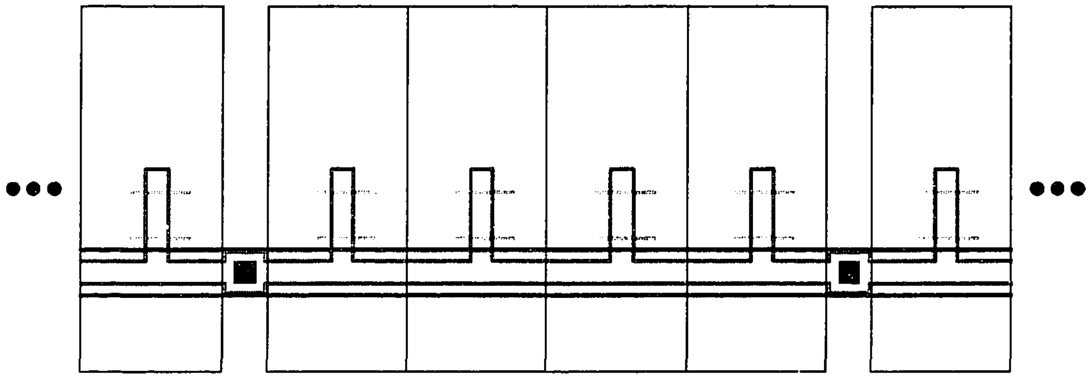


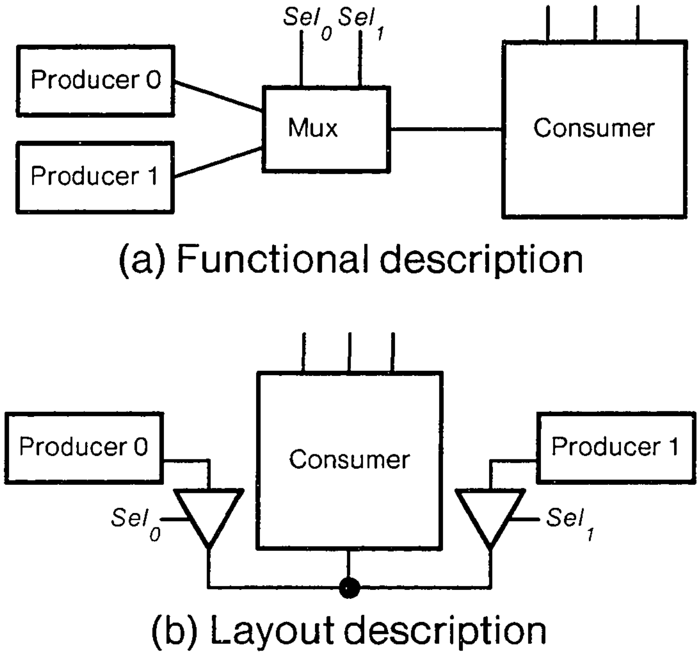
Figure 2.6: Periodic Contacts in an Array

Another way to send a signal through a large array is to insert buffers every few elements. Again, the layout hierarchy could use extra intermediate structure. In this case the buffers could appear at some of the higher levels of abstraction—specifically, at the electrical and switch-level. But at even higher levels—Boolean and above—the buffering is not significant. This is a case where two views of the same circuit should differ not only in hierarchical structure, but in flat structure as well.

Another layout consideration is the need to align the hierarchy with the geometry. Most hierarchical layout systems (the analysis tools play a significant role here) require that: (1) cells have rectangular areas, (2) the area of a cell be a superset of the area of every child cell, and (3) the areas of siblings usually be mostly disjoint. Thus, parts that are in the same layout cell will tend to be close to each other, and vice versa. The geometric relations between circuit parts can be different from the logical relations because of the need to minimize things like the area of the circuit and the lengths of critical wires. A layout hierarchy thus has geometric reasons to differ from the best hierarchy for a more abstract view.

A simple case of this occurs in the MIPS-X design: in the functional view, a latch appears in the execute section, where it logically belongs; in the layout view, the latch appears in the instruction register, where it fits better.

A more complicated case occurs in the register file in MIPS-X, and it goes, in



(a) Functional description

(b) Layout description

Figure 2.7: Split Multiplexor

essence, as follows (see Figure 2.7). In the functional description there is a multiplexor, which takes two inputs from two producer cells, chooses one according to some control signals, and sends it on to a consumer cell. In the layout, for reasons having to do with routing other signals, the geometrical arrangement places the consumer of the mux output between the two producers of the mux inputs. In this situation it is very advantageous to implement the multiplexor by two tri-state drivers, separated and placed as shown in the figure. In this layout, there is no point in making a cell for the whole multiplexor; in the functional description, using the multiplexor instead of the two tri-state drivers is clearer.

An even more vexing and common case concerns the implementation of Boolean functions. In a Boolean view, simple functions like *AND* are leaves—they have no internal structure, as far as that view is concerned. But in the layout, those Boolean functions are not leaves—transistors are. Each simple Boolean function is implemented by a few transistors and some wiring. Often the transistors implementing a

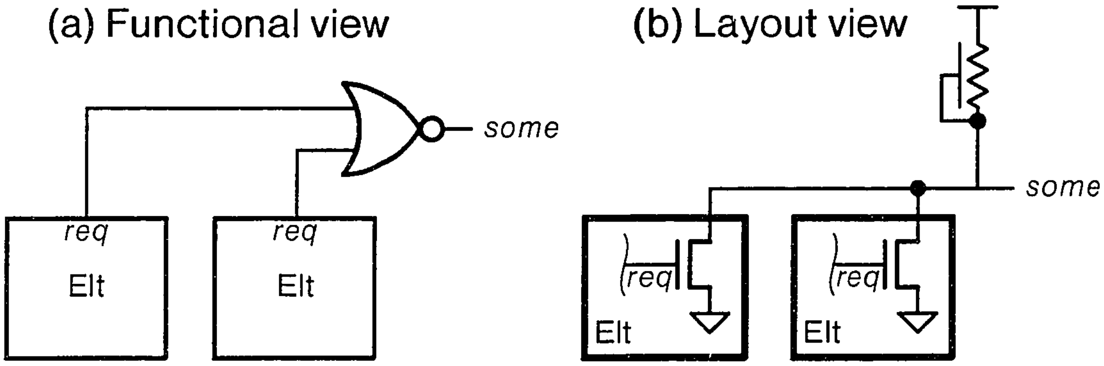


Figure 2.8: Scattering of Transistors Implementing a Boolean Operator

single simple Boolean function are scattered throughout a few cells (see Figure 2.8 for an example); also the transistors implementing several simple Boolean functions commonly are grouped together into one cell. A PLA is a structured example of the latter; there are also many unstructured examples. In these examples, some leaves of the high-level view have no corresponding cell in the low-level view. These are thus examples of differences that cannot be reconciled simply by flattening.

In summary, the different views focus on different interactions, and this means that different hierarchies are sometimes preferable. Other work that uses alternate views with different hierarchies can be seen in [Blackburn85] [Blackburn88] [Katz86] [Parker84] [Sequin83] [Walker85] [Walker87] [v.d.Wolf88]. The differences in hierarchy complicate the comparison problem, as will be seen in the next section. However, with the use of Informed Comparison, the comparison problem is not so difficult that it should preclude the use of different hierarchies.

2.1.3 The Comparison Problem

The preceding sections explain why it is desirable to use multiple views, at different levels of abstraction and with different hierarchies, in a VLSI design. In this methodological context, two related problems arise: testing consistency between the views, and discovering the correspondence between the entities used in the views.

This section explains exactly what these problems are and why they arise.

Consistency

Although two alternate views used in a VLSI design are at different levels of abstraction, there is some overlap in their information content; for this reason, it is important to be able to test for consistency of this common information. For example, consider an electrical view and a Boolean view of the same chip. Although the electrical view focuses on issues not even present in the Boolean view, the behavior of the electrical circuit should be consistent with the behavior of the Boolean description. Although the electrical behavior is more detailed, there should be some way of abstracting from it a simple Boolean behavior that matches the behavior of the Boolean view.

Note that the problem is not simply to determine whether two views are consistent; if they are not consistent, it is helpful to identify the causes of the inconsistency. For example, if two distinct wires in one view are accidentally merged together in another view, the designers would benefit from an indication of that situation.

Correspondence of Entities

When multiple views are used, design tasks that involve more than one view involve knowing which entities[1] in one view correspond to which entities in another view. For example, consistency checking is often given some of this correspondence, and then discovers more of it in the course of doing the comparison. For another example, a designer revising the design needs to know parts of the correspondence in order to make a consistent revision.

When the views follow identical hierarchies, the correspondence between their entities is simple: each entity in one view corresponds to exactly one entity in the other view (unless one view isn't as structurally detailed as the other or the entity is concerned solely with issues that aren't addressed in the other view, in which case it corresponds to nothing). When the views use different hierarchies, the correspondence is complex. For an example, recall the different hierarchies presented in Figure 2.5.

[1] Entities include the structural ones (cell types and instances, ports, and wires), and possibly also some non-structural ones.

Consider the left inverter in the *inv pair* cell; what in the functional view corresponds to it? The most obvious answer is the even-numbered inverters in the simple array—but that is not the whole story. It would be more accurate to say that the left inverter of the 0^{th} *inv pair* instance in the pair array corresponds to the 0^{th} inverter in the simple array, and that the left inverter of the 1^{st} *inv pair* instance in the pair array corresponds to the 2^{nd} inverter in the simple array, and so on. For an even more complicated example, recall the hierarchy difference illustrated in Figures 2.3 and 2.4; both the *Shifter* and the *BitSlice* have no corresponding cell type in the other view. However, in the style of the previous example, *paths*—from the *Datapath* to the *Register File Slice*, the *ShiftCell*, and the *ALU Bit*—can be used to give a precise correspondence between the two views. Also as before, vague indications can be given of the correspondences of problematic entities. An even more problematic example occurs with the difference illustrated by Figure 2.8; the best that can be done is to indicate that the one NOR gate of the functional description corresponds to the paths to its implementing transistors. Trouble of a different kind arises when multiple cell types of one view all correspond to the same cell type of the other view. This could happen, for example, when there are several different layouts for latches (because different driving strengths are required) but only one latch cell type in the functional description. Note that these examples illustrate two different senses—one conjunctive, one disjunctive—in which one entity can correspond to many. Of course, these two senses must be kept distinct.

The full correspondence between design entities is quite detailed. Since it is a large quantity of information, this suggests that most of it should be generated by a program. The consistency checker is an obvious candidate. For this reason, and because of their logical closeness, consistency checking and discovery of the correspondence between entities are put together to make the *comparison problem*.

2.2 Existing Techniques for Consistency Checking

The previous section sets forth the problem that informed comparison solves: comparison of alternate views at different levels of abstraction and with different hierarchies. This section discusses several known consistency-checking techniques. Some of these techniques don't solve exactly the same problem, either because they can't compare views at different levels of abstraction, or because they can't compare views that use different hierarchies. The remainder of these techniques don't use hierarchy well (they flatten it out). Nevertheless, understanding these techniques provides important background for understanding Informed Comparison. The existing techniques cf comparison fall into three groups: those that compare flat views at the same level of abstraction, those that address the issues of comparison between views at different levels of abstraction, and those that seize some of the opportunities offered by hierarchically structured views.

2.2.1 Flat, Same-Abstraction Comparison

The techniques for flat same-abstraction comparison fall into three classes: simulation, algebraic comparison, and structural comparison. The variety provides different trade-offs between speed, completeness, and soundness. There cannot be one technique that is fast (runs in less than exponential time), complete (able to verify the consistency of any two consistent views at the same level of abstraction—no false negatives), and sound (never claims two inconsistent views to be consistent—no false positives), because comparison problems are hard. For example, comparing two Boolean functions to see if they produce the same result for every combination of inputs is NP-complete,[2] which means that all solutions take an amount of time (in the worst case) that grows exponentially with the size of the problem (unless someone proves that $P = NP$).

[2] This comparison is easily shown to be equivalent to the Boolean satisfiability problem, which is well known to be NP-complete [Hopcroft79, Theorem 13.1].

Simulation

In simulation techniques both views are simulated against consistent stimuli, and their responses are compared. These techniques have the advantage of being relatively simple to implement—in fact, it is even feasible to implement simulators in hardware [Pfister82] [Beece88]. A serious issue in simulation is its soundness: inconsistencies may be overlooked if the right stimuli are not tried. To assure that the two views are really consistent requires extensive simulation. For example, two purely combinational[3] Boolean circuits of I inputs must be tested against 2^I input patterns.[4] When the circuits to be compared have state (which most do), another complication arises: the output may depend on past inputs as well as the present ones. The number of input patterns that must be applied to do a sound comparison depends on how much analysis of the circuits is done. The following techniques illustrate this, while making two simplifying assumptions: (1) the circuits are synchronous, with one clock (see Figure 2.9), and (2) the circuits should have consistent inputs and outputs on each cycle. The term *test vector* is used to mean the pattern of inputs for one clock cycle.

The minimum analysis that enables a sound comparison is counting the number of state bits. When comparing two finite state machines with M state bits each, the shortest sequence of test vectors that exposes an inconsistency may be as long as $2^M - 1$ or longer. Thus, a sound comparison by simulation requires applying all sequences of test vectors that are $2^M - 1$ long. If the machines each take I input bits, there are $2^{I \times (2^M - 1)}$ such sequences. Even for very modest numbers of inputs and state bits, this is wildly impractical.

The following techniques place a restriction on the circuits: they must both have the same number of memory bits, they must both go through the same number of states, and they must both use the same encoding of states. As Figure 2.9 makes clear, when this restriction holds it is necessary only to compare the purely combinational

[3] A combinational circuit is stateless—the outputs are a function of nothing more than the current inputs.

[4] The number of input patterns can be reduced by using 3-valued logic [Stabler87], but this increases the cost of simulation [Chang87]; since this is equivalent to Boolean satisfiability, there is no fast solution.

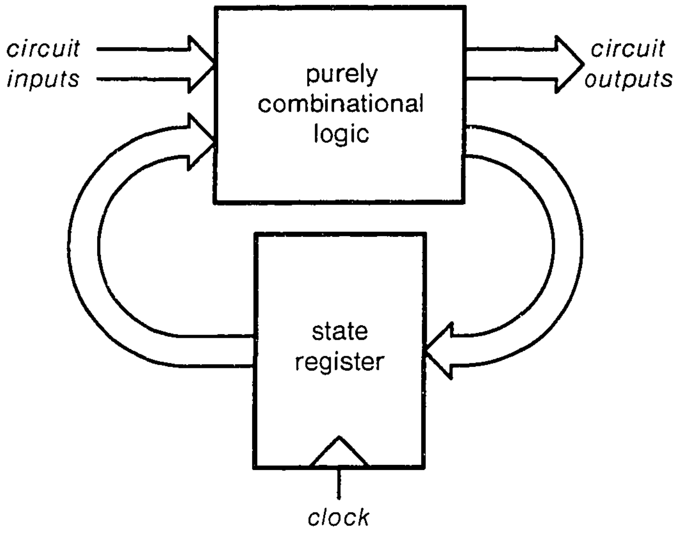


Figure 2.9: A Synchronous Edge-Triggered Circuit

logic that computes the circuit outputs and memory inputs from the circuit inputs and memory outputs. With I circuit inputs and M memory bits, this comparison can be done soundly with 2^{I+M} test vectors. Even very modest circuits today have dozens of input and memory bits, which makes this technique decidedly impractical. For example, if $I + M = 100$ (a small chip), and, if a trillion (10^{12}) vectors could be tried every second (much faster than currently possible [Beece88]), the comparison would take about 40,000,000,000 years—on the order of the estimated age of the universe!

Fortunately, there is a faster way to compare 'interesting' purely combinational circuits. This technique relies on two observations. The first is that for circuits with multiple outputs, the comparison can be broken up into a number of independent comparisons, one for each output. For each output, it is only necessary to compare that segment of the circuit that contributes to the computation of that output. Figure 2.10 shows an example. Only the part of the circuit outlined with a dashed line is necessary to compare the way this circuit computes the *sum* output with the way another circuit computes *sum*. Similarly, only the part outlined with a dotted line

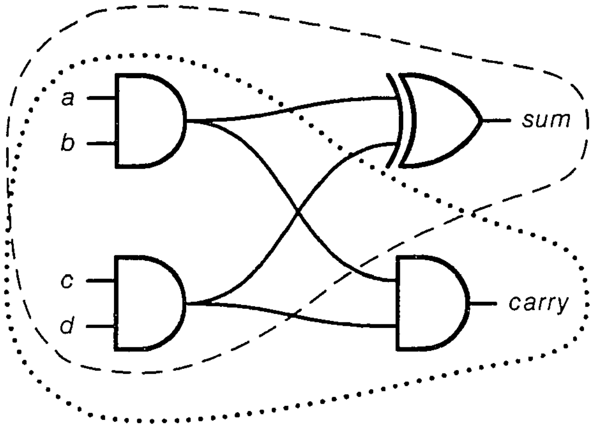


Figure 2.10: Segmenting a Circuit

needs to be considered for the *carry* output. The second observation is that for 'interesting' circuits, the size of each segment is limited. This limit comes from the facts that (1) any real implementation technology has limits on computation and propagation speed and component fan-in, and (2) a circuit is generally allotted only a small amount of time in which to compute its results.[5] These two observations together mean that a single output can depend on no more than a certain constant number of inputs, where that constant is fixed by the technology and the cycle time (which does not vary much for a given technology). This in turn means that the amount of time necessary to verify each output is bounded, and thus the time necessary to verify a whole circuit grows no faster than proportionally with the number of outputs. This is much better than the exponential growth seen with earlier techniques. However, even though this is very good asymptotic behavior, the constants can be very bad. For example, 32-bit adders are not uncommon, and the most significant bit of the result depends on all 64 inputs. Since the maximum number of inputs is thus at least 64, at least 2^{64} test vectors may be necessary to test some outputs. If a billion vectors could be tested each second,[6] it would take almost 600 years to verify the most significant

[5] A long computation time means that each component spends most of each computation idle; more can be computed in many short steps than a few long ones.

[6] This is not currently possible, but only by a few decimal orders of magnitude.

bit of a 32-bit adder. Thus, even this technique is not practical for sound comparison.

Most inconsistencies that actually occur are revealed long before the last vector is applied. Many designers trade off soundness for speed by trying only a small, carefully chosen, fraction of the necessary vectors. However, choosing a revealing fraction is problematic, and even a miniscule fraction of these astronomical numbers can be quite large. Nevertheless, many designers consider the unique ability to make this trade-off to be an advantage of simulation techniques.

Simulation is one of the most popular verification techniques. Many hardware description languages suggest simulation as a verification technique [Waxman86] [Crawford84] [Veiga84], many simulators exist [Bryant81] [Beece88] [Grodstein87], and in many design environments comparison is done primarily through simulation techniques [Acosta88] [Morison87] [Saunders87] [Suzuki85] [Suzuki87].

Algebraic Comparison

The next class of flat same-abstraction comparison techniques is algebraic comparison. Although these techniques are devoted to solving the same NP-complete problem, and so must take exponential time on some instances, they offer the hope of usually being faster than sound comparison by simulation.[7] The central idea of algebraic comparisons is to find an analytical expression for the function of a circuit, and to compare two circuits by attempting to prove their functions equal. An example would be to compare two views at the Boolean level of abstraction by converting each into a set of Boolean equations, and then testing for equality in Boolean algebra.

When the circuits to be compared have state, the problem of dependence on past inputs arises again. The same two solutions are available: restrict the two circuits to have identical states, or try to compare circuit operation across many cycles. The first solution again yields a technique whose running time is asymptotically proportional to the number of outputs, for the same reasons (the size of each output's segment is bounded). And at the Boolean level of abstraction, the constants are not so large. For example, PRIAM is able to verify a 32-bit adder against its specification in under a

[7] Unsound comparisons are flatly rejected in many design projects, in recognition of the high cost of overlooking inconsistencies.

minute of SPS9 RIDGE/62 CPU time [Madre88]. Another fast Boolean example can be seen in [Chandrasekhar87]. However, when the level of abstraction is raised above the Boolean, to include arithmetic and data structures, verification becomes much more difficult—the HAVE project at Manchester tried several specification languages and theorem provers, and the quickest one took half a man-day to verify a simple adder [Stavridou88].

The techniques that do not require identical states [Supowit86] [Devadas87] are much slower, both because they cannot factor the circuits as the restricted technique can and because they have to compare the operation of the circuits over multiple cycles. These techniques may take an amount of time that is polynomially equivalent to 2^M [Devadas87]; while this is better than robust simulation, it is still impractical for large circuits.

Other examples of algebraic comparison appear in [Barrow84], [Gordon81a], [Gordon81b], [Gordon83], [Hwang87], [Malik88], [Odawara86], [Maruyama85], [Milne84], [Narendran88], and [Roth77]. All of these techniques are best suited for the Boolean level of abstraction and above, because none of them can model the bidirectional nature of MOS transistors well.

Structural Comparison

The inspiration of the third class of flat comparison techniques, structural comparison, is that two circuits are equivalent if (although not only if) they are constructed from equivalent interconnections of equivalent components. Because the implication does not hold in the other direction, structural comparison techniques lack the power of the previous classes: many pairs of consistent views cannot be verified by structural comparison. In exchange for completeness, structural comparison techniques gain speed: many take an amount of time that is proportional (or nearly so) to the size of the circuits being compared.

One popular way to do structural comparison is to convert the circuits into labelled graphs, and then use a graph isomorphism checking algorithm. Figure 2.11 shows a sample circuit and its corresponding graph. Both the devices and the wires in the circuit appear as vertices in the graph; the connections in the circuit become the

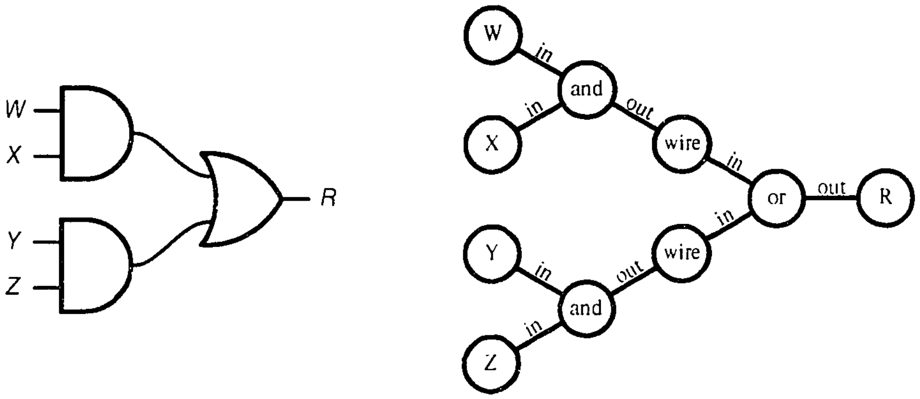


Figure 2.11: A Circuit and Its Graph

edges in the graph. The edges are labelled to indicate the role of the connection. Device vertices are labelled to indicate the device type. The vertices for input and output wires are labelled with unique labels; internal wires are all labelled with the same bland label. The graphs model interchangeability by giving the interchangeable entities the same label (as the input edges to the *OR* gate in Figure 2.11).

Graph isomorphism, whether the graphs are labelled or not,[8] is in general a hard problem, although its complexity is not known: it is clearly in NP, but it is not known to be NP-complete [Hoffman82] [Johnson81] [Read77]. However, the graphs that result from circuits tend to be easier to compare. For example, there is an algorithm [Kubo79] that usually terminates in $\mathcal{O}(N \log N)$ time. Other examples can be seen in [Ablasser81], [Baker80], [Ebeling83], [Takashima82], [Tygar85].

Structural comparison techniques are less powerful than algebraic ones in a number of ways. Figure 2.12 shows one example, in which two views differ by the inversion of the sense of the unlabeled wires. This difference is insignificant in Boolean algebra, but definitely significant structurally (a *NAND* gate is not equivalent to an *AND* or an *OR* gate). Figure 2.13 shows another example: two identical CMOS implementations

[8] Unlabeled graph isomorphism is polynomially equivalent to vertex and edge labelled graph isomorphism. The labelled problem can be polynomially reduced to the unlabeled one by adding subgraphs to encode the labels, as in [Hoffman82, proof of Lemma 1 of Chapter 2]; the unlabeled problem can be reduced to the labelled one by choosing a trivial labelling.

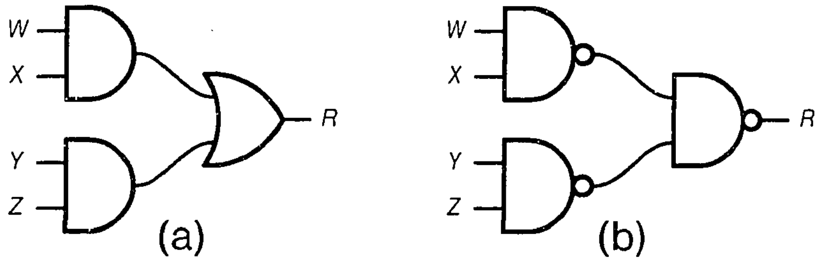


Figure 2.12: Structural Comparison Does Not Understand Inversion

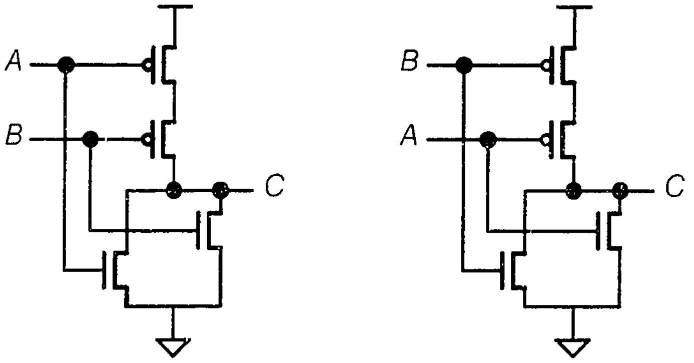


Figure 2.13: Structural Comparison Does Not Understand Commutativity of Boolean Function Inputs

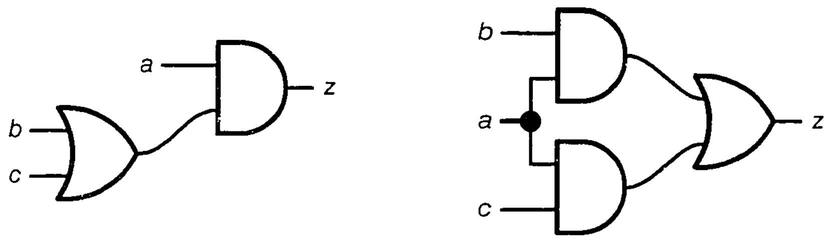


Figure 2.14: Structural Comparison Cannot Distribute AND Over OR

of a Boolean *NOR* gate, with swapped inputs. The *NOR* operator is commutative in Boolean algebra, but there is a structural difference between the two inputs to the transistor netlists: one input is connected to a transistor closer to *Vdd* than the other. Figure 2.14 shows a final example: *AND* distributes over *OR* in Boolean algebra, but the circuits have very different structure.

Some techniques are based on structural comparison and include extensions to recapture some of the power of algebraic comparison. For example, the technique presented in [Shiran86] can handle the swapping of inputs in Figure 2.13. Also, a technique presented in Section 5.5.3 of this dissertation can handle the example of Figure 2.12. But no technique can have all the power of algebraic techniques without also having an exponential worst-case running time.

Flat Comparison Summary

The trade-offs made by the flat same-abstraction comparison techniques are summarized in Table 2.2. The table shows that we must either (a) live with certain restrictions, (b) face the possibility that an inconsistency will go undetected, or (c) keep the problem size small.

2.2.2 Hierarchical Comparison

Hierarchical techniques improve the speed and quality of comparison by taking advantage of the hierarchical organization of the views. All these techniques wcrk essentially

Technique	Speed[*]	Restrictions	Failings
Simulation			
Complete	absurdly slow—$\mathcal{O}\left(2^{I\times(2^M-1)}N\right)$	none	none
Restricted	extremely slow—$\mathcal{O}(O)$ [†]	identical states	none
Partial	moderate—$\mathcal{O}(m \times N)$ [‡]	none	false positives
Algebra			
Restricted	moderate—$\mathcal{O}(O)$ [§]	identical states, high level of abstr.	none
Unrestricted	slow—$\mathcal{O}\left(P\left(2^M\right)\right)$ [¶]	high level of abstr.	none
Structure	fast—$\mathcal{O}(N \log N)$	essentially identical structure	none

[*] I is the number of inputs, O is the number of outputs, M is the number of memory bits, and N is the total circuit size

[†] constant is extremely bad

[‡] only m test vectors are applied

[§] constant is not good

[¶] for some polynomial P

Table 2.2: Tradeoffs of Flat Comparison Techniques

the same way. They start by requiring, or establishing, that the two views to be compared have essentially identical hierarchies. There is thus a one-to-one correspondence between the cell types of the two views, and consequently the whole views can be compared by independently comparing the corresponding cell types. The advantages of this factoring of one big problem into many small problems are many:

- The sum of the small problem sizes is smaller than the big problem size.
- Algorithms with worse-than-linear complexity benefit greatly from the small size of each cell type comparison.
- The cell type comparisons can be done in parallel, or serially, or according to any convenient schedule.
- Different techniques can be applied in different cell type comparisons.

- The independence of the cell type comparisons fosters incrementality: once two whole views have been compared, and one cell type is then edited, it is only necessary to re-compare that cell type.

- Flattening would multiply errors and obscure their origins.

A simple example of this is the COMPARE program from Valid Logic Systems [Tygar85]. It compares each pair of cell types structurally. Another example is the system used at NEC for the design of the SX-1/SX-2 supercomputer [Suzuki85]; this system compares each pair by simulation. Another example is Barrow's VERIFY [Barrow84], which compares structural and behavioral views by an algebraic technique.

As mentioned earlier, different flat techniques can be mixed. For example, Silica Pithecus [Weise87] compares the lowest level composite cell types algebraically, and the higher ones structurally. This is a member of the class of *structural/semantic hierarchical comparison techniques*, which are particularly attractive. These techniques in general identify a low frontier in each view, compare structurally above that frontier, and use a more powerful (semantically-oriented, such as simulation or algebraic comparison) technique to compare the corresponding frontier cells. The use of structural comparison above a low frontier gives these techniques speed; the use of a more powerful technique at the frontier gives these techniques an often-sufficient amount of power.

Many of these techniques allow slight deviations from absolute identicalness of hierarchy. For example, Silica Pithecus, which compares a switch-level view to a digital one, allows the switch-level hierarchy to *go lower* than the digital one. That is, every digital cell type has a corresponding switch-level cell type, but some switch-level cell types may have no corresponding digital cell type (but they must be descendants of cell types that do); these two hierarchies are *identical modulo bottom*. In general, when comparing a high-level view to a low-level one, it must be expected that their hierarchies are no more than identical modulo bottom—the leaves of a Boolean circuit (the Boolean operators) *cannot* be leaves in an electrical circuit. Another deviation from identicalness allowed by Silica Pithecus concerns wiring: the power supply wiring appears in the switch-level view, but not in the digital.

Neither of those two variances seriously redresses the fundamental wrong of requiring identical hierarchies. There is one variance that begins to make a fundamental difference: some comparison programs (this option is logically available to any technique) allow extra intermediate cell types (as exemplified by the *Register File*, *Shifter*, and *ALU* cell types in Figure 2.3 and the *BitSlice* cell type in Figure 2.4) to be present in the views—and flatten out these extra intermediate cell types before doing comparison. An example that does this is COMPARE [Valid87]. However, this variance alone is not enough, for two reasons: (1) flattening reduces the amount of hierarchy, and thus also the amount of benefit received from it, and (2) when comparing views at different levels of abstraction, flattening may not be powerful enough to establish identical hierarchies (recall the example from Figure 2.8 of a Boolean leaf cell with no corresponding electrical cell). Reason (2) does not apply to some algebraic and simulation techniques, because they only require that the hierarchies be identical above some frontiers.

