Understanding Hierarchical Design

Thesis by

James Allely Rowson

In Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

California Institute of Technology

Pasadena, California

1980

(Submitted April 15, 1980)

Acknowledgements

Special thanks are due to Carver Mead. His vision and energy have created an exciting new field which is the basis for this thesis. He also hires dynamite secretarial help.

I am indebted to Jim Kajiya, in whose office this thesis was born. Besides having such a great first name, he is directly responsible for any good mathematics in this thesis.

For creative office-mates, I have the best in Sally Browning and Bart Locanthi (in order of increasing height).

I must also thank Doug Fairbairn for having curly hair, and Martin Newell for many valuable discussions and ideas.

The production of a thesis is often more a psychological trial than a technical one. In that regard, I am appreciative to the many people who helped me to keep grinding. To my parents, without whom I wouldn't be; thank you for being such incredibly good parents and for being so supportive about everything, no matter how bizarre. To Bruce, who provided comic relief. To Bart Locanthi, Sally Browning, Jim Kajiya, and Carver Mead who convinced me that my work might be worthwhile. And most of all, to Donna whose unfailing good spirits and support are the real reason this thesis is finished.

Abstract

With the exponential improvement in integrated circuit technology comes the problem of how to design systems containing millions of devices. This thesis presents a new look at hierarchical design based on the Caltech structured design methodology.

The hierarchy is separated into two parts: leaf cells, containing no instances of other cells, and composition cells, containing only instances of other cells. A leaf cell can be implemented in many different representations. A representation consists of a set of leaf cells and a composition rule that builds correct higher level cells.

The separated hierarchy is suitable for mathematical analysis by the use of Curry's theory of combinators. In this form, a hierarchy is represented by a mathematical operator that produces a digital system from the leaf cells. The question of hierarchical equivalence is examined.

Three sample composition rules, or algorithms, are presented as examples. The SLAP system provides a geometry composition rule that produces the mask description of a system given the geometries of the leaf cells. In analogy to TYPEing in a programming language, two representations that enforce a certain design style are discussed. The first TYPE system guarantees signal integrity. The second TYPE system guarantees mutual exclusion between the sources on a bus.

Table of Contents

List of Figures and Tables

Chapter 1

An Overview

1.1 The Problem -- 10^7 Transistors

VLSI technologies will be fabricating chips containing 100,000 transistors by 1982 [Lattin 1979]. With today's design techniques, it would take around 60 man years to design, and another 60 to debug, such a chip. In the theoretical limits, VLSI chips will contain near 10 million transistors. Without some method for reducing the complexity of design, a 10 million transistor chip would take somewhere near 6000 man years to design.

These numbers clearly point out that there is a widening gap between what VLSI technologies can produce and what system designers can design. As it is, the only chips that approach the available complexity are memory chips, and those only because of their extremely regular patterns. With less regular systems, like microprocessors, designers are having real difficulty just completing complex designs.

An analogy to help visualize the complexity available from VLSI technology has been proposed by Charles L. Seitz of

Caltech [Seitz 1979a]. His analogy relates the basic separation of features on silicon with the size of a typical city block. Wires and buses in an integrated circuit correspond to streets and highways of a city. Just examining this analogy with scaling in mind produces some astounding conclusions.

As illustrated in Figure 1.1, the separation sizes of the mid-1960's, a typical chip was about as complex as the street network of San Jose or Pasadena. Most people can deal with a city of that size from memory. The technology circa 1978 allowed chips whose complexity was near that of the entire San Fransisco Bay Area or that of the Los Angeles Basin. With a 1-micron technology, say around 1985, chips will have the complexity comparable to that of a street network of urban density covering all of California and Nevada! The ultimate physical limits allow chips whose complexity rivals a street network covering the entire North American Continent. Some new approaches are clearly needed in design methodology and design aids to attack this serious design problem.

1.2 Finding a Solution

The Caltech "structured" design methodology [Mead 1979] is an approach to VLSI system design that directly attacks the problems of complex designs. By introducing regularity into a system, the design problem is reduced in complexity. Even traditionally irregular control structures have their regular counterparts in ROM and PLA.

Hierarchical techniques have been the major tool used to design complex systems [Simon 1962][Koestler 1967].

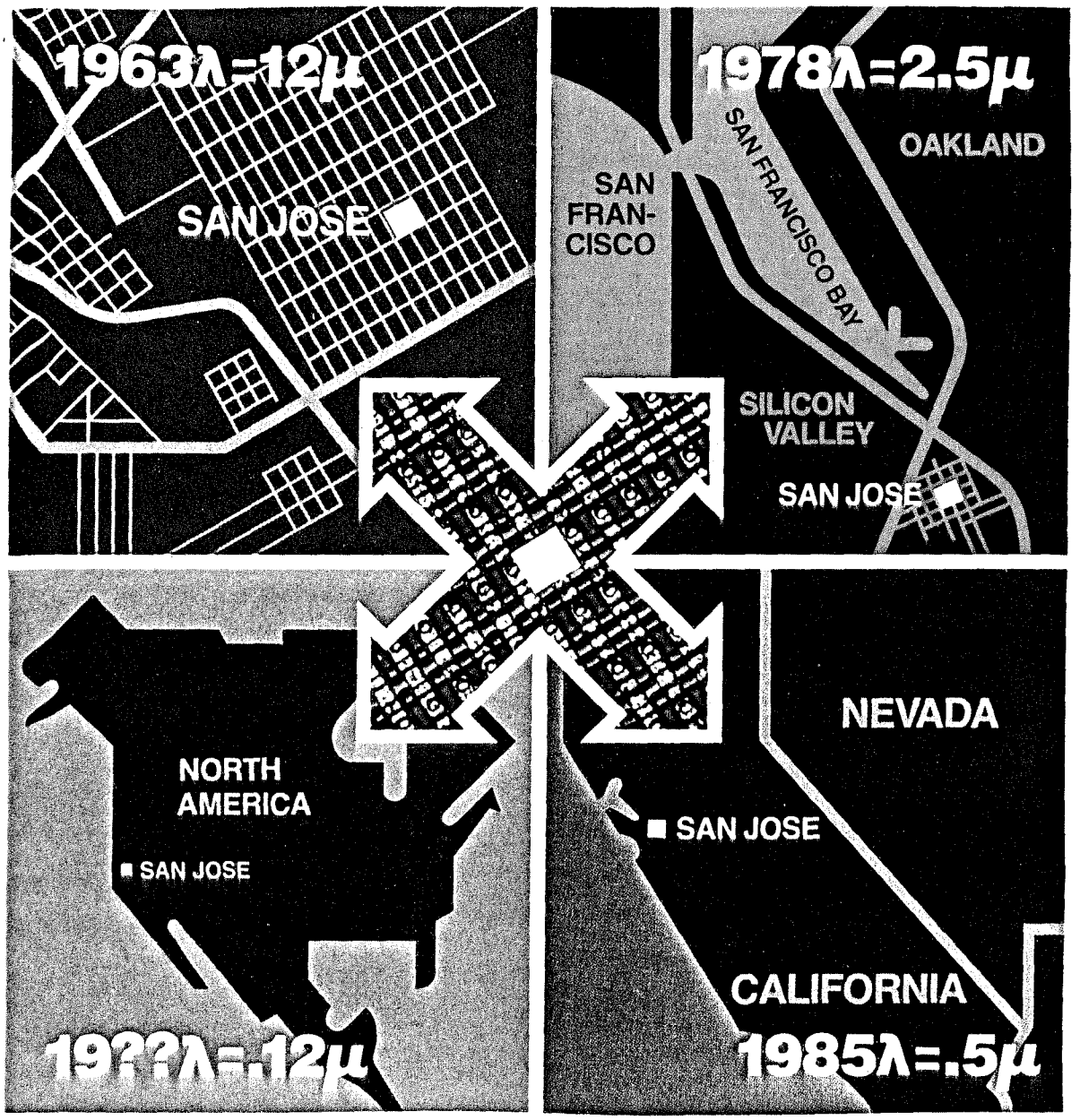


Figure 1.1 VLSI/City Analogy

Hierarchies are used to help partition designs among the design team. As in programming languages, common parts of a design can be factored out and specified only once.

This thesis introduces a new look at hierarchical design in the light of the structured design methodology. By completely separating the "leaves" of the hierarchy from the rest of the tree, mathematical analysis of hierarchies is possible.

When a design is completely separated into its two constituents, the leaves and the hierarchical description, then the role of "composition rules" becomes more important and general. A composition rule is similar to the second step to an inductive proof. Given a correct first case, a theorem is true if, assuming it is true for N, it can be shown true for N+1. Composition rules are a method for guaranteeing some property at the next level in the hierarchy, given that the hierarchy is built from parts that satisfy the same property.

The separated hierarchy becomes a "representation independent" language for specifying a design. Once the hierarchy is extracted, it can be looked at as an operator that composes systems from primitives. In this sense, the hierarchy is completely representation independent. Given that each primitive is described for that representation, and that there is a composition rule for the representation, then the hierarchy can be "implemented" using that representation. Since the hierarchy is representation independent, consistency checking need only be done on the leaves and the composition rules. This approach is much less complex than the traditional

approaches to consistency checking.

In the context of this separated hierarchy, several composition algorithms have been implemented. A method of composing cells geometrically is developed that enhances the adaptability and regularity of designs. The geometric composition algorithm supports the structured design methodology idea of butting cells and "floor plans". Two typing systems to check for legal compositions have also been implemented, with the types propagating up the hierarchy automatically.

1.3 In This Thesis...

The new hierarchical structure is introduced in Chapter 2. In this chapter, the traditional goals of hierarchies are examined in the light of the Caltech structured design methodology. The separated hierarchy is developed as a way of dealing with the hierarchy in a mathematical way.

A mathematical model for a hierarchically described system is developed in Chapter 3. Using Curry's theory of combinators, the hierarchy is completely separated from the functions in the leaves. Using this technique, the hierarchy can be examined in much the same way as mathematical operators like the differentiation operator.

Chapters 4 and 5 present some algorithms consistent with the separated hierarchy. Chapter 4 discusses an algorithm that combines geometrically defined leaves. The geometry algorithm is consistent with the hierarchy in that the "composition" cells introduce no extra geometric information, specifying only logical structure. Chapter 5

presents two typing systems that are used to restrict the legal compositions. The typing systems support the assumptions used in Chapter 3 to develop the mathematics of hierarchies.

Chapter 6 states some of the conclusions from this research and also discusses some of the unanswered questions and fertile areas for further work.

Chapter 2

The Separated Hierarchy

Before we can do much analysis of hierarchies, we need to have a clean, simple model to analyze. This chapter attempts to present and justify the model used throughout this thesis.

The hierarchy presented here is a fusion of two separate ideas: the classical divide-and-conquer type of hierarchical design and the "structured design" style as developed at Caltech for VLSI system design. The classical hierarchical design methodology merely proposes a kind of macro expansion to take advantage of similarities or commonalities in a particular design. The "structured design" style from Caltech is an attempt to adapt to the peculiar constraints of VLSI systems.

The resultant model turns out to be non-VLSI specific, and may have applications in other areas where hierarchical design might be effective.

2.1 Goals of the Hierarchy

Hierarchical design is certainly not a new concept. With the advent of assemblers and compilers that allow macros

and procedures, programming has long been a discipline that encourages the progressive breakdown of large, complex problems. Similarly, almost all designers of complex electronics systems have dealt with the "block diagram" of a system, each block of which represents a less complex subsystem.

One reason for the power of hierarchical design may be that it is the synthesis analog to the inductive proof. An inductive proof is a proof in two steps. An initial case of the theorem is identified and shown correct. Then the theorem is assumed true for some general case, and shown true for the next case. Many times this second step results in assuming the theorem true for some "n" and proving it true for "n+1". Hierarchies resemble the inductive proof when every two adjacent levels of the hierarchy are related by a simple rule. This rule is often called a composition rule. A complete design system will consist of many composition rules that cover the whole range of design tasks. Each composition rule deals with one specific property of the cell that is propagated up the hierarchy.

During the synthesis of a large system, the designer(s) need to make abstractions at various "levels" of the design in order to deal with a manageably small amount of information. Whether designing "top down" or "bottom up", a system is usually broken into pieces whose atomic units are the subsystems. The designer will deal with an abstraction of the subsystems he is using to design the next larger system, not with the complete description of the subsystem. In a "top down" fashion, the designer will specify an abstraction of what the subsystems will do when

he designs them. The incomplete specification of subsystems is the key that allows the designer to work within one piece of a large system without worrying about too many other pieces.

The next few sections will enumerate some of the attributes of a "good" design system. The effectiveness of hierarchies will be examined in this light.

Handling Complexity

With the number of features per chip doubling every two years, complexity handling is clearly necessary in a design system. Any system that purports to aid in the design of VLSI chips has to provide some method of organizing and specifying an enormously complex design.

The problem with handling complexity is the well known limitation of the human short-term memory. Most people have a difficult time dealing with more than around 7 short term ideas [Miller 1956]. The design of a VLSI chip in one fell swoop, i.e. with no levels of abstraction, requires the consideration of well over 7 ideas at any time. The constraints on a particular part of the layout can come from any other place on the chip. Without superhuman foresight or an almost incredible memory for details and interrelationships, no small group of people could successfully complete a design much more complex than those of today.

In hierarchical design, each level of the design is as complicated, or simple, as the designer wants. With the proper self-discipline, and knowledge of how to really use

the hierarchy, the designer can keep the complexity of each level down to a manageable level. The complexity is kept down because the only design data needed are completely local: what the local design problem is, what the available building blocks are and an abstraction of how the building blocks work. Of course, reducing the complexity doesn't come for free. The designer has to be constantly on guard against the natural urge to "patch" in a fix that violates the locality inherent in a hierarchy.

Partitioning

Even though a design system helps to handle the complexity of the design problem, there is still a lot of work to be done. The urge to pile mountains of people on a project is usually an irresistible one, especially with the "first product" economics at work now. However, it's clear that at some point adding more people to a design is non-productive and may, in fact, delay the project [Brooks 1975].

Key to effectively utilizing many people to design one chip is to limit their intercommunication requirements. In this way, each person is free to get on with his piece of the design without spending too much time mis-communicating.

With the proper exploitation of hierarchies, the interfaces between separate design groups can be specifically and carefully designed to reduce the amount of added communication. Hierarchies are inherently full of places where interfaces must be specified.

Understanding

Hand in hand with the complexity arguments comes understandability. After all, not only is it difficult to keep more than 7 things in your head during the design, it is even more difficult to understand them later.

With the proper abstractions, or specifications, available for each subsystem in a hierarchical design, "reading" a design should be easier. The inclusion of some "redundant" descriptions, the abstractions, may help an outsider understand the original intent behind a subsystem.

Sharing

Being able to design using a library of useful subsystems has always seemed like an idyllic situation. The designer need only sit at his desk with a "part catalog" from which he picks the exact pieces he needs to implement a particular function. Indeed, this approach worked fairly well in the digital hardware world with the TTL part catalogs.

One problem with libraries is that changes in technology tend to make a particular library obsolete, requiring a large effort to reimplement it. Making the libraries technology independent has not been successful in the past through various efficiency and generality problems.

Another limitation with libraries is the lack of complete characterization of the parts. No specification language exists that is adequate to describe the quirks that tend to be in any particular implementation. The effort involved

in understanding how to successfully use a subsystem must be less than designing it from scratch.

The third limitation is that of generality or adaptability. No designer will use a library full of huge numbers of very specific parts. The effort involved in searching for a needed part would soon become more than the effort needed to design it. Not only must a library function be general, its implementation must be adaptable. Adaptability of a part means, for example, that the part must be able to "fit" independent of its environment. The more adaptable a part is, the easier it will be to use it since less pre-preparation will need to be done.

In order for a hierarchical design system to support libraries, the specification/abstraction of a particular subsystem has to be complete enough to characterize without being so complex as to be incomprehensible.

2.2 Structured Design Philosophy

The Caltech "structured design" methodology was developed to formalize the interaction between designing a digital system and laying out a VLSI chip. Digital system design has been based on a cost function that minimized active components, since the component cost was the economic limit. When designing a VLSI system, one that is implemented on one or a few chips, a whole new cost function arises based on the limitations and advantages of the technology.

The nMOS technology was used as an example while developing the structured design methodology. When designing with

nMOS, the number and length of the interconnection wires are the limiting factor, and not the number of transistors involved. Generally speaking, the cost, whether area or speed, of communicating some piece of data is more than the cost of any particular operation on the data. In fact, a reasonable size estimate for a function is just the area it will take to route the wires needed for data and control [Mead 1979 -- Chapter 8].

The length of a wire generally determines how much energy or how much time is needed to transmit a piece of data along its length. Thus, designs with lots of long wires will either be high power or slow, depending on what is optimized. Since a design is limited to the two dimensions, the important optimization is the placement in the plane based on the amount of intercommunication [Sutherland 1977] [Mead 1979].

The recognition of these cost constraints in the technology has led to a design style that is commonly called the Caltech structured design methodology. Chips designed in a "structured" way tend to be very regular, with many similar cells and very little "random" wiring. The classic example is the OM chip set designed by Dave Johannsen and Carver Mead [Mead 1979].

Some of the relevant aspects of the structured design style will be discussed in the next few sections. These pieces include floor planning with abutting cells, pieced buses, and perpendicular wiring.

Floor Plans

Floor plans are a major tool used in the planning stage of many chips. In the standard usage, areas for various functions are estimated by people with vast experience, and these areas are laid down on the chip in an attempt to minimize the random interconnection between them. The floor plan is used to get estimates on the size and performance of the chip. The chip sizes can be used to help partition multi-chip systems. Floor plans are also used to control the interactions between independent designers. An example floor plan, of the microcode controller from the OM chip set, is shown in Figure 2.1.

In the structured design philosophy, floor planning is carried further into the design. In particular, the random interconnect is largely eliminated by designing the cells so that they "abut" perfectly, i.e. the signals that need to interconnect come out on the edges of the cells at the same place on both cells. By designing the cells with the same geometric interface, just placing instances of the cells next to each other will also wire them together correctly.

This philosophy tends to make chips in which large numbers of cells have the same "pitch", i.e. the same width or height. These cells also have certain interface signals at the same places so that they "plug" together with no extra interconnect. Data paths, as in the OM, begin to look more like memory: very regular patterns.

In addition to data paths, some control structures have been developed that have the same butting, regular

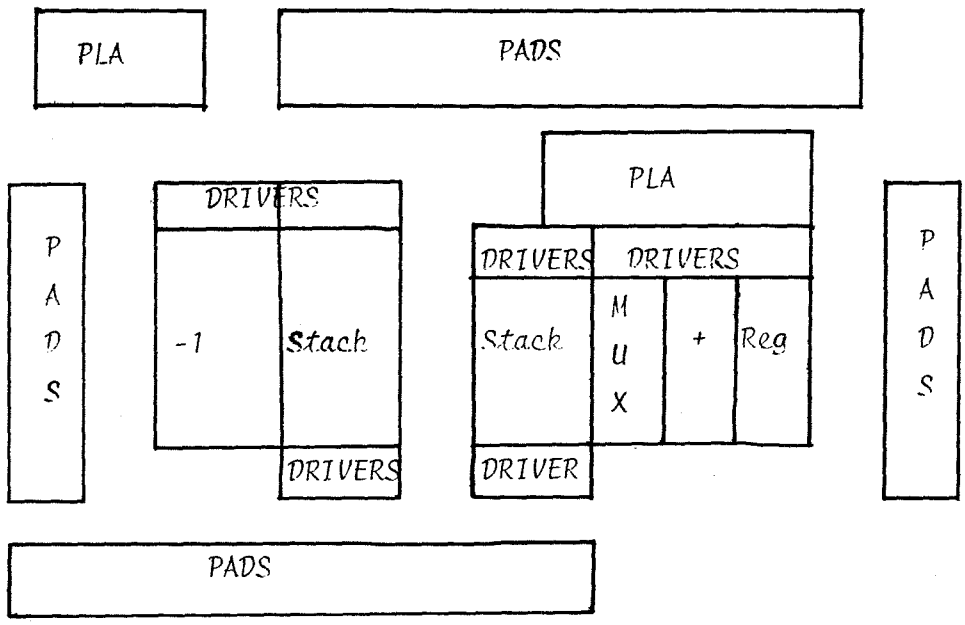


Figure 2.1 A Floor Plan

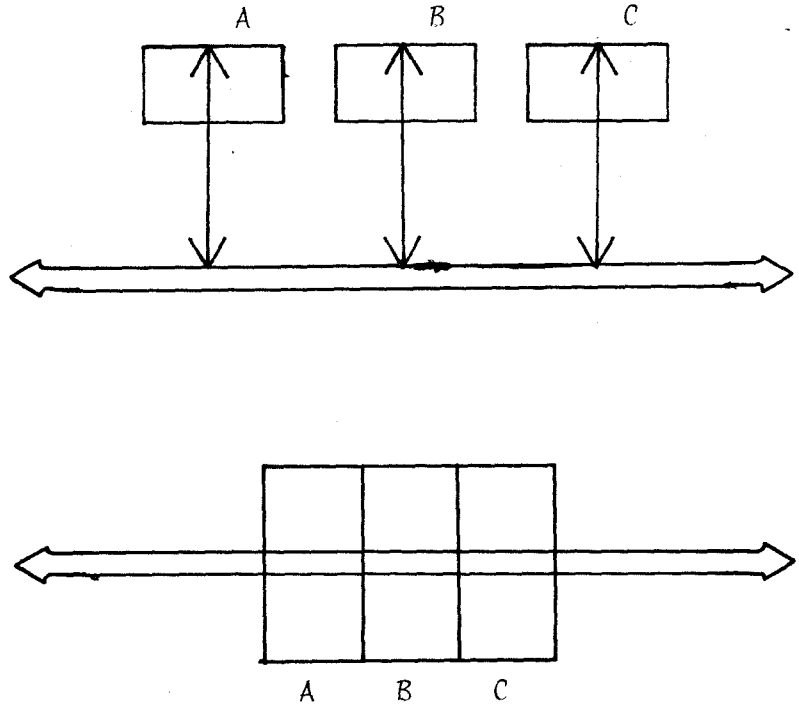


Figure 2.2 Included Bus

structure: PLAs ROMs and RAMs all have this property.

Occasionally, some extra wiring is needed to interconnect parts of a design. Often, VDD and GND from various parts need to be interconnected, or instances of cells need to be connected that are not matched geometrically. When that is the case, a wiring cell is usually used. The needed wires make up the definition of a separate wiring cell. By packaging them separately, the designer can localize groups of random wiring, making them relatively independent and easy to modify.

Included Buses

When drawing the logical block diagram of a system, one tends to draw boxes labeled REGISTER and ADDER and hook them up by drawing lines representing a whole bus between them. This simple drawing device represented very nicely what was going on with wire-wrap, stitch weld, and printed circuit technologies. In particular, the emphasis was on the boxes and not on the wires. They were intentionally de-emphasized in the notation.

In some design styles, the logical drawing notation seems to have been embraced whole hog. These design styles tend to have functional blocks interconnected by buses, using quite a bit of area for interconnect. Examples are the 8080 [Intel 1974], the Z8000 [Zilog 1978], and Standard Cells [Persky 1976].

Often, the functional blocks tap signals off the bus and then must communicate those signals within themselves. The structured design style recognizes this and proposes that

the buses be run through rather than around the functional blocks. The difference between these two styles is illustrated in Figure 2.2. Notice that if the three functions are designed to abut, there is no random interconnect at all. Including the buses in the functions is even likely to save area, since the bus is no longer tapped and extended.

Incidental Computation

One effect of expensive communication is the phenomenon of incidental computation. Most of the effort in structured design is spent on the communication flow. Once the communication plan of an algorithm is designed, filling in the operations is mostly a matter of throwing in a few transistors between passing wires. The real trick is in designing an algorithm that provides all the data in the same location ready to be combined.