The cell type comparisons are not completely independent in some of the hierarchical techniques. For example, Silica Pithecus generates constraints in cells, then propagates them up the hierarchy, until they are discharged in some higher cell. This places constraints on the scheduling of, and communication between, the cell type comparisons. It also reduces the incrementality: after a cell is edited, all its ancestors may have to be re-compared.

The amount of benefit derived by hierarchical techniques depends on the structure of the views. One of the largest sources of benefit is repetition. If a cell type has many instances in a hierarchy, hierarchical techniques can spend effort only once, on that cell type, that flat techniques must spend on every instance. Some views have more repetition than others.

In summary, hierarchical techniques are attractive because they factor the whole comparison problem into many smaller, independent problems. Unfortunately, the existing techniques require that the views to be compared have essentially identical hierarchies. This is an undesirable restriction, and the only existing way to deal with substantially different hierarchies is to flatten out the differences—and

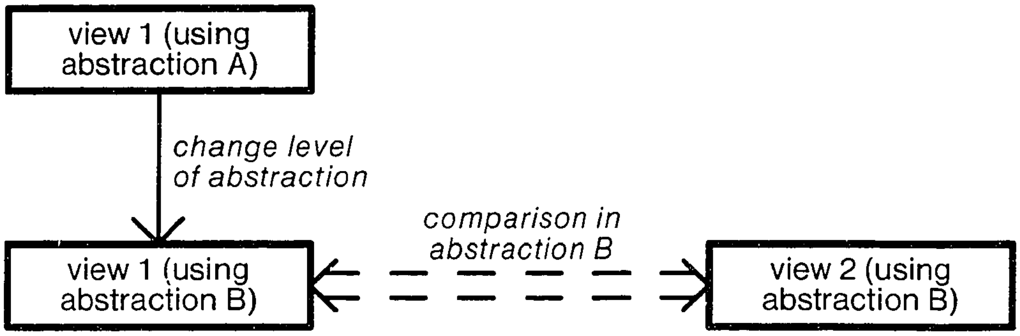

view 1 (using abstraction A)
change level of abstraction
comparison in abstraction B
view 1 (using abstraction B)
view 2 (using abstraction B)

Figure 2.15: Factorization of Comparison Between Views at Different Levels of Abstraction

this is not a completely satisfactory solution.

2.2.3 Abstraction Crossing

The previous sections set forth techniques for comparing views at the same level of abstraction, and for taking advantage of hierarchy; however, the *real* problem involves comparing views at different levels of abstraction. The techniques for doing this can be factored into a technique for changing the level of abstraction of one view and a technique for comparing views at the same level of abstraction; see Figure 2.15. The techniques for changing the level of abstraction of a view again fall into three classes: those that raise the level of a view, those that lower the level of a view, and those that change the level of simulation traces. Many of these techniques are acceptably fast, but some impose constraints on the choices of hierarchies.

Raising the Level of Abstraction of a View

The most obvious, and one of the most popular, class of level-crossing techniques is raising the level of abstraction of one view to match that of the other. For example, for comparing masks to higher views, there are many programs available [Chiang88] [Ablasser81] [Baker80] [Gupta83] [Wong85] for extracting electrical or switch-level circuits from masks. There are also programs that abstract Boolean, or similar, views from electrical or switch-level views [Bryant87b] [Wu87] [Weise87]. And there are programs that work between even higher levels of abstraction [Lathrop87].

These techniques vary greatly in their efficiency. Electrical circuit extraction can be done in $\mathcal{O}(N \log N)$ time (where N is the number of mask features) [Chiang88].

Boolean circuit extraction has a more complicated analysis. For example, in Bryant's method [Bryant87a] [Bryant87b], conversion of a switch-level circuit of N transistors into a set of Boolean equations may produce a result of size $\mathcal{O}\left(N^{3/2}\right)$ and take a proportional amount of time. However, most circuits produce $\mathcal{O}(N)$ results in $\mathcal{O}(N)$ time. Unfortunately, Bryant's method is alone in its efficiency; the others can require exponential time and produce exponential results. Fortunately, the Boolean extraction problem can be factored into many small sub-problems by noting that there is no communication through the power supply wires and that information flows unidirectionally into a MOS transistor's gate. Also, technology and clock cycle considerations limit the size of the resulting circuit segments, and so the time required to do Boolean circuit extraction actually grows linearly with the size of the whole circuit. Although the constant is not small, it is usually acceptable. For example, Bryant's COSMOS is able to extract the Boolean view of a 64-bit nMOS ALU, containing 1664 transistors, in under 5 CPU minutes on a DEC MicroVax-II.

Many techniques for raising the level of abstraction of a view have interesting interactions with hierarchical concerns. For example, techniques for extracting electrical circuits from masks have to work harder when there is overlap between the areas of sibling cells. Another kind of interaction is exemplified by Boolean circuit extraction: all the transistors that implement a given Boolean gate have to be in the same cell (otherwise some cells do not have an appropriately[9] Boolean interface); the scattering illustrated in Figure 2.8 violates this restriction. Requiring the hierarchies to be identical modulo bottom prevents this problem. Another popular solution is to extend Boolean algebra with a value that roughly means 'high-impedance'. When this is done, most MOS cells can be abstracted—but the structure of the results will not match that of the natural Boolean description. For example, the result of raising

[9] As COSMOS demonstrates, switch-level behavior can be cast in Boolean terms; however, the Boolean cast of the switch-level behavior of a fragment of the implementation of a Boolean gate will not, in general, correspond to anything in the Boolean view.

the description in Figure 2.8b will not have an *OR* gate in it, like the description in Figure 2.8a. And even such extended algebras are not able to describe the operation of some MOS cells (those in which some transistor transmits information in both directions, and this is visible at the cell's interface).

Lowering the Level of Abstraction of a View

The level of abstraction of a view can be lowered, as well as raised. Although doing a good job of this is very hard (it is the synthesis problem) doing a poor but correct job is often quite easy. The biggest concern with this technique is the extra information introduced in the process—what if it is inconsistent with the corresponding information in the other view? For example, suppose one view is Boolean and the other is switch-level. The Boolean view could be lowered by replacing every Boolean primitive with a standard switch-level implementation. But recall from the example of Figure 2.13 that the ordering of the inputs to a transistor network can matter (to structural comparison, for instance), even though that network is implementing a commutative Boolean function.

This problem has two resolutions. One is to live with it. On the face of it, this sounds like a bad idea—the lower view can be no better than a poor synthesis from the higher view. It is actually not so bad, for two reasons. One is that the synthesis does not have to be poor. The other is that in addition to lowering the level of abstraction of one view, the level of abstraction of the other can be raised—which means the views can differ by more than the synthesis. This is one way of looking at the comparison done in the DATools at PARC [Barth88], where masks are compared to schematics containing arbitrary abstractions.

The other resolution is to use a same-level comparison technique that can overlook the differences in the added information. An example is reported by Roth [Roth77]. One view is in PL/R, which is similar to the PL/I programming language. The other view is a Boolean circuit. A compiler generates an inefficient but correct Boolean circuit from the PL/R view, then the two Boolean circuits are compared in Boolean algebra—in which efficient and inefficient circuits can be proven functionally equivalent.

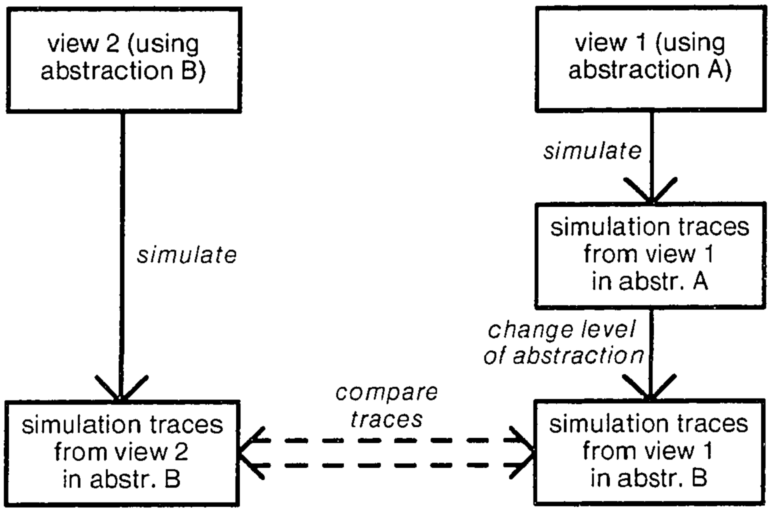

view 2 (using abstraction B)
view 1 (using abstraction A)
simulate
simulate
simulation traces from view 1 in abstr. A
change level of abstraction
compare traces
simulation traces from view 2 in abstr. B
simulation traces from view 1 in abstr. B

Figure 2.16: Comparison Across Levels of Abstraction by Simulation and Raising the Level of Abstraction of Simulation Traces

Changing the Level of Abstraction of Simulation Traces

The final class of techniques work by raising the level of abstraction of simulation traces. These techniques extend simulation techniques to enable comparison across levels of abstraction; see Figure 2.16 for a schematic of this process. These techniques generally are fast and don't place any additional restrictions on the circuits—beyond the obvious one that says the less abstract view can in fact be converted into a more abstract one.

2.2.4 Summary of Consistency Techniques

The preceding sections present several existing comparison techniques, each of which does well on some aspects of the comparison problem. However, none of the existing techniques solves the problem of comparing alternate views at different levels of abstraction with different hierarchies well. They either don't solve it at all (because they require the views to use identical hierarchies) or they solve it poorly (by flattening out the hierarchy or accepting only flat input). The one that deals with differing hierarchies best, Valid's COMPARE, can be considered a simple version of Informed Comparison.

2.3 Existing Techniques for the Correspondence of Entities

The previous section presents existing techniques for solving one part of the comparison problem: verifying consistency between views with redundant information. This section presents existing techniques for the other part: discovering and maintaining the correspondence between the entities of alternate views.

The most trivial techniques simply insist that all the views of a design use the same hierarchy. This makes the correspondence trivial: it is one-to-one between the entities of every pair of views. A leading exponent of this is the Mead/Conway design methodology [Mead80]. The Version Server [Katz86] is another example: it is able to note one-to-one equivalences, but nothing more sophisticated. These techniques are clearly incapable of handling correspondences between views that differ by hierarchy transformations.

The CORAL-II program [Blackburn88] is more sophisticated. It is part of the System Architect's Workbench [Walker87], which uses alternate views with independent hierarchies. CORAL-II is responsible for maintaining the links between the views as the views are synthesized and transformed; it specifically addresses the problems of the correspondence between an original view and a transformed version of it. CORAL-II relaxes the restriction that the correspondence be one-to-one. Each entity is tagged with the set of entities to which it corresponds. Unfortunately, that is not good enough to accurately describe the correspondence across transformations that break cell boundaries. Recall the examples of Section 2.1.3, where the correspondence must be between paths, not just single entities.

DDS [Parker84], which is a data structure for use in synthesis from a register-transfer-level description, employs a very sophisticated treatment of the correspondence between view entities. In DDS, there are four views, called *domains*: the behavioral domain, the structural domain, the physical domain, and the timing and control domain. There are also five relations between the entities of those domains. Three of those relations are binary, and the other two are 3-way. This complexity is needed to handle the fact that a value from the behavioral domain appears at different places in the structural domain at different times. While DDS gives a sophisticated treatment to correspondences between alternate views, it does not address, and cannot well represent, correspondences across hierarchy transformations.

This section presents some existing techniques for conceiving of, discovering, and maintaining the correspondence between entities. Few techniques even address the problems of correspondences across hierarchy transformations, and those that do are not very capable.

Chapter 3

Introduction to Informed Comparison

The previous chapter presents the comparison problem and some of the existing techniques for solving it. These techniques are not fully satisfactory, because they do not handle different hierarchies well (they either flatten out the differences or refuse to accept them). This chapter presents a new comparison technique, called *Informed Comparison*, that allows the views to use different hierarchies (as well as different levels of abstraction) and has many of the benefits of other hierarchical methods.

3.1 The Schema

The inspiration of Informed Comparison is that if the designers keep track of the intended relationship between the different hierarchies of the views, then the views can be compared by simply applying hierarchy transformations, under the guidance of that intended relationship (called the *key*), to copies of the views until they have sufficiently similar hierarchies that an existing hierarchical comparison technique can be applied. See Figure 3.1. The process of transforming the hierarchies is called the *reconciliation of the views*, and the *base comparison* completes the comparison. The method of Informed Comparison is so named to emphasize the importance of the key.

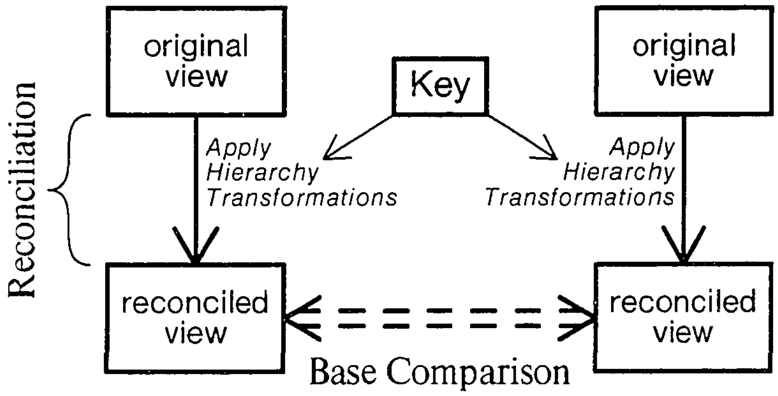


Figure 3.1: An Informed Comparison

During reconciliation, Informed Comparison builds up the correspondence between the original view entities and the transformed ones. Knowing this, error reports from the base comparison can be transformed to speak in terms of the original entities. Furthermore, once the base comparison determines the correspondence between the reconciled entities, that correspondence can be composed with the correspondences back to the original entities, to yield the correspondence between the original entities.

In a *cleanly divided* Informed Comparison, the transformations of the reconciliation change only the hierarchical organization of a view. In particular, the transformations are *flat-insignificant*: if the original and reconciled versions of a view were completely flattened, they would be identical. A cleanly divided Informed Comparison is thus as sound as its base comparison.

Although the transformations focus on the hierarchical structure of a view, it must be remembered that there is more to a view than structure. An Informed Comparison considers the non-structural information to be attached to the structural. Thus each structural transformation must also transform the non-structural information, in such a way as to have no effect that would be discernable after flattening. By virtue of the attachments between the structural and the non-structural information, and of the requirement to be flat-insignificant, the required non-structural changes can be derived from a purely structural specification of each transformation.

It is not logically necessary that the transformations be flat-insignificant. What *is* necessary is that the reconciliation not change any of the information being compared for consistency. A *blurred* Informed Comparison is one whose reconciliation changes more than the hierarchy of the views. Such a comparison is called "blurred" because the reconciliation is doing some of the base comparison's work. A simple example is deleting, during the reconciliation, wires not connected to anything. The reconciliations of blurred Informed Comparisons are more powerful and complex than those of cleanly divided ones. The soundness of a blurred Informed Comparison depends on the soundness of its flat-significant reconciliation transformations as well as the soundness of its base comparison. Although the two Informed Comparisons studied in the following two chapters are slightly blurred, a complete study of blurred Informed Comparisons is beyond the scope of this dissertation.

In what language should the key state the intended relationship between the hierarchies? How can a computer program determine from such a statement the transformations to apply? Informed Comparison answers both questions thusly: the key consists of invocations of transformations. The transformations are a valid representation of the relationship between the hierarchies of the original views—*if* the hierarchies of the reconciled views are identical. If they are not identical, the difference is not represented in the key. That is acceptable, because that difference will be in only a few, easy-to-resolve areas (such as whether the power supply wiring is explicit) that the base comparison can handle. The key also contains more information than the relationship between the hierarchies: it also implies a choice for the hierarchies of the reconciled views, and a strategy for changing the original hierarchies into the reconciled ones. Of course, even a plain statement of the relationship between the original hierarchies would have to employ some strategy for expressing that relationship in terms of the available relational primitives.

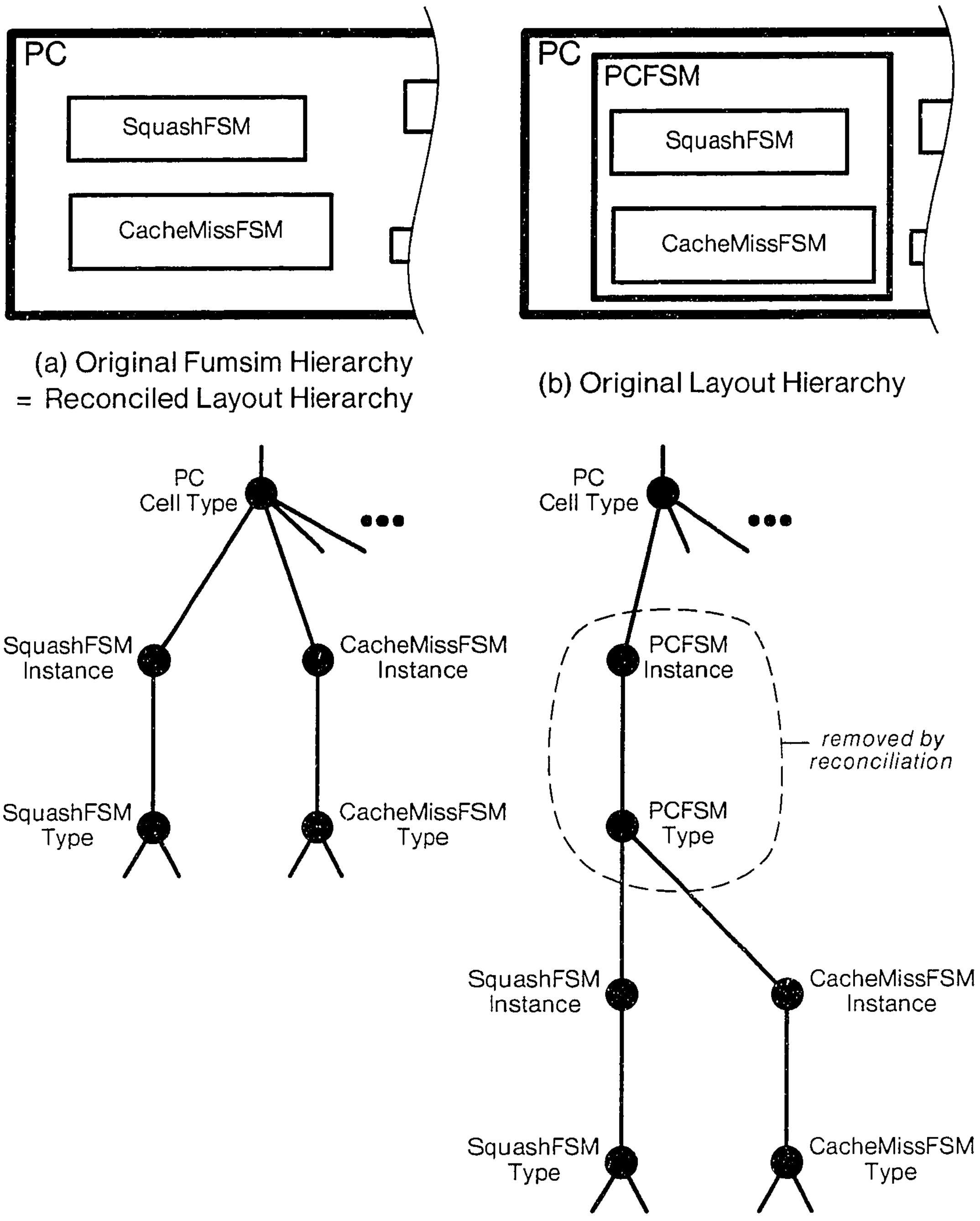
In order for informed comparison to be advantageous, creation and maintenance of the key must not be too onerous. One possible difficulty can quickly be laid to rest. Most of the discussion in this dissertation focuses on the problem of comparing two views, but designs can use more than two views. When N views are used, there are $\binom{N}{2}$, which is $\frac{1}{2}(N^2 - N)$, different pairs of views to be compared. Since $\binom{N}{2}$

grows quickly with N, it would be tedious if the key had to have $\binom{N}{2}$ sections (one for every pair of views). There are two design practices that keep the key size small. In one, the views can be ordered by information content. When this is done, it is only necessary to compare each view with its immediate successor and predecessor. Thus $N - 1$ comparisons are done, and the key size is linear, rather than quadratic, in N. This practice has benefits even when followed only partially: if only a subset of the views can be ordered by information content, the number of comparisons required between members of that subset is only linear, not quadratic, in the subset size. The second practice can restrict the size of the key even when the number of comparisons is quadratic. This is done by choosing one "ideal" hierarchy, and independently reconciling each view to it. The key thus has only N sections, one for each view.

3.2 Examples

The two example steps of reconciliation here are drawn from MIPS-X, a 32-bit microprocessor designed at Stanford. The two views being reconciled are a functional simulation (called the *funsim*) and the layout. Figure 3.2 shows the first example, in which the layout has some "extra" structure, the *PCFSM* cell type and instance. The reconciliation step removes this difference by flattening out the *PCFSM* cell type; this promotes the *SquashFSM* and *CacheMissFSM* cell instances to be contained directly in the *PC* cell type, as is the case in the funsim view.

The other example concerns the split multiplexor, introduced in Figure 2.7. The difference of hierarchies in this example is that the funsim has a multiplexor whereas the layout has instead two tri-state drivers. Figure 3.3 depicts the reconciliation of this difference, accomplished by adding some structure, the multiplexor cell type and instance, to the layout. In this example the reconciliation does not yield identical hierarchies. In general the reconciliation does not need to yield identical hierarchies: it only needs to make them similar enough for the base comparison. In this example it is *impossible* to make the hierarchies identical, because the multiplexor cell type of the funsim is atomic and the multiplexor cell type of the layout is not. There *are* base comparison methods for which this reconciliation brings the hierarchies close enough:



(a) Original Fumsim Hierarchy
= Reconciled Layout Hierarchy

(b) Original Layout Hierarchy

Figure 3.2: A Simple Reconciliation

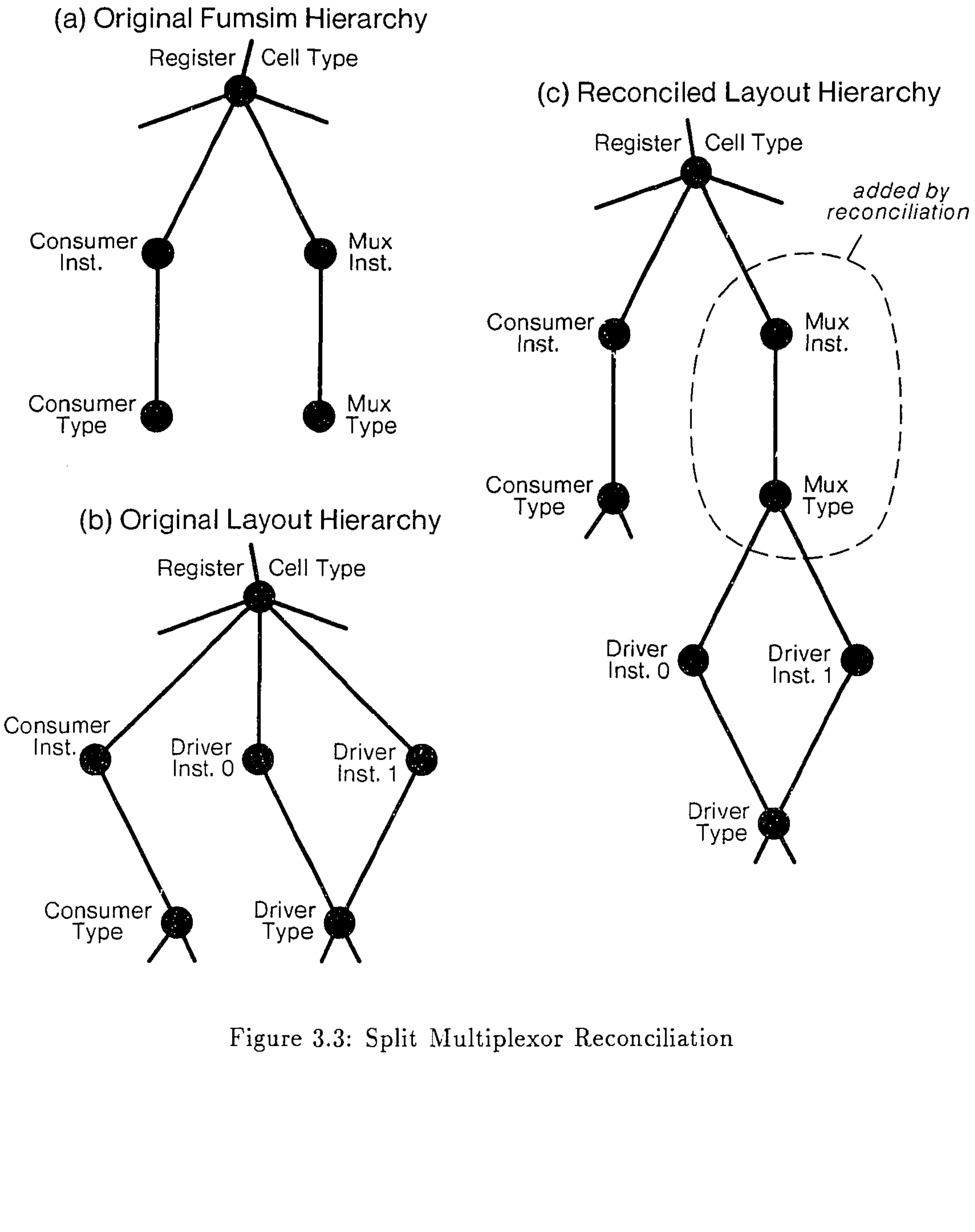


Figure 3.3: Split Multiplexor Reconciliation

an example is comparing the simulated input/output behaviors of the multiplexor cell types of the reconciled views.

3.3 Choices

The method of Informed Comparison is actually a "meta-method": it is a modification (prefacing by the reconciliation) of another method (the base comparison). Many qualities of Informed Comparison vary with the base comparison and the repertoire of transformations available to the reconciliation. The transformation repertoire gives Informed Comparison most of its power to see through hierarchy differences; all the other comparison power comes from the base comparison. Even once these factors are fixed, there are some choices left to the designers that also affect the Informed Comparison.

The transformation repertoire affects the length and complexity of the key, and the power and efficiency of the reconciliation. A large, complicated transformation can be composed from small simple ones. Borrowing from group theory terminology,[1] we can speak of the set of transformations *generated* (through taking all possible compositions) by a given set. The set of flat-insignificant transformations can be generated by a small set of simple transformations. However, composing a real reconciliation from such small pieces could be very tedious. To keep the key small, the repertoire of transformations should match the designers' abstractions concerning hierarchy relationships. This could become problematic, because designers, being human, can invent new abstractions. However, the study in chapter 5 suggests that a fixed repertoire (of modest size and complexity) can enable reasonably short keys.

Conciseness of the key also requires that the amount of information necessary to specify each transformation be small. For example, when moving a cell around, the key should not need to explicitly mention what happens to all of the attached wiring. In general, when only the effects on cell structure are specified, alternatives for the wiring effects remain. However, for each transformation in the two example

[1] Unfortunately, hierarchy transformations do not form a group: most transformations are not applicable to every hierarchy, which means that either the closure property or the existence of inverses cannot be achieved.

systems of later chapters, there is one 'most natural' alternative, whose automatic use leads to reasonable keys. Explicitly describing what happens to the non-structural information is not necessary either, as mentioned earlier.

A repertoire of transformations is called *complete* if it generates every flat-insignificant transformation. While is possible to do Informed Comparison with an incomplete repertoire, the power of the Informed Comparison may be reduced. When the hierarchies of two views cannot be reconciled, and the base comparison cannot handle the remaining differences, Informed Comparison cannot verify the consistency of the views. Even an incomplete repertoire can enable reconciliation of any two flat-identical[2] views, for the reconciliation involves transforming both original hierarchies to new ones, not one original to the other. Consider a repertoire that consists only of flattening transformations. Any two flat-identical views can be reconciled by this repertoire, by complete flattening. However, such a reconciliation leaves the base comparison with no hierarchy to take advantage of. Furthermore, views at different levels of abstraction are unlikely to be flat-identical: normally one hierarchy will at least go lower than the other, and may be even more different. Some base comparison techniques do not require the reconciled views to have identical cell structure, but *do* require that a frontier can be chosen for each reconciled view so that they have identical cell structure above their chosen frontiers. If only flattening transformations are available, those frontiers may have to be as high as the root cell types. Figures 3.4 and 3.5 show an example. In the switch-level view, the transistors implementing a Boolean gate (B in the digital view) are not all in the same lowest composite cell. Suppose the base comparison uses a frontier in each reconciled view: above the frontier the views are compared structurally, and at the frontier the switch-level view's level of abstraction is raised to digital and then digital equivalence is checked. Silica Pithecus [Weise87] is such a method. In this example, the frontiers must be the root cells. The B and C cells cannot be in the frontier; even though the use of a 'high-impedance' value makes it possible to generate a digital description of the switch-level B cell, that digital description will not match the description of the original digital

[2] Two views are ***flat-identical*** when either can be changed into the other with only flat-insignificant transformations. Put another way, if both views were completely flattened, the results would be identical.

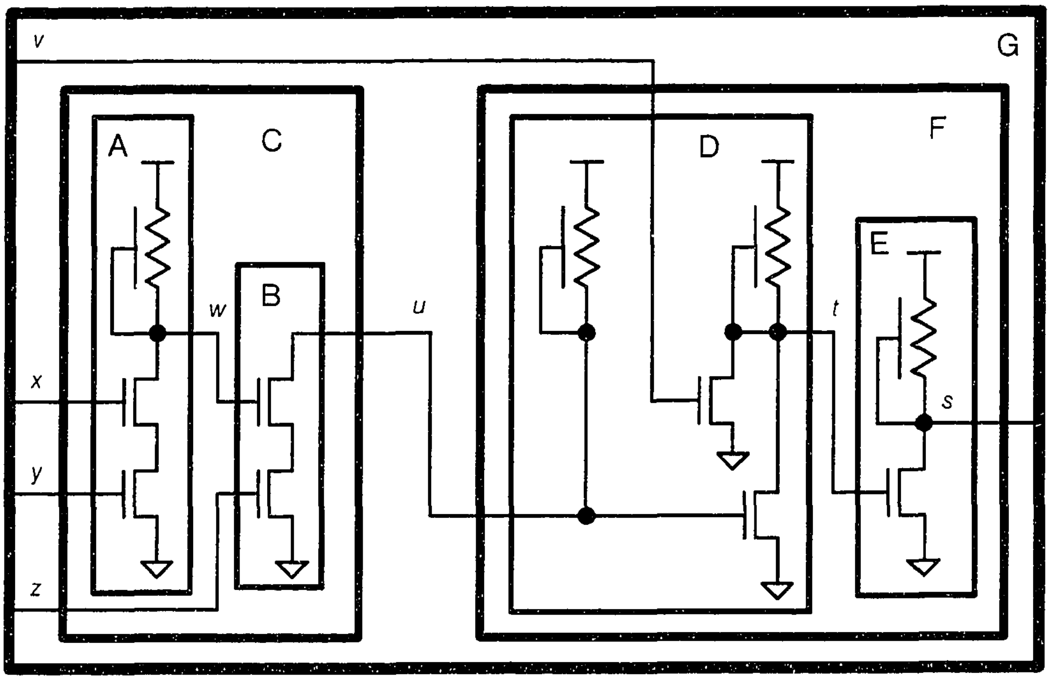
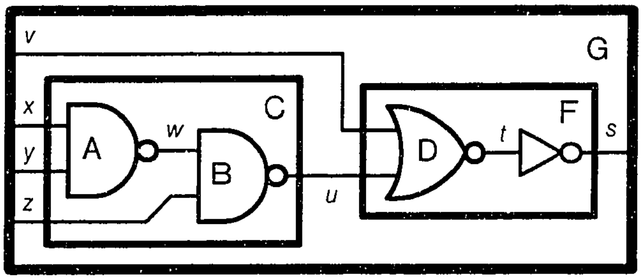


Figure 3.4: A Switch-Level View, with Problematic Pullup

```
(df (A x y) (not (and x y)))
(df (B w z) (not (and w z)))
(df (C x y z) (B (A x y) z))
(df (D u v) (not (or u v)))
(df (E t) (not t))
(df (F u v) (E (D u v)))
(df (G x y z v) (F (C x y z) v))
```

(a) A Digital View



(b) Its Hierarchy

Figure 3.5: A Clean Digital View

B cell, which does not use the 'high-impedance' value. The *A* cell cannot be in the frontier because the frontier must be above *A*'s parent *C*. The frontier cannot include *D*, *E*, or *F* for similar reasons. In this small example, the transformation repertoire's lack of power and the difference in hierarchical placement of the pullup force both views to be completely flattened. This can also happen in large designs, where a large amount of flattening is very disadvantageous.

The key can be stored in a number of ways. It can be an independently maintained file or can be distributed throughout the views, by annotating the view entities. The transformations can be specified textually or graphically. Where synthesis tools are used, they can generate germane parts of the key. These choices can greatly affect the "user-friendliness" of Informed Comparison.

For a given Informed Comparison system and pair of alternate views, the designers generally have some freedom in choosing the reconciled hierarchies and the transformations used to make them. These choices also affect the conciseness of the key. Flat-insignificant transformations come in pairs, where each member of a pair is the inverse of the other. One member generally takes less information to specify than the other. For example, flattening is simpler to specify than un-flattening.

The order in which the transformations are used is also important. For example, it is easier to specify applying some transformation T to the contents of some cell type followed by flattening out that cell type than it is to specify flattening out that cell type followed by applying T at every former instance of the now flattened out cell type.

Although the designers must, in general, pick a strategy for reconciling two views, in some methodologies some kinds of differences are easier to reconcile: there is a canonical form to which each view can be automatically reduced, so that views that differ in these certain ways no longer differ. A transformation that reduces a view to a certain canonical form is called a *canonicalization transformation.* A common example is deletion of wires with no connections.

3.4 The Nature of the Correspondence

Informed Comparison requires a careful treatment of the correspondence between the entities of views. Since the reconciliation is composed from many transformations, the correspondence between the original entities and the reconciled ones is composed from many correspondences across the individual transformations. It is important that those component correspondences give detailed accounts of how the entities are related; vagueness, compounded over and over, would be practically useless. The exact nature of the correspondences will depend, like many things, on the repertoire of transformations and the base comparison. Two detailed formulations of correspondence appear in later chapters; a few generalities are discussed here.

Informed Comparison's treatment of correspondences rests on two foundations: (1) instance paths and wire paths are used to give a completely detailed accounting of transformations that 'break cell boundaries', and (2) the correspondence is, in essence, simply a binary relation, and composition of correspondences is accomplished mainly by simply composing binary relations. Since (1) shifts the focus away from cell instances and wires to paths, when a designer asks for the correspondence of a single instance or wire (or even cell type) the answer may be complex. That is proper—the correspondence may actually *be* complex. The complete answer is contained in the correspondence between the paths. If the designer does not want the full complexity, a vague but shorter answer can be given. Notational shortcuts can also be used: an example is using a short path to stand for all the longer paths of which it is a prefix. In simple cases, the binary relation is one-to-one; unfortunately, complications commonly make it many-to-many.

3.5 Why Informed Comparisons Are Better

Existing comparison methods handle differing hierarchies poorly, either by rejecting them or flattening them. Informed Comparison can compare views with differing hierarchies and has many of the benefits of other hierarchical methods. In addition to checking consistency, Informed Comparison can also discover the correspondence between the entities of the views—and in a more flexible way than existing comparisons do.

Informed Comparison is faster than existing methods that allow differing hierarchies. The existing methods handle differing hierarchies by complete or partial flattening. Even those that flatten only partially may be required to create very large and complex cell types. Many base comparison methods take an amount of time that grows much faster than proportionally to the size of the description of a cell type. Increasing the complexity of cell types is thus very disadvantageous.

Conversely, Informed Comparison is more flexible than the existing methods that are fast. These require identical hierarchies, which is an undesirable restriction.

The additional costs of Informed Comparison, beyond those of existing methods, are the creation and maintenance of the key and doing the reconciliation. These costs depend on the transformation repertoire and the base comparison. Creation and maintenance of the key are not great burdens. The examples of the following two chapters show that keys are small: in one example, the key is ‘free’ (it is taken automatically from design data present for other purposes); in the other the key is explicitly maintained and larger, but still much smaller than the views being compared. The knowledge of the key has to be distributed among the designers in order for them to be able to do their work, regardless of whether Informed Comparison is used. Informed Comparison does the designers a service by enabling them to write the key down in a machine-checkable form.

Informed Comparison enables hierarchical comparison of views with different hierarchies. Of course, this does not solve all the problems of comparison. For example, there are still the problems arising from the fact that the views are at different levels of abstraction. Also, as pointed out in Section 2.2.2, not every design benefits greatly from hierarchical techniques. Furthermore, there are still reasons to keep the hierarchies somewhat similar—hierarchy differences still have a negative impact on overall design clarity.

Chapter 4

A Simple System

Previous chapters present the comparison problem and introduce Informed Comparison; this chapter presents a particular, simple, Informed Comparison method: *PWCoreLichen.*[1] It was developed in the Computer Science Laboratory of Xerox PARC as a member of the *DATools* [Barth88], which have a simple, well-defined methodology. The requirements of that methodology precisely determine the transformation repertoire and base comparison for PWCoreLichen. In the DATools methodology, the key is small and "free": it is determined by design information already captured for other purposes. The correspondence between entities of alternate DATools views is relatively simple.

4.1 The DATools Methodology

The DATools are an integrated suite of programs: they all operate on a common in-memory design representation called the *Core* data structure. The basic paradigm for design data flow is this: first *source Core* is created, and then layout is generated according to it. PWCoreLichen compares the source core with the layout.

[1] "Lichen" is the name of the general system studied in the next chapter; the prefix "PWCore" is added here to keep the distinction clear. "PWCore" is the name of the main component of the "PatchWork" system, introduced later. "PW" is an abbreviation of "PatchWork", and "Core" refers to the central data structure.

4.1.1 The Core Data Structure

The Core data structure focuses on the hierarchical structure of a view; the non-structural information is attached to the structure through the widespread use of property lists. Core uses a formulation of hierarchy different from the one adopted in this dissertation (see Section 2.1.1). In Core, there are no ports; instead, some wires are simply declared public. The atomic cell types have public wires and no others. In Core, wires are structured: each wire either is atomic, or has a non-empty sequence of child wires. A wire may be used as a child more than once; thus, the structure of the wires may form any directed acyclic graph (DAG).

In Core, the composite wires serve only to highlight regularity in the connectivity of the atomic wires: at every composite connection, the corresponding children are also connected. The composite wires are thus superfluous; they could be deleted without losing any information about the communication in the view. Since the composite wires are no more significant than the intermediate cell types, transformations that change only the composite wiring are considered flat-insignificant.

A picture editor called ChipNDale and its data structure are used for layouts and schematics. ChipNDale pictures are also hierarchical: there are picture types and picture instances. ChipNDale has nothing analogous to ports or wires; the layout for a wire consists of instances of picture types for rectangles of various shapes and colors. Other important atomic picture types include transistors and texts. The ChipNDale picture entities also have property lists, and there are links between the Core data structure and ChipNDale. For example, on the property list of a Core cell type may be found the ChipNDale picture type for the cell's layout.

4.1.2 The Methodology

Source Core is created either by program or by extraction from schematic pictures; the programs and schematics are created by hand using editors. Low level composite layout cells also are created by hand. The higher level layout cells are created by programs, *layout generators*, by combining the lower cells following the structure of the source Core. A program called *PatchWork* is responsible for establishing and following the links between the source Core and the layout, for calling the layout generators as necessary, and for other mediation in support of layout generation. A source Core cell type may specify in its property list a *layout key*, which is the name of the layout generator to use for that cell type.

The two views (source Core and layout) are independent only for the low level composite cells; for the higher ones, the layout generators fix the layout as a function of the source Core. However, the DATools are an open system, and thus designers can (and do) add layout generators. In fact, the designers and the tool builders are essentially the same people. Some layout generators are design-specific, in the sense that they are used for only one design (so far); however, they are usually non-specific, in the sense that they implement a generally interesting layout technique that could be used in a later design. The layout generators are 'meta-data': the knowledge they embody is about VLSI design in some generality, not one specific design. The layout generators are thus beyond the scope of Informed Comparison, which is for comparing views, not verifying general design knowledge. PW-CoreLichen does not directly check the layout generators, but it *does* check their work.

The level of abstraction of the source Core is similar to the switch-level (some high-level directives concerning layout are also present), but it is easy for designers to think of it as being more abstract, because the schematics extractor is extensible. Special graphical sub-languages can be invented for various formalisms. For example, there is an extension for finite-state-machines: the designer draws the state diagram, and the extraction produces Core for the finite-state-machine. That Core goes all the way down to transistor netlists, but the designer need not see anything below the finite-state-machine. Of course, the programming language that provides the other way to create source Core also supports abstraction.

A simple charade makes the design data flow appear uniform: there is a layout generator that follows a naming convention to simply fetch the desired layout from a ChipNDale file. Thus, even though some layout cells are created by hand, all layout can be considered to come from layout generators. Other layout generators implement general layout techniques (such as abutment, routing, and standard cells) and more specific ones (such as various styles of logic arrays, decoders, memories, and datapaths).

4.1.3 The Relationship Between the Hierarchies of the Views

PWCoreLichen compares source Core and layout. The difference in level of abstraction is handled by calling a circuit extractor to get *extracted Core* from the layout. The extractor is trusted to do a sound translation, and it leaves links between the two descriptions. That leaves PWCoreLichen with the problem of comparing the two Core descriptions. The level of abstraction of the extracted Core is the same as that of the source Core: mainly switch-level, with a few annotations concerning layout. The DATools methodology stipulates the kinds of differences allowable between the hierarchies of the two Core views.

The DATools methodology is rather restrictive about the ways the cell structure of the two views may differ—there are only two. One difference is the presence of an extra intermediate cell type in either view (or, equivalently, the lack of an intermediate cell type in the other view). This difference gives welcome freedom to both designers and the layout generators. The other allowable difference is the appearance in the layout of several transistors in parallel, corresponding to one (wide) transistor in the schematic (this is a common design practice—at least in CSL). The parallel transistors must be of the same type and length.

The DATools methodology allows somewhat more complicated differences in wiring. The extracted Core never has any composite wires, because the layout system has no way of representing them. This is not a problem, since composite wires serve only to highlight regularity in atomic ones. For the atomic wires, the methodology requires that there be a partial surjection[2] from extracted public wires to source public wires. This allows two kinds of difference: an extracted public wire may not correspond to any source public wire, and a single source public wire may correspond

[2] A surjection is a function that is onto: every member of the declared range is related to at least one member of the domain. A partial function is one that may not have a mapping for every member of the declared domain.

to several extracted public wires. The first kind of difference arises from a feature of the extractor: it sometimes makes wires public that need not be so. The second kind of difference arises from a common design practice, *delaying connectivity up the hierarchy*. Where a source Core cell type has one wire, the corresponding layout cell may have several, because connecting them in that cell would cost more area than connecting them higher in the cell hierarchy. The methodology also allows a small difference between the atomic private wires: either view may have atomic private wires that are not connected to anything and do not correspond to any wire in the other view.

4.2 How PWCoreLichen Checks Consistency

PWCoreLichen starts by calling a circuit extractor to translate the layout into an approximately switch-level description using Core. Then copies of the source and extracted Core are made and reconciled. Because of the simplicity of the transformation repertoire, the reconciliation is done during the copying process: the reconciled copies are produced directly from the original Core. The base comparison is hierarchical structural comparison. PWCoreLichen extracts its key from information already known to PatchWork for the support of the layout generators.

PWCoreLichen gives transistors special attention. The transistor cell types are the atomic cell types of both the source and extracted Core. There are many different transistor cell types, each for transistors of a particular type (e.g., n, p, depletion), length, and width. PWCoreLichen first checks consistency and determines the correspondence between entities (including transistors) without regard to transistor shape, but *with* regard to transistor type; it then checks that the shapes of corresponding transistors agree within a designer-specified tolerance.

PWCoreLichen conceptually chops the hierarchies up into many smaller hierarchies, by dividing at the cell types that have corresponding cell types in the other view. The resulting sub-views are compared independently. Figure 4.1 shows such a division. The solid lines indicate which cell types use (via cell instances, which are not shown) which cell types, and the dashed lines indicate the correspondence

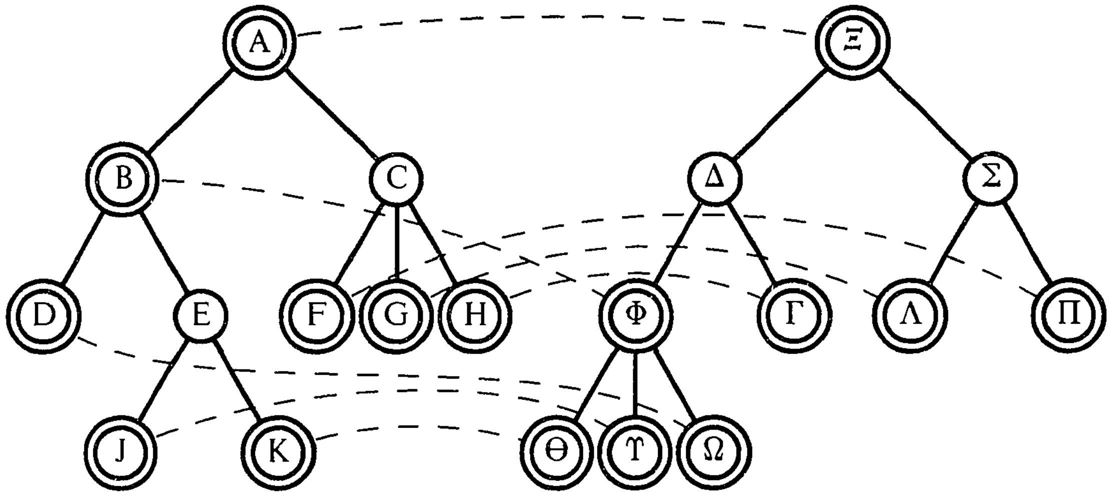


Figure 4.1: Division of Two Views by Corresponding Cell Types

between the cell types of the two views. The singly-circled cell types are flattened out in the reconciliation. In this example, each view is divided into two sub-views; the four sub-views are:

1. one with root A and leaves B, F, G, and H,

2. one with root B and leaves D, J, and K,

3. one with root Ξ and leaves Φ, Γ, Λ, and Π, and

4. one with root Φ and leaves Θ, Γ, and Ω.

Two sub-view comparisons are made: 1 vs. 3 and 2 vs. 4. PWCoreLichen works in a bottom-up fashion: it is applied to a sub-view only after it has been applied to the sub-views below it. This method meshes well with the rest of the DATools, wherein generators construct layout in a bottom-up way.

4.2.1 PWCoreLichen's Reconciliation

The key for PWCoreLichen is the correspondences, already known to PatchWork, between the source and extracted Core cell types, and between the source and extracted atomic public wires. These correspondences can be easily interpreted as invocations of transformations: a cell type in one view with no corresponding cell type in the other is to be flattened out; multiple extracted public wires that correspond to the same source public wire are to be merged; and an extracted public wire that corresponds to no source public wire is to be retracted (made private). Because of the DATools methodology, PWCoreLichen also applies the following three canonicalization transformations without even consulting the key: (1) every composite wire is removed; (2) every private atomic wire with no connections is removed; and (3) every set of parallel extracted transistors of the same type and length is merged. Because the last two are flat-significant, PWCoreLichen's version of Informed Comparison is blurred.

Flattening out intermediate cell types and removing composite wires are always flat-insignificant, while merging parallel transistors and deleting unconnected wires are always flat-significant but sound, because of the kind of consistency being checked. However, the remaining two wiring transformations are flat-insignificant only under certain conditions, and PWCoreLichen checks that they hold. A public wire can be retracted only if at every instance of its containing cell type that public wire is connected to a wire that is connected to nothing else. If PWCoreLichen finds that this condition does not hold for a public wire it must retract, an error message is given to the designer.

Merging atomic public wires is even trickier. It is only flat-insignificant if at every instance of the wires' containing cell type either

- those public wires are all connected to the same wire, or
- the wires they are connected to themselves will be merged.

Figure 4.2 shows some examples. Merging the four *Vdd* wires of the *Driver Array* is flat-insignificant;[3] merging the four *Gnd* wires and merging the last two *In* also are flat-insignificant. However, merging all four *In* wires is not. The figure does

[3]... assuming that there are no other instances of the *Driver Array* cell type, or that if there are, they also connect all four *Vdd* wires together.

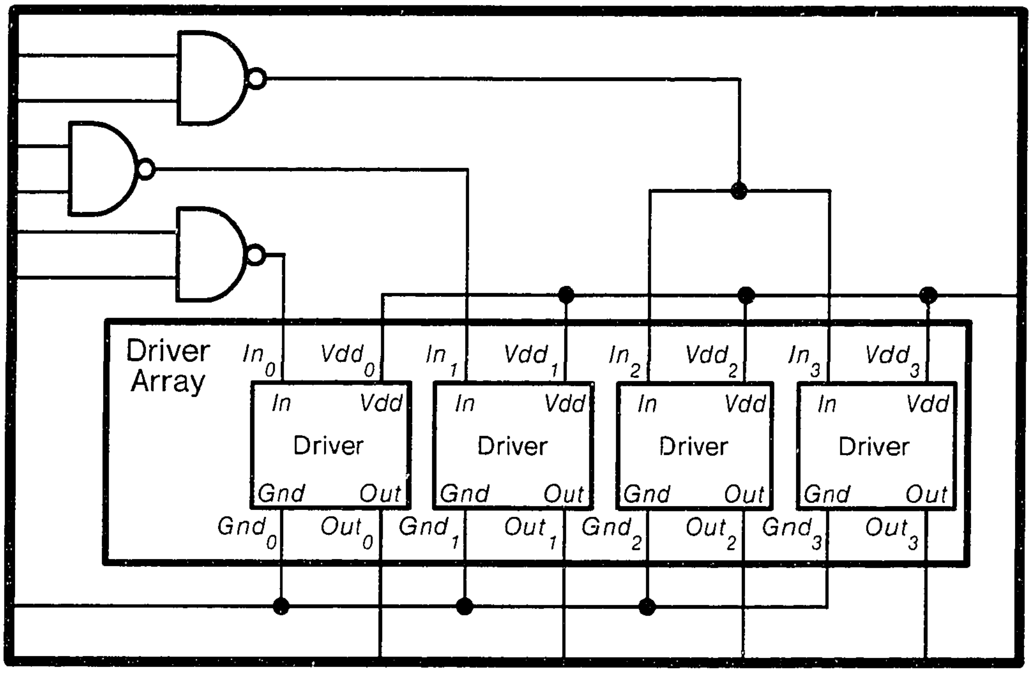


Figure 4.2: Array with Delayed Connectivity

not contain enough information to tell whether merging the four *Out* wires is flat-significant, which illustrates a problem PWCoreLichen has: within a sub-view it may not be possible to tell whether merging a set of public wires is flat-significant. PWCoreLichen solves this by posting, propagating, and discharging constraints. The constraints are posted on original extracted public wires of cell types that root sub-views. Each constraint is simply a subset of the atomic public wires of that cell type, and corresponds to the above 'it's-valid-to-merge' condition. Thus, the discussion above tells when to post these constraints, how to propagate them, and when they are discharged. Such a constraint reaching the root cell type of a chip is not necessarily an error—it means only that certain pins of the chip must be wired together externally.

4.2.2 PWCoreLichen's Base Comparison

Two views that are consistent according to the DATools methodology will have identical hierarchies after reconciliation. Thus a very simple and efficient base comparison, hierarchical structural comparison, will suffice. PWCoreLichen and the reconciliation guarantee that the base comparison starts with a known one-to-one correspondence between the cell types of the two reconciled views, and a known one-to-one correspondence between the public wires of those cell types. Each pair of corresponding cell types is compared structurally. No flattening is done—the subcells are the atoms of the structures compared. The structural comparison is done by labelled graph isomorphism, as discussed in Section 2.2.1. The reconciled views, which are produced directly from the original Core, are represented in the graph data structure used by the graph isomorphism checker.

4.2.3 A Note on Performance

Most of the time and space required by the algorithms and data structures of PWCoreLichen are linear, or nearly so, in the size of the inputs and outputs. The graph isomorphism checker takes a little worse than linear time. It uses a 'refine-an-automorphism-partition' technique modeled closely on Gemini [Ebeling83]. The step that merges parallel transistors takes an amount of time proportional to the sum of the squares of the number of transistors in each original extracted Core cell type; this could be improved to linear (by using hash tables), but has not been a practical problem. The remaining algorithms are linear.

Taking time and space proportional to the size of the inputs and outputs is rather good asymptotic performance—but what are the input and output sizes? Because PWCoreLichen can do little more to cell structure than flatten it, the size of a reconciled view can be an exponential function of the size of the original view. As discussed in Section 3.3, if flattening is the only reconciliation transformation of cell structure available, a great deal of flattening may be required to reconcile two views. This problem is inherent in the DATools methodology. Designers (and layout generator writers) avoid provoking this problem: they keep the original views

Seconds	Step
602	Extract schematics
965	Generate Layout
666	Extract circuit from layout
383	Copy & reconcile both views
61	Base Comparison

Table 4.1: Times to Generate and Compare the SIC

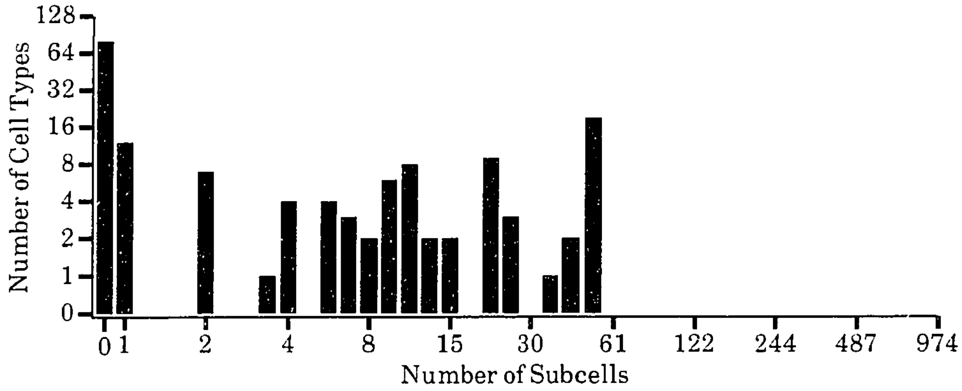


Figure 4.3: Original Source Core Cell Type Size Distribution in the SIC

close enough to avoid an excessive amount of flattening.

Table 4.1 gives the times for the steps in the generation and comparison of one of the chips designed with the DATools, the Scanner Interface Chip (SIC). The table shows that PWCoreLichen takes a minor portion of the time. This chip has about 40,000 transistors, and Figures 4.3, 4.4, and 4.5 show the distribution of the sizes[4] of the original source, original extracted and reconciled cell types, respectively. These histograms reveal that although there was a significant amount of flattening, especially for the reconciliation of the source Core, the resulting cell types were of a definitely manageable complexity.

How much better is PWCoreLichen than a program that flattens completely? This depends on the regularity of the design and the degree to which the hierarchies of

[4] Here the size of a cell type is the number of its subcells.

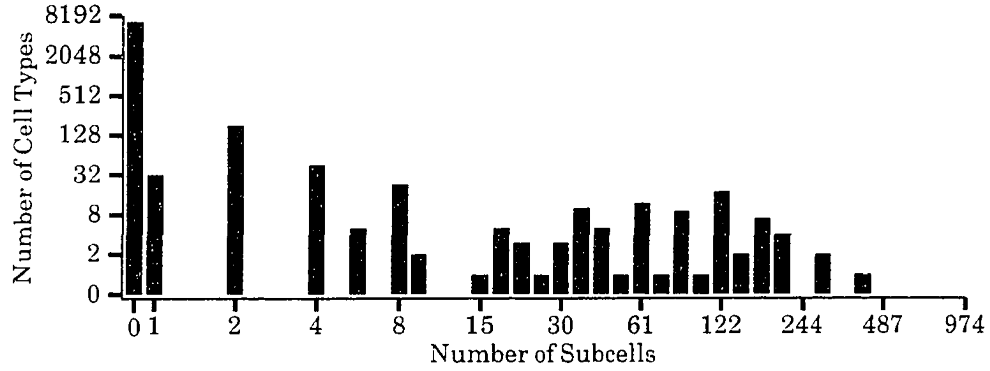


Figure 4.4: Original Extracted Core Cell Type Size Distribution in the SIC

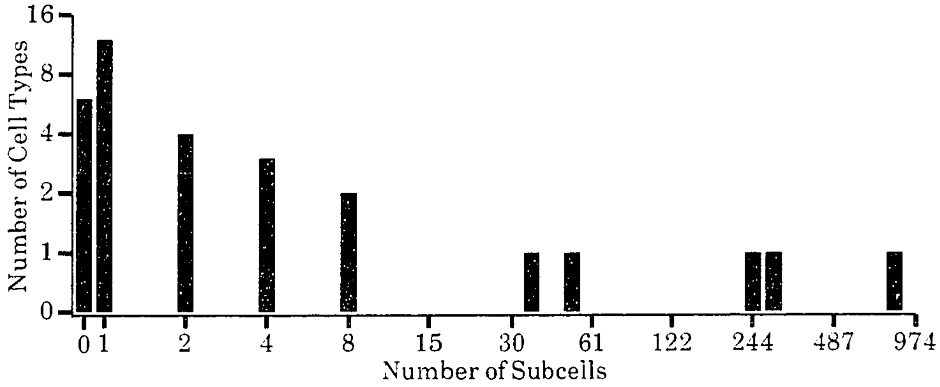


Figure 4.5: Reconciled Cell Type Size Distribution in the SIC

the two views differ. For the SIC, an estimate[5] can be made based on the above data and some statistics from the SIC. The following analysis rests on the conservative assumption that the structural comparison is simply linear in the graph sizes. Because of this assumption, the time for the base comparison increases by the *regularity factor* of the SIC, which is the ratio of the number of cell instances and wires in the completely flattened views to the total number in the hierarchical views. Each of the two reconciled hierarchical views of the SIC has about 1,500 cell instances and 2,600 wires, for a total of about 4,100 graph vertices. The completely flattened and reconciled SIC has about 35,000 transistors and 18,000 wires, for a total of about 53,000 vertices in the completely flattened graph, and thus a regularity factor of about 13. So the time for the base comparison would become about $13 \times 61 \approx 800$ seconds. The time to copy and reconcile the views also increases—flattening is even more time-consuming than simply copying. Assuming the time required to copy and reconcile is proportional to the size of the output, that time increases to $13 \times 383 \approx 5,000$ seconds. Actually, a fraction (experimentally measured to be less than 10%) of those 383 seconds are spent on reconciliation procedures that wouldn't take appreciably longer when completely flattening, and thus 4,500 seconds is a better estimate. A program that uses complete flattening thus would take about 5,300 seconds to flatten, reconcile, and structurally compare the two views of the SIC, which is about 12 times longer than PWCoreLichen takes to copy, reconcile, and structurally compare.

4.3 The Entity Correspondence Determined by PWCoreLichen

Figure 4.6 illustrates the correspondences relevant to PWCoreLichen. The *grand correspondence* relates the entities of the source Core with those of the layout; this correspondence is the composition of the *main correspondence* and the *extraction correspondence*. The extraction correspondence is maintained by PatchWork and is very straightforward (because the hierarchies of the layout and the extracted Core are very

[5] I tried to completely flatten the SIC—and ran out of memory!

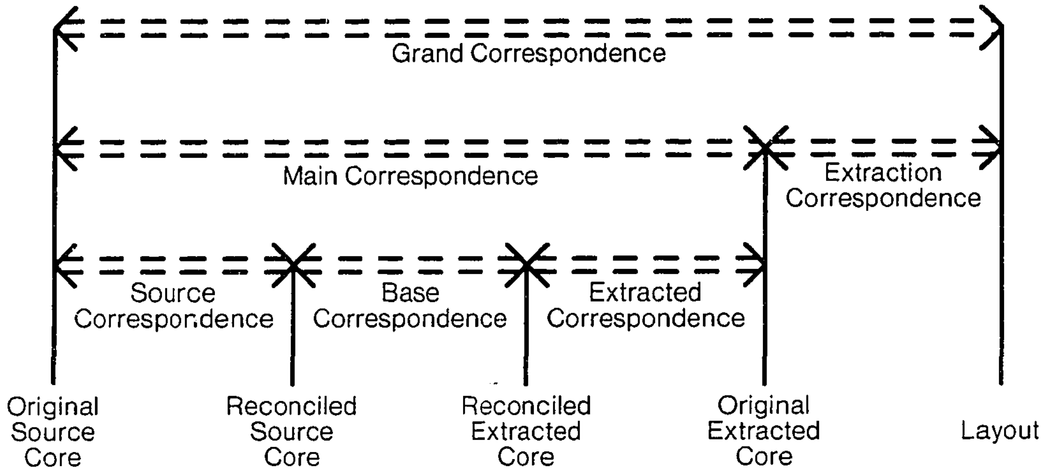


Figure 4.6: Correspondences in PWCoreLichen

similar). The rest of this section is concerned with the main correspondence, which is more illustrative of the correspondences produced by Informed Comparisons. The main correspondence is the composition of three subsidiary correspondences: (1) the *source correspondence*, between the original source Core and the reconciled source Core; (2) the *extracted correspondence*, between the original extracted Core and the reconciled extracted Core; and (3) the *base correspondence*, between the reconciled source Core and the reconciled extracted Core. This tripartite decomposition is characteristic of the correspondences produced by Informed Comparison: two are from the reconciliation of the two views, and the remaining one is from the base comparison. Because of the DATools methodology, the base correspondence is simple: it has a one-to-one association between the cell types of the two reconciled views, between the cell instances of the views, and between the wires of the views. Because of the transformations applied during the reconciliation, the reconciliation correspondences[6] are more complicated than the base correspondence. Because the main correspondence incorporates the reconciliation correspondences, it too is more complicated than the base correspondence. The ways in which the reconciliation transformations increase

[6] The *reconciliation correspondences* are those between reconciled entities and their origins; in PWCoreLichen, they are the source and extracted correspondences.

the complexity of the main correspondence are presented in turn.