The classic example is the barrel shifter in the OM data path. This function requires 32 input bits, a shift constant, and produces 16 output bits. The convenient way to bring in the shift constant is a 1 of N code on 16 wires. When all of these wires are brought into conjunction, with the 32 input lines brought in horizontally and over sections of 16 vertical wires, the 16 shift lines brought in vertically and run diagonally, and the 16 output lines running horizontally, it is simply a matter of putting in a transistor between a vertical input wire and a horizontal output wire controlled by a diagonal shift wire. The entire array has N^2 transistors in it, which fit completely underneath the wires needed to move the data and control. A transistor diagram of a 4-by-4

version of this barrel shifter is shown in Figure 2.3.

A key observation of structured design is that efficient use of the silicon comes from proper analysis of the communication of an algorithm.

2.3 Separating Out the Hierarchy

This section describes an extreme form of hierarchy that will be used throughout this thesis. The structured design style suggests, through the wiring cell technique, that a chip design is a mosaic of instances of cells, with no extra wires or extraneous geometric information involved. The extreme form of this technique is to completely separate the cell instance level of design from the wire/polygon level of design. This results in the separated hierarchy.

The extreme hierarchy will separate completely the leaf cells of the hierarchy from the composition cells. A leaf cell is the atomic unit, having no further internal structure. Composition cells contain only instances of cells interconnected, or composed, in some manner. Leaf cells can be instanced at any level in the hierarchy, even from the topmost cell. Only leaf cells have any "real" data in them.

Leaf Cells

A leaf cell might be as simple as a single transistor or even a section of wire, or it may be as complex as an entire PLA or multiplier. Leaf cells are only important for their function, not for their implementation. The

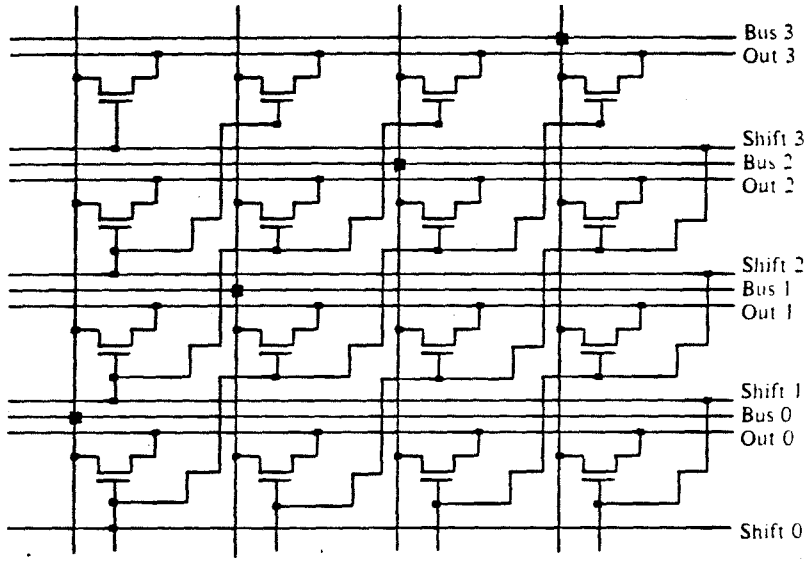


Figure 2.3 Incidental Computation in a Shifter

design of leaf cells is much better understood than is the design of large systems.

Leaf cells represent the basic semantic units of a design. They are the vocabulary the designer uses to express the function of the whole system. Alternatively, the leaf cells can be looked on as the initial cases of an inductive proof. Being the basic elements that make up the design, how the leaf cells are implemented is somewhat flexible. As will be discussed later, the ability to have multiple implementations of the leaves of a design has repercussions in consistency checking.

A leaf cell in a mask layout system that adhered to this hierarchical structure might contain any combination of rectangles, polygons, wires, or any other geometric structure. However, it may not contain instances of other cells.

Composition Cells

Composition cells specify how instances of other cells are interconnected. Any kind of cell can be instanced, whether leaf or composition, and both kinds of cell can be intermixed within one composition cell. Interconnections between instances are heavily restricted. Specifically, the interconnection mechanism must not, itself, introduce any new functionality to the composition cell.

Composition cells are inherently implementation independent. By allowing only instances of cells and their (functionless) interconnection, the composition cell cannot specify anything that depends on the implementation of the

leaves. If the leaf cells are the initial case of an inductive proof, then composition cells represent the general case.

In contrast, the mechanism that interconnects instances without introducing new functionality is implementation dependent. While the mechanism is not allowed to introduce any implementation "material", polygons for example, the concept of "connection" varies from implementation to implementation. For geometry, i.e. mask layout, interconnection between instances is accomplished by the intersection of two layout features (see Chapter 5 for more details on geometry interconnection). The interconnection mechanism is the analog of the induction step in an inductive proof. The interconnection mechanism will be termed a composition rule or composition algorithm.

An Observation

Composition cells are actually a representation independent design notation. Independent of how the leaf cells are implemented, if they are composed as specified in the composition cells in a hierarchy, they will perform the same function. Representation independence hinges on composition rules introducing no new functionality.

Given a description of a hierarchy, implementations for the leaf cells, and a composition rule, one could, in principle, produce a working system in that implementation domain. Given the hierarchical description for a system, many functionally identical implementations of it could be produced given only two things: an implementation of each leaf cell function and the composition rule for that type

of implementation. An illustration of the use of this organization is the production of the chip layout and simulator input from the same description. The functions of these two representations would be guaranteed identical IF the leaf cells were described consistently for each AND the two composition rules were correct.

A pleasing conclusion is that consistency checking reduces to the problem of checking the leaf cells, which can be amortized over all designs, and verifying only one rule, the composition rule. Notice that, with a reasonably large set of predefined cells and a composition rule for each implementation, a design that behaves identically in all implementations can be produced from only one description.

It should be noted that an assumption has been made that all implementations will fit in the same hierarchy. Classic design automation tools have generally assumed the opposite, that different representations of a design will have different hierarchical breakdowns. Allowing these different hierarchies makes consistency checking a knotty problem. Not only are there a large number of different representations to keep consistent, but each has its own hierarchical breakdown.

As an example, consider the difference between the "logical" and "physical" hierarchies. Many people would like to have the logical description of their chip be in a register transfer notation. The physical implementation of this same function might be a bit slice. The two hierarchies are just plain different, since the logical hierarchy is split by function first, then by bit while the physical one is by bit first, then by function.

If the leaves of the two hierarchies end up being the same, as in the above example, then consistency between the two implementations reduces to proving the two hierarchies identical and then using our original consistency checks. With some mathematics and cleverness, we may even be able to do the hierarchy equality check easily. However, if the leaves are different, and one implementation ends up with logic gates and the other with transistors, then consistency checking hinges on being able to describe the semantics of the leaves, a problem that is still unsolved.

Relating Back

The separated hierarchy certainly has the potential for satisfying all of the goals from section 2.1. Being a special case of the hierarchy, where cells above the leaves only compose, the separated hierarchy has the same benefits as hierarchies in general: complexity reduction, partitioning, sharability, and understandability.

Complexity reduction in each cell in the hierarchy is enhanced with the separated hierarchy. In the leaf cells, only the function that is being implemented need be considered since composition with other functions is handled by the composition rules. Composition cells can contain as much or as little complexity as desired. The data abstraction properties that are essential for composition rules also reduce the amount of information needed from other cells. This reduces the degree of interdependence between cells.

The other three properties, partitioning, understandability, and sharability, describe the separated hierarchy as well. Partitioning is enhanced by the functional abstraction properties that come along with the composition rules. Similarly, understandability is enhanced because there is less information required within a level of the hierarchy. Libraries depend on several properties. The functional abstraction enhances the ability of a designer to pick the appropriate library function. However, cells must be adaptable to be really sharable. As will be shown later, the generality that is needed to make composition rules work also enhances the adaptability of cells.

The separated hierarchy supports the key concept of the Caltech structured design methodology in that it stresses the communication between cells, rather than the computations. Composition cells really define the communication requirements between instances.

The geometric aspects of the structured design methodology are also well supported by the separated hierarchy. The concept of butting cells and floor plans is almost directly represented in the hierarchy through the lack of physical interconnection between instances of a composition cell. Since the logical interconnections cannot be represented by anything that adds some function, the superposition of connectors is a good way to implement that constraint. More detail on a geometry composition rule is developed in Chapter 4.

Chapter 3

The Mathematics of Hierarchies

The separated hierarchy developed in Chapter 2 lends itself to mathematical analysis. The ability to reason precisely about hierarchical designs will become more important as complexity increases. For that reason, this chapter is devoted to discussing ways of modeling hierarchies.

In the first section, a functional method for modeling digital systems will be discussed and related to the leaf cells of a hierarchy. By staying with a functional notation, the theory of Combinatory Logic [Curry 1958] is available to help in analysis.

Using Combinatory Logic, composition cells are modeled and separated from the leaf cells. After separation, the hierarchy is seen to behave like a mathematical operator. Some examples are given to help clarify the use of combinators.

The important question of equivalence between hierarchies is examined in some detail in Section 3.3. Establishing equivalence between two hierarchies will allow important consistency checking to be done.

The multiple-writer bus is a particularly troublesome structure for any functional model of a digital system. Section 3.4 discusses a possible model for buses that allows the "pieced" buses common in structured designs.

3.1 Functions, State, and Combinators

This section discusses a functional method for describing systems with internal state. Since this functional notation deals with digital systems having internal state, it can be used to describe any node in a hierarchy. The functional notation used is based on a precise mathematical base that inherits a large body of theory. One particular branch of functional analysis is presented that will be used to describe hierarchies.

Throughout this section, the lambda calculus [Church 1941] will be used as a functional notation. The lambda calculus is a convenient notation for expressing functional ideas because it is based on a notion of functional abstraction [Curry 1958 -- Chapter 3]. Also, lambda calculus is the basis for the theory of combinators that will be used to model hierarchies.

State

The lambda calculus provides a method for modeling systems having no internal state. The result of a function can only depend on constants and input arguments. In order to model the behavior of a memory cell, for instance, some modification is required. A system containing internal state can be represented as a function that is history dependent, i.e. the output of a memory cell depends on when

it was last written.

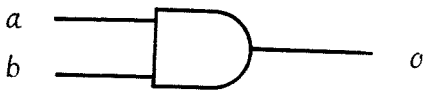
One method for dealing with history dependent functions, like memory cells, is to make the inputs include their whole history. By defining functions that operate on state sequences, rather than single valued inputs, memory cells and other systems with internal state can be modeled.

A state sequence is a possibly infinite sequence of values that represent the entire history of that input or output. A sequence represents an event sequence. The "sequence step" could represent the basic clock cycle of a synchronous system, where succeeding elements in the sequence represent the value during successive clock periods. In a similar way, the elements in a sequence might represent successive states in a self-timed system [Seitz 1979] that had no basic clock. Whether there is a constant length of time represented by each element in a sequence or not, the important characteristic is that a state history is represented, not a time history.

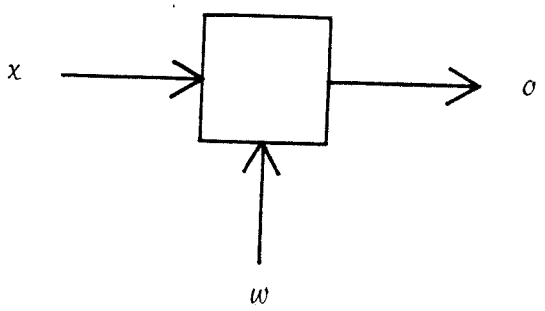
Two examples of functions over state sequences are illustrated in Figure 3.1. The top figure represents a 2-input AND gate. Having no internal state, the expression for its output sequence need refer only to the "current" inputs. The output sequence elements can be generated by the following sequence generator:

$$o_i = a_i \text{ AND } b_i$$

Shown in the lower half of Figure 3.1 is one bit of a typical storage register. The output of the storage register should be the same as the input when w was last a "1". The time history of the register might look something



2-input AND gate



Storage Register

Figure 3.1

like the following sequence.

x = (... 0 1 0 0 1 1 0 0 1 1 0 ...)
w = (... 0 0 0 1 0 0 0 0 1 0 0 ...)
o = (... 1 1 1 1 0 0 0 0 0 1 1 ...)