4.3.1 PWCoreLichen's Base Correspondence

The base correspondence can be represented by a binary relation between the entities of one view and those of the other. The symbol $\sim$ is used to denote such a relation. Since there are three kinds of entities being related, $\sim$ is the disjoint union of three smaller relations ($\overset{t}{\sim}$, $\overset{i}{\sim}$, and $\overset{w}{\sim}$), one for each of cell types, cell instances, and atomic wires. The relation $\sim$ is one-to-one and total: every cell type, every cell instance, and every atomic wire of one view is related by $\sim$ to exactly one entity in the other view. The composite wires need not be included in $\sim$: since each view describes the relationships between its composite wires and its atomic ones, that plus $\overset{w}{\sim}$ suffices to describe the correspondences of a view's composite wires. Because of this, the deletion of composite wires adds no complexity to this formulation of the correspondence between view entities.

To compose two such correspondences, say $\sim_{12}$ and $\sim_{23}$, ordinary binary relation composition is used: $\sim_{13} = \sim_{12} \circ \sim_{23}$.

4.3.2 The Complexity Due to Flattening

The possible expansion of cell types adds considerable complexity. To describe the correspondences across such transformations, $\overset{t}{\sim}$ must be allowed to omit some cell types (the ones flattened out), $\overset{i}{\sim}$ must be changed to relate *introductory instance paths* instead of instances, and $\overset{w}{\sim}$ must be changed to relate *introductory wire paths* instead of wires. An *instance path* is a sequence of cell instances, where each instance is contained in the type of its predecessor. An instance path is introductory when it *starts at* a *tagged* cell type, *ends at* another tagged cell type, and does not *pass through* any tagged cell type. A tagged cell type is one related by $\overset{t}{\sim}$ to some other cell type. An instance path starts at the cell type that contains its first element, ends at the cell type instantiated by its last element, and passes through every cell type instantiated by non-final elements (or, equivalently, every cell type containing non-initial elements). A *wire path* is like an instance path, except that its last element is a wire. A wire path is *subrooted* when it starts in a tagged cell type and does not pass through any tagged cell type. A wire path is introductory when it is subrooted and ends with a wire that is either private or the only element in the path; a wire path is *secondary* when it is subrooted but not introductory. There is a one-to-one correspondence between the introductory instance paths of a sub-view and the cell instances of the result of flattening that sub-view. Similarly, there is a one-to-one correspondence between the introductory wire paths of a sub-view and the wires of the result of flattening that sub-view. An introductory wire path directly describes the 'highest' origin of one of the wires of the flat sub-view; the secondary wire paths describe the lower origins, of which there may be several. Because the main correspondence relates entities forwards through flattening and then backwards, paths must be used on both sides.

A given correspondence can be represented by many different binary relations. For every binary relation $\sim$ that directly states the correspondence of two intermediate cell types $t_1 \sim t_2$, there is an equivalent binary relation $\sim'$ that does not directly state that correspondence. Because $\sim'$ tags different cell types, its set of introductory instance and wire paths is different. In particular, the introductory paths of $\sim$ that start or end at t_1 or t_2 are not introductory in $\sim'$, but some of their concatenations are. Thus, the correspondence of t_1 and t_2 is represented indirectly in $\sim'$ by the correspondences of the longer paths. A binary relation $\sim$ is said to be *tighter* than an equivalent binary relation $\sim'$ if $\sim$ tags more cell types than $\sim'$. There is an analogous degree of freedom in the reconciliation: for every reconciliation that does not flatten t_1 and t_2, there is an alternate reconciliation that does (albeit unnecessarily). However, the binary relation used to represent a correspondence can be tightened or loosened without changing the reconciliation. Each correspondence has one tightest and one loosest representation. The loosest tags only the root and atomic cell types. The tightest tags every cell type that any other representation tags.

When two of these more complex correspondences are composed, simple composition of binary relations may not suffice. Consider composing $\sim_{12}$ (which relates V_1 with V_2) with $\sim_{23}$ (which relates V_2 with V_3) to get $\sim_{13}$ (which relates V_1 with V_3). If $\sim_{12}$ and $\sim_{23}$ tag the same cell types of V_2, simple composition of binary relations suffices: $\sim_{13} = \sim_{12} \circ \sim_{23}$. However, if $\sim_{12}$ and $\sim_{23}$ tag different cell types of V_2, this different must first be canceled. This cancellation is done by finding tighter or looser $\sim'_{12}$ and $\sim'_{23}$ that *do* tag the same cell types of V_2 (there is always at least one solution). Once this is done, composition of binary relations completes the composition of correspondences: $\sim_{13} = \sim'_{12} \circ \sim'_{23}$.

4.3.3 The Remaining Complexity

The remaining four transformations require relaxing the restrictions that $\sim$ be one-to-one and total. Because of the possibility of merging parallel extracted transistors, $\overset{i}{\sim}$ might relate one source introductory instance path to many extracted ones. Because of the possibility of merging public wires of a tagged extracted cell type, $\overset{w}{\sim}$ might relate one source introductory wire path with many extracted ones. Because of the possibility of retracting public wires of tagged extracted cell types, $\overset{w}{\sim}$ might be partial.[7] The possibility of deleting private wires with no connections is another reason $\overset{w}{\sim}$ might be partial. No change is required in the procedure for computing the composition of two correspondences.

In summary, the rules for $\sim$ are as follows.

- $\sim$ is the disjoint union of $\overset{t}{\sim}$, $\overset{i}{\sim}$, and $\overset{w}{\sim}$.
- $\overset{t}{\sim}$ is a partial one-to-one relation between the source Core cell types and the extracted ones. The tagged cell types include at least the root and atomic cell types, and maybe others.
- $\overset{i}{\sim}$ is a total one-to-many relation between the source introductory instance paths and the extracted ones.
- $\overset{w}{\sim}$ is a partial one-to-many relation between the source introductory atomic wire paths and the extracted ones.

[7] A partial relation between two sets S and T is one that does not guarantee that every member of S and T is related to at least one member of the other set.

4.3.4 A Simpler Presentation of a Correspondence

It should be possible for a designer sitting at a workstation to point at a cell instance (or wire or cell type) in one view get the DATools to highlight the corresponding part(s) of the other view. That seems inconsistent with the representation of the correspondence in terms of paths instead of single entities. This inconsistency is a manifestation of the fact that the correspondence may be so complicated that it is difficult to say what a single entity corresponds to. However, there are three mitigating factors:

- To the degree that the correspondence actually is simple, that simplicity can be determined from the paths, and the dialogue with the designer can be carried on in those simpler terms.

- The designer may not want all the details kept in $\sim$, in which case the dialogue can be simplified further.

- There are graphical, as well as textual, ways of interacting in terms of paths instead of single instances or wires.

This section suggests a way in which PWCoreLichen could accept simple queries of the correspondence, and provide relatively simple answers.

Each query consists of a single cell instance, cell type, or wire. The answer sought is the corresponding part(s) of the other view. The answer is determined by following three steps:

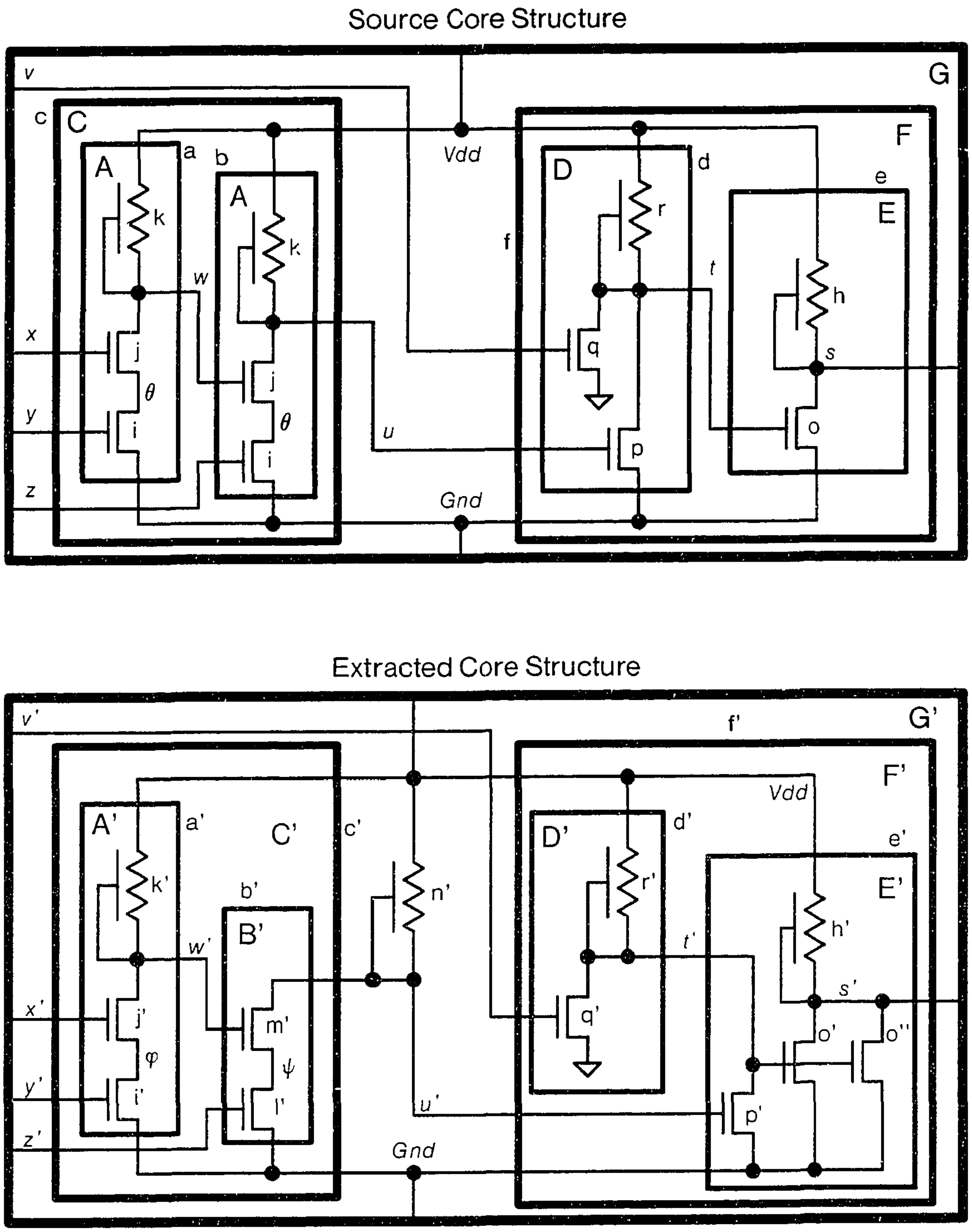
1. *normalize* the given entity to an equivalent structure of introductory paths in its own view,

2. follow $\sim$ across to a structure of introductory paths in the other view, and

3. *simplify* that structure of paths as much as possible.

Steps (1) and (3) use the same equivalences, in opposite directions.

A few examples follow. These examples concern the views shown in Figure 4.7; Table 4.2 shows $\sim$. Each example includes a summary of the form



Source Core Structure

Extracted Core Structure

Figure 4.7: Views for the Examples

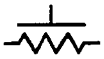
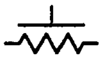
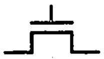
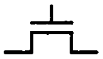
Source		Extracted
Cell Types		
F	$\overset{t}{\sim}$	F'
G	$\overset{t}{\sim}$	G'
	$\overset{t}{\sim}$	
	$\overset{t}{\sim}$	
Intr'y Instance Paths		
$\langle f\rangle$	$\overset{i}{\sim}$	$\langle f'\rangle$
$\langle c,a,i\rangle$	$\overset{i}{\sim}$	$\langle c',a',i'\rangle$
$\langle c,a,j\rangle$	$\overset{i}{\sim}$	$\langle c',a',j'\rangle$
$\langle c,a,k\rangle$	$\overset{i}{\sim}$	$\langle c',a',k'\rangle$
Intr'y Instance Paths (continued)		
$\langle c,b,i\rangle$	$\overset{i}{\sim}$	$\langle c',b',l'\rangle$
$\langle c,b,j\rangle$	$\overset{i}{\sim}$	$\langle c',b',m'\rangle$
$\langle c,b,k\rangle$	$\overset{i}{\sim}$	$\langle n'\rangle$
$\langle d,p\rangle$	$\overset{i}{\sim}$	$\langle e',p'\rangle$
$\langle d,q\rangle$	$\overset{i}{\sim}$	$\langle d',q'\rangle$
$\langle d,r\rangle$	$\overset{i}{\sim}$	$\langle d',r'\rangle$
$\langle e,h\rangle$	$\overset{i}{\sim}$	$\langle e',h'\rangle$
$\langle e,o\rangle$	$\overset{i}{\sim}$	$\langle e',o'\rangle$
$\langle e,o\rangle$	$\overset{i}{\sim}$	$\langle e',o''\rangle$
Intr'y Wire Paths		
$\langle Vdd\rangle$	$\overset{w}{\sim}$	$\langle Vdd\rangle$
$\langle Gnd\rangle$	$\overset{w}{\sim}$	$\langle Gnd\rangle$
$\langle z\rangle$	$\overset{w}{\sim}$	$\langle z'\rangle$
$\langle y\rangle$	$\overset{w}{\sim}$	$\langle y'\rangle$
$\langle x\rangle$	$\overset{w}{\sim}$	$\langle x'\rangle$
$\langle v\rangle$	$\overset{w}{\sim}$	$\langle v'\rangle$
$\langle s\rangle$	$\overset{w}{\sim}$	$\langle s'\rangle$
$\langle u\rangle$	$\overset{w}{\sim}$	$\langle u'\rangle$
$\langle c,w\rangle$	$\overset{w}{\sim}$	$\langle c',w'\rangle$
$\langle f,t\rangle$	$\overset{w}{\sim}$	$\langle f',t'\rangle$
$\langle c,a,\theta\rangle$	$\overset{w}{\sim}$	$\langle c',a',\varphi\rangle$
$\langle c,b,\theta\rangle$	$\overset{w}{\sim}$	$\langle c',b',\psi\rangle$

Table 4.2: Correspondence for the Examples

$$\alpha \approx \omega$$
$$[\beta \sim \chi]$$

where α is the single entity of the query, β is the equivalent structure of introductory paths in α's view, χ is the corresponding structure of introductory paths in the other view, and ω is the simplified answer. Although each example shows and discusses the intermediate steps, the designer need only give α and get back ω.

In the first example, the cell instance f in one view simply corresponds to the instance f' in the other.

$$f \approx f'$$
$$[\langle f\rangle \sim \langle f'\rangle]$$

This is what happens when the correspondence really is simple. The correspondence is directly represented in $\overset{i}{\sim}$ by two *unit* paths (a unit path is one that has only one element).

The next example shows that ambiguity arises when PWCoreLichen is asked about a cell instance k contained in a cell type A that gets flattened out in the reconciliation.

$$k \approx k' \mathrm{or}\, n'$$

$$[\langle c,a,k\rangle \text{ or } \langle c,b,k\rangle \quad \sim \quad \langle c',a',k'\rangle \text{ or } \langle n'\rangle]$$

Because k's containing cell type A is flattened out, simply giving k is not enough to uniquely determine what in the other view corresponds to k. The answer is that k corresponds to one or the other of two extracted transistors, depending on which k of the source is started from. On the other side, k' appears in only one introductory instance path, and thus can be used as a shorthand for that path. In this example and the previous one, the simple statements are completely faithful to $\sim$; in the following examples that is not so.

The next example shows that a "structure of introductory instance paths" can be complex.

$$A \quad \approx \quad A' \text{ or}(B' \text{ and } n')$$

$$\left[\begin{pmatrix} \langle c,a,i\rangle \text{ and } \langle c,a,j\rangle \text{ and } \langle c,a,k\rangle \\ \text{or} \\ \langle c,b,i\rangle \text{ and } \langle c,b,j\rangle \text{ and } \langle c,b,k\rangle \end{pmatrix} \sim \begin{pmatrix} \langle c',a',i'\rangle \text{ and } \langle c',a',j'\rangle \text{ and } \langle c',a',k'\rangle \\ \text{or} \\ \langle c',b',l'\rangle \text{ and } \langle c',b',m'\rangle \text{ and } \langle n'\rangle \end{pmatrix}\right]$$

This example uses both disjunctive and conjunctive structure, reflecting the facts that there is ambiguity about which A is meant and that A must be represented by its components. This example discards some of the information in $\sim$ by not considering the wire paths that pass through the cell types.

The final example restates the previous correspondence in an even less precise way.

$$A \quad \approx \quad \{A', B', n'\}$$

$$\left[\left\{\begin{matrix} \langle c,a,i\rangle, & \langle c,a,j\rangle, & \langle c,a,k\rangle, \\ \langle c,b,i\rangle, & \langle c,b,j\rangle, & \langle c,b,k\rangle \end{matrix}\right\} \sim \left\{\begin{matrix} \langle c',a',i'\rangle, & \langle c',a',j'\rangle, & \langle c',a',k'\rangle, \\ \langle c',b',l'\rangle, & \langle c',b',m'\rangle, & \langle n'\rangle \end{matrix}\right\}\right]$$

Instead of keeping the conjunctive structure and the disjunctive structure distinct, this example simply uses undifferentiated sets. This vaguer answer is easier to indicate with the typically limited highlighting facilities of most graphical editors. While this vaguer answer is also more confusing, it is good enough to give the designer a rough idea of what is going on—and that is all that some tasks require.

4.3.5 Summary of PWCoreLichen's Correspondence

The correspondence between the entities of the source Core and the entities of the layout is composed from the following four subsidiary correspondences:

1. the source correspondence, which relates the entities of the original source Core with those of the reconciled source Core;

2. the base correspondence, which relates the entities of the reconciled source Core with those of the reconciled extracted Core;

3. the extracted correspondence, which relates the entities of the reconciled extracted Core with those of the original extracted Core; and

4. the extraction correspondence, which relates the entities of the original extracted Core with those of the layout.

Because of the DATools methodology, the base correspondence is extremely simple: a one-to-one association between the cell types, cell instances, and wires of the reconciled views. The extraction correspondence is also simple: it associates each entity of the extracted Core with the layout picture elements that form its graphical representation. The other two correspondences, the source and extracted, are more complex, because of the possible reconciliation transformations. The added complexities consist of (1) changing the focus from instances and wires to introductory instance and wire paths, in order to handle the flattening of cell types, and (2) relaxing the restriction that the association be one-to-one, in order to handle merging and removal of features. Although the correspondence is represented by an association between paths, dialogues with designers can often be carried out in simpler terms.

4.4 PWCoreLichen Summary

PWCoreLichen is a simple example of Informed Comparison. The DATools methodology determines PWCoreLichen's reconciliation repertoire and base comparison. PW-CoreLichen is limited in two ways: (1) its only reconciliation transformation that can make a major change in the cell structure is flattening out a cell type, and (2) it compares views at the same level of abstraction. Because of these limits, PWCoreLichen can find its key in design data already captured for other purposes. Another consequence of the limited reconciliation repertoire is that the reconciled views may be very flat, which makes for both a costly reconciliation and a costly base comparison. Designers avoid these costs by restraining the degree to which the hierarchies differ.

PWCoreLichen is fast enough and useful enough that its use is a standard step in chip design in CSL. PWCoreLichen has caught real bugs, in both hand work and in layout generators.

Since PWCoreLichen is an Informed Comparison technique, the correspondence it determines between the entities of the two compared views is more complex than that determined by many existing techniques. However, because of PWCoreLichen's limitations, its correspondences are not as complex as Informed Comparison correspondences can be.

Chapter 5

The Lichen and MIPS-X Study

The previous chapter presented a simple, restricted instance of Informed Comparison: PWCoreLichen. This chapter presents a more general, and therefore more complex, instance: the Lichen and MIPS-X study. Lichen is a general program for transforming views,[1] while MIPS-X is a microprocessor designed at Stanford. An Informed Comparison of two views of part of MIPS-X was studied. PWCoreLichen is simple and restricted because of the design methodology it supports. The Lichen and MIPS-X study was more general because it was intended to develop and explore the method of Informed Comparison. The experience with Lichen and MIPS-X shows how a richer set of transformations enables a tighter correspondence between the reconciled hierarchies (that is, generally smaller cells), which increases the benefits of hierarchical methods at the cost of a more complex key and reconciliation.

[1] The name invites comparison with the plant, which is composed of a fungus and an alga in a symbiotic relationship. In a rough analogy, the program Lichen enables two views of a circuit to have a symbiotic relationship; also, by virtue of the near independence of the cell type comparisons in the base comparison, Lichen enables two base comparison techniques to have a symbiotic relationship. Furthermore, lichens grow on rocks, in which silicon figures prominently; Lichen is concerned with VLSI, in which silicon also figures prominently. The name "Lichen" is also apt because its pronunciation is the same as that of "liken", to "compare". Finally, the name "Lichen" fits well into the botanical tradition of CSL. Thanks to Carl Black for encouraging and helping me to find such a good name.

5.1 Overview

The study initially focused on reconciliation, because that is the central new idea of Informed Comparison. The purpose of the study was to explore the possibilities of reconciliation in general and to try it on a real example. The questions were:

- What is a good set of transformations to have available?
- How large and complex will a real key be?
- How much detail must be specified in the transformations?
- How abstract must the transformations be in order to keep the key size small?

The plan was to make an initial hypothesis of what a good set of transformations is; to implement them in a program called Lichen; and to reconcile a pair of views from a real design project, enhancing Lichen's repertoire of transformations as required. Later sections report on the resulting repertoire of transformations in Lichen and the key used for the actual reconciliation.

I initially expected that the reconciliation would transform the two views so that they could then be compared in a straightforward way (using simulation and structural comparison), which I will call *Plan A*. However, doing the reconciliation revealed that there are differences between the two MIPS-X views that cannot be removed by flat-insignificant transformations, and which Plan A cannot handle. I therefore expanded the study to include devising a new base comparison method that can verify the consistency of two views that differ in those ways.

While working on the reconciliation I discovered that the two views of the MIPS-X are inconsistent: certain debugging features, added late in the design, are represented in one view but not in the other. I then edited my copy to remove these features (in a real application, the designers would have made the views consistent by adding the debugging features to the view that lacked them). This chapter presents the comparison of the consistent views. While Informed Comparison *per se* is not confused by inconsistencies, the base comparison method used in this study would generate many secondary error reports as a consequence of the primary inconsistencies.

5.2 Introduction to MIPS-X

I chose the MIPS-X design [Horowitz87] to be the test case because of its availability and difficulty. MIPS-X has about 150,000 transistors and an area of about 0.6 cm^2. Although the design project was headed by two members of my reading committee, I was not involved in the design of MIPS-X, and the MIPS-X design team did not take Informed Comparison or my study into consideration when choosing their methodology and producing their design. Most of the MIPS-X design work was completed before I decided to use it in my study.

The design team at Stanford created and used two views in the MIPS-X design: a functional simulation, called the *funsim*, and a layout. For the most part, the funsim was created first, and then the layout; however, some of the layout work (exploring alternatives for critical structures) began very early. The MIPS-X was divided into eight major parts, and the same designer(s) were generally responsible for both the funsim and the layout of each part.

The methodology employed in the MIPS-X design places very little emphasis on keeping the hierarchies of the two views similar. At the top level in both views, the MIPS-X is divided into the same eight major parts. Below that, the methodology places no constraints on the relationship between the hierarchies used in the views; the designers were free to do whatever made their tasks easiest, and they did. For this reason, the MIPS-X views make a strong test of the ability of Lichen to reconcile two real views.

The funsim consists of two parts: a general purpose simulation kernel, and a model of MIPS-X. Both are written in Modula-2 [Wirth83]. The kernel takes a flat circuit structure, and simulates it with an event-driven selective-trace technique. The wires are modeled by 32-bit Modula-2 integers. The computational models of the components are Modula-2 co-routines that fetch values from the wires, compute, and then selectively drive values onto wires. The circuit structure is described in the model by calls on kernel procedures to create and connect components and wires. Although the circuit structure given to the kernel is flat, the model code that describes that structure has hierarchy. Unfortunately, little discipline was followed in creating this program structure. I would have preferred to extract the hierarchy of the funsim by program, but I found it more expedient to do so by hand; Lichen thus was given a hand-translated version of the funsim view.

The layout, created with the Magic [Ousterhout84] system from U. C. Berkeley, is essentially a hierarchical description of a set of colored rectangles. Magic includes a circuit extractor, which produces an essentially electrical view of the design. A few geometric abstracts, such as perimeters, areas, and positions, also appear in the extracted circuit description. The extracted circuit is put into ".ext files", one composite cell type per file. The atomic cell types are transistors and parasitic capacitors, which are fully described at each instantiation. In Magic's formulation of hierarchy, cells have no explicit interfaces; connections can reach down into descendant cells to pick out the desired wire.

The comparison studied in this chapter was between the funsim and the electrical view presented in the ".ext" files.

The comparison experiment focused on the program counter unit (the *pc*),[2] which is one of the eight major parts of MIPS-X. The funsim view of the pc is particularly detailed, going all the way down to simple Boolean gates and memories; this makes the best opportunity for testing reconciliation. Although the experiment focused on the *pc*, it was necessary to pay some attention to the rest of MIPS-X because some of the differences between the views concern whether certain components are in the *pc* or another of the eight major parts.

5.3 Lichen

Lichen is a program for doing general reconciliations. It is written in Cedar [Swinehart86], and thus run on a Dorado [Lampson81]. Lichen considers a view to consist of a structural hierarchy with other non-structural information attached. Although the transformations are focused on the hierarchy, the non-structural information must also be transformed accordingly and sometimes is used to help specify

[2] ...for two reasons: (1) I expected that would be somewhere near the point of diminishing returns, and (2) Lichen runs out of memory when it tries to read in the whole MIPS-X layout view.

the transformations. The transformations known to Lichen fall into four categories: flat-insignificant hierarchy transformations, flat-significant but behaviorally insignificant transformations, transformations of Lichen's representation, and transformations of the non-structural information.

5.3.1 Lichen's Notion of a View

Lichen considers a view to have a structural hierarchy and other non-structural information. The hierarchy is constructed out of cell types and instances, wires, and ports, as outlined in Section 2.1.1. In Lichen, wires and ports themselves have hierarchical structure. Lichen has a special-case representation for cell types with a certain kind of regularity, which are called *arrays*. The non-structural information manipulated by Lichen consists of names and some geometrical data.

Wire and Port Structure

In Lichen, wires and ports have hierarchical structure, without a type/instance duality, and without explicit interfaces. Simply put, wires and ports have tree structure: a wire (port) either is a leaf, or has a decomposition into a sequence of child wires (ports). The composite wires and ports simply make very evident some regularity among the leaves; in particular, a composite port and wire are connected at a site if and only if their corresponding children are.

Lichen allows wires and ports to have structure in order to approximate the designer's way of thinking. A very common example of how designers apply structure to wires and ports is to think of a set of wires running in parallel as one wire (called a *bus*). Designers also go further than Lichen, sometimes applying more than one structure to the same set of wires. An example is using alternate ways of breaking up an instruction word. This richer structure could be captured by allowing the wires and ports to have directed acyclic graph (DAG) structure, instead of just tree structure. With DAG structure, a wire or port could be used more than once as a child. The current version of Lichen forbids this richer structure because allowing DAG structure would complicate the transforming of ports and wires, and it is not clear that this added complexity is necessary. It was not needed in the MIPS-X study.

Non-Structural Information

In addition to structure, Lichen views include names and some physical information. This is actually a rather limited subset of all the kinds of non-structural information used in VLSI design. However, it suffices for the purposes of the Lichen and MIPS-X study. In general, the only non-structural information that must be manipulated in an informed comparison is that required by the base comparison; in this case, the names alone would suffice. By manipulating *some* non-structural information, the Lichen and MIPS-X study suggests that general non-structural information can be handled.

Lichen profits from acknowledging two common facts about names: names are structured, and one entity can have multiple names. All four kinds of Lichen's structural entities can have multiple names (or even no names at all). The names of three kinds—ports, wires, and cell instances—are structured: each name is a sequence of steps, where each step is a string or an integer. The names for ports and wires are directly associated with the ports and wires, even those that are children of others; this contrasts some systems in which name steps are attached to links in the wire structure.

In a Lichen view, physical information can be associated with structural entities in the following ways: a cell type may have a bounding box, and a cell instance may have a geometrical transformation (composed only of translation, 90-degree rotation, and mirroring in X or Y). The bounding box of a cell type must include the transformed bounding boxes of its subcells.

Arrays

Arrays are common, regular structures. They offer many opportunities for efficiency, because they contain multiple instances of the same cell type, and because they are regular. Lichen has two representations for the contents of a cell type: one directly follows the discussion of Section 2.1.1 and is completely general; the other is specialized for representing array structure. Lichen's notion of arrays is as follows.

In an array, all the subcells are instances of the same cell type. The subcells, also called the array's *elements*, are arranged on a finite two-dimensional grid. Each position is uniquely associated with an *index*, which is a pair of integers. The indices start at $\langle 0,0\rangle$. Lichen uses the names X and Y for the dimensions in which an array is arrayed, but Euclidean geometry is not relevant here—the structure of an array is only concerned with the topological properties of the connectivity.

In a Lichen array, all the connectivity is a consequence of regular connections between adjacent elements; any connectivity established by a context of an array is handled in Lichen by a cell instance and the context in which it instantiates the array. Lichen makes a distinction between the regular *starter* connections between adjacent elements and the *ultimate* connectivity, which is the transitive closure of the starter connectivity. Because of edge effects, the ultimate connectivity can have irregularities, even though the starter connections are absolutely regular.

Lichen extends the applicability of its array concept beyond traditional limits by handling concepts like mirroring every other element. It does this by considering an array to have a two-dimensional *period*, where the starter connectivity is regular with that period. A period will often be denoted by the symbol τ or the pair $\langle \tau_x, \tau_y\rangle$. For example, an array wherein the odd rows are mirrored would have a period of $\tau = \langle 1,2\rangle$. Continuing to borrow from signal-processing terminology, we can also talk about *phases*. A phase is a position within a period: each index $i = \langle i_x, i_y\rangle$ can be divided (independently in each dimension) by the array's period τ to yield a cycle number k and a phase ϕ:

$$\begin{aligned} i_x &= k_x\tau_x + \phi_x \\ i_y &= k_y\tau_y + \phi_y \\ 0 \le \phi_x &< \tau_x \\ 0 \le \phi_y &< \tau_y \end{aligned}$$

Phases are interesting because all the elements and starter connections at a given phase have certain regularities.

In a Lichen array, the physical information (the instantiation transformations) is also regular: all the instances at a given phase have the same rotation and mirroring, and their translations progress in a linear fashion—with the structural X dimension aligned with the physical X dimension, and similarly for Y. The translations for a given phase ϕ are characterized by two vectors $\langle \dot{x}_\phi, \dot{y}_\phi \rangle$ and $\langle \bar{x}_\phi, \bar{y}_\phi \rangle$: the translation for an index with phase ϕ and cycle number k is

$$\begin{aligned} \Delta X &= k_x \dot{x}_\phi + \bar{x}_\phi \\ \Delta Y &= k_y \dot{y}_\phi + \bar{y}_\phi \end{aligned}$$

Thus, Lichen's representation of the instantiation transformations does not depend on the size of an array, only on the period.

5.3.2 Flat-Insignificant Hierarchy Transformations

The preceding sections present Lichen's formulation of views; this and following sections present Lichen's transformations. Those transformations fall into four broad categories, and this section presents the first: flat-insignificant hierarchy transformations. There are 16 of these transformations, in two subgroups: those that manipulate only ports and/or wires, and those that focus on cell structure. Their level of abstraction is not very high (although, as will be seen later, it is high enough to make keys reasonably succinct).

Transformations come in pairs: for every transformation, there is an inverse transformation. This is not evident in PWCoreLichen, because its repertoire of transformations is so restricted. Even Lichen does not implement both transformations of every pair, due to lack of need. Nevertheless the pairs are at least conceptually complete.

These transformations are not expensive to apply. Most take an amount of time that is linear in the number of entities and relationships that obviously must be touched. For example, flattening out a cell instance is linear in the number of wires and subcells of that instance's type.

Port and Wire Transformations

The wiring of two flat-identical views can differ in five ways. Fortunately, four of these five are of such a restrictive form that they can be reconciled almost completely by canonicalization transformations. This is fortunate because choosing the ultimate hierarchy and planning the reconciliation does not require much intelligence; there is no

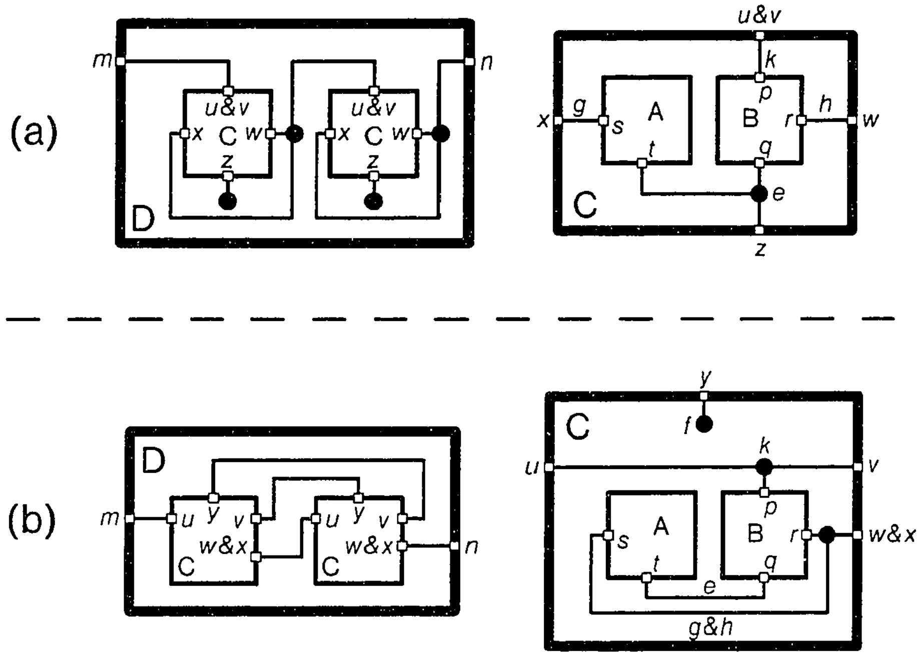


Figure 5.1: Two Flat-Identical Views with Different Wiring

designer involvement. Although the fifth kind of difference can be quite complicated in general, it occurs in only a simple way in MIPS-X for methodological reasons, and Lichen has a canonicalization transformation that reconciles this restricted difference.

The five ways in which the wiring of flat-identical views can differ follow. Figure 5.1 shows[3] examples of the first four.

1. Where one view has a private wire, the other view can have a *pointlessly public* one. A pointlessly public wire is one that is not connected to anything else outside its cell type. Wire e[4] in cell type *C* of Figure 5.1a is an example of a pointlessly public wire (assuming that the two instances of *C* shown are the only ones).

[3] This figure, unlike many others, explicitly represents (with open boxes in the cell boundary) and labels the ports.

[4] That wire's name actually is ⟨"e"⟩, because wire names in Lichen are sequences. In this example, and others, all the name sequences have only one element, which is a string, and thus the notational overhead can be (and is) dropped.

2. Where a cell type in one view *pointlessly imports a wire*, the corresponding cell type in the other view can refrain from doing so. A pointlessly imported wire is a public wire that is connected to nothing else in its cell type. Wire f in cell type C of Figure 5.1b is an example of a pointlessly imported wire; it has no corresponding wire in Figure 5.1a. This difference and the previous one are *inside/outside duals* of each other: the roles of the 'inside' of a cell type and the 'outside' are reversed.

3. Where a public wire in one view is exported through a certain number of ports,[5] the corresponding public wire in the other view can be exported through a different number of ports. Furthermore, the external connections can be distributed differently among those ports. A multiplicity of ports exporting one wire is called a *split port*. The ports u and v exporting wire k of cell type C in Figure 5.1b are a split port. In Figure 5.1a the corresponding public wire, also named k, is exported by only one port (which has two names, u and v).

4. Where a cell type in one view has a set of ports that are all connected together at every instance of that cell type (the ports are *unnecessarily distinguished*), the corresponding cell type of the other view can have just one port (or a set of a different size, or with a different distribution of internal connections). This is the inside/outside dual of the previous difference. The ports w and x of cell type C in Figure 5.1a are unnecessarily distinguished; they correspond to the single port named both w and x in Figure 5.1b. The root cell type of a view presents a special problem: the whole view is itself of a part of a larger system, which may always connect certain ports of the root cell type together—but the fact that those ports are unnecessarily distinguished cannot be determined by examining the given view. Lichen solves this by allowing the key to mark sets of ports of the root cell type as unnecessarily distinguished. Of course, whatever is verifying the larger system needs to check this assertion.

5. The two views can differ in their composite wiring, even though their atomic wiring is identical. In MIPS-X, this takes a particularly simple form: the funsim

[5] Lichen's formulation of hierarchy allows more than one port to export the same wire.

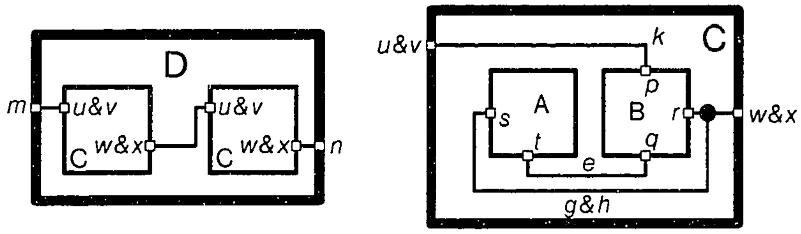


Figure 5.2: Wiring Differences Reconciled

has composite wiring, and the layout does not (Magic does not have composite wiring).

The first two kinds of difference are removed by a canonicalization transformation called simply *cleaning up the view*, which *retracts* (makes private) every pointlessly public wire and *sweeps up* (deletes it and its exporting ports) every pointlessly imported wire. Retracting or sweeping up one wire may make another wire pointlessly public or pointlessly imported; cleaning up the view removes not only the initially pointless features, but also the ones revealed in the process.

The third and fourth kinds of difference are removed by a canonicalization transformation called *unifying ports*, which merges split ports and merges *undesirably distinguished* ports. A set of ports are undesirably distinguished if they are unnecessarily distinguished and also, if their cell type has array structure, merging them would not make their cell type irregular. For example, recall Figure 4.2; the four *Vdd* ports of the *Driver Array* are undesirably distinguished; the last two *In* ports are unnecessarily, but not undesirably, distinguished. Also, ports w and x of cell type C in Figure 5.1a are undesirably distinguished: their cell type does not have array structure. Figure 5.2 shows the result of cleaning up the two views of Figure 5.1 and unifying ports; both views give the same result.

The fifth kind of difference is removed by a canonicalization transformation called *deducing port and wire structure*. It creates composite wiring along lines suggested by the names. For example, in the original layout view of MIPS-X, there is a wire named ⟨"*PCBus*", 0⟩, another wire named ⟨"*PCBus*", 1⟩, and so on to ⟨"*PCBus*", 31⟩;

deducing port and wire structure creates a composite parent wire for them, named simply ⟨*"PCBus"*⟩.

Sometimes it is necessary for the key to explicitly invoke transformations that create pointless imports or exports, because of the interaction of wire structure with the above canonicalization transformations. When cleaning up a view, what should be done to a composite wire of whose components some are pointlessly public and some are not? Rather than break up a composite wire in order to give its components disparate treatment, Lichen, when cleaning up a view, alters a composite wire only if all its components should be altered. This reluctance to break up composite wires helps to keep the layout view consistent with the funsim view, which, in its original form, is unable to express disparate connections for the components of composite ports and wires.[6] However, the original layout has disparate treatment of some bus elements. This difference is reconciled by firstly exporting and importing the missing bus elements to and from certain cells, secondly deducing port and wire structure, and thirdly cleaning up the view. Thus these pointless publics and imports are 'held on to' by their composite parents by the time the view is cleaned up, in both the layout and funsim views.

Cell Transformations

The cell transformations are the most significant—they change the basic organization of a view. When transforming the cell structure, changes to the port and wire structure must also be made. Fortunately, the key does not need to mention these small details: for each cell structure transformation there is one *most natural* way to handle the port and wire structure.

Lichen offers six pairs of cell structure transformations:

1. create/delete a cell type that *contributes nothing to the flat view*;

2. distinguish/undistinguish cell types;

3. flatten/un-flatten a cell type or instance;

[6] All the disparate treatment of bus components is done in the Modula-2 code for atomic cell type behavior.

4. split/merge cell types;

5. raise grandchildren/lower children;

6. and transpose (which is its own inverse).

The first two pairs reconcile certain kinds of trivial differences. Each of the next three can be expressed in terms of the other two in almost any combination. The last transformation can be expressed in terms of preceding ones, in a number of different ways. Lichen offers this diversity because use of the most direct transformation makes for more succinct keys.

A cell type contributes nothing to the flat view if it has no subcells and no split ports, or if it is not instantiated. Thus, deletion of such cell types is clearly flat-insignificant, as is the inverse transformation. The canonicalization transformation of cleaning up the view, introduced earlier, deletes all the cell types that contribute nothing to the view. Thus, designers need not invoke the creation or deletion transformations directly.

Some of the cell transformations, such as split/merge and raise/lower, affect every instance of a given cell type. However, in some cases the relationship between the two hierarchies being reconciled is such that only some of the instances of a cell type should be affected. The transformation of distinguishing cell types removes that conflict, by changing those instances to be instances of a new cell type that is equivalent to the given cell type.

The flatten/un-flatten transformations can be applied to cell types or cell instances. Flattening out a cell instance replaces that instance with the contents of its type, suitably interconnected. This is the smallest possible amount of flattening. Flattening out a cell type consists of flattening out each of its instances (Lichen deletes a cell type after its last instance is flattened out). The inverse transformations are un-flattening to a cell instance and un-flattening to a cell type. Flatten/un-flatten transformations can be used to reconcile the difference illustrated in Figure 2.5 on page 14: Flattening out the *inv pair* cell type converts the structure of part (c) into that of part (d), and un-flattening to the *inv pair* cell type does the reverse. To specify the type-flattening transformation requires only a name of the cell type to flatten;

the inverse transformation requires much more: (1) all the names of the cell type to create, (2) all the names of the instances to create, and (3) for each new instance, the set of old instances to which it corresponds. In the above example, the last two parts of the un-flattening specification can be succinctly expressed:

$$\text{for } j \in [0..15] : \langle\text{“}\mathit{pair}\text{”}, j\rangle \Leftarrow \{\langle\text{“}\mathit{inv}\text{”}, 2j\rangle, \langle\text{“}\mathit{inv}\text{”}, 2j+1\rangle\}\,.$$

A cell type can be split in two, and two cell types can be merged into one (provided their instances can be paired up appropriately). Splitting a cell type replaces one cell type with two that partition the original type's subcells amongst themselves. The ports and wires are not exactly partitioned—some ports and wires may need to show up in both[7] of the new cell types in order to keep the same communication pattern.

Raising grandchildren and lowering children are like fractional flattening and un-flattening. Raising grandchildren takes a subset of a cell type's subcells out of that cell type and puts a copy of them (suitably interconnected, of course) next to each instance of that type; lowering children is the inverse. Raising grandchildren can be done by splitting the cell type and then flattening out one of the two resultant cell types; lowering children can be done by un-flattening to a cell type and then merging it with another. Also, flattening and un-flattening can be constructed from raising grandchildren and lowering children (and creation and deletion of cell types that contribute nothing to the flat view). The reconciliation of the difference illustrated in Figure 2.8 on page 17 could use raising grandchildren to move the pulldown transistors out of the *E!t* cells of part (b), or could use lowering children to move pulldown transistors into the *Elt* cells of part (a).

The final cell structure transformations are *transpositions*: they interchange two adjacent levels of structure. This can be made clear by an analogy with programming-language array and record structures: transposing changes an array of records into a record of arrays, and vice versa. For example, transposition can convert between the function slicing structure of Figure 2.3 on page 12 and the bit slicing structure of Figure 2.4.

[7] Lichen's formulation of hierarchy does not allow one port or wire to be in two cell types; what actually happens is that in each of the two cell types there is a port or wire corresponding to the port or wire of the original cell type.

A Note on Generality

Any flat-insignificant transformation of a view without composite ports or wires can be expressed as a combination of flattening out some cell types and unflattening to some other cell types, if the unflattening transformation is parameterized by the wiring alternative to effect. This is easily seen: a transformation is flat-significant if the original and transformed views would be equivalent after flattening; thus, the combination of flattening out all the original cell types and unflattening to all the transformed cell types would serve. In many cases, simpler combinations also suffice.

If the unflattening transformation is not parameterized, but always takes the "most natural" wiring alternative, the following three pairs of transformations restore the power lost by that restriction: (1) pointlessly import/unimport a wire, (2) pointlessly export/unexport a wire, and (3) split/merge ports. These pairs change exactly those aspects of wiring fixed by choosing the "most natural" alternative when unflattening.

5.3.3 Flat-Significant Behaviorally Insignificant Transformations

There is only one pair of these: creation and deletion of a wire with no connections. These transformations need not be directly invoked from the key: the canonicalization transformation of cleaning up the view, introduced earlier, also deletes every wire with no connections. These transformations, and the non-structural ones introduced below, are flat-significant. Thus Lichen can be used in blurred Informed Comparisons.

5.3.4 Transformations of Lichen's Representation

Lichen has some choice in how it represents the internal structure of an array-structured cell type: either the general representation or the one for arrays can be used. Furthermore, in the array representation, there is some freedom of choice for the period: if a period $\langle \tau_x, \tau_y \rangle$ is acceptable, so is $\langle n\tau_x, m\tau_y \rangle$, for any integers n and m (although if n and m are so big that the resulting period is larger than the size of the array this is rather pointless). Lichen has two pairs of transformations that do nothing more than change which of these alternatives is used.

One pair changes the representation of a cell type's internal structure between the general representation and the one for arrays. One direction, from the array representation to the general, can always be applied. The other has restricted applicability: the cell must be sufficiently regular.

The other pair changes the period of arrays. Again, one direction (multiplying the period) always can be done, and the other (dividing the period) can be done only if the cell has sufficient regularity. Lichen offers a canonicalization transformation that ensures that every array's representation uses the smallest possible period.

5.3.5 Non-Structural Transformations

Lichen offers some transformations of the naming and of the physical information. The only transformation of physical information offered is simply deleting it. This is sound because the base comparison technique does not use the physical information. Deleting it is desirable because that increases the regularity of the pair arrays (presuming the pair cells have been flattened): because every other row (or column) is mirrored, the physical information is periodic with a period of 2 in one dimension—but all the other information is periodic with a period of $\langle 1, 1\rangle$.

Names are important to Informed Comparison, for three reasons.

- Names guide the deduction of composite ports and wires.
- Names are useful hints to the base comparison.
- Names are often used in communicating with people.

Lichen has three transformations, *renaming*, *inheriting names*, and *pruning less interesting names*, that change names to better serve these purposes.

In the Lichen and MIPS-X study, renaming was used mainly to name wires not named in the original layout. Neither Informed Comparison nor Lichen requires every wire to have a non-machine-generated name; but naming some wires is helpful. A minor use was to remedy inconsistent naming. In a design carried out with Informed

Comparison in mind, these uses would not be necessary—the original views would be edited to correct these deficiencies, if they arise at all. However, renaming would still be useful. Renaming is done before deduction of structured ports and wires; instead of reconciling the composite wire structure of the funsim with that of the layout after the deduction, the names of the layout's atomic wires are manipulated so that the subsequently deduced structure matches that of the funsim. For example, a certain 32-bit bus in the MIPS-X layout corresponds to a 31-bit bus and an independent wire in the funsim; the key renames the 0^{th} element of the 32-bit bus to give it a distinctive name and renames the other 31 elements to shift their subscripts down by 1 (there is a concise notation for this); wire structure deduction is done later, and the result matches the wire structure of the funsim.

The name inheritance and pruning transformations work together to propagate names throughout a view to an appropriate degree, so that the designers are not unduly burdened with labelling layout and yet most ports and wires have some names that are reasonably suggestive of their roles. Name inheritance propagates names 'up' the cell structure: a port inherits the names of the wire it exports, and a wire connected to a port at a cell instance inherits the concatenations of the instance's names and the port's. Actually, a name is inherited only if it is at least as *interesting* as the names already on the inheriting entity. Lichen determines the interestingness of a name from how many strings it has, how many integers it has, whether the last string looks like a *global* name,[8] whether it looks like a machine-generated name, and whether it looks like the name of a power supply. A label placed high in the hierarchy suppresses inheritance of non-global names from lower in the hierarchy. Pruning of less interesting names is also used alone as a canonicalization transformation that visits each entity and prunes.

The sections above present Lichen's repertoire of transformations, which fall into four categories: flat-insignificant hierarchy transformations, flat-significant but behaviorally insignificant transformations, transformations of Lichen's representation, and non-structural transformations. Following sections discuss the experience of using Lichen to reconcile the two views of the MIPS-X *pc*.

[8] This is a concept from Magic that indicates an expectation of ultimate connectivity.

5.4 The MIPS-X pc Key and Reconciliation

I was able to reconcile many of the differences between the two views of the *pc*. The reconciled differences are all of the flat-insignificant ones, plus a few others that do not constitute behavioral inconsistencies. The unreconciled, behaviorally insignificant, differences are handled by the base comparison, presented later.

The answers to the four questions posed in Section 5.1 are as follows.

- The repertoire of Lichen, presented in Section 5.3, is good.
- The reconciliation of the two views of the *pc* of MIPS-X consists of 61 invocations of transformations. These 61 invocations take 223 "words".[9]
- The invocations have very few details. The above figures lead to an average of under 4 words per invocation. Each invocation explicitly addresses only the major concerns.
- The transformations do not have to be very abstract. The modest transformations of Lichen's repertoire suffice.

5.4.1 Reconciled Differences

Table 5.1 shows the breakdown of the transformations in the reconciliation of the *pc*. For the layout, each of the three port and wire canonicalization transformations presented in Section 5.3.2 is applied once. Cleaning up the view is counted as a port and wire transformation because that is the main thrust of its effects, even though it also invokes some transformations from other categories. There were also two explicit importations and one explicit exportation. Deducing port and wire structure is not necessary for the funsim because only the layout originally lacks structured wires. Each of the four non-structural transformations of Section 5.3.5 is applied once[10] to the layout. Physical information need not be dropped from the funsim

[9] "Words" include names (of both transformations and view entities) and numbers, but not syntactic overhead like commas and brackets.

[10] The one application of renaming reads a file of name changes for the whole view. That file lists 241 changes (some of them using patterns, to, for example, shift indices down by one).

Funsim	Layout	Transformation
2	6	Port and Wire Transformations
3	36	Cell Structure Transformations
5	42	Total Flat-Insignificant Transformations
0	1	Physical Transformations
2	3	Naming Transformations
2	4	Total Non-Structural Transformations
0	8	Transformations of Lichen's Representation
7	54	Grand Total

Table 5.1: Transformations of the Reconciliation of the MIPS-X pc unit

Funsim	Layout	Type
0	4	Flatten out cell instance
0	8	Flatten out cell type
0	1	Flatten nested arrays
0	11	Un-flatten
3	6	Raise Grandchildren
0	1	Lower Children
0	5	Transpose
3	36	Total

Table 5.2: Cell Structure Transformations in the Reconciliation of the MIPS-X pc

because it does not have any to start with. Also, the funsim needs no renaming. The representation transformations of the layout consist of (a) switching seven cell types from the general representation to the array representation, and (b) one application of the canonicalization transformation that minimizes the period of every array. These representation transformations are part of the process of changing arrays of pair cells into simple arrays (see Figure 2.5 on page 14); the funsim's arrays are simple to start with, and so no representation transformations are needed. The remaining transformations, of cell structure, are broken down by type in Table 5.2.

The uses of the cell structure transformations are exemplified in the reconciliation of the pc incrementer (*pcinc*), which is one of the major parts of the *pc*. Figure 5.3 shows the funsim view and Figure 5.4 shows an introduction to the layout view of the *pcinc*. There are several differences.

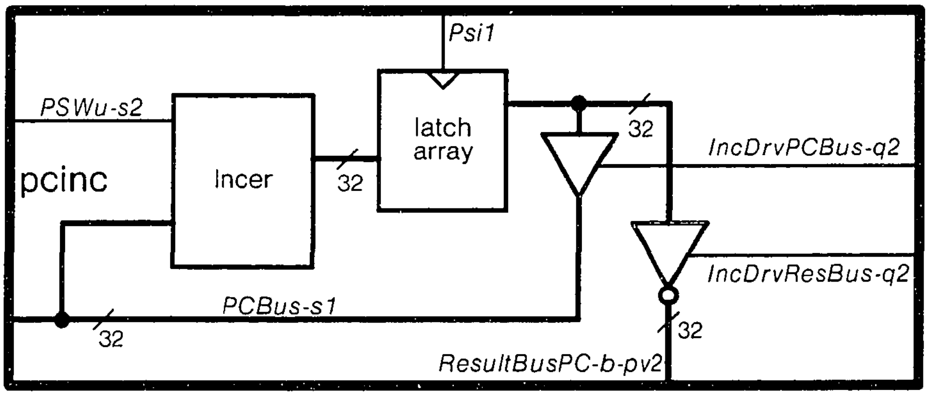


Figure 5.3: Funsim View of the pcinc

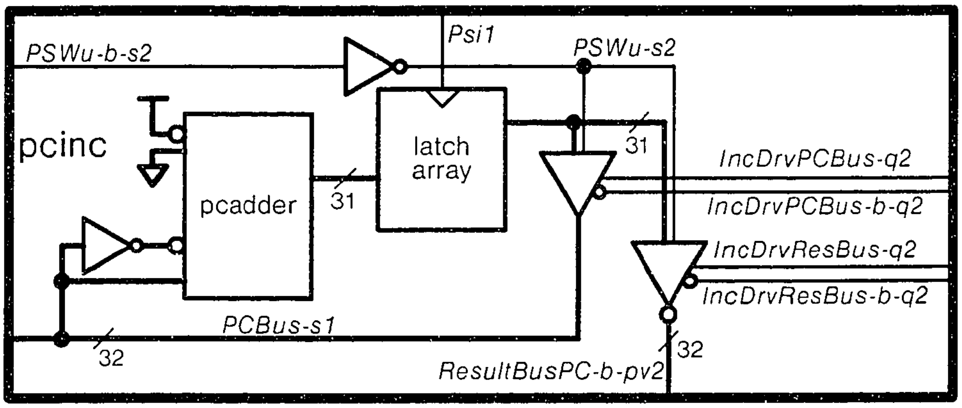


Figure 5.4: Introduction to Layout View of the pcinc

- In the funsim view, the *pcinc* is given the positive sense of *PSWu-s2*; in the layout view, *pcinc* gets the negative sense (*PSWu-b-s2*). There is thus an inverter in the layout view that does not correspond to anything in the funsim.

- Where the funsim has an incrementer (the *Incer*), the layout has a general adder with one of its inputs wired to 0 and the carry-in set. This general adder cell type is also instantiated elsewhere in the pc.

- The most sirnificant bit of the MIPS-X program counter is special: it indicates whether the machine is executing in user or supervisor mode. In other words, the 0^{th} bit of the program counter is one and the same as the *s/u* bit of the program status word (PSW). Whether this bit changes from one instruction to the next should not be determined by whether the lower 31 bits of the previous instruction's address are all 1, but rather by whether an appropriate instruction has been executed. Thus, in the funsim, the Modula-2 code for the *Incer* reads the 32-bit value of *PCBus-s1*, increments it, smashes *PSWu-s2* into the most significant bit, and then drives the new program counter value onto the output. The layout works differently: the adder only computes the 31 least significant bits of the sum, and the proper s/u bit is connected directly into the circuitry that accepts the new program counter.

- The adder in the layout takes both the positive and negative sense of its inputs; the incrementer only takes the positive sense. Also, the drivers in the layout take both the positive and negative sense of their enable inputs, whereas the drivers in the funsim take only the positive sense.

- In the funsim, *PSWu-s2* is latched on *Psi1* before going to the drivers; in the layout, it is fed directly to the drivers. MIPS-X uses two-phase non-overlapping clocks, named *Phi1* and *Phi2*, and *Psi1* is a qualified version of *Phi1*. The "-s2" timing signature means "stable during *Phi2*", and the "-q2" signature means "qualified by *Phi2*". Since the drivers only drive during *Phi2* and *PSWu-s2* is already stable then, the *Psi1*-clocked latch for *PSWu-s2* is unnecessary: its input is as good as its output needs to be. The other 31 latches

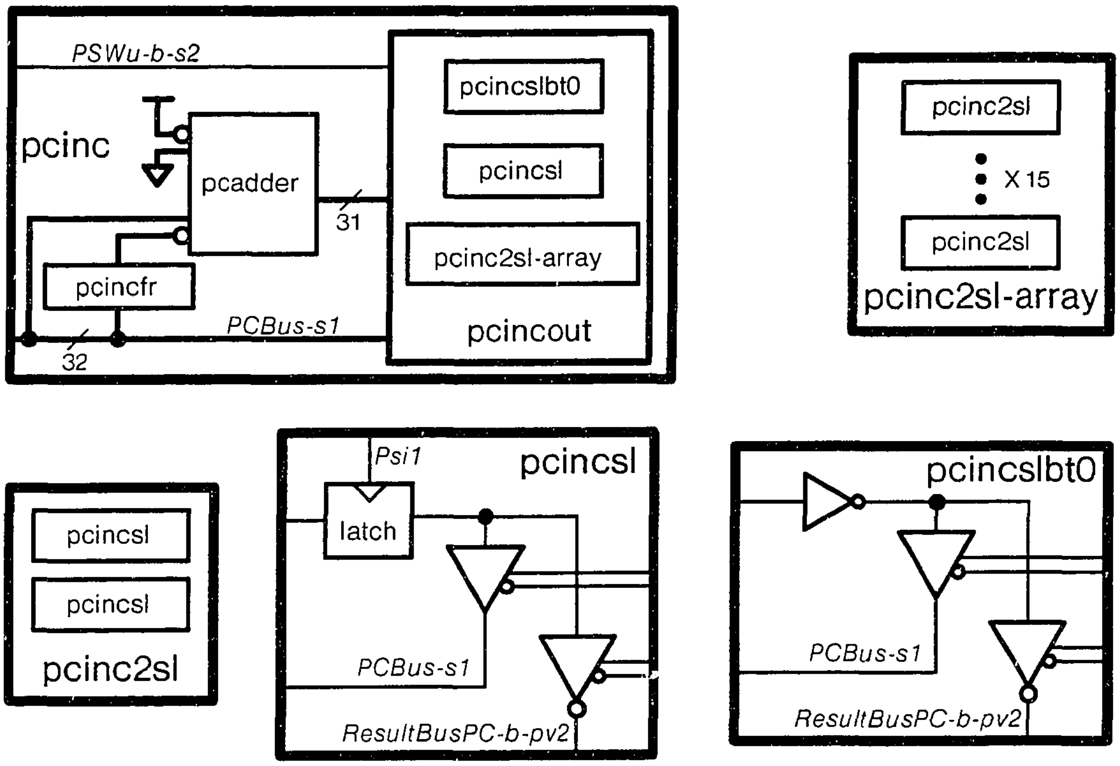


Figure 5.5: Layout View of the pcinc

are necessary, and are present in both views.

- Although not drawn in the figure, the power supply (*Vdd* and *Gnd*) wiring is explicit throughout the layout view. In the funsim view, *Vdd* and *Gnd* are used in a few places as constant inputs that specialize general cells; the power supply wiring is only as extensive as needed for that specialization. Another constant voltage supply, *vbias*, appears in the layout, for use in making resistors out of transistors. This signal does not appear in the funsim.

Figure 5.5 shows the actual layout view of the *pcinc*. It organizes the drivers, latches, and the inverter into 32 bit slices, each of which has a positive driver, an inverting driver, and either a latch or an inverter. The 0^{th} bit slice is thus irregular (it has an inverter instead of a latch). 30 of the other 31 bit slices are organized into bit slice pair cells, of type *pcinc2sl*, and arrayed in the *pcinc2sl-array* cell type. The *pcincfr* cell type arrays the inverters for the adder input.

The two views of the *pcinc* are reconciled by the following transformations.

- Un-flatten the *pcadder* and *pcincfr* instances of the layout into a new cell type corresponding to the *Incer* of the funsim. The "most natural" way to handle the wiring leaves the fixing of one of the adder's inputs to 0 inside the new cell type.
- In the funsim, raise grandchildren to bring the 2 most significant elements out of the latch array and the two driver arrays.
- Flatten out the *pcincout* cell type of the layout.
- Flatten out the *pcinc2sl* cell type of the layout; this makes the *pcinc2sl-array* into a simple array of 30 *pcincsl*.
- Transpose the top two levels of structure of the *pcinc2sl-array* cell type; this converts between the bit-slice and the function-slice organization.
- Flatten out the *pcincslbt0* cell type and the lone *pcincsl* cell instance in the layout *pcinc*; this breaks up the top two bit slices, leaving their subcells directly in the *pcinc* to correspond to the raised grandchildren in the funsim.

Figures 5.6 and 5.7 show the reconciled structures. The rest of the *pc* was no more difficult than the *pcinc*.

5.4.2 Reconciliation Performance

The asymptotic costs of a Lichen reconciliation are good. Each transformation takes an amount of time that is proportional to the number of entities that are affected. Because Lichen's representation of views is hierarchical, it is concise—and it minimizes the number of entities affected by each transformation.