In the absence of a "write command", a "1" in the w sequence, the output sequence remains unchanged in time. The output sequence value changes to the previous value of the input sequence during the sequence step after w is a 1. A sequence generator for the storage register is:

o_i = IF w_{i-1}=1
THEN x_{i-1}
ELSE o_{i-1}

Again, the sequence steps could represent the clock cycle in a synchronous system, as in the OM processor, or they could directly represent the sequencing of a self timed system.

Sequences are not a terribly natural notation. In particular it would seem more natural to have the internal state of the cell be explicitly called out and brought in as an extra input. However, bringing out the internal state as a feedback term introduces some unnecessary information, namely the exact form of the internal state, in the notation. With state sequences, nothing need be known about the internal structure. [Nerode 1958]

Leaves

Leaf functions in the separated hierarchy represent the primitive functions for a given system. As such, they need not be described more fully. In order to analyze what a

particular digital system does, the semantics of its leaf, or primitive, functions must be precisely defined. However, this thesis is concerned more with modeling the composition, rather than the system.

For the purposes of analyzing the hierarchy, the functions of the leaf cells need not be precisely defined. As will be seen later, the separation of leaves and composition implies that analysis of the composition cells does not depend on the leaf functions. Since no knowledge of the leaf cell functions will be required, expressing them precisely will not be needed. All that need be said is that leaf functions that map state sequences into state sequences are sufficiently powerful to represent the leaf cells used in VLSI design, by virtue of their ability to model internal state.

Combinators

Combinatory logic [Curry 1958] deals with operators that perform mappings on the function space, i.e. operators whose result is a function. The theory is based on the lambda calculus and is equally powerful. The operators that map from function space to function space can all be expressed in terms of two primitive operators.

In this section combinators will be introduced and related to several common functional notations. Starting with a garden variety functional notation, the lambda calculus will be introduced. The same function will also be written in ALGOL to illustrate different types of "variable binding." Finally, combinators will be introduced and contrasted with the lambda calculus.

Combinatory logic is a notation that is equivalent to the lambda calculus. The two notations are equal in power, differing only in how "variables" are "bound." The lambda calculus is a formal notation that deals with the "binding" of variables into expressions. Take, for example, the quadratic function:

$$F(x) = x^2+2x+1$$

The lambda calculus version of this function could be written as

$$F = \lambda x . x^2+2x+1$$

This notation makes explicit the declarations of variables used to define the function. The variables are declared between the lambda and the dot, with the expression defining the function following the dot. (This syntax is not the only one used for lambda calculus expressions, but it is the one that will be used uniformly throughout this thesis.) The general form for a lambda calculus function might be written as follows.

⟨function⟩ ::= "λ" ⟨variables⟩ "." ⟨expression⟩

Within the expression, further occurances of a declared variable, like "x", are bound to the same value. Thus, invoking the function F with value x=-3 will cause the following bindings to be made:

$$\begin{aligned} F(-3) &= (\lambda x . x^2+2x+1)(-3) \\ &= (-3)^2+2(-3)+1 \\ &= 4 \end{aligned}$$

Variables are not restricted to numerical values and can even represent a lambda calculus function. The LISP

programming language is based on a notation of this kind [McCarthy 1965]. The binding schemes in a programming language like ALGOL [Naur 1963] are very similar to the lambda calculus schemes. An ALGOL procedure to calculate the function F might be written as:

```
real procedure F(x); real x;
F:= x**2 + 2*x + 1;
```

When this procedure is invoked, the actual value of the parameter x is substitued wherever the formal variable x is used.

- Combinatory logic takes a different approach to variable binding. In particular, the use of combinators eliminates the concept of variables altogether. The implicit binding process represented by the declaration of variables in a lambda expression is replaced by an explicit functional description of the binding in combinatory logic. Combinators rely on the concept of currying and on some simple primitive functions to perform the binding function.

Currying is a method of expressing functions of several variables in terms of a series of functions of one variable, each of which returns a new function as its value. To illustrate this concept, consider the addition function, ADD(x,y). As it stands, the addition function requires two numeric arguments and returns a numeric result. Equivalently, the addition could be defined as

```
plus x y  =  x+y
```

The plus operator is defined to take an argument, say x, and return another function that adds "x" to its argument, y in this case. Thus the piece of equation "plus 2" has a

perfectly valid meaning: it is the function that adds 2 to things. (Expressions in combinatory logic are left associative, i.e. they are read from left to right. Thus the expression "plus x y" is actually read as "((plus x) y)".) The function "plus" is a curried version of the addition operator.

By the introduction of two operators, the full power of lambda calculus variable binding is available in a combinatory logic expression. These two operators are S and K, and can be described in lambda calculus as follows.

$$Sxyz = \lambda xyz.xz(yz)$$
$$Kxy = \lambda xy.x$$

Several things need to be said about these two equations. First, this seems to be a rather mixed notation with curried functions, or combinators, on the left side and a lambda notation on the right side. It turns out that since the two notations, lambda calculus and combinatory logic, are equivalent, they can be expressed in terms of each other. In fact, the two notations can be combined in the same expression. The examples discussed in this thesis will be introduced in a lambda calculus form, and then gradually transformed into a combinatory logic form. The transformation, as illustrated in section 3.2, consists of introducing combinators into a lambda calculus expression until all of the variables are moved to the right end in the same order as in the declaration. The residue, that between the dot and the variables piled up at the right end, can be removed and dealt with as a combinator.

The two primitive combinators defined above are enough to provide the full power of lambda calculus binding. By

judicious use of the S operator, arguments can be copied and functions can be composed. The K operator is used to remove dependence on an argument.

In a sense, these two operators can be viewed as merely massaging the text description of the arguments. Neither the S nor the K operator actually performs any arithmetic function, it merely rearranges the order and composition of other functions. S and K represent the essence of combination and composition.

While all combinators can be expressed in terms of S and K, there are some other convenient operators, as defined below.

$$I = \lambda x.x$$
$$= SKK$$

$$B = \lambda xyz.x(yz)$$
$$= S(KS)K$$

$$C = \lambda xyz.xzy$$
$$= S(BBS)(KK)$$
$$= S((S(KS)K)(S(KS)K)S)(KK)$$

The I operator is the identity. The B operator is useful for moving variables and functions out of parenthesis. It is a form of primitive association. The C operator performs a primitive permutation on its arguments.

As you can see, combinators can be very long and tedious when expressed in S and K operators. This is alleviated very little by the addition of I, B, and C. The combinator notation is not meant to be a programming language for expressing composition. It is instead a formal tool that

can be used to put composition on a more precise footing.

3.2 Modeling Hierarchy

The hierarchy as described in Chapter 2 matches with the combinator notation rather nicely. Combinators were meant to compose functions into other functions. As such, they should be a convenient notation for use in describing the essence of a hierarchy, composition. Some examples illustrate the process of separating the hierarchy from the leaf functions.

The Hierarchy as Combinator

A digital system designed consistently with the separated hierarchy of Chapter 2 has two kinds of cells: leaf cells and composition cells. As discussed in the previous section, characterizing composition cells can proceed quite independently from the functions performed in the leaves. The separation between leaf and composition cells is what allows this characterization.

The combinatory logic discussed in the previous section seems to describe nicely what happens in a composition cell. Remember that composition cells are not allowed functional interconnects; the act of composition is purely that of identifying which outputs plug into which inputs. Since combinators are very good at rearranging arguments and functions, some way should be made to model a hierarchy, which is just a composition of leaves, into a combinator.

By equating combinator and hierarchy, a very simple view of a digital system is reached. The hierarchy combinator expects as inputs a list of leaf functions. When these leaves are composed as specified in the combinator, the result is the specified digital system. Therefore, digital systems can be represented, where H is the hierarchy, as:

H⟨leaves⟩ => ⟨systems⟩
⟨systems⟩⟨inputs⟩ => ⟨outputs⟩

H provides no function but to compose the leaf cells, now arguments to this hierarchy operator, into interesting systems. Thus, the hierarchy H, which is composed only of combinator primitives like S and K, takes the leaf functions as arguments and produces a "system". This system maps from state sequences to state sequences.

As the examples will illustrate, in order to separate the hierarchy from the leaves, the leaf functions must be "moved out as arguments." By appropriate introduction of combinators, the inputs and leaf functions can be slowly shifted to the right end of the expression describing the system. This will result in an expression with only combinator primitives, on the left, then a group of leaf functions, and finally the inputs and outputs on the right. Since the expressions are left associative, the combinator primitives can be considered separately as the operator that represents the hierarchy.

Combinators, and hence hierarchies, map from the function space back into the function space. The first level of compositions map from the set of leaf functions, F, into the set of hierarchical functions, H. Every level higher than the lowest maps from the set of hierarchical functions

into itself.

$$H \cong [H \rightarrow H] + [F \rightarrow H]$$

For mathematical solutions to such recursive domain equations see [Scott 1975].

An Example

As a simple example, we will write down the combinator for the 3-input AND gate example shown in Figure 3.2. The circuit contains two 2-input ANDs, whose function can be written as

$$A_2 = \lambda ab.ANDab$$

There are three inputs and one output. A lambda calculus version of the 3-input AND gate function is as follows.

$$A_3 = \lambda abc.A_2(A_2ab)c$$

Developing a combinator to represent this composition is like a game whose object is to move all the arguments to the right end of the expression. The function at the beginning can then be removed and dealt with separately. Doing that with the 3-input AND gate might be as follows.

$$\begin{aligned} A_3 &= A_2(A_2ab)\ c \\ &= BA_2(A_2a)\ bc \\ &= B(BA_2)A_2\ abc \end{aligned}$$

So far, we have removed the state sequences to the right. These can then be removed from consideration while we attempt to remove the 2 input AND gates and get down to the real hierarchy combinator.

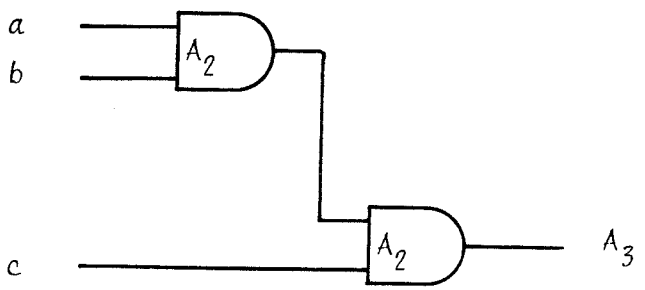


Figure 3.2 3-input AND gate

$$A_3 = B(BA_2)A_2$$
$$= BBBA_2A_2$$
$$= BBBA_2(IA_2)$$
$$= S(BBB)I\ A_2$$

Thus, the composition of two 2 input AND gates into a 3 input AND gate is represented by the combinator S(BBB)I. This combinator also represents the composition of any other 2 input, one output function into a 3 input, single output function. The combinator describing this hierarchy just represents the composition, not the function.

Feedback

Another example, illustrating the use of combinators to model feedback, is a synchronous toggle flipflop. This function, illustrated in Figure 3.3, takes one input, t, and produces one output, f. As long as the input sequence is inactive, i.e. t_i=0, the output sequence remains unchanged from its last value. If the input sequence is active, t_i=1, then the next output state will be inverted from its previous value. This action is shown by the state sequences below.

t = (... 0 0 0 1 0 0 1 1 1 0 0 ...)
f = (... 1 1 1 1 0 0 0 1 0 1 1 ...)

The toggle flipflop will be built using two leaf functions, a delay leaf with function D, and an exclusive-or leaf referred to as E. The delay function, D, is characterized by having its output sequence delayed one sequence step from its input. The output sequence from E is a "1" if exactly one of its inputs is a "1" and "0" otherwise. In terms of these primitives, the function F, for flipflop, can almost be written as below.