The breadth of Lichen's repertoire of transformations enables Lichen to produce reconciled views that correspond tightly. Figure 5.8 compares the Lichen reconciliation

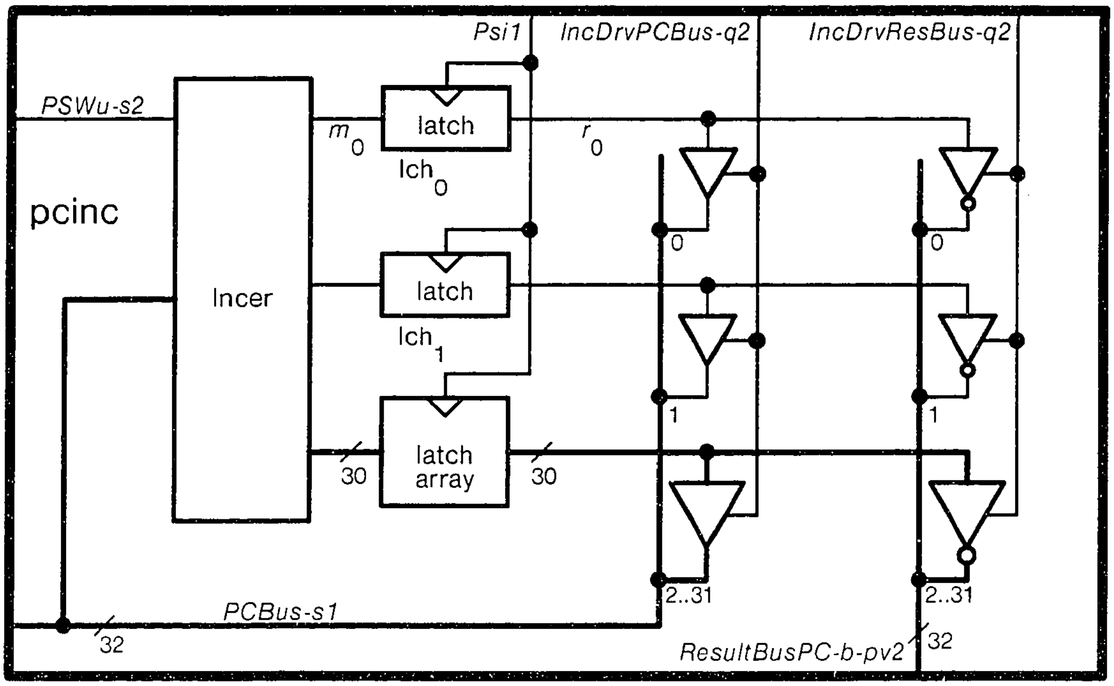


Figure 5.6: Reconciled Funsim pcinc

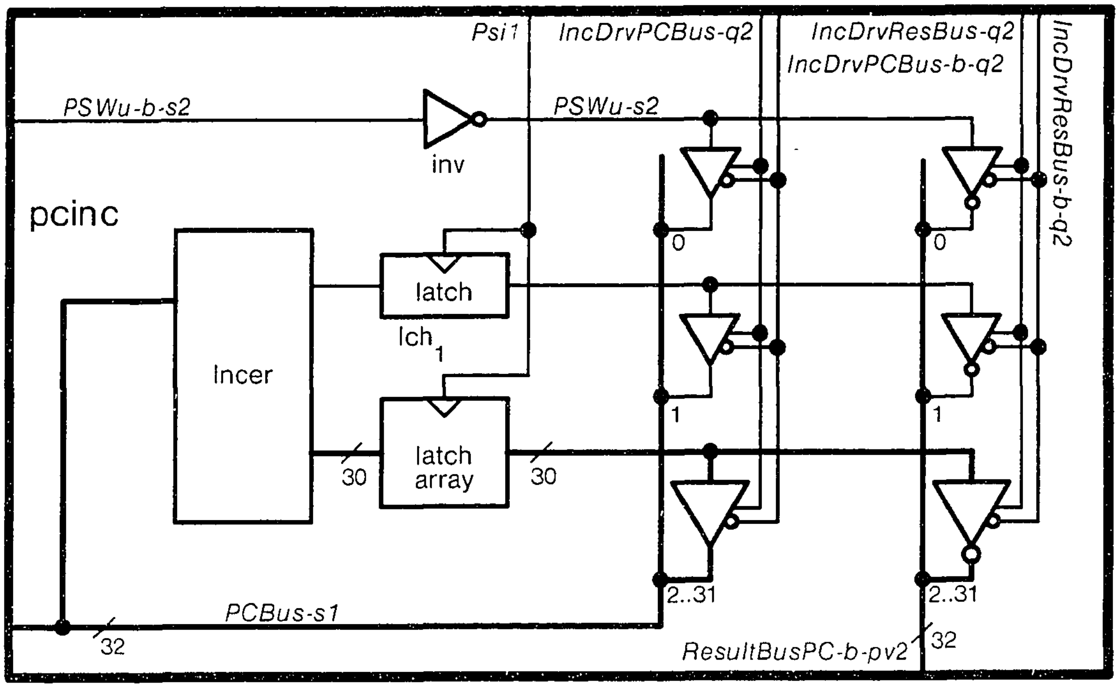


Figure 5.7: Reconciled Layout pcinc

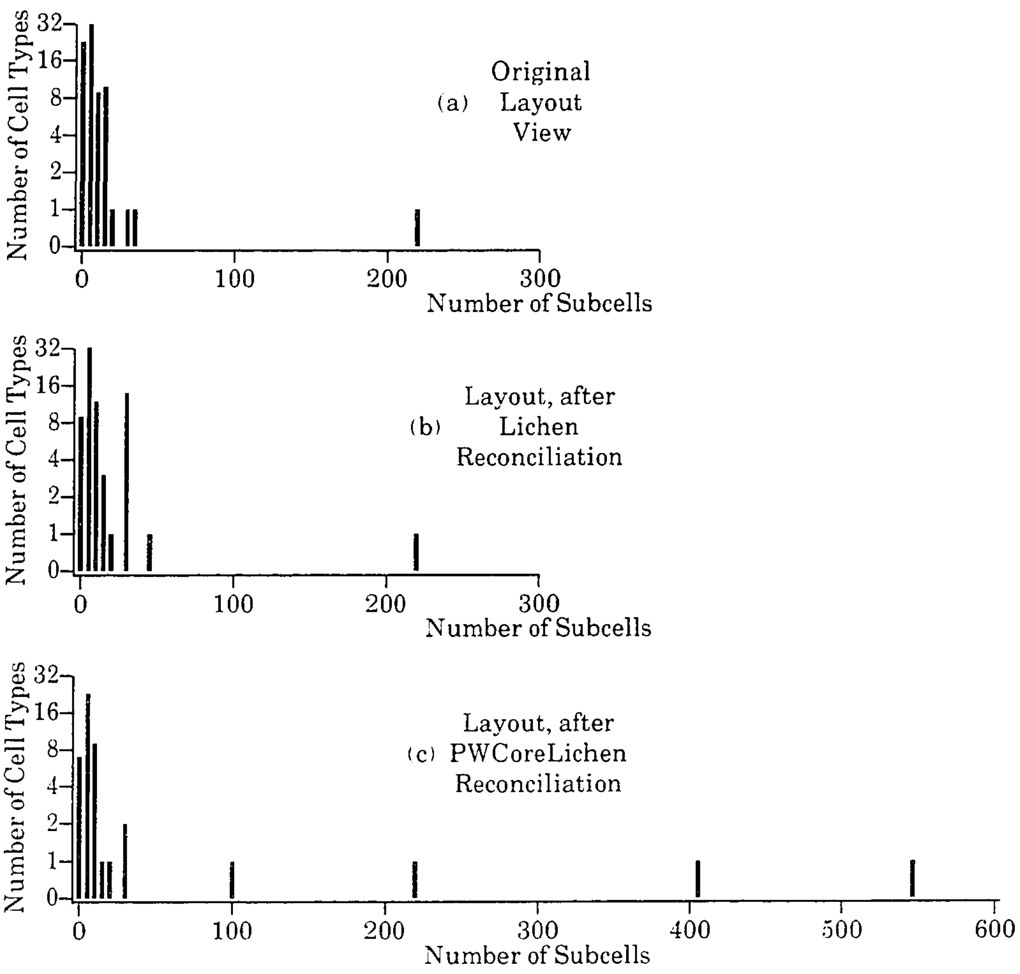


Figure 5.8: Tightness of Lichen and PWCoreLichen Reconciliations of the MIPS-X pc

of the *pc* with the reconciliation that PWCoreLichen would do. The one original cell type with a large number of subcells is a PLA. The reconciled layout produced by Lichen is not much flatter than the original. Because PWCoreLichen's only way of dealing with major differences in cell structure is to flatten out cell types, the reconciled layout produced by PWCoreLichen is significantly flatter than that produced by Lichen. In fact, Figure 5.8 understates the degree to which PWCoreLichen has to flatten. The *pc* cell type appears in part (c) of the figure with about 545 subcells. However, since the reconciliation moves some components between the *pc* and others of the eight major parts of MIPS-X, PWCoreLichen would have to flatten out the *pc* and other major parts, replacing the large *pc* and other major cells with one even larger cell for the whole MIPS-X.

5.5 The MIPS-X Base Comparison

I initially expected that after the reconciliation by Lichen, the two views of MIPS-X could be compared by a straightforward structural/semantic (see Section 2.2.2 on page 28) technique, to be called Plan A. This technique would compare frontier cells by simulation, and the higher cells structurally. However, the reconciled views turned out to have behaviorally insignificant differences that Plan A would flag as inconsistencies. Most of these differences are due to the difference in level of abstraction of the views. Remember, the comparison problem includes other difficulties besides differing hierarchies. To demonstrate that hierarchy reconciliation does not lead to a dead end, I devised a new base comparison method, called *Comparison Modulo Boring Components*, that can correctly complete the Informed Comparison of the two MIPS-X views.

5.5.1 Plan A

Figure 5.9 shows the steps of Plan A and where they were expected to fit into the whole Informed Comparison of the two MIPS-X views. Being a structural/semantic technique, Plan A is a combination of structural comparison and a more semantically

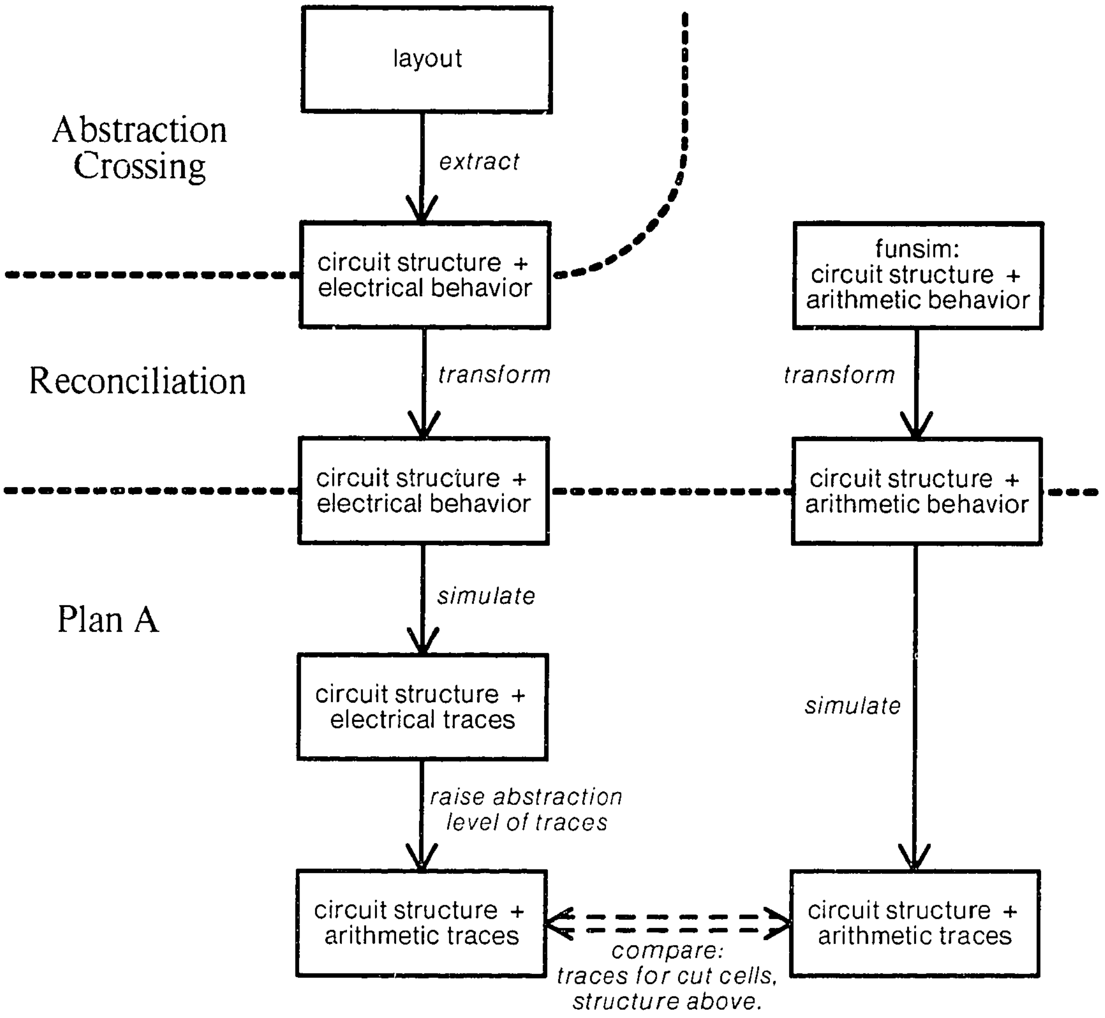

layout
Abstraction Crossing
extract
circuit structure + electrical behavior
funsim: circuit structure + arithmetic behavior
Reconciliation
transform
transform
circuit structure + electrical behavior
circuit structure + arithmetic behavior
Plan A
simulate
circuit structure + electrical traces
simulate
raise abstraction level of traces
circuit structure + arithmetic traces
circuit structure + arithmetic traces
compare: traces for cut cells, structure above.

Figure 5.9: An Informed Comparison for MIPS-X Using Plan A

powerful technique (in this case, simulation). Plan A requires me to choose a low frontier in each view, and calls the cell types of those frontiers the *cut* cell types (because a frontier is a cut through a hierarchy). Under Plan A, the corresponding cut cell types are compared by simulation, and the higher cell types are compared structurally. The structural comparison is efficient, and the simulation-based comparison is powerful.[11]

The behaviors of corresponding cut cells do not need to be (and sometimes are not) exactly equivalent. There are 'don't care' conditions: corresponding outputs need to be equivalent only at the times when they are read as input by other cells, and corresponding cells need to respond equivalently only to inputs that the rest of the circuit actually provides. A familiar example of a 'don't care' condition arises at the output of any cell feeding into a latch: that output must be consistent with the corresponding output in the other view only at the times when the latch is enabled. A familiar example of a 'don't care' condition on the inputs of a cell arises at a multiplexor's unary-encoded select inputs: the rest of the circuit guarantees that the select inputs are mutually exclusive, and so the multiplexor needs to be equivalent to the corresponding multiplexor in the other view only under the condition that the select inputs are mutually exclusive.[12]

To compare the whole circuit when there are 'don't care' conditions on the cut cells requires three things:

A picking the 'don't care' conditions to be used in the comparison,

B checking that the cut cells are equivalent modulo the 'don't care' conditions, and

C checking that the higher structure respects the 'don't care' conditions.

[11] The simulation-based comparison is also unsound, because some of the cut cells are too complex for exhaustive simulation. However, Plan A still beats the comparison done at Stanford: that comparison also used simulation, but it had to be slower than Plan A's simulation (for reasons explained near the end of this section), and thus covered fewer possibilities than Plan A can. Another reason that Plan A uses simulation is that the Modula-2 code of the funsim would be difficult to use for anything else.

[12] Actually, it is more realistic to suppose that the rest of the circuit guarantees the mutual exclusion of the select inputs only at certain times; but then the rest of the circuit can only depend on the multiplexor's choice at those times, and so it is still the case that the corresponding multiplexors need to behave equivalently only when their select inputs are mutually exclusive.

Stricter 'don't care' conditions than those actually imposed by the circuit can be used for comparison—if the cells' behaviors really are consistent to that greater degree or if false claims of inconsistency can be tolerated. For an example that illustrates the necessity of these three steps, consider using a precharged inverter in an electrical view where a static inverter appears in a Boolean view. The precharged inverter charges its output high during *Phi1*, and may discharge its output during *Phi2*; the static inverter continuously assigned the inverse of its input to its output. A reasonable choice for the 'don't care' conditions on the inputs and outputs follows.

- The outputs are significant only at the end of *Phi2*.
- The input must be low at the end of *Phi1*.
- If the input changes, it must be high at the end of *Phi2*.

Establishing the consistency of the electrical and Boolean views requires showing not only that the corresponding inverters are equivalent given proper behavior of their inputs, but also that the outputs of the inverters are used only at the end of *Phi2* and that the inputs provided by the rest of the circuit actually meet their conditions.

The output conditions in MIPS-X are concerned only with timing, and thus present few problems. Dave Noice's timing discipline [Noice83] was followed in the MIPS-X design. In this discipline, every signal is assigned one of a few *clocking types*, which give information about when the signal changes, when it must have its logic value, and whether it is inverted. An efficient analysis of the electrical circuit extracted from the layout suffices to verify that the layout respects this labelling. Thus, most of the work for output 'don't care' conditions has been done: the clocking types indicate what the conditions are, and the static analysis has verified that the higher structure respects these conditions. The remaining task—verifying that the cut cells are equivalent modulo 'don't care' conditions—can simply be done by simulating and then comparing responses only at the relevant times, as indicated by the clocking types.

The input conditions of the MIPS-X cut cells are unrestricted, and thus more problematic. Plan A takes the stimuli for the cut cells from a simulation of the whole funsim. This accomplishes A, B, and C in one stroke, because the cut cells are compared using inputs derived from the operation of the rest of the circuit. However, simulation of the whole funsim is not a hierarchical process: the simulator executes a flat circuit. Does this mean Informed Comparison cannot work hierarchically? There are two answers.

- Informed Comparison *can* work hierarchically. The problem in this case is that the designers did not prepare for a hierarchical comparison. Any hierarchical technique more powerful than structural comparison would be frustrated by the 'don't care' conditions on the inputs of the MIPS-X cells; the problem is not due to the unique features of Informed Comparison. The kind of solution used for the output conditions also would work for the input conditions: use a carefully designed language of constraints to annotate the view so that the satisfaction of the constraints can be efficiently checked with a static analysis. I chose the simpler solution because I wanted to avoid editing the MIPS-X design data or learning more than necessary about how and why MIPS-X works.

- Even an Informed Comparison that is not completely hierarchical can be faster than other methods. The comparison done at Stanford was slower (see Section 5.5.4 for an estimate of how much), because it simulated the whole of *both* views. The electrical circuit extracted from the layout was simulated using RSIM [Terman83], which works at a lower level of abstraction (than the funsim) and is thus significantly slower. Also, the electrical circuit is significantly more detailed than the funsim. Thus, while improving the asymptotics is good, improving the constants is also good.

In summary, then, Plan A requires choosing, in each view, a low frontier of cell types, to be called the cut cell types. Above that frontier. the two reconciled views are compared by structural comparison. The cut cells themselves are compared by simulation. The 'don't care' conditions are factored out by comparing responses only at times indicated by the clocking types of the signals and by taking the stimuli (as well as the funsim cut cells' responses) from a simulation of the whole reconciled funsim.

5.5.2 Problematic Differences

Plan A is not sufficiently powerful to complete the Informed Comparison of the two views of the MIPS-X. Those views have certain kinds of differences that cannot be removed by Lichen because they are flat-significant and not of the trivial nature reconciled by Lichen's non-structural transformations. These differences appear to be inconsistencies to Plan A, because they involve structural differences at and above the frontiers. The kinds of differences are as follows.

- Where the funsim has one sense of a signal, the layout has the opposite sense, or both senses. For example, the drivers in the *pcinc* of the funsim (see Figure 5.6) take only the positive sense of their enabling signal; in the layout (see Figure 5.7), the drivers take both senses.
- The layout has extra wiring, for *Vdd*, *Gnd*, and *vbias*.
- The layout has inverters and buffers that do not appear in the funsim. An example is *inv* in the *pcinc* in Figure 5.7.
- The funsim has a latch (lch_0 in Figure 5.6) that is useless and does not appear in the layout.
- Some or all of the function of some funsim cells is accomplished simply with wiring in the layout. For example, the funsim *Incer*'s use of *PSWu-s2* in Figure 5.6 is accomplished by wiring in Figure 5.7. Another example is a funsim cell type named *TrapMask*, whose function is entirely implemented with wiring in the layout.

All but the second-to-last difference are due to the difference in level of abstraction of the views. The next section presents a new comparison method that is not confused by any of the above kinds of differences.

5.5.3 Comparison Modulo Boring Components

Comparison Modulo Boring Components follows the same schema as Informed Comparison: apply some transformations to remove problematic differences, and then apply a base method. In this case, the transforming step is called the *boring reconciliation* and the base method is called Plan A^+ (because it is a slightly enhanced version of Plan A). The boring reconciliation is done between the simulation and comparison steps of Plan A^+. Figure 5.10 shows the steps of Comparison Modulo Boring Components and where they fit into the whole comparison of the two MIPS-X views. The boring reconciliation and the enhancements of Plan A^+ add the power required to handle the differences of the previous section.

Sense-Abstract Views

Comparison Modulo Boring Components uses *sense-abstract* views, whose purpose is to make differences in the senses of wires ignorable. In a sense-abstract view every wire and port carries both the positive and negative sense of its signal, and any connection may connect those two senses either reversely or normally. Thus an inverter does nothing that any other cell cannot do, and two wires carrying opposite senses of a signal do nothing that a single wire cannot do.

Connections are more complex in sense-abstract views than in ordinary views. Every connection is between a wire and a port at a site. A sense-abstract connection additionally has a *polarity*, which is either *normal* or *reverse*. A normal connection connects the positive sense of the wire with the positive sense of the port, and the negative with the negative; a reverse connection connects the positive with the negative and the negative with the positive. For example, Figure 5.11 shows a sense-abstract view of an implementation of an XOR gate (the "X"s indicate reverse connections); Table 5.3 lists the connections. Any ordinary view (at a high enough level of abstraction) can be considered to be a sense-abstract view in which every connection's polarity is normal.

The Transformations of Boring Reconciliation

Boring components are those that simply copy inputs to outputs, perhaps with inversion and an insignificant change of timing. A component that copies some of its inputs to some of its outputs but also does more interesting computation is called *fractionally boring*. The boring reconciliation deletes boring components and chops

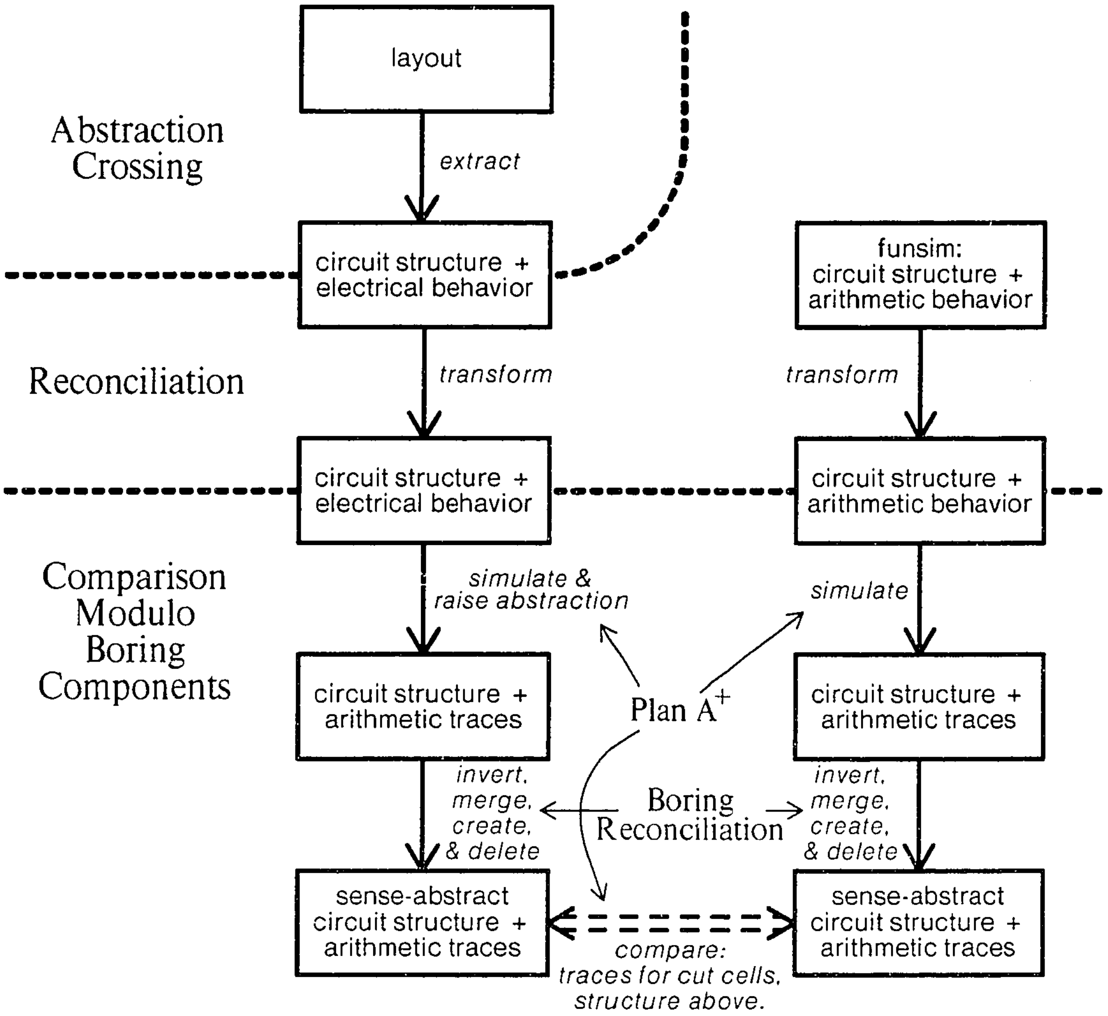

Abstraction Crossing
layout
extract
circuit structure + electrical behavior
funsim: circuit structure + arithmetic behavior
Reconciliation
transform
transform
circuit structure + electrical behavior
circuit structure + arithmetic behavior
Comparison Modulo Boring Components
simulate & raise abstraction
simulate
circuit structure + arithmetic traces
Plan A+
circuit structure + arithmetic traces
invert, merge, create, & delete
Boring Reconciliation
invert, merge, create, & delete
sense-abstract circuit structure + arithmetic traces
compare: traces for cut cells, structure above.
sense-abstract circuit structure + arithmetic traces

Figure 5.10: An Informed Comparison for MIPS-X Using Comparison Modulo Boring Components

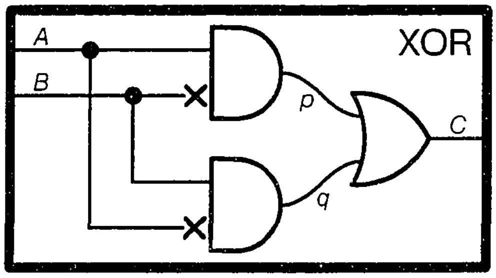


Figure 5.11: A Sense-Abstract View of an XOR Implementation

Wire	Port	Site	Polarity
A	A	XOR	normal
A	In_1	And_1	normal
A	In_2	And_2	reverse
B	B	XOR	normal
B	In_2	And_1	reverse
B	In_1	And_2	normal
p	Out	And_1	normal
p	In_1	Or	normal
q	Out	And_2	normal
q	In_2	Or	normal
C	Out	Or	normal
C	C	XOR	normal

Table 5.3: Connections in a Sense-Abstract View of an XOR Implementation

out the boring part of fractionally boring components. The boring reconciliation also merges the wires between which boring components copy values. In order to be able to merge some such wires, the boring reconciliation also swaps the senses carried by some wires. Finally, the boring reconciliation creates and deletes power supply wiring.

The boring reconciliation cannot work directly on the reconciled layout. While the layout's boring components do not constitute inconsistency with the funsim, they are very important to the way the layout works. The output of a buffer changes faster than it would if the buffer were deleted and the input and output were merged. The correct operation of the MIPS-X layout cannot be observed without a certain amount of quantitative timing analysis (this is why the designers at Stanford simulated it with RSIM instead of a pure switch-level simulator). Thus, the boring reconciliation would significantly change the functioning of the layout view.

The problem is that boring reconciliation can only be applied to a view at a level of abstraction in which the changes made by the boring reconciliation are truly insignificant—and it should be easy for a program to verify that insignificance. This means that I could not even apply the boring reconciliation to the reconciled funsim view, because I had no tool that could verify that the Modula-2 code for a buffer simply copies input to output. Comparison Modulo Boring Components solves this problem by applying the boring reconciliation after the simulation (and the raising of the abstraction level of the layout traces from electrical to arithmetic). Once arithmetic simulation traces are the behavioral description of the cut cells, the boring reconciliation's changes can easily be verified to be insignificant. This also makes it easier to chop out the boring fraction of a component: rather than editing a block of Modula-2 code, the boring reconciliation simply drops some simulation traces.

The first transformations applied in the boring reconciliation swap the senses carried by ports and by wires; these transformations are called *inverting a port* and *inverting a wire*. Inverting a wire involves switching the polarity of each of that wire's connections; similarly, inverting a port switches the polarity of each of that port's connections. For example, Figure 5.12 shows the result of inverting the wires and ports named *PSWu-b-s2*, *IncDrvPCBus-b-q2*, and *IncDrvResBus-b-q2* of Figure 5.7.

After the necessary wire and port inversions, the next transformations merge wires

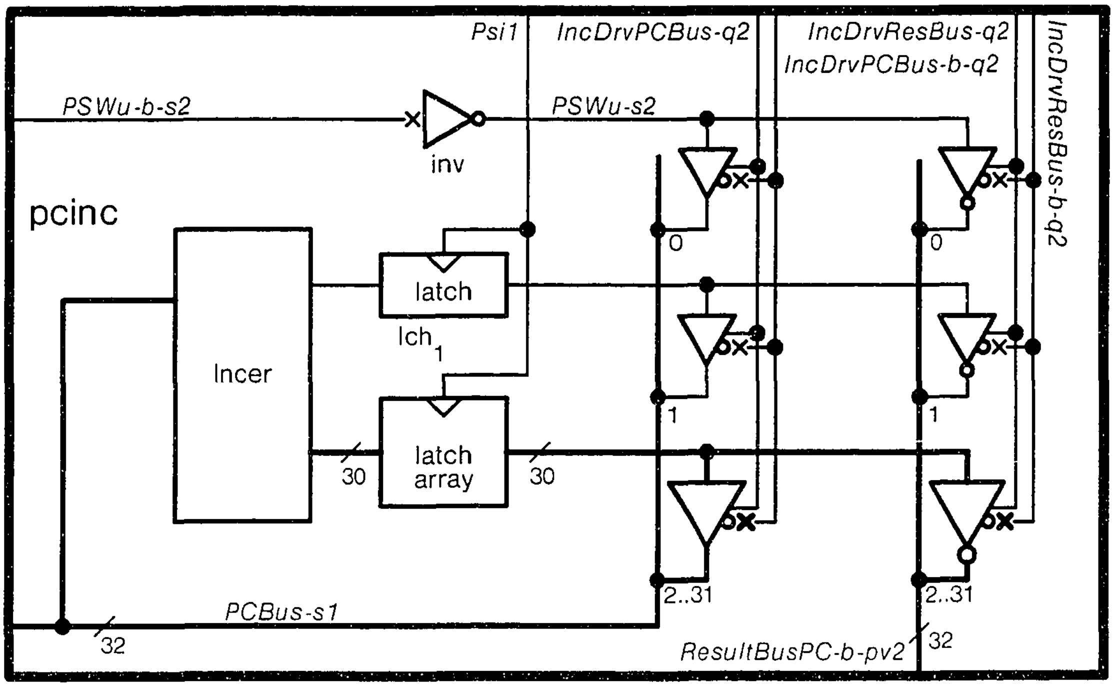


Figure 5.12: Reconciled Layout pcinc After Inverting Certain Wires and Ports

and merge ports. These transformations do not clearly preserve behavior, and so must be checked. This is done by comparing simulation traces. When two wires are merged, their simulation traces are checked for consistency. Specifically, the traces for the two wires are compared for equality at the times indicated by their clocking types (which must not be incompatible). The trace for the merged wire is constructed from the traces of the wires being merged.