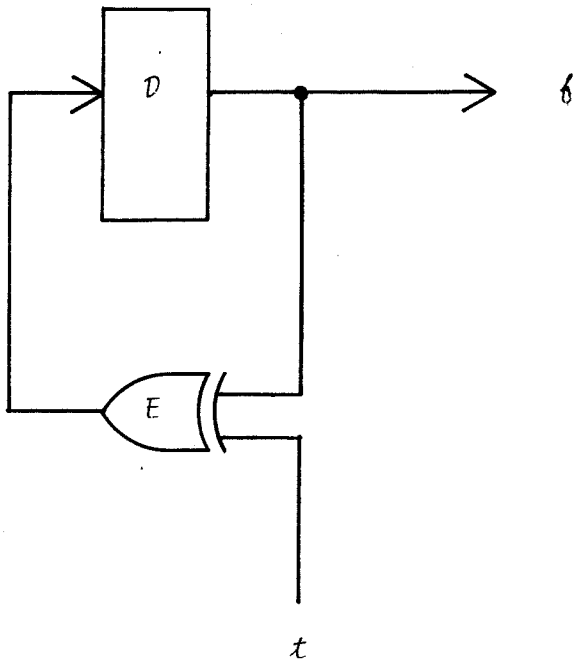


Figure 3.3 Toggle Flipflop

$$F = \lambda t.D(EtF)$$

Unfortunately, this is a recursive function definition. The function F is used to define itself. This really means that there is a feedback loop in the circuit, as is obvious in Figure 3.3. Luckily, there is a body of theory that deals with just this sort of equation: fixpoint theory. The solution to the recursive equation describing F is called the fixed point of the function D(Et). There is a fixed point operator that represents that solution: the fixed point combinator, Y.

The operator Yx represents a possibly infinite string of recursive "calls" on the function x. The operator Yx is a function that remains unchanged by the function x. Thus the following equalities all hold.

$$Yx = x(Yx) = x(x(Yx)) = \ldots$$

The Y operator is the combinator analog of recursion in standard programming languages. The Y operator can be written in terms of S and K, the basic combinators. The S and K definition, along with a slightly more understandable lambda calculus definition, is:

$$\begin{aligned} Y &= SSI(SB(K(SSI))) \\ &= SS(SKK)(S(S(KS)K)(K(SS(SKK)))) \\ &= (\lambda uv.uv(uv))(\lambda yz.y(zz)) \end{aligned}$$

For more detail about the Y operator, see [Curry 1958] and [Burge 1975].

Using the Y operator, we get the following solution for F.

$$\begin{aligned} F &= \lambda t.Y(D(Et)) \\ &= \lambda t.BY(BDE)\ t \end{aligned}$$

dropping the argument, t

$= B(BY)(BD)\ E$

$= B(B(BY))B\ DE$

$H_F = B(B(BY))B$

Thus, the hierarchy that represents the basic feedback loop as diagrammed in Figure 3.3 is B(B(BY))B. The properties of the Y operator will become more important when hierarchy equivalence is discussed in Section 3.3.

A CounterExample

To illustrate more completely how to express the hierarchy via combinators let's examine a slightly more complicated example: a two-bit ripple carry counter. The counter, represented as a one level hierarchy, is composed of two toggle flipflops and two 2-input AND gates, as in Figure 3.4.

A convenient sequence step for this system matches the transitions on the counter input, t. Since the entire counter is run by the input transitions, and nothing interesting happens between transitions, that sequence step will model everything interesting about the counter. (Notice that the input sequence, t, reduces to an infinite sequence of alternating 1's and 0's.)

The leaf functions for this hierarchy are A and F, the AND and Flipflop functions. The sequence generator for the AND gate, A, was presented in the first section of this chapter. The flipflop sequence generator is similar to that of the register presented in the first section. A toggle flipflop should invert its value after every zero to one transition of its input. A sequence generator that models this activity is as follows.

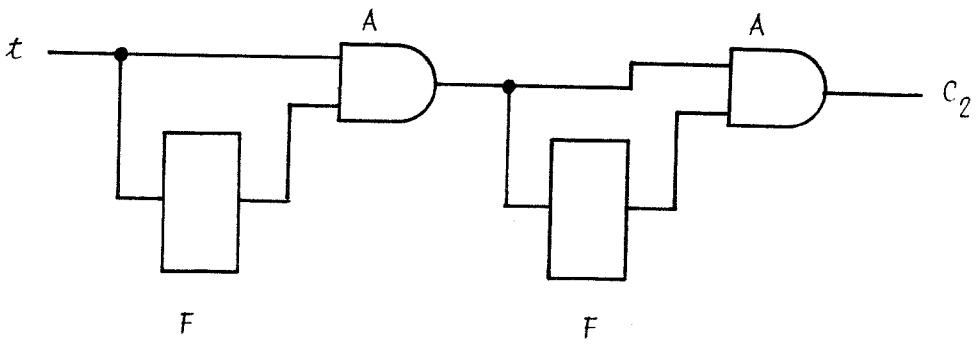


Figure 3.4 A 2-bit Counter

f_i = IF x_{i-1}=1
THEN NOT f_{i-1}
ELSE f_{i-1}

This sequence generator results in the state sequences below:

x = (... 0 1 0 1 0 1 0 1 0 1 0 1 ...)
f = (... 1 1 0 0 1 1 0 0 1 1 0 0 ...)

A lambda calculus description of the counter in terms of the AND gates and flipflops might be

$$C_2 = \lambda t.A(At(Ft))(F(At(Ft)))$$

Separating the hierarchy for this implementation of the counter can be done in two steps. The first step is to separate out the variable "t". After that is moved to the right, the rest of the expression can be grouped, because of the left association, and worked with as an operator. The second step separates out the leaf functions, A and F. After these are moved to the right, what is left on the left is the combinator representing this hierarchy.

In the equations that follow, the first step, that of removing the variable "t", is done. The added combinators are underlined.

$$C_2 = \lambda t.A(\underline{SAF}t)(F(\underline{SAF}t))$$
$$= \lambda t.\underline{SAF}(SAFt)$$
$$= \lambda t.\underline{B}(SAF)(SAF)\ t$$

Now that the input has been moved to the right, the rest of the expression can be treated separately as an operator that will work on any single input. To further distill this operator, the leaf functions A and F can be removed.

C_2 = B(SAF)(SAF)

= B(SAF)(I(SAF))

= SBI(SAF)

= SBI(SAF) = (SBI)(SAF)

= B(SBI)(SA) F

= B(B(SBI))S AF

Removing the leaf functions leaves a combinator built from the primitive combinators only. This combinator represents that "hierarchy" for the two bit counter.

H_C = B(B(SBI))S

Another way of building this two-bit counter is to build it out of two one-bit counters. Each one-bit counter in the Figure 3.5 is represented by the cell with function C. In the following mathematical representation of this implementation, the combinators are underlined as before.

C = λs.As(Fs)

= λs.SAFs

C_2 = λt.C(Ct)

= λt.BCCt

In order to derive the actual hierarchy from this pair of functions, it is necessary to substitute in C and solve for a combinator that does not include the leaf functions, A and F. We will solve for H_C again, the combinator that represents the full hierarchy of this 2-bit counter.

C_2 = BCC

= B(SAF)(SAF)

= B(SAF)(I(SAF))

= B(SBI)(SA) F

= B(B(SBI))S AF

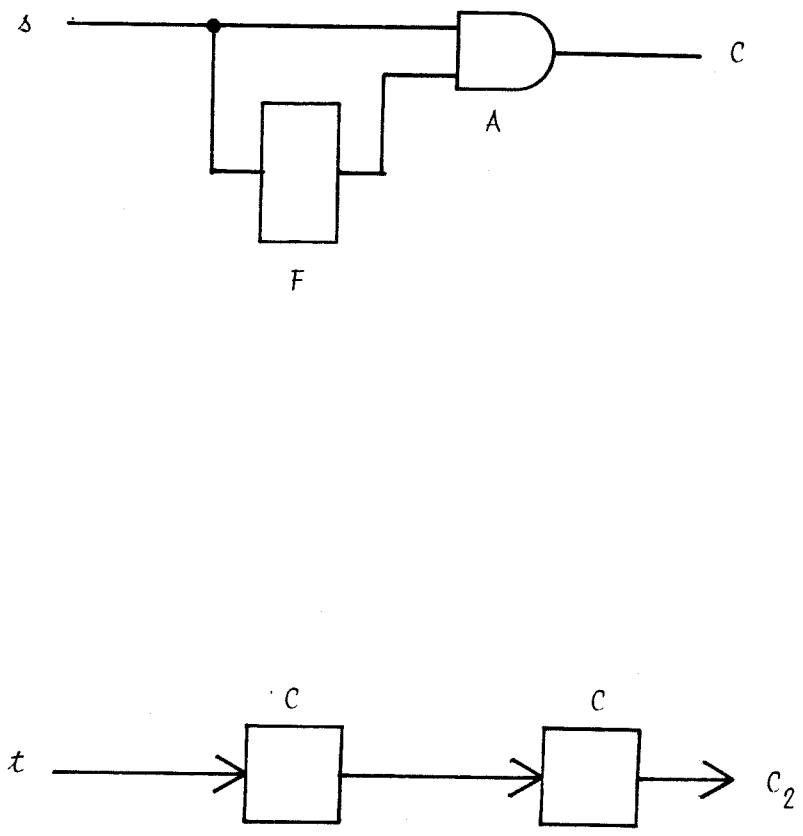


Figure 3.5 A Different 2-bit Counter

$H_C = B(B(SBI))S$

Notice that the combinator representing the first implementation of this counter is exactly identical to the second. Of course that was no accident. This example was used to illustrate the manipulative properties of combinators. The fact that two different hierarchical descriptions of the same functions could be shown to be identical through "algebraic" manipulations is one of the reasons why combinators are a powerful representation tool.

The ability to manipulate hierarchies algebraically to show two hierarchies identical was shown to be important for consistency checking, as indicated in Chapter 2, especially between different representations. Being able to design in different representations using a different hierarchical breakdown, but to still be able to prove the two designs identical, is a powerful tool for the designer. A more complete treatment of hierarchical equivalence is the subject of the next section.

3.3 Equivalence Between Hierarchies

There is no single, best hierarchical breakdown for a complex system. No two people will produce the same hierarchy. In fact, the same person is apt to want different hierarchical decompositions for different purposes. The chip layout might be most easily accomplished as a bit slice, while the functional simulation would be more efficient and understandable in a register transfer representation.

As was mentioned in Chapter 2, given two hierarchies that are built on the same leaf cells, consistency checking reduces to showing the two hierarchies equivalent. This section is dedicated to exploring the concept of

equivalence between hierarchies.

Types of Hierarchical Equivalence

Many different criteria can be used to judge two systems equivalent. Different criteria result in different classes of systems being equivalent. The difficulty here is to pick an appropriate method for judging two hierarchies equivalent that will fit nicely with the separated hierarchy. All of these methods for determining equivalence will assume that the hierarchies under question build from the same pool of leaf cells, as was discussed in section 2.3.

Three equivalence criteria will be discussed. Named identical, topological, and functional equivalence, the criteria can be ordered by "strictness." Criterion A is more strict than B if A implies B but not B implies A. In other words, systems that are A-equivalent are also always B-equivalent, but not vice versa. The criteria discussed here are ordered in this way: identical is more strict than topological which is more strict than functional. The discussion will move from identical, through functional and on to topological equivalence.

A definition for identical equivalence reads as follows:

> Identical: Two systems are equivalent only if every cell in every level of the hierarchical tree composes the same instances of the same cells with the same interconnection.

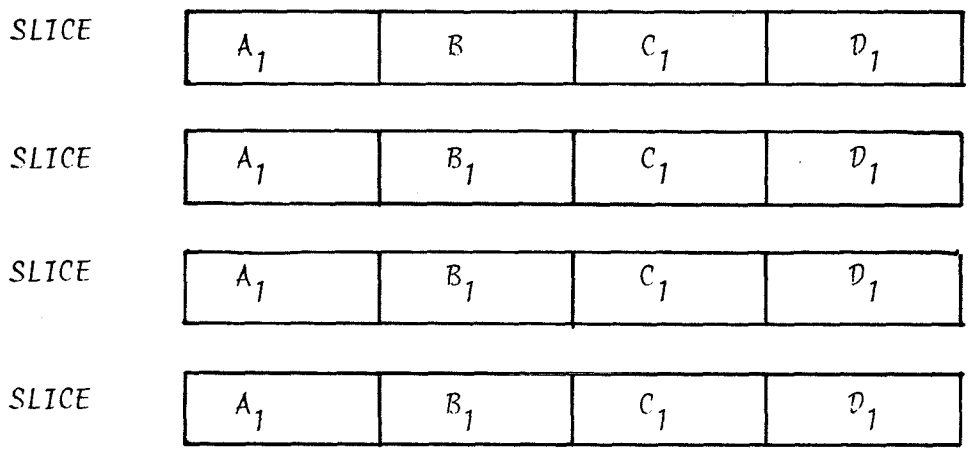
This is the strictest definition of equivalence because an exact match is required. Checking for identical equivalence reduces to verifying that the corresponding composition cells are described by "equal" combinators.

The issue of equality between combinators is discussed below. The consistency checking of Chapter 2 would be restricted to representations that map into identical hierarchies, however. The restriction of a single hierarchy prevents potentially useful representations, such as a functional slice representation for simulation illustrated in Figure 3.6. Unless nothing better comes up, this is an unsatisfactory equivalence criterion.

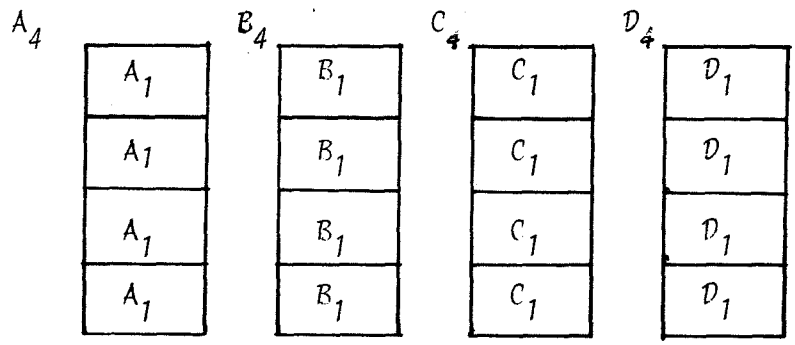
Functional equivalence is the least strict criterion requiring only that the two systems have the same function.

> Functional: Two systems are equivalent if, given the same input sequences, they produce identical output sequences.

Using functional equivalence would allow the functional slice representation of Figure 3.6 since the two systems performed the same transformation from input sequences to output sequences. However, this criterion may be too unrestrictive: it attempts to guarantee equivalence between systems that the philosophy inherent to the separated hierarchy disallows. An example of two systems that are functionally equivalent is shown in Figure 3.7. The two systems are equivalent becuase of the properties of the instance labeled A. This instance performs no useful function and so removing it, as was done in the leftmost system, does not effect the functional equivalence of the two systems. However, the functional equivalence of these two systems depends on the semantics of the leaves. This thesis is based on a system description that is independent of the semantics of the leaves. To base equivalence on a criterion that includes the properties of the leaves is inconsistent.



Bit Slice Organization



Functional Slice Organization

Figure 3.6

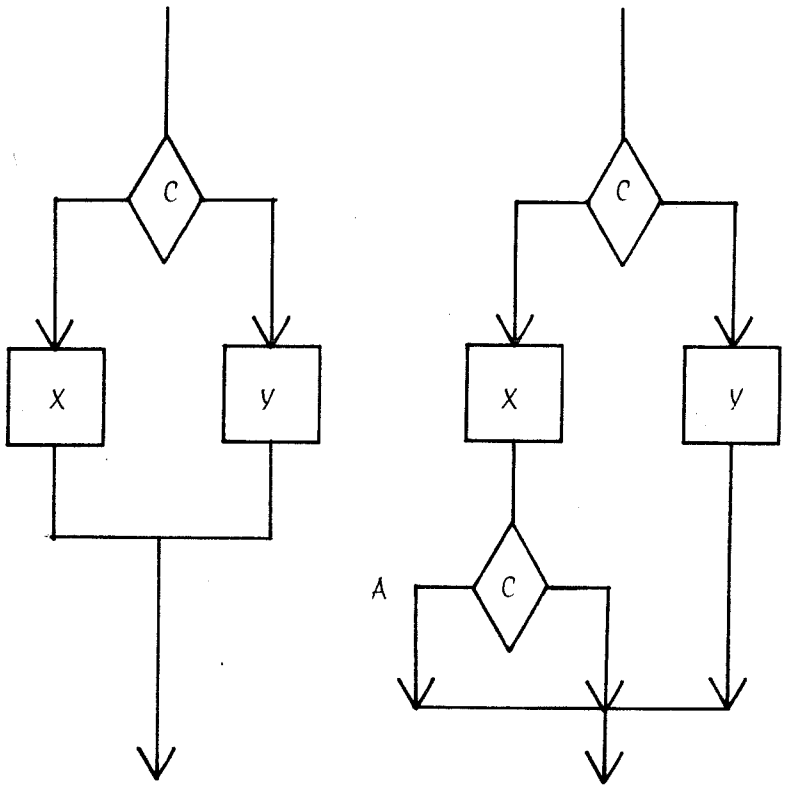


Figure 3.7 Functionally Equivalent Systems

The last method for determining equivalence between systems is topological.

> Topological: Two systems are equivalent if the hierarchies that define them result in identical compositions of the identical leaves.

This definition may warrant some explanation. Topological equivalence between two systems implies that, if the two hierarchies that describe them are "flattened out", i.e. all instances of composition cells are replaced by copies of their definitions, the result will be the same interconnections of the same leaf cells. In a sense, this proposes a normal form for a hierarchical description that has only one composition cell that describes how the leaf cells are interconnected. Hierarchies are then equivalent if they reduce to the same normal form. Figure 3.8 illustrates the "flattening" process and the resulting equivalence test. This equivalence criterion is termed "topological" because the final test for equivalence is one of comparing the topologies of two graphs. The topological equivalence criterion would call the two systems of Figure 3.6 equivalent but not the two systems of Figure 3.7.

Topological equivalence would be the first choice for an equivalence criterion. Its feasibility rests on some combinatory logic results since using the topological equivalence criterion is the same as requiring that the two combinators representing the systems be "equal". This can easily be shown by noticing that "running" a combinator, i.e. applying it to a set of leaves, results in a composition of leaf cells. This composition of leaf cells is the normal form discussed above. The notion of combinator equality will be discussed in the next section.

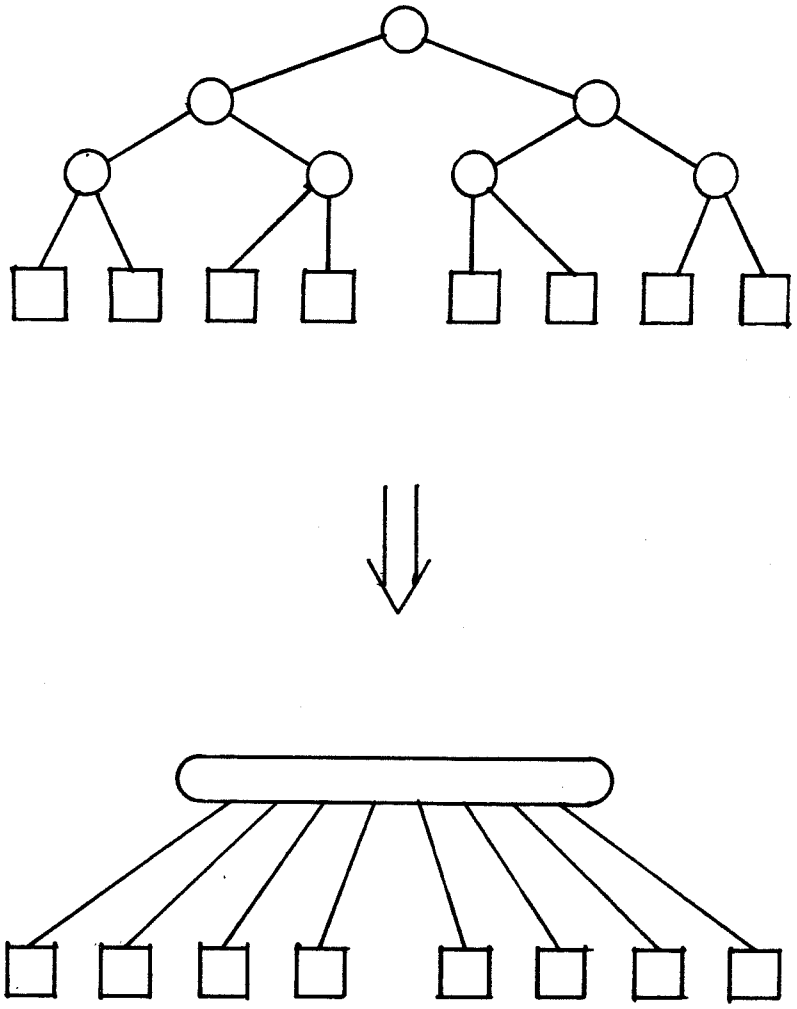


Figure 3.8 Flattening the Hierarchy

The Y Operator and Turing Machines

Unfortunately, equality between combinators is not an easily decided issue. Results from combinator theory show combinators equal in power to Turing machines [Curry 1958]. One of the basic results from the study of Turing machines is that the problem of showing two programs equal is undecidable [Rogers 1967]. Thus, since the combinator notation used to describe digital systems is as powerful as Turing machines, equality between digital systems is an undecidable issue. Notice that this applies to both identical equivalence and topological equivalence.

The reason for equality being an undecidable issue is the Y operator. Being a recursive operator, there is no way to decide whether a hierarchy described using the Y operator will terminate when run to find the normal form. (Note that preventing the use of the Y operator does not eliminate the problem since the Y operator can be defined in terms of S and K operators, and they are the necessary primitives.) The Y operator not terminating would imply that an infinite hierarchy had been defined, a meaningless concept in terms of an implementable VLSI system. However, results from Blum [Blum 1967] suggest that the ability to express infinite objects, while not useful in itself, is an essential quality for any useful notation. Blum's conclusion is that, by restricting the notation to prevent infinite recursion, i.e. by limiting the use of the Y operator, the notation may be much less suited for describing finite problems.

The Horns of a Dilemma

The conclusion reached from this discussion of hierarchy equivalence is that it is an undecidable question. That is an unsatisfying conclusion. In retrospect it seems like an obvious conclusion, however. It is clear that at some point, a system will be developed that allows the full power of a programming language (read Turing machine) for describing composition cells. In fact, the geometry composition algorithm presented in this thesis is such a system. Once the desirability of that step is granted, then it is clear that the algorithmic definition of a composition cell will automatically make equivalence an undecidable issue.

However, even though a chip is described through the use of a general purpose programming language, it is still a finite object. Being finite, it can be described without the full power of a programming language, i.e. without recursions and WHILE loops. The finite description of a chip is the output of the programming language description, it is the result of "running" a program. While there is no way to guarantee that the program describing the chip will stop, if it does stop the finite description can be checked. In this way, two algorithmic descriptions of a chip can be checked for equivalence: by generating specific, finite descriptions and checking those.

Combinators are clearly as powerful as any programming language and can be used to notate an algorithmically described chip. However, combinators can also describe chips in a finite way. A combinator that is in its "normal form" is one that can be compared to other normal form

combinators. The normal form for a combinator is defined to be in a state where no "reductions" are possible. A reduction can be performed whenever a primitive combinator has enough arguments to be performed, i.e. whenever the K operator has two arguments, the operator and its operators can be replaced by the first argument. When a combinator has been reduced to normal form, it is a finite description: it can be compared with other normal form combinators for equivalence. The comparison step is strictly syntax: two combinators that reduce to normal form are equal if and only if they reduce to the same normal form [Curry 1959 -- The Church-Rosser Theorem]. Thus, given two combinators in normal form, equivalence checking is a simple text string comparison for equality.