Ports are merged, as well as wires. This is necessary because the ports of the frontier cells and their ancestors are significant to the structural comparison of Plan A^+. The question of whether a port merge preserves funsim-level behavior is converted into a question of whether wire merges preserve funsim-level behavior. Before merging the ports, the wires connected to the ports at each site are merged—and the validity of those merges is checked in the usual way. Once this is done, the ports are unnecessarily distinguished and merging them is flat-insignificant (see Section 5.3.2); no further checking is required. Figure 5.13 shows the result of merging the following three pairs of wires and three pairs of ports of the *pcinc* of Figure 5.12: wires *PSWu-s2* and *PSWu-b-s2*, wires *IncDrvPCBus-q2* and *IncDrvPCBus-b-q2*, wires *IncDrvResBus-q2* and *IncDrvResBus-b-q2*, ports *PSWu-s2* and *PSWu-b-s2*, ports *IncDrvPCBus-q2* and *IncDrvPCBus-b-q2*, and ports *IncDrvResBus-q2* and *IncDrvResBus-b-q2*. Although Figure 5.13 does not show it, the boring reconciliation also inverts the negative enable ports of the drivers and then merges them with the positive ones.

Once the wires and ports have been inverted and merged as necessary, the boring components and fractions can finally be deleted. Figure 5.14 shows the layout *pcinc* after the deletion of boring components and fractions. The *inv* has been deleted. The funsim *pcinc* has the same structure; its reconciliation involved: (1) merging the three wires *PSWu-s2*, m_0, and r_0; (2) deleting the boring component lch_0; and (3) deleting the *Incer*'s boring fraction, the ports formerly connected to *PSWu-s2* and m_0.

Figure 5.14 does not show all of the ports and wires: the power supply and *vbias* wiring is omitted. The boring reconciliation involves two transformations that reconcile the power supply and *vbias* usage of the two views. These two transformations need not be done at any particular time relative to the other three transformations

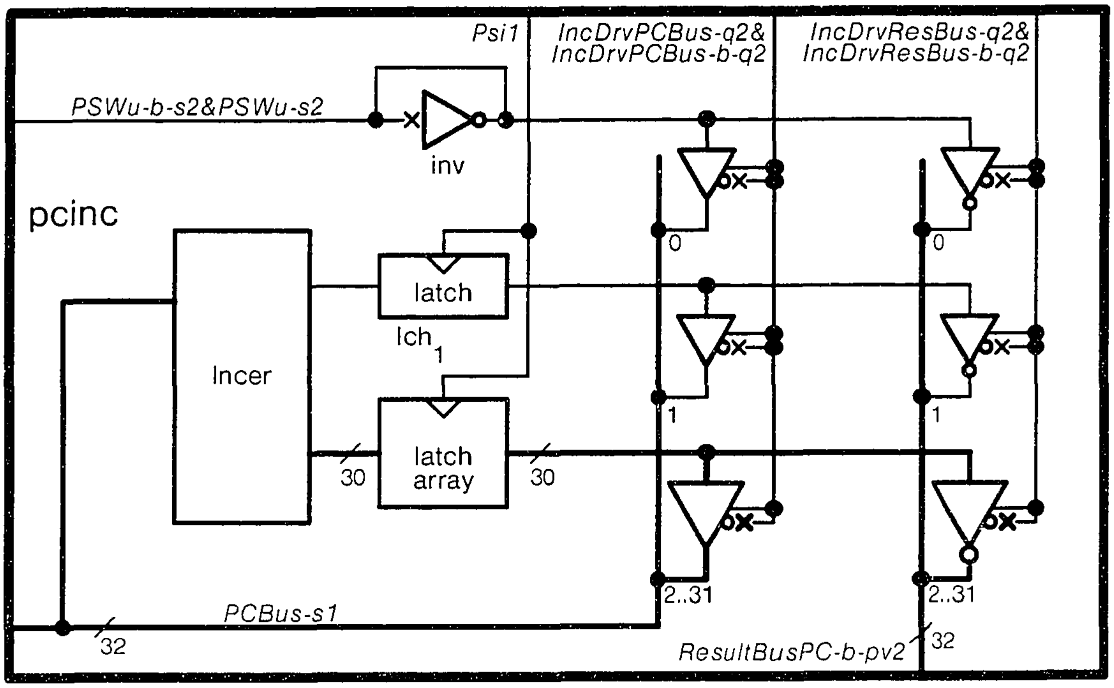


Figure 5.13: Reconciled Layout pcinc After some Inversions and Merges

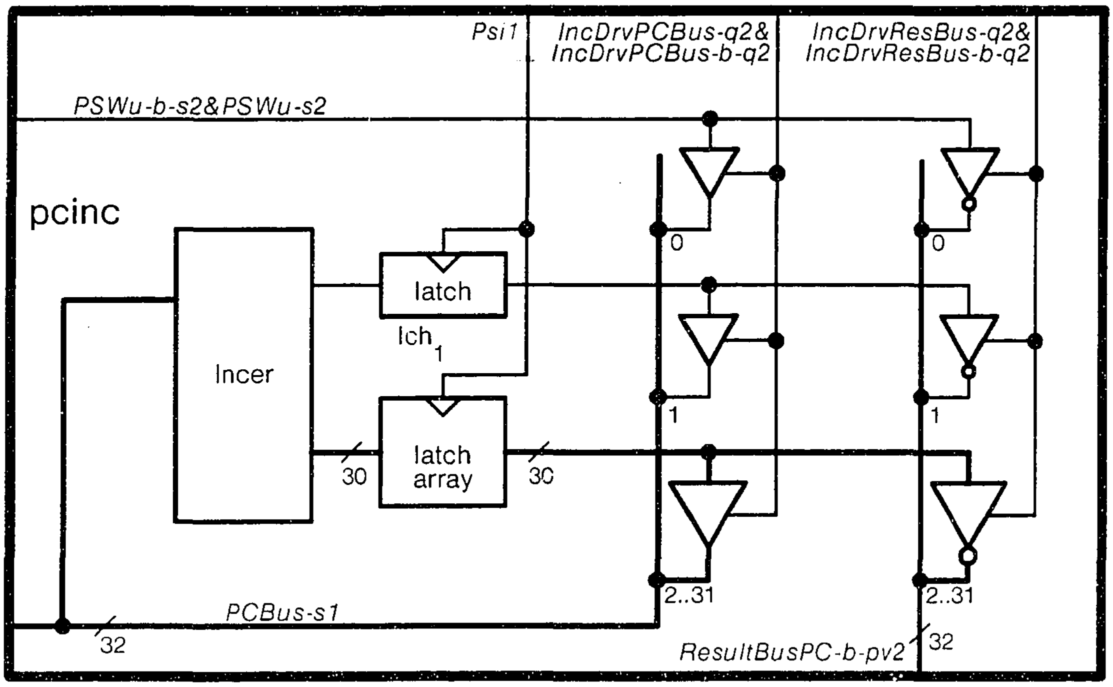


Figure 5.14: Layout pcinc After Deletion of Boring Components and Fractions

of the boring reconciliation. One of these transformations deletes a wire and every port and wire hierarchically connected to it above a frontier. This transformation is invoked with an RSIM voltage range, and its soundness is checked by testing whether all those ports and wires stayed within that range during the whole simulation. This transformation is used to remove the *vbias* wiring from the electrical view above the cut cells.

The other transformation adds *Vdd* and *Gnd* ports to every cell type of the funsim that doesn't already have them, and also adds wires and connections as necessary to globally distribute the power supplies. This transformation also adds the appropriate constant simulation traces for the added ports and wires. This transformation takes a list of exception cell types, which need not get *Vdd* and *Gnd* ports. There is one cell type in the funsim that does not logically need the power supplies (it corresponds to a single transistor). It is easier to add *Vdd* and *Gnd* to the funsim than to remove them from the electrical view because removal would involve even more exceptions, in order to leave *Vdd* and *Gnd* as constant inputs to various cells.

The five transformations of the boring reconciliation remove the problematic differences listed in Section 5.5.2. Comparison Modulo Boring Components then is completed by Plan A^+.

Plan A^+

Plan A^+ differs from Plan A in three minor ways.

1. The comparison of corresponding simulation traces takes into account the sense of the correspondence (see Section 5.6.2).

2. The structural comparison is enhanced to take the polarities of the connections into account.

3. The structural comparison is further enhanced by some knowledge of De Morgan's laws.

The first two are required for the sake of soundness. The third is easy and supplies some of the power needed by some of the differences in the MIPS-X views.

The first enhancement requires no further description. The second and third are both accomplished by small changes to the way structure is compared. That comparison is done by converting the circuits into labeled graphs and then testing labeled graph isomorphism. The enhancements of Plan $A^+$are accomplished by changing the way labels are assigned. Each wire and each cell instance becomes a vertex, and each connection becomes an edge. An edge's label is a function of the port and polarity of its connection. In particular, a reverse connection gets the *reversal*[13] of the label that connection would get if it were normal; the label for a normal connection is purely a function of the connection's port. The label of a cell instance's vertex is purely a function of the instance's cell type. Cell types whose behaviors differ only by inversion of the values that pass through some ports are assigned the same label. Corresponding ports of those cell types are assigned identical or reversed labels, as appropriate. The reversal for labels of reverse connections meshes with the reversal of the labels for inversely corresponding ports to encode knowledge of De Morgan's laws. For example, a normal connection to the output of a *NAND* gate would get the same label as a reverse connection to the output of an *AND* gate. Thus, after the boring reconciliation inverts the appropriate ports and wires, this choice of labels produces isomorphic graphs from views that originally differed in ways described by De Morgan's laws. The test of graph isomorphism then completes the Comparison Modulo Boring Components, which completes the Informed Comparison of the two MIPS-X views.

5.5.4 Base Comparison Performance

This section compares the estimated speed of the base comparison presented above with the speed of the full-circuit simulation done by the designers at Stanford. Comparison Modulo Boring Components can be broken down into three activities:

- simulation of the cut cell types,
- hierarchical structural comparison above the simulation frontiers, and

[13] The reversal can be any function R of labels such that $R(R(l)) = l$ for every l; identities of R, if there are any, cannot be used.

- boring reconciliation.

The second two require little time. As demonstrated in PWCoreLichen and other work, hierarchical structural comparison is faster than any significant amount of simulation. Each reconciled hierarchical view of the SIC (see Section 4.2.3 on page 59) has a total of about 1500 cell instances and 2600 atomic wires, for a total of about 4100 vertices in all the view's labelled graphs. PWCoreLichen on the 4-MIPS Dorado performs the whole hierarchical structural comparison of the two SIC views in 61 seconds. The reconciled hierarchical *pc* views each have about 400 cell instances and 1800 atomic wires above their simulation frontiers, and thus the hierarchical structural comparison of those clipped views should take about half a Dorado minute.[14] RSIM on the 1-MIPS DEC MicroVax-II simulates the approximately 50,000 transistors of MIPS-X minus its instruction cache memory at the rate of one clock cycle per minute, and thus that half a Dorado minute should be enough for simulating about 14 cycles of the 7100-transistor flattened reconciled *pc* layout. The designers at Stanford simulated roughly 30,000 cycles of the MIPS-X in their comparison, which included finding and fixing some bugs and then simulating again to verify the fixes.

The quantity of work to be done in the boring reconciliation is actually very small. Only five boring components in the *pc* layout and three in the funsim need to be identified and deleted.[15] The comparison of simulation traces for merged wires can be done during the simulation, costing little time. An experimental procedure, not tuned for performance, is able to delete the *vbias* wiring from the 39 cell types of the *pc* layout at and above the simulation frontier in 15 seconds on the Dorado; adding the *Vdd* and *Gnd* wiring to the 33 cell types of the funsim *pc* should take a similar amount of time.

Simulation speed is of primary importance. The functional simulation runs about 100 times faster than the RSIM simulation, and so can be ignored. Table 5.4 shows the number of transistors and atomic wires resulting from completely flattening each of the cut cell types in the reconciled layout view of the *pc*. Completely flattening that

[14] Structural comparison is little worse than linear in the graph sizes.

[15] Although other components are also boring, only those eight need to be deleted in order to enable the hierarchical structural comparison to succeed.

Transistors	Atomic Wires	Layout Cut Cell Type	Transistors	Atomic Wires	Layout Cut Cell Type
1005	562	DispAdder	6	8	pccmlatch
941	513	Incer	6	8	pclatch
656	319	PCRandomLogic	5	6	pcdislatch
320	132	pcdriver-array	4	6	pcresdrv
10	8	pcincdrvr	4	6	pcnor
7	10	2InputLatch	4	6	pcnand
7	9	pc2inultch	4	4	pcinvsigdr
7	9	pc2inpltch	4	4	pcclamp
7	9	pc2indltch	2	4	pcnot
7	9	pc2ingltch	1	3	nfet[48,32,24,1,1]
6	8	pcinclatch	3013	1643	Total

Table 5.4: Cut Cell Type Sizes

view of the *pc* yields 7123 transistors and 3039 atomic wires. Thus, the completely flattened cut cell types add up to about half the circuitry of the whole completely flattened *pc*. This is not a great improvement. Whether Informed Comparison can, in general, greatly improve simulation-based comparisons depends on certain structural details of the views. In the MIPS-X *pc*, over 90% of the transistors and wires in the flattened cut cells come from just four of those cut cells. Each of those four large cut cell types corresponds to a funsim cell type that either is atomic or has no duplication in its internal structure. Thus lowering the simulation frontier, where possible, would not decrease the total number of transistors and wires in the flattened cut cell types. Simply put, the improvement is small because there is not much regularity in the funsim when the funsim elements are weighted according the amount of layout used to implement them.

However, this Informed Comparison decreases simulation time by more than the ratio of the total quantities of circuitry: the time required for comparison by simulation grows faster than linearly in the circuit size, and the cut cell types are compared individually, rather than en masse. In other words, each pair of corresponding cut cell types can be compared by simulating fewer cycles than required for comparing the whole *pc* or MIPS-X at once. How many fewer cycles give an equivalent level of confidence?[16] This depends on how well the simulations of the whole *pc* and MIPS-X exercised the individual cut cells—and I have no data on this. However, the Informed Comparison has matched up many of the latches;[17] factoring out the dependence on their state could greatly reduce the number of cycles required.

By virtue of its reconciliation of the hierarchy differences, this Informed Comparison of these two views is as good as a hierarchical comparison based on simulation can be; to get better, either a different base comparison technique must be used or the funsim must be edited to move some of the remaining regularity out of the Modula-2 code and into the circuit structure.

5.6 The MIPS-X Correspondences

The previous sections present the reconciliation and base comparison of the MIPS-X views, with an emphasis on testing consistency. This section discusses the correspondences between the entities of the MIPS-X views. These correspondences are similar to those of PWCoreLichen, and are shown in Figure 5.15. The following sections focus on the *main* correspondence, between the original funsim and the original electrical view. This correspondences is composed of three subsidiary correspondences: (1) the *funsim* correspondence, between the original funsim and the reconciled funsim; (2) the *electrical* correspondence, between the original electrical view and the reconciled electrical view; and (3) the *base* correspondence, between the reconciled funsim and the reconciled electrical view. Lichen's reconciliation correspondences (the funsim and the electrical) are similar to those of PWCoreLichen. However, the MIPS-X base correspondence is more complicated than PWCoreLichen's, due to the complexity of Comparison Modulo Boring Components. The difficulties of composing the base correspondance with the reconciliation correspondences make the main correspondence even more complex.

[16] It is a question of confidence because practical comparisons of large circuits by simulation are unsound, as discussed in Section 2.2.1 on page 20.

[17] Of the four big cut cell types, only one (*PCRandomLogic*) keeps state; the other three, which account for approximately 3/4 of the total flattened cut cell type circuitry, are combinational (although the general adder used in the layout *DispAdder* and *Incer* uses precharged logic).

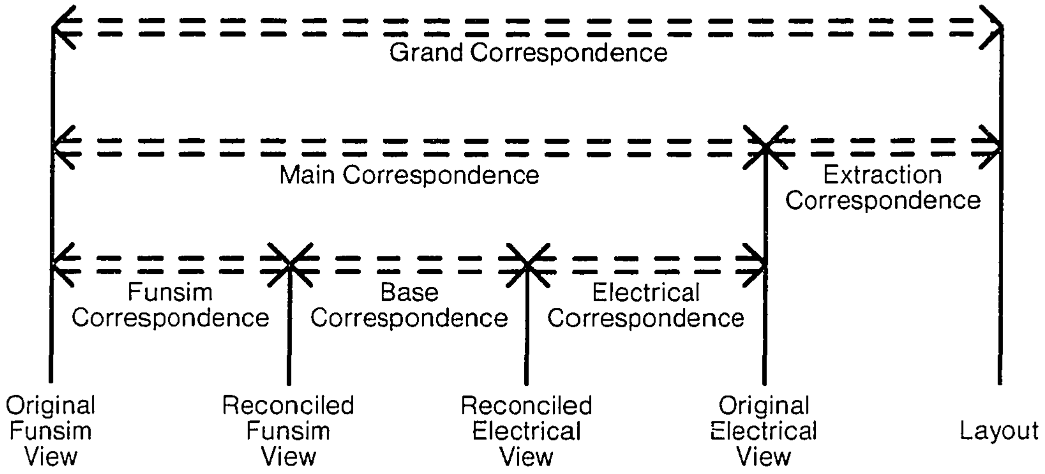


Figure 5.15: Correspondences in the Informed Comparison of the MIPS-X Funsim and Layout

5.6.1 The Reconciliation Correspondences

A Lichen reconciliation correspondence can be represented with a binary relation $\sim$, in a way very similar to the PWCoreLichen reconciliation correspondences. In Lichen, $\sim$ has the following features.

- $\sim$ is the disjoint union of $\overset{t}{\sim}$, $\overset{i}{\sim}$, $\overset{w}{\sim}$, and $\overset{p}{\sim}$.
- $\overset{t}{\sim}$ is a partial one-to-one relation between the funsim cell types and the electrical ones. The tagged cell types include at least the root and atomic cell types, and maybe others.
- $\overset{i}{\sim}$ is a total one-to-one relation between the funsim introductory instance paths and the electrical ones.
- $\overset{w}{\sim}$ is a partial relation between the funsim introductory atomic wire paths and the electrical ones.
- $\overset{p}{\sim}$ is a partial relation between the funsim atomic ports and the electrical ones.

Paths are needed because Lichen can flatten and un-flatten. Because Lichen does not merge parallel transistors, $\overset{i}{\sim}$ is one-to-one. Because of the possible unification of ports in either view, $\overset{w}{\sim}$ cannot be restricted to being one-to-one or even many-to-one. Also, $\overset{w}{\sim}$ is partial because of the possible exporting and retracting of wires of tagged cell types, as well as because of the possible creation and deletion of wires with no connections. Finally, $\overset{p}{\sim}$ is added because these views use explicit ports; $\overset{p}{\sim}$ shares many features with $\overset{w}{\sim}$.

Although this formulation of correspondences has no features directly related to some of the cell structure transformations (split/merge, raise/lower, transpose), it can represent the correspondence across any of Lichen's transformations. This is because those "unrepresented" cell structure transformations can be effected with flattening and un-flattening.

5.6.2 The Base Correspondence

The base correspondence relates views that differ in different ways than the views related by the reconciliation correspondences. The complexities of the base correspondence are introduced in turn, starting with the correspondence established by the Plan A$^+$ comparison. That comparison is done with a combination of

- structural comparison of a clipped version of the funsim with a clipped version of the electrical view, and
- comparison by simulation of the cut cells that specify the clippings.

The structural comparison establishes a total, nearly one-to-one, correspondence between all four kinds of structural entities of the two clipped views. This correspondence might not be one-to-one because multiple layout cell types can correspond to a single funsim cell type. In the MIPS-X, for example, there are multiple layout cell types for single simple funsim cell types like drivers and latches. For simplicity's sake, $\overset{t}{\sim}$ may relate multiple electrical cell types only to atomic cell types of the clipped funsim. Because of this restriction, no further complexities are introduced by those one-to-many correspondences. The comparison by simulation adds nothing to the correspondence, because it treats each atomic cell type of the clipped views as a black box.

The power supply and *vbias* transformations require that $\overset{w}{\sim}$ and $\overset{p}{\sim}$ be allowed to be partial. The deletion of boring fractions also requires that $\overset{p}{\sim}$ be allowed to be partial. The deletion of entirely boring components requires that $\overset{i}{\sim}$ and $\overset{t}{\sim}$ be allowed to be partial. The merging of ports and wires precludes $\overset{w}{\sim}$ and $\overset{p}{\sim}$ from being one-to-one. The inversion of ports and wires requires changing $\overset{w}{\sim}$ and $\overset{p}{\sim}$ from binary to three-way relations, adding a *sense of correspondence* (*positive* or *inverse*) to each tuple. Because they have different arities, the four subsidiary relations can no longer be put together in one union. Instead, a correspondence is represented by a 4-tuple of the subsidiary correspondences $\left\langle \overset{t}{\sim}, \overset{i}{\sim}, \overset{w}{\sim}, \overset{p}{\sim} \right\rangle$

A cell instance i in a view might not be related by $\overset{i}{\sim}$ to anything for one of two reasons:

- i is deleted during the boring reconciliation, and thus does not correspond to anything in the other view, or
- i is below the clipping frontier, and thus corresponds to part of whatever its ancestors that survive the clipping correspond to.

Knowing the clipping frontiers is necessary to distinguish these two cases. Thus, the representation of a correspondence must also represent the relevant clipping frontiers.

In summary, base correspondences have the following features.

- A base correspondence between reconciled views V_1 and V_2 can be represented by a six-tuple $\left\langle F_1, \overset{t}{\sim}_{12}, \overset{i}{\sim}_{12}, \overset{w}{\sim}_{12}, \overset{p}{\sim}_{12}, F_2 \right\rangle$, where F_1 and F_2 are clipping frontiers of V_1 and V_2 (respectively).
- $\overset{t}{\sim}$ is a partial, nearly one-to-one, binary relation between the cell types of one clipped view and the cell types of the other.
- $\overset{i}{\sim}$ is a partial one-to-one binary relation between the cell instances of one clipped view and the cell instances of the other.

- $\overset{w}{\sim}$ is a partial three-way relation between the atomic wires of one clipped view, the atomic wires of the other, and the senses of correspondence.

- $\overset{p}{\sim}$ is a partial three-way relation between the atomic ports of one clipped view, the atomic ports of the other, and the senses of correspondence.

5.6.3 The Main Correspondence

The base correspondence and the reconciliation correspondences have different features, and those differences make the main correspondence even more complex. The following formulation of correspondences covers the base correspondence, the reconciliation correspondences, *and* the main correspondence.

- A correspondence between view V_1 and view V_2 can be represented by a five-tuple $\left\langle F_1, \overset{t}{\sim}_{12}, \overset{iw}{\sim}_{12}, \overset{p}{\sim}_{12}, F_2 \right\rangle$, where F_1 and F_2 are clipping frontiers of V_1 and V_2 (respectively). The symbol $\sim_{12}$ is used to stand for this tuple. A correspondence only addresses the parts of V_1 that survive clipping by F_1 and the parts of V_2 that survive clipping by F_2.

- $\overset{t}{\sim}_{12}$ is a partial, nearly one-to-one, binary relation between the cell types of V_1 above F_1 and the cell types of V_2 above F_2. Thus $\overset{t}{\sim}$ is *not* required to tag any of the atomic cell types of either (unclipped) view. A cell type is considered to be tagged if either it is related by $\overset{t}{\sim}_{12}$ to another cell type or it is a member of F_1 or F_2.

- $\overset{iw}{\sim}_{12}$ is a partial three-way relation between the introductory instance and wire paths of V_1 above F_1, the introductory instance and wire paths of V_2 above F_2, and the senses of correspondence.

- $\overset{p}{\sim}_{12}$ is a partial three-way relation between the ports of V_1 above F_1, the ports of V_2 above F_2, and the senses of correspondence.

So far, this formulation is not very different from earlier ones. The biggest difference, and added complexity, is in the procedure for composing two correspondences. This procedure is similar to the one for composing PWCoreLichen correspondences (see Section 4.3 on page 62). The procedure is given the representation $\left\langle F_1, \overset{t}{\sim}_{12}, \overset{iw}{\sim}_{12}, \overset{p}{\sim}_{12}, F_{2a} \right\rangle$ of a correspondence between the entities of V_1 and those of V_2, and the representation $\left\langle F_{2b}, \overset{t}{\sim}_{23}, \overset{iw}{\sim}_{23}, \overset{p}{\sim}_{23}, F_3 \right\rangle$ of a correspondence between the entities of V_2 and those of V_3. The following steps yield a representation of the composite correspondence between the entities of V_1 and those of V_3. This procedure assumes that no cell type of F_{2b} is below the frontier F_{2a}. This assumption is valid because $F_{2a} \neq F_{2b}$ only for the composition of a base correspondence with a reconciliation correspondence, wherein the reconciliation correspondence can take the role of $\sim_{12}$, and thus F_{2a} is the atomic cell types of (unclipped) V_2.

1. Compute $\sim'_{12}$ from $\sim_{12}$ and $\sim'_{23}$ from $\sim_{23}$ by tightening and/or loosening so that $\sim'_{12}$ and $\sim'_{23}$ tag as nearly as possible the same cell types of V_2. The only exceptions necessary are for composite cells of F_{2b} that are tagged by $\overset{t}{\sim}_{23}$ but not $\overset{t}{\sim}_{12}$.

2. Use *lopsided loosening* to compute $\overset{iw''}{\sim}_{23}$ from $\overset{iw'}{\sim}_{23}$, accommodating for the differences in which cell types of V_2 are tagged.

3. Compose the relations to get the result: $\left\langle F_1, \overset{t'}{\sim}_{12} \circ \overset{t'}{\sim}_{23}, \overset{iw'}{\sim}_{12} \circ \overset{iw''}{\sim}_{23}, \overset{p'}{\sim}_{12} \circ \overset{p'}{\sim}_{23}, F_3 \right\rangle$.

The major difference between this procedure and the one for PWCoreLichen correspondences is that Step 2 may be needed to finish the job that would be accomplished for PWCoreLichen by Step 1 alone. Because correspondences only address the parts of a view above some clipping frontier, when two correspondences being composed use different frontiers (that is, $F_{2a} \neq F_{2b}$) for their common view (V_2) the ordinary tightening and/or loosening may not be enough to get the two representations to tag the same cell types of V_2. This inconsistency can arise whenever there is some composite cell type t_2 of V_2 that is a member of F_{2b} but is above F_{2a} and is not tagged by $\overset{t}{\sim}_{12}$. Cell type t_2 must be tagged by $\sim'_{23}$ because t_2 is atomic in the clipped (by F_{2b}) view, and thus ordinary loosening cannot 'expand' it into anything. If tightening can make $\sim'_{12}$ tag t_2, there is no problem. Otherwise, lopsided loosening must be used. Lopsided loosening is a last-ditch effort to prepare $\overset{iw'}{\sim}_{23}$ for Step 3. Lopsided loosening expands t_2 into a set of instance and wire paths that cover it, by replacing the paths that end at t_2, and thus would not match up with any paths in $\overset{iw'}{\sim}_{12}$ (because t_2 is not tagged by $\sim'_{12}$), with the longer paths that pass through t_2 and *do* match up with paths in $\overset{iw'}{\sim}_{12}$. In particular, lopsided loosening replaces every tuple $\langle p_2, s, p_3\rangle$, where p_2 is an instance path that ends at t_2 and s is the sense of the correspondence, with all the tuples $\langle p_2 + q_2, s, p_3\rangle$, where q_2 is an introductory (according to $\sim'_{12}$) instance or wire path that starts at t_2. Lopsided loosening only needs to adjust the instance/wire path relation; the other four components of a correspondence representation can be used as they are.

The three-way relations are composed by linking according to the common paths and "XORing" the senses:

$$\overset{iw'}{\sim}_{12} \circ \overset{iw''}{\sim}_{23} = \left\{ \langle p_1, s, p_3\rangle \;\middle|\; \exists u, v, p_2 \ni \left(\langle p_1, u, p_2\rangle \in \overset{iw'}{\sim}_{12}\right) \wedge \left(\langle p_2, v, p_3\rangle \in \overset{iw''}{\sim}_{23}\right) \wedge \left(s = u \oplus v\right)\right\}.$$

Table 5.5 shows $\oplus$.

5.6.4 Correspondence Summary

The correspondences encountered in the Lichen and MIPS-X study are broadly similar to those of PWCoreLichen. The reconciliation correspondences are very similar, but the base correspondence for MIPS-X needs additional complexity because the base comparison is more powerful. The paradigm of using paths and mathematical relations works even for Lichen and MIPS-X, provided that sufficient attention is paid to the clipping frontiers of the base comparison and the difficulties of composing correspondences.

$\oplus$	positive	inverse
positive	positive	inverse
inverse	inverse	positive

Table 5.5: How to Combine Senses of Correspondence

5.7 Lichen and MIPS-X Summary

The Lichen and MIPS-X study investigates a more powerful informed comparison. Lichen's reconciliation repertoire has no *a priori* limits (as PWCoreLichen's does, due to the DATools methodology). The two views of MIPS-X make a good test of reconciliation, because they were designed with no constraints on the similarity of the hierarchies (below the top level of decomposition). This study shows that a repertoire of 26 transformations of moderate power and complexity suffices to describe reconciliations reasonably succinctly: the reconciliation of the *pc* is described with 61 invocations, taking 223 words. Lichen reconciliations are asymptotically efficient: each transformation takes an amount of time merely proportional to the number of entities affected in a hierarchical representation. The reconciled views have essentially identical hierarchies, and yet are not appreciably flatter than the originals; thus the base comparison also can be asymptotically efficient.

The base comparison method, Comparison Modulo Boring Components, is also more powerful than PWCoreLichen's. It must be, because some of the behaviorally insignificant differences between the views are flat-significant, and thus cannot be removed by the reconciliation. These differences are due to boring components (or boring fractions of components), which simply copy signals from input ports to output ports with possible inversion. Boring components include buffers and inverters, as well as wiring cells implemented by Modula-2 code. Most of the differences involving boring components are consequences of the difference in level of abstraction between the views. Comparison Modulo Boring Components reduces the amount of circuitry to be compared through simulation by a factor of a little more than two for the *pc*; this factor is disappointingly small because the funsim view of the *pc* does not have much regularity when its elements are weighted by the amount of layout used to implement them. However, because one large comparison is replaced by many smaller ones, the number of cycles needed is also reduced, by a factor that is difficult to quantify.

Because the limitations of PWCoreLichen do not apply to MIPS-X, the correspondence between the entities of the two MIPS-X views is more complex, and more illustrative of the correspondences produced by Informed Comparison. These correspondences are formulated as simple mathematical relations. Instance paths and wire paths are used in place of simple cell instances and wires, in order to be able to represent the correspondence between views with major differences in cell structure. Since the base comparison establishes the correspondence only of entities above certain clipping frontiers, extra care must be taken when composing such correspondences.

Chapter 6

Concluding Remarks

Previous chapters present the comparison problem, introduce Informed Comparison, and give two examples. This final chapter summarizes Informed Comparison, discusses its problems and limitations, and suggests several directions for future research.

6.1 Summary

Informed Comparison is a hierarchical comparison technique that can compare two views at different levels of abstraction with different hierarchies. Informed Comparison first reconciles the differences in hierarchy by applying hierarchy transformations, and then finishes with a hierarchical base comparison technique that can take advantage of the similarity of the hierarchies. The reconciliation is directed by the key, which describes the intended relation between the hierarchies of the views. The key simply consists of invocations of hierarchy transformations. Many qualities of Informed Comparison depend on the repertoire of transformations available to the reconciliation and on the base comparison technique.

Informed Comparison is superior to existing techniques, which either require the views to use essentially identical hierarchies or reconcile the differences with (usually complete) flattening. Requiring the views to use essentially identical hierarchies is an undesirable restriction, because different divisions are preferable at different levels of abstraction. Flattening is disadvantageous for many reasons, the foremost of which is a potentially large degradation of asymptotic performance. Informed Comparison does not require any particular similarity between the hierarchies of the views being compared, and has better ways of reconciling those differences than flattening.