3.4 Modeling Buses

In this section, some of the subtleties of buses spread throughout a hierarchy are examined. Buses are to VLSI what shared memory and critical sections are to multiprogramming, with similar problems of synchronization. To emphasize the similarity between buses and classic synchronization problems, I have chosen to call a bus driver a writer and a bus receiver a reader. The bus is a resource shared between many contending processes and so relates closely to the classic synchronization problem.

Buses also pose a problem for any functional notation because of multiple readers and writers. Functional notations do not naturally express constructs that connect together outputs of functions. Some method is needed to map buses into the functional domain so that they can be analyzed.

The Problem

To this point, interconnections between instances of cells have been strictly directed, one instance output values to an input of another instance. This has led to a functional notation that seems to describe fairly well what is happening in the hierarchy. The problem is that most chips are not designed this way. Bus structures are rampant in computer design; whole companies are formed to build devices compatible with one type of bus.

Buses are still common in VLSI, due to the ease with which many technologies form buses. The OM processor designed at Caltech has two independent buses running completely across and off the chip. Bus drivers tend to be simple in terms of number of transistors, two or three transistors in the OM, if not easy to design from an electrical point of view.

In the structured design style, and in the separated hierarchy, buses tend to be cut into pieces, a short piece running right through any cell having access. The cells that include a piece of the bus may do nothing but pass it on to a neighbor, or they may compute a new value to be conditionally written onto the bus. Control of who writes on the bus generally comes from above in the hierarchy, from a PLA or some other central control. A collision on the bus, two sources trying to write on the bus at the same time, can cause a major disaster. The collisions can be data dependent and therefore difficult to debug.

With their prevalence and potential for hard to find bugs, some way of notating and of guaranteeing good behavior for

buses needs to be found. Included buses pose a special problem in a hierarchy since the control tends to be spread between levels of the hierarchy. Some way has to be found to guarantee good bus behavior without needing to examine the entire hierarchy.

Before examining a general bus, the least complex bus, having only two potential writers, will be examined.

Two Writers

A physically inaccurate, although mathematically and logically correct, method for illustrating a two writer bus is through the use of multiplexors. If write enabled, each writer selects his own new value, and if not write enabled the writer selects the output of his neighbor. In this way, when not writing, the bus writer is actively reinforcing what should be on the bus. (This reinforcement is actually never done, physically.) A schematic drawing of this multiplexor model is shown in Figure 3.9.

The two bus wires are physically the same piece of wire in the real layout as illustrated in Figure 3.10, but they will be kept separate here for purposes of discussion. By keeping them separate, the outputs of the multiplexors are not interconnected, and functional assumptions remain intact. Examining the functions that represent each bus we get

$$B_1 = Ms_1 i_1 B_2$$
$$B_2 = Ms_2 i_2 B_1$$

Breaking the feedback by introducing the intermediate value o, as in Figure 3.11, we get

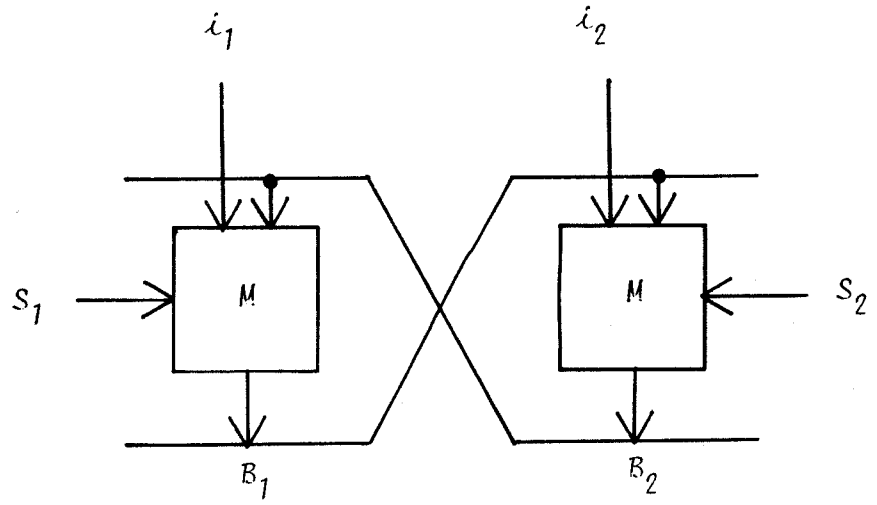


Figure 3.9 Two Writers

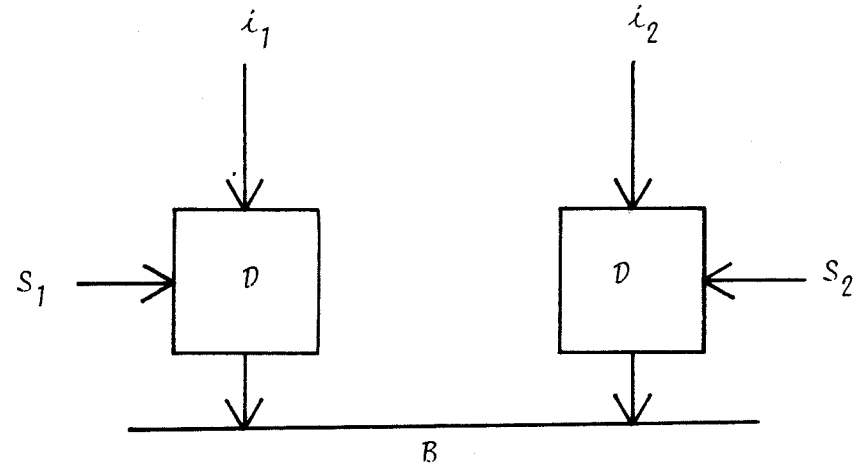


Figure 3.10 Physical Bus

$$B_1 = Ms_1 i_1 (Ms_2 i_2 o)$$
$$= B(Ms_1 i_1)(Ms_2 i_2)\ o$$

Since B_1 must be equal to o, it is a fixed point of the right hand side. The fixed point operator, Y, computes just that, giving this result for B_1:

$$B_1 = Y(B(Ms_1 i_1)(Ms_2 i_2))$$
$$= B(BY)B\ (Ms_1 i_1)(Ms_2 i_2)$$

and, by symmetry,

$$B_2 = B(BY)B\ (Ms_2 i_2)(Ms_1 i_1)$$

Since the two buses will be physically the same piece of wire, the functions describing them must be equal or our model fails. The equality between B_1 and B_2 is maintained if the functions operated on by the B(BY)B combinators are equal.

$$Ms_1 i_1 (Ms_2 i_2) = Ms_2 i_2 (Ms_1 i_1)$$

From the definition of M, as a selector, it is clear that if s_1 does not equal s_2, then these two functions are equal independent of the values of i_1 and i_2. If the two selectors are not guaranteed to be unequal, then the two input values must be equal when the selectors are not.

This mutual exclusion ensures that the two multiplexors will have the same output value. It also models the obvious conclusion that multiple writers on a bus should not write on the bus at the same time.

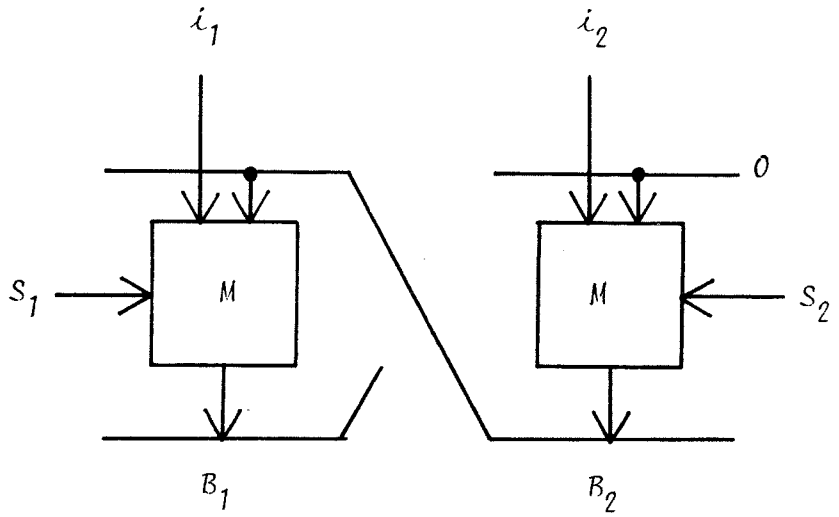


Figure 3.11 Breaking the Feedback

When both multiplexors are selecting the bus, the bus value is undefined. This state is not allowed and can be fixed by having another source on the bus that drives it to a known state when neither selector is enabled.

N Writers

We extend the bus notation to N writers on the bus by connecting the writers in a ring as shown in Figure 3.12. This ring formation is a dual of the actual physical formation in Figure 3.13.

The conclusions drawn from the study of the 2 writer case can be generalized for N writers very easily. Each equation from the 2 writer case is generalized by replacing the 2 and 1 with i and i-1, unless i=1 when 2 and 1 represent 1 and N respectively. Each bus output must be equal to keep the model consistent with reality which leads to the conclusion that exactly one multiplexor must be selecting its input rather than the previous bus value.

By including an "extra" bus writer that is enabled by the NOR of all the other select inputs, as shown in Figure 3.14, the value of the bus is always defined. The extra writer allows this model to handle precharged buses. A precharged bus is set to a known state during an idle time. A bus writer need only modify the bus if it needs to transmit a value different than the precharged value. This type of bus is common in nMOS because of the asymmetric speeds of transmitting a 1 versus transmitting a 0. If no writers are selected to transmit over a precharged bus, the value is the known state. The extra writer can model this behavior.