Informed Comparison's contribution is the reconciliation of hierarchy differences; the rest of the comparison problem remains. Notable remaining difficulties arise from the use of different levels of abstraction in different views, and the fact that not all comparisons benefit greatly from a hierarchical solution.

PWCoreLichen and the Lichen and MIPS-X study illustrate the range of possibilities offered by Informed Comparison. PWCoreLichen has a limited repertoire of reconciliations and fits into a well-defined methodology. The key for a PWCoreLichen comparison is contained in design information captured for other purposes; thus, no extra effort is required from the designers to create and maintain the key. PWCoreLichen's base comparison technique is hierarchical structural comparison, which is very fast but has limited power. PWCoreLichen's only major transformation of cell structure is partial flattening, and thus the reconciled views could be very wide and flat; PWCoreLichen's users take care to avoid provoking this problem. A PWCoreLichen comparison beats a completely flat structural comparison by nearly the regularity factor of the reconciled views; in an example studied in Chapter 4 that is 13.

In contrast to PWCoreLichen, the Lichen and MIPS-X study has a large repertoire of transformations and a more powerful base comparison method (the new method of Comparison Modulo Boring Components). The key is not contained in existing design data, and thus has to be created by hand. Even so, the key is reasonably small: the reconciliation of the two views of the *pc* unit of MIPS-X consists of 61 transformation invocations, with an average of under 4 "words" for each invocation. The greater power of Lichen reconciliations means the reconciled views are not much wider or flatter than the originals. The successful comparison of the two views of MIPS-X also requires a more powerful base comparison method than that used in PWCoreLichen, mainly because the two MIPS-X views are at different levels of abstraction. Comparison Modulo Boring Components is able to correctly compare two views that differ by the presence/absence of power supply wiring and boring components, which simply copy signals from input ports to output ports with possible inversion. Boring components include buffers and inverters, as well as wiring cells implemented by Modula-2 code. The Comparison Modulo Boring Components of the two views of the MIPS-X *pc* would have to compare a little less than half the amount of circuitry that a comparison of the whole, completely flattened, views would; this fraction is disappointingly large because the regularity of the *pc* funsim, when weighted by the amount of layout corresponding to each element, is low (around 2). The Comparison Modulo Boring Components also benefits from requiring simulation of fewer cycles of the circuitry being compared, because it compares multiple, smaller circuits instead of one larger circuit.

Informed Comparison enables efficient comparison of alternate views at different levels of abstraction with different hierarchies. The crucial enabling factor is the capture of a small amount of additional design information, the key. While the use of different hierarchies still has some disadvantages, designers are no longer forced to use essentially identical hierarchies because of the inefficiency or incompetence of their comparison programs.

6.2 Problems and Limitations

6.2.1 Remaining Comparison Difficulties

Informed Comparison only removes one difficulty of the comparison problem: the use of different hierarchies for organizational purposes. The comparison problem includes other difficulties, many of which are harder to resolve than hierarchy differences. In some instances Informed Comparison's reconciliation can not greatly ease the comparison problem, because of the magnitude of the fundamental differences between the views.

Some of the remaining comparison difficulties stem from the fact that the alternate views are at different levels of abstraction. An example is comparing an expression like "$a \times b$" in a high-level language with the layout of a serial multiplier. Verifying the consistency of those two requires an understanding of computation; Informed Comparison's reconciliation uses only a simple understanding of structure. Another example is verifying relationships that take time into account. Such relationships are often used in work on synthesis driven by register-transfer level descriptions, where, for example, a variable in the register-transfer level description might have its value stored in different Boolean-level components at different times. The differences between the views in such relationships go "deeper" than circuit structure (while having structural differences as consequences), and no amount of flat-insignificant transformation is going to remove these deeper differences (although certain flat-insignificant transformations might still ease the comparison).

Two views can have deep differences even if they are at the same level of abstraction. Figure 2.14 on page 28 shows a low-level example: two different ways of computing $a \wedge (b \vee c)$. A high-level example is sorting by bubble sort vs. sorting by QuickSort. When deep differences are confined within low-level cells, the higher levels of the views' hierarchies can be reconciled, and hierarchical base comparison techniques can derive benefit from the essential similarity of the upper hierarchies. The higher the levels of hierarchy affected by the deep differences, the less reconcilable hierarchy is left, and the less the amount of benefit to be derived from hierarchical base comparison techniques. The Informed Comparison of the two views of the MIPS-X *pc* reduces the amount of circuitry to be compared by a disappointingly small factor because of the height at which the two views have deep differences.

Hierarchical comparison techniques may be less than greatly beneficial, for the reasons given above and more. One of the big benefits of hierarchical techniques is reducing total problem size: because a hierarchical description succinctly represents repetition using the cell type/instance duality, it can be much shorter than the equivalent flat description. However, some circuits simply do not have much repetition. Also, the formulation of hierarchy used in this dissertation leads to lower regularity factors than some other formulations, when wires are counted as well as cell instances (as is appropriate, for example, when considering the performance of structural comparison). Due to the use of explicit interfaces, every net is cut into different wires at every cell boundary. For example, an n-bit bus that is passed up k levels of hierarchy contributes a total of $n \times k$ atomic wires to the hierarchical description. In formulations (such as Magic's) that do not use explicit interfaces, fewer explicit wires are needed, leading to higher regularity factors.

The need to compare modulo 'don't care' conditions is another of the remaining difficulties of comparison.

6.2.2 Comparison versus Synthesis

A strong trend in VLSI design is the growing use of automatic synthesis programs. This work sometimes adopts the slogan "correctness by construction". If one view is correctly synthesized from another, need those two views be compared? In the grand and glorious future, when significant programs can be proven correct, those views will not need comparison. However, in the present situation, comparison is useful as a check of a synthesis program's work (as for a human designer's work). Of course, the comparison program is not proven correct either; thus comparison only increases confidence. There are two reasons why a comparison can increase confidence of equivalence.

- Comparison is easier than synthesis. Many views are equivalent to a given one, and the synthesis problem involves picking one that maximizes some criteria; comparison is not concerned with those criteria—it only needs to verify the equivalence. To the degree that a comparison program is simpler than a synthesis program, the comparison program can be expected to have fewer bugs than the synthesis program.

- An independent check increases confidence. Even if the comparison and synthesis programs are of comparable complexity, if the comparison program works in a different way than the synthesis program or is written by different people, it can be expected to have different bugs. Thus a synthesized view that passes comparison is more likely to be truly consistent with its specification than a synthesized view that has not been compared.

Checking the work of synthesis programs is of practical value; PWCoreLichen has found bugs in layout generators.

6.3 Future Work

Difficulties, opportunities, and questions encountered in this work suggest many possible further investigations.

- **Performance constants.** In the PWCoreLichen example reported in Chapter 4, the copying of the views took about six times as long as the base comparison—even though copying seems like it should be an easier activity. Even the fact that this copying set up pointers between the original and copied data structures seems insufficient to explain this large cost. Why should that step take so long? Lichen was not tuned for performance, and is very slow. How fast could it be? Both Lichen and PWCoreLichen run out of memory disappointingly quickly—in a 32-megabyte address space! How little memory can Informed Comparison be made to use?

- **Comparing other circuits; having real designers write some keys.** These would make good tests of the generality of the techniques of this dissertation.

- **More abstraction transformations.** Although it is easy to make a repertoire that enables reconciliation of any two flat-identical views, it is not easy to guarantee that the keys will be concise. The conciseness of the key depends in part on how well the repertoire of transformations approximates the designers' repertoire of ways of thinking about relationships between hierarchies. The modestly abstract repertoire of Lichen serves surprisingly well for the MIPS-X *pc*. Even so, an examination of Appendix A will suggest more abstract transformations. For example, very early in the layout transformations, six latches are moved from the *ireg* into the *pc*.[1] The technique used to identify the latch components—listing transistors by their machine-generated names—is tedious and fragile (different names could be assigned after small changes in the

[1] The six uses of the special case of unflattening, *UnflattenOnce*, could be replaced by one use of the general case—if I had taken the time to implement it. Because the six latches end up below the simulation frontier, it does not matter whether they are all instances of the same cell type or of six distinct but equivalent cell types.

layout). A more satisfying technique would involve some ability to recognize the latches, perhaps given a prototype. This could be formulated in a purely structural way, or via deeper semantics.

- **Key development.** How do the designers develop and debug the key? I developed the key presented in Appendix A incrementally, by applying it, then printing and examining the resulting hierarchies, and then making changes. My experience is atypical, because I was also learning about the MIPS-X design and developing Lichen at the same time. If the Informed Comparison program is good at translating messages concerning transformed entities into messages concerning the original entities, the designers may not need to print out and examine the transformed views.

- **Improving array data structures and algorithms.** Lichen's representation for the connectivity within an array grows proportionally with the number of elements in the array; I suspect there is a way to represent most arrays that is insensitive to the number of elements. Also, Lichen's test for whether merging a set of ports of an array cell type preserves the "arrayness" of that cell types is conservative. It would be interesting to find a way to test that condition precisely.

- **Generalizing arrays.** What are the costs and benefits of generalizing arrays to enable the insertion of components, such as buffers, every few elements? What about generalizing to 'programmed' arrays, such as PLAs and decoders?

- **Implementing correspondence-keeping and Comparison Modulo Boring Components.** A good way to test the designs presented in Chapters 4 and 5.

- **Manipulating other non-structural information.** The examples of this dissertation manipulate few kinds of non-structural information, compared to the full spectrum employed in VLSI design.

- **Improving the correspondences.** More sophisticated representations for correspondences may be worthwhile. For example, the correspondences of this dissertation have features that directly correspond to the flattening/unflattening transformations, but none that directly correspond to the other major cell structure transformations; perhaps this should be changed. Also, further theoretical and practical work on simple ways of presenting these complex correspondences would be interesting.

- **Letting the key state only the intended relationship between the views;** not the choice of reconciled hierarchies nor the transformations. The reconciliation program would be responsible for choosing the reconciled hierarchies and planning a transformation strategy.

- **Omitting information from the key.** The reconciliation program would use search techniques to determine the missing information from the views and the given key information. The difficulty of this task is increased by the possibility of the two views having some inconsistencies.

- **Adding temporal structure.** By enhancing the structural theory used in Informed Comparison to include time, enough power may be gained to express the relationships (and thus transform) between the views used in register-transfer-driven synthesis. However, the reconciliation repertoire would thus become the same as the repertoire of transformations used in the synthesis—which removes much of the benefit of using a comparison to check the synthesis. Instead, comparison and synthesis become two applications of the same underlying mechanism (as noted by Parker, Kurdahi, and Mlinar [Parker84]).

- **Verifying relationships more general than equivalence.** The key could easily invoke transformations that change information tested for equivalence in the base comparison; the whole Informed Comparison thus tests not equivalence, but a more general relationship stated in the key. It is unclear, however, whether there is any methodological need for this generalization.

Appendix A

The MIPS-X pc Relation

A.1 The Repertoire

Following are the transformation procedures used in the reconciliation of the MIPS-X *pc*.

UnifyPorts: PROC [];

> *The canonicalization transformation.*

DeducePortAndWireStructure: PROC [];

> *The canonicalization transformation.*

CleanupDesign: PROC [];

> *The canonicalization transformation.*

ImportAtomicWireOnce: PROC [cellType, wire: REF ANY[1]];

> *A special case of pointlessly importing a wire, wherein the cell type (*cellType*) into which the wire is being imported is instantiated only once. This special case is significantly easier to specify than the general one, which would have to include a pairing of cell instances with wires.*

[1] In Cedar, REF means "reference-counted pointer". Cedar provides the ability to safely discriminate on the type of the actual referent of a REF ANY. Thus the key may specify *wire* by its name or by passing it directly.

ExportWires: PROC [from, wires: REF ANY];

Pointlessly exports wires from their cell type.

FlattenType: PROC [cellType: REF ANY];

Flattens out a cell type.

FlattenInstance: PROC [instance: REF ANY];

Flattens out a cell instance.

FlattenNestedArrays: PROC [];

Flattens nested arrays into simple arrays.

UnflattenOnce: PROC [iName, tName: ROPE,[2] parent, siblings: REF ANY];

A special case of the inverse of flattening out a cell type, wherein the cell type (un)flattened is instantiated only once. iName *and* tName *are the names of that instance and cell type, respectively;* siblings *are the cell instances to be gathered together, and* parent *is the cell type in which they are found.*

RaiseGCs: PROC [childType, grandchildren: REF ANY]
RETURNS [raised: set of cell instance];

Raises grandchildren *out of the cell type* childType.

LowerKidsOnce: PROC [parent, kids, sibling: REF ANY]
RETURNS [lowered: set of cell instance];

*A special case of lowering children wherein the cell type (*parent*) into which the children (*kids*) are lowered has only once instance (*sibling*).*

ShortenArrayInstance: PROC [instance: REF ANY, end: {low, high}, by: NAT];

[2] ROPE is the standard type for strings in Cedar.

A special case of raising grandchildren wherein the grandchildren are the by *elements at the* end *end of a one-dimensional array cell type. Starts by distinguishing cell types to make* instance *the only instance of its type.*

Transpose: PROC [outerType: REF ANY];

Interchanges the top two levels of structure below outerType.

DropPhysical: PROC [];

Deletes the physical information.

InheritNames: PROC [renamingFileName: ROPE];

Both renames and inherits names in one pass. The renamings are read from the given file.

PruneLessInterestingNames: PROC [];

The canonicalization transformation.

UseArrayRepresentation: PROC [cellTypes: REF ANY];

Changes Lichen's representation of the given cell types from the general to the one for arrays; checks that those cell types do in fact have array regularity.

MinimizeArrayPeriods: PROC [];

The canonicalization transformation that ensures that every array representation uses the smallest possible period.

Subcells: PROC [parentTypes, instanceTypes, instanceNames: REF ANY];

A utility procedure for identifying sets of cell instances. Arguments may be name patterns, using the '' character to match any substring.*

A.2 The Key

Below are the transformation invocations that apply to the funsim view of the *pc*.

```
InheritNames[“/dev/null”];
UnifyPorts[];
PruneLessInterestingNames[];
CleanupDesign[];
ShortenArrayInstance[“PCIncer/pcdrv4”, low, 2];
ShortenArrayInstance[“PCIncer/pcdrv5”, low, 2];
ShortenArrayInstance[“PCIncer/pcincl”, low, 2];
```

Below are the transformation invocations that apply to the extracted view of the *pc*.

```
InheritNames[“pc-l.renames”];
UnflattenOnce[“branchLatch”, “BranchLatch”, “decoderIR”,
  LIST[“Q11#”, “Q12#”, “Q28#”, “Q35#”, “Q40#”, “Q44#”, “Q51#”]];
UnflattenOnce[“jpcLatch”, “JpcLatch”, “decoderIR”,
  LIST[“Q7#”, “Q8#”, “Q22#”, “Q27#”, “Q32#”, “Q39#”, “Q49#”]];
UnflattenOnce[“jpcrsLatch”, “JpcrsLatch”, “decoderIR”,
  LIST[“Q9#”, “Q10#”, “Q23#”, “Q33#”, “Q34#”, “Q43#”, “Q50#”]];
UnflattenOnce[“jspciLatch”, “JspciLatch”, “decoderIR”,
  LIST[“Q5#”, “Q6#”, “Q18#”, “Q25#”, “Q26#”, “Q38#”, “Q48#”]];
UnflattenOnce[“movfrsLatch”, “MovfrsLatch”, “decoderIR”,
  LIST[“Q3#”, “Q4#”, “Q17#”, “Q21#”, “Q24#”, “Q31#”, “Q47#”]];
UnflattenOnce[“movtosLatch”, “MovtosLatch”, “decoderIR”,
  LIST[“Q1#”, “Q2#”, “Q16#”, “Q19#”, “Q20#”, “Q30#”, “Q46#”]];
UnflattenOnce[“pcOther”, “pcOther”, “pc”, LIST[
  LowerKidsOnce[“mipsx”,
    RaiseGCs[“ireg”,
      RaiseGCs[“decoderIR”, LIST[“branchLatch”, “jpcLatch”, “jpcrsLatch”,
        “jspciLatch”, “movfrsLatch”, “movtosLatch”]] ],
    “pc”],
    “p0”, “p1”, “pc1”, “pc2”, “pcp”, “pcph”]];
[] ← RaiseGCs[“pc”, RaiseGCs[“pcdisp”, “Q1#”]];
FlattenType[“pccntdrvs”];
FlattenType[“pcdvcntdrv”];
FlattenType[“pcaludrv”];
FlattenType[“pcdisout”];
FlattenType[“pcincout”];
FlattenType[“pcfsm”];
UnifyPorts[];
PruneLessInterestingNames[];
ExportWires[“pc”, LIST[“PCBus_b_sl/1”, “PCBus_b_sl/2”,
    “PCBus_b_sl/26”, “PCBus_b_sl/27”, “PCBus_sl/1”, “PCBus_sl/2”]];
FOR i: NAT IN [1 .. 16] DO
    [] ← ImportAtomicWireOnce[“pcOther”,
        IO.PutFR[“pc/Immed_sla/%g”, [integer[i]]] ];
    ENDLOOP;
[] ← ImportAtomicWireOnce[“pcdisp”, “pc/Immed sla/0”];
DeducePortAndWireStructure[];
CleanupDesign[];
UseArrayRepresentation[LIST[“pcaludr2sl”, “pcinc2sl”, “pcxor2”,
    “pcdis2sl”, “pcincfr2sl”, “pla2driver”, “pcchn2sl”]];
FlattenNestedArrays[];
DropPhysical[];
MinimizeArrayPeriods[];
UnflattenOnce[“pcl4”, “2InputLatch”, “pcchnsl”, LIST[“pcn”, “pcp”, “Q1#”]];
Transpose[“pcchn2sl[0:0:1103][0:15:220](pcchain.p)”];
Transpose[“pcdis2sl[0:0:307][0:15:220](pcdisout.pcd)”];
Transpose[“pcaludr2sl[0:0:561][0:15:220](pcaludrv.pc)”];
Transpose[“pcincfr2sl[0:0:61][0:15:220](pcincfr.p)”];
Transpose[“pcinc2sl[0:0:293][0:14:220](pcincout.p)”];
FlattenInstance[“pcchain/p”];
```

```
FlattenInstance[“pcdisp/p/pc”];
FlattenInstance[“pcdisp/pcd/pcd”];
FlattenInstance[“pcinc/pci/p”];
FlattenType[“pcincsl”];
FlattenType[“pcincslbt0”];
UnflattenOnce[“PCRandomLogic”, “PCRandomLogic”, “pc”, LIST[
  Subcells[“pc”, “and*”, “*”],
  “pcOther”,
  RaiseGCs[“cmfsm”, LIST[“pcn2”, “pcno1”, “pcnot0”, “pcnot1”]],
  RaiseGCs[“sqfsm”, LIST[“p0”, “p1”, “pc0”, “pc2”, “pcs”]] ]];
UnflattenOnce[“pcincer”, “Incer”, “pcinc”, LIST[“p”, “pc”]];
UnflattenOnce[“pcdispadder”, “DispAdder”, “pcdisp”,
  LIST[“pca”, “p/pc/pc0”, “p/pc/pc1”]];
```

Bibliography

[Ablasser81] I. Ablasser and U. Jäger.
Circuit recognition and verification based on layout information.
ACM IEEE 18th *Design Automation Conference*, 1981.

[Acosta88] Ramón D. Acosta, Mark Alexandre, Gary Imken, and Bill Read.
The role of VHDL in the MCC CAD system.
ACM IEEE 25th *Design Automation Conference*, 1988.

[Baker80] Clark M. Baker and Chris Terman.
Tools for verifying integrated circuit designs.
Lambda, I(3):22–30, 4th quarter 1980.
Name changed from *Lambda* to *VLSI Design* as of Volume II, Number 3, 3rd Quarter 1981.

[Barke84] Erich Barke.
A network comparison algorithm for layout verification of integrated circuits.
IEEE Transactions on Computer-Aided Design, CAD-3(2):135–141, April 1984.

[Barrow84] Harry G. Barrow.
Proving the correctness of digital hardware designs.
VLSI Design, V(7):64–77, July 1984.

[Barth88] Richard Barth, Louis Monier, Bertrand Serlet, and Pradeep Sindhu.

VLSI design aids: Capture, integration, and layout generation. Xerox Palo Alto Research Center, July 1988.

[Beece88] Daniel K. Beece, George Deibert, Georgina Papp, and Frank Villante.
The IBM engineering verification engine.
ACM IEEE 25th *Design Automation Conference*, 1988.

[Blackburn85] Robert L. Blackburn and Donald E. Thomas.
Linking the behavioral and structural domains of representation in a synthesis system.
ACM IEEE 22nd *Design Automation Conference*, 1985.

[Blackburn88] Robert L. Blackburn, Donald E. Thomas, and Patti M. Koenig.
CORAL II: Linking behavior and structure in an IC design system.
ACM IEEE 25th *Design Automation Conference*, 1988.

[Bryant81] Randal Everitt Bryant.
A Switch-Level Simulation Model for Integrated Circuits.
PhD thesis, Massachusetts Institute of Technology, 1981.

[Bryant87a] Randal E[veritt] Bryant.
Algorithmic aspects of symbolic switch network analysis.
IEEE Transactions on Computer-Aided Design, CAD-6(4):618–633, July 1987.

[Bryant87b] Randal E[veritt] Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler.
COSMOS: A compiled simulator for MOS circuits.
ACM IEEE 24th *Design Automation Conference*, 1987.

[Chandrasekhar87] Mandalagiri S. Chandrasekhar, John P. Privitera, and Kenneth W. Conradt.

Aplication of term reqriting techniques to hardware design verification.
ACM IEEE 24th *Design Automation Conference*, 1987.

[Chang87] Hongtao P. Chang and Jacob A. Abraham.
The Complexity of Accurate Logic Simulation.
IEEE International Conference on Computer-Aided Design, 1987.

[Chiang88] Kuang-Wei Chiang, Surendra Nahar, and Chi-Yuan Lo.
Time efficient VLSI artwork analysis algorithms in GOALIE2.
ACM IEEE 25th *Design Automation Conference*, 1988.

[Crawford84] John D. Crawford.
An electronic design interchange format.
ACM IEEE 21st *Design Automation Conference*, 1984.

[Devadas87] Srinivas Devadas, Hi-Keung Tony Ma, and A. Richard Newton.
On the verification of sequential machines at differing levels of abstraction.
ACM IEEE 24th *Design Automation Conference*, 1987.

[Ebeling83] Carl Ebeling and Ofer Zajicek.
Validating VLSI circuit layout by wirelist comparison.
IEEE International Conference on Computer-Aided Design, 1983.

[Gordon81a] Mike Gordon.
A model of register transfer systems with applications to microcode and VLSI correctness.
Department of Computer Science, University of Edinburgh, March, 1981.

[Gordon81b] M[ike] Gordon.
A very simple model of sequential behaviour of nMOS.

In J. Gray, editor, *Proceedings of the VLSI International Conference*, Academic Press, 1981.

[Gordon83] Mike Gordon.
Proving a computer correct with the LCF_LSM hardware verification system.
Computer Laboratory, University of Cambridge, England, [1983].

[Grodstein87] Joel J. Grodstein and Tony M. Carter.
SISYPHUS—an environment for simulation.
IEEE International Conference on Computer-Aided Design, 1987.

[Gupta83] Anoop Gupta.
ACE: A circuit extractor.
ACM IEEE 20th *Design Automation Conference*, June 1983.

[Hoffman82] Christoph M. Hoffmann.
Group-Theoretic Algorithms and Graph Isomorphism.
Springer-Verlag, 1982.

[Hopcroft79] John E. Hopcroft and Jeffrey D. Ullman.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, 1979.

[Horowitz87] Mark Horowitz, John L. Hennessy, Paul Chow, P. Glenn Gulak, John M. Acken, Anant Agarwal, Chorng-Yeung Chu, Scott A. McFarling, Steven A. Przybylski, Steve E. Richardson, Arturo Salz, Richard T. Simoni, Don C. Stark, Peter A. Steenkiste, Steve W. K. Tjiang, and Malcolm J. Wing.
A 32b microprocessor with on-chip 2Kbyte instruction cache.
IEEE International Solid-State Circuits Conference, February 1987.

[Hwang87] Seung H. Hwang and A. R. Newton.
An efficient design correctness checker of finite state machines.
IEEE International Conference on Computer-Aided Design, 1987.

[Johnson81] David S. Johnson.
The NP-completeness column: An ongoing guide.
Journal of Algorithms, 2(4):393–405, Academic Press, December 1981.

[Katz86] R. H. Katz, M. Anwarrudin, and E. Chang.
A version server for computer-aided design data.
ACM IEEE 23rd *Design Automation Conference*, 1986.

[Kubo79] Noburo Kubo, Isao Shirakawa, and Hiroshi Ozaki.
A fast algorithm for testing graph isomorphism.
Proceedings of the International Symposium on Circuits and Systems, 1979.

[Lampson81] Butler W. Lampson and Kenneth A. Pier; Butler W. Lampson, Gene A. McDaniel, and Severo M. Ornstein; Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier.
The Dorado: A high-performance personal computer, three papers.
Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California, 94304. CSL-81-1, January 1981.

[Lathrop87] Richard H. Lathrop, Robert J. Hall, and Robert S. Kirk.
Functional abstraction from structure in VLSI simulation models.
ACM IEEE 24th *Design Automation Conference*, 1987.

[Madre88] Jean-Christophe Madre and Jean-Paul Billon.
Proving circuit correctness using formal comparison between expected and extracted behavior.
ACM IEEE 25th Design Automation Conference, 1988.

[Malik88] Sharad Malik, Albert R. Wang, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli.
Logic verification using binary decision diagrams in a logic synthesis environment.
IEEE International Conference on Computer-Aided Design, 1988.

[Maruyama85] Fumihiro Maruyama and Masahiro Fujita.
Hardware verification.
Computer, 18(2):22–32, IEEE Computer Society, February 1985.

[Mead80] Carver Mead and Lynn Conway.
Introduction to VLSI Systems.
Addison-Wesley, 1980.

[Milne84] George J. Milne.
A model for hardware description and verification.
ACM IEEE 21st Design Automation Conference, 1984.

[Morison87] J. D. Morison, N. E. Peeling, T. L. Thorp, and E. V. Whiting.
EASE: A design support environment for the HDDL ELLA.
ACM IEEE 24th Design Automation Conference, 1987.

[Narendran88] Paliath Narendran and Jonathan Stillman.
Formal verification of the Sobel image processing chip.
ACM IEEE 25th Design Automation Conference, 1988.

[Noice83] David C. Noice.
A clocking discipline for two-phase digital integrated circuits.
Technical Report, Stanford University, January 1983.

[Odawara86] Gotaro Odawara, Masahiro Tomita, Osamu Okuzawa, Tomomichi Ohta, and Zhen-quan Zhuang.
A logic verifier based on Boolean comparison.
ACM IEEE 23rd *Design Automation Conference*, 1986.

[Ousterhout84] John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George S. Taylor.
Magic: A VLSI layout system.
ACM IEEE 21st *Design Automation Conference*, 1984.

[Parker84] Alice C. Parker, Fadi Kurdahi, and Mitch Mlinar.
A general methodology for synthesis and verification of register-transfer designs.
ACM IEEE 21st *Design Automation Conference*, 1984.

[Pfister82] Gregory F. Pfister.
The Yorktown Simulation Engine: Introduction.
ACM IEEE 19th *Design Automation Conference*, 1982.

[Read77] Ronald C. Read and Derek G. Corneil.
The graph isomorphism disease.
Journal of Graph Theory, 1:339–363, John Wiley & Sons, 1977.

[Roth77] J. Paul Roth.
Hardware verification.
IEEE Transactions on Computers, C-26(12):1292–1294, December 1977.

[Saunders87] Larry F. Saunders.
The IBM VHDL design system.
ACM IEEE 24th *Design Automation Conference*, 1987.

[Sequin83] Carlo H. Séquin.
Managing VLSI complexity, an outlook.
Proceedings of the IEEE, 71(1):149–166, January 1983.

[Shiran86] Yehuda Shiran.
YNCC: A new algorithm for device-level comparison between two functionally isomorphic VLSI circuits.
IEEE International Conference on Computer-Aided Design, 1986.

[Stabler87] Edward P. Stabler and Haluk Bingol.
Boolean comparison by simulation.
ACM IEEE 24th *Design Automation Conference*, 1987.

[Stavridou88] V. Stavridou, H. Barringer, and D. A. Edwards.
Formal specification and verification of hardware: A comparative case study.
ACM IEEE 25th *Design Automation Conference*, 1988.

[Supowit86] Kenneth J. Supowit and Steven J. Friedman.
A new method for verifying sequential circuits.
ACM IEEE 23rd *Design Automation Conference*, 1986.

[Suzuki85] Shigenobu Suzuki, Kazutoshi Takahashi, Takao Sugimoto, and Mikio Kuwata.
Integrated design system for supercomputer SX-1/SX-2.
ACM IEEE 22nd *Design Automation Conference*, 1985.

[Suzuki87] Shigenobu Suzuki, Tatsushige Bitoh, Masao Kakimoto, Kazutoshi Takahashi, and Takao Sugimoto.
TRIP: An automated technology mapping system.
ACM IEEE 24th *Design Automation Conference*, 1987.

[Swinehart86] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann.
A structural view of the Cedar programming environment.
ACM Transactions on Programming Languages and Systems, 8(4):419–490, October 1986.

[Takashima82] Makoto Takashima, Takashi Mitsuhashi, Toshiaki Chiba, and Kenji Yoshida.
Programs for verifying circuit connectivity of MOS/LSI mask artwork.
ACM IEEE 19th *Design Automation Conference*, 1982.

[Terman83] Christopher J. Terman.
RSIM—a logic-level timing simulator.
IEEE International Conference on Computer Design: VLSI in Computers, 1983.

[Tygar85] J. D. Tygar and Ron Ellickson.
Efficient netlist comparison using hierarchy and randomization.
ACM IEEE 22nd *Design Automation Conference*, 1985.

[Valid87] Valid Logic Systems.
COMPARE reference manual.
SCALDstar Release 9.1. San Jose, CA, 1 April 1987.

[Veiga84] Pedro M. B. Veiga and Mário J. A. Lança.
HARPA: A hierarchical multi-level hardware description language.
ACM IEEE 21st *Design Automation Conference*, 1984.

[Walker85] Robert A. Walker and Donald E. Thomas.
A model of design representation and synthesis.
ACM IEEE 22nd *Design Automation Conference*, 1985.

[Walker87] Robert A. Walker and Donald E. Thomas.
Design representation and transformation in the System Architects's Workbench.
IEEE International Conference on Computer-Aided Design, 1987.

[Waxman86] Ron Waxman.
The VHSIC hardware description language—a glimpse of the future.
IEEE Design and Test of Computers, 3(2):10–11, April 1986.

[Weise87] Daniel Weise.
Formal verification of MOS circuits.
ACM IEEE 24th *Design Automation Conference*, 1987.

[Wirth83] Niklaus Wirth.
Programming in MODULA-2.
Springer-Verlag, 1983.

[v.d.Wolf88] P. van der Wolf and T. G. R. van Leuken.
Object type oriented data modeling for VLSI data management.
ACM IEEE 25th *Design Automation Conference*, 1988.

[Wong85] Yiwan Wong.
Hierarchical circuit verification.
ACM IEEE 22nd *Design Automation Conference*, 1985.

[Wu87] Ching-Farn E. Wu, Lionel M. Ni, and Anthony S. Wojcik.
Functional recognition of static CMOS circuits.
IEEE International Conference on Computer-Aided Design, 1987.