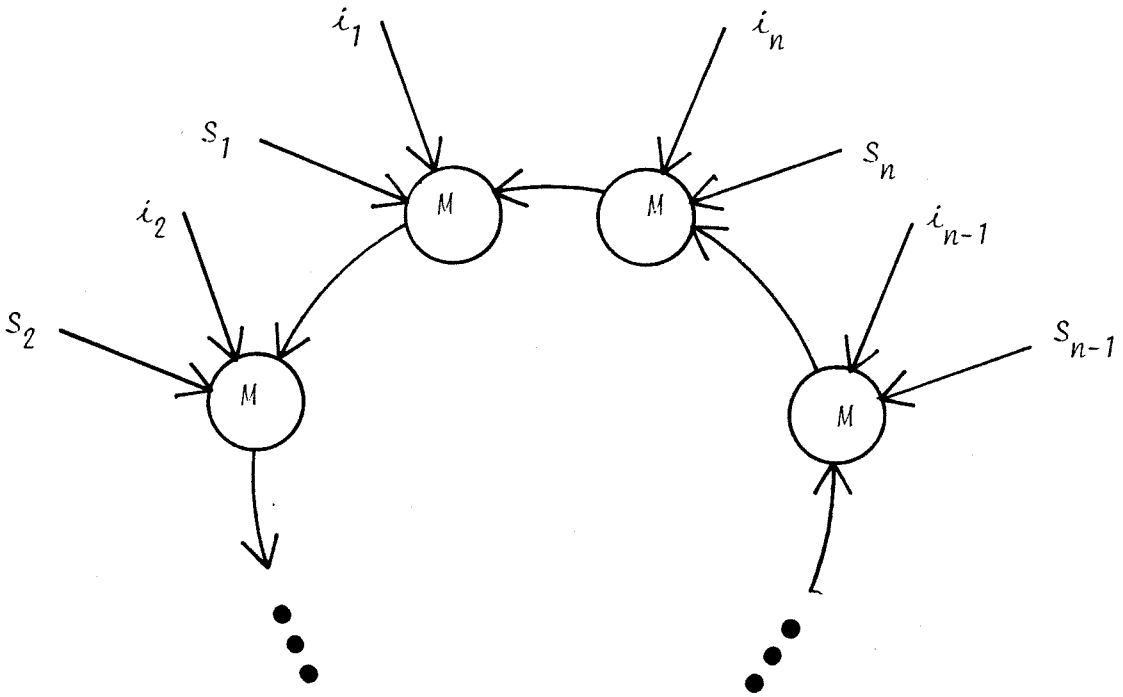


Figure 3.12 N-Writer Bus

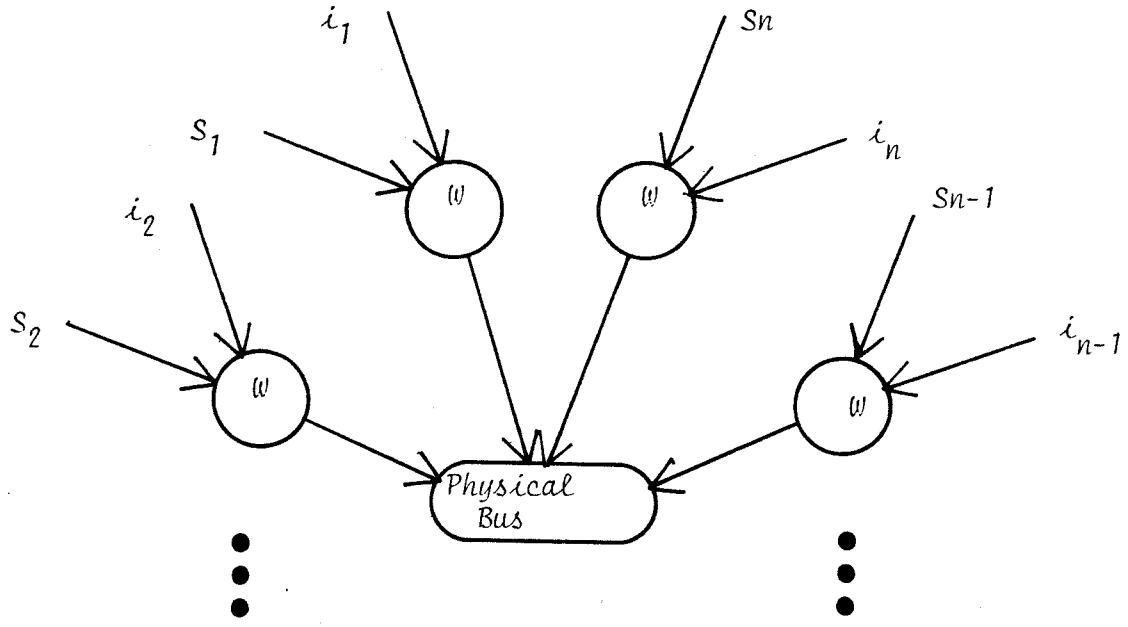


Figure 3.13 Physical Dual

Capping the Bus

The N writer bus model has a peculiar property, it doesn't work at all unless the entire bus is present. The model is incomplete unless the first bus writer has a connection to the last bus writer. The formation of the ring is the method by which the bus values are propagated around the mathematical model.

This peculiar property has some nice philosophical implications. It implies that a bus is not "complete", in some sense, until its full extent is known. The closing of the ring might be termed a "capping off" of the bus. After the bus is capped off, then it can be examined for correctness. Equivalently, a bus can be extended provided some assurances can be made about mutual exclusion. Two buses can be connected IF the enables are mutually exclusive.

The formalism developed here will be the basis for a system for guaranteeing "safe" pieced buses in a hierarchy. This system is developed in Chapter 5.

3.5 Conclusions

Hierarchical systems can be successfully modeled through the use of a functional notation for the leaf cells and combinators for the composition cells. State within leaf cells is modeled through the use of state sequences. The combinator notation allows the hierarchy to be mathematically separated from the leaf functions.

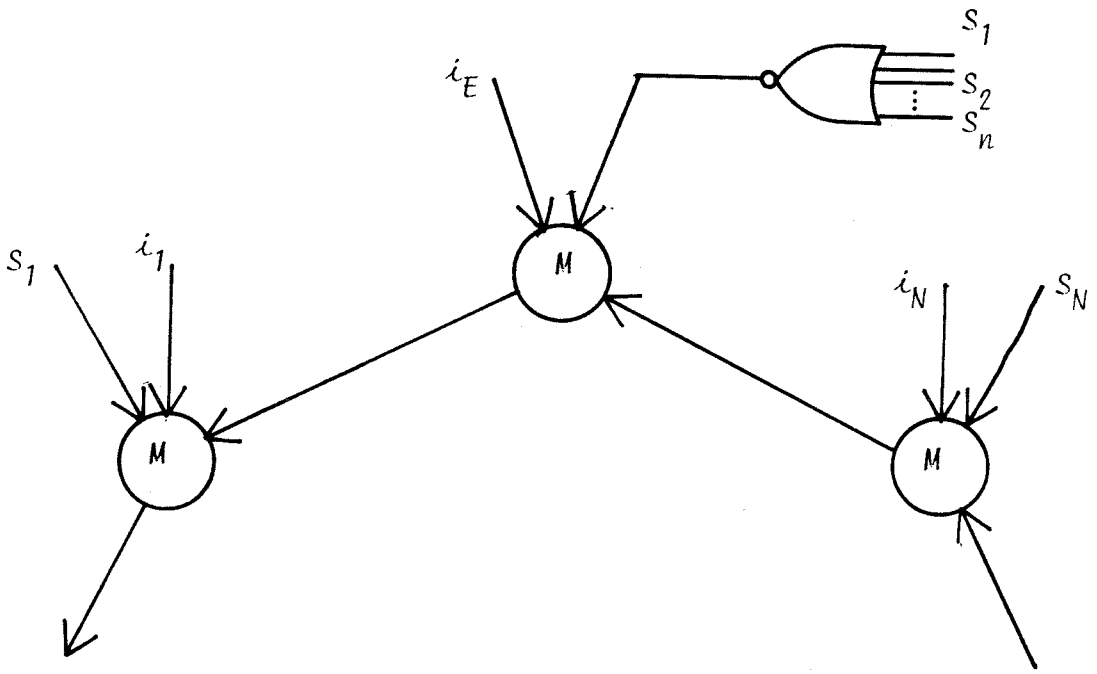


Figure 3.14 Extra Writer

Identity transformations on hierarchies were shown to be possible provided the combinators describing the hierarchies "terminate". Determining whether a combinator terminates is, unfortunately, an undecidable problem. Work needs to be done to determine a decidable subset of combinators that is sufficient for the description of VLSI systems.

A model for buses was presented that allowed multiple writers. This model was used to verify the rather obvious conclusion that only one writer could be enabled at a time. Two buses can be concatenated with the restriction that their two sets of enables are shown to be mutually exclusive.

Chapter 4

Geometry Composition

In this chapter, a composition algorithm for the geometry portion of a VLSI design system will be presented. Composition algorithms, as discussed in Chapter 2, must not add any functionality when interconnecting instances. This poses special problems with the geometric implementation of a design since traditional approaches have used wires as the interconnect mechanism.

When looking through the history of software development, symbolic addresses were available in some of the very first assemblers. The introduction of symbolic addresses relieved the programmer from the onerous and error prone process of cleaning up all the absolute addresses when adding or deleting code. Symbolic addresses were the first, and probably most basic, feature that made hierarchical design realistically possible because they helped to decouple independent parts of the system.

The VLSI analogs to symbolic addresses and their modern counterparts are the cell size and connector (externally available signal) positions. Adjusting to size and connector position changes is probably the single most important hierarchical feature needed in today's design

systems.

A look at some existing mask layout aids is presented as a precursor to developing the geometry composition algorithm. This examination is conducted from the structured design point of view, i.e. which existing systems encourage the designer to build structured designs.

A proposed structure, called SLAP, is presented that will provide the environment for geometry composition. The structure is embedded in an object oriented programming language allowing the cells to be parameterized.

The composition algorithm is described in the next section. By judicious restrictions, the algorithm is decomposed into two identical one dimensional problems. The algorithm places one instance at a time, recursively re-placing instances where the interconnection constraints cannot be satisfied.

Some examples, including a programmable PLA, are given along with some conclusions and proposed extensions to the algorithm. Included are the run times for the PLA example.

4.1 Existing Systems

Existing mask layout systems separate easily into three categories: mask editors, symbolic layout systems, and automatic layout systems. The systems in the mask editor category generally provide an interactive graphics environment that allows the user to massage mask artwork directly. Symbolic layout systems represent the mask layout symbolically to reduce the complexity of the design

task. By imposing a regular structure on the chip layout, automatic layout systems provide very fast mask layouts directly from some sort of netlist input. Each of these areas will be examined in the light of the structured design style to try and discover where the good and bad ideas lie.

At the end of this section, a "silicon compiler" [Johannsen 1979] developed at Caltech will be briefly discussed.

Mask Editors

Following in the footsteps of rubylith cutting and pencil and paper, mask editors provide a graphical medium in which to create and modify the shapes that make up a mask layout. Most commercially available mask design aids fall into this category, including the Calma [Calma 1979] and Applicon [Applicon 1979] systems. Mask editors generally allow the user to create and modify geometry shapes with the aid of a graphics display, a pointing device, and a keyboard. Some systems also interface with digitizers allowing hand drawn layouts to be digitized. The ICARUS [Fairbairn 1979] system developed at XEROX PARC is another example of a mask editor.

Mask editors do not generally presume to support a particular design philosophy. The three systems mentioned above all provide a form of hierarchy, some with a limited number of hierarchical levels. The hierarchy provided is a mixed one, as opposed to the separated hierarchy used here. Geometric shapes can be intermixed with instances of cells in any level of the hierarchy. Mask editors can be used to design in a structured way, but the design discipline is

not enforced by the system. Experience has shown that very few designs produced using these systems are structured.

Symbolic Layout

The designer of a complex chip is often overwhelmed by the number of details involved with representing even a simple circuit. In an attempt to reduce this "can't see the forest for the trees" syndrome, symbolic design introduces a notation that removes detail better handled by the computer. Specifically, most symbolic layout systems introduce a graphic representation of the layout that is an abstract view of the layout. This symbolic representation allows the designer to get a more global view which often results in better designs.

One of the first symbolic layout systems was the SLIC system [Gibson 1976] developed at AMI. Systems of this type restrict all features to lie on grid points. The grid itself is usually rather coarse, i.e. the distance between adjacent grid points is approximately the same size as the smallest allowed mask feature. Each grid point represents the local mask geometry with a single symbol, usually a normal letter or punctuation mark. The mask geometries represented by these single symbols can be arbitrarily complex, i.e. a transistor or contact. The systems provide an algorithm for transforming the symbolic representation into actual mask layouts. The analysis of this symbolic representation is much less complex than the corresponding analysis of a full geometric representation.

The latest rage in symbolic layout are variations of the STICKS system developed by John Williams at MIT and

Hewlett-Packard [Williams 1977]. These systems use stick diagramming as their symbolic representation of the layout. A stick diagram represents the mask layout in a very general way, indicating the general relationships between circuit features but not their absolute positions. Generally, a stick system will represent a circuit with symbolic components, like transistors and contacts, connected together by skeletal wires, i.e. wires with no width. An automatic, or semi-automatic, algorithm can then convert this stick diagram into the full geometric mask layout. The actual positions of geometric features are resolved by this "compaction" algorithm in such a way that all design rules are satisfied. Other stick systems have been developed at Bell Labs [Dunlop 1978] and Berkeley [Hsueh 1979].

The hierarchy provided by symbolic systems has been very similar to those of the mask layout systems. The ability to compose systems is somewhat enhanced with a stick system since the compaction algorithm adjusts for positional variations between instances being composed. SLIC type systems provide no composition aid being just a symbolic mask editor. None of the symbolic systems support the structured design style. In fact, it can be argued that symbolic layout systems tend to produce less structured chips since they allow much more bigger, more complex, cells to be designed. Since the designer can more easily produce random logic, the structure of a design becomes less of a factor. However, when chips begin to approach the 10 million transistor level, some method for structuring designs will be needed. Sticks and SLIC may be a cost effective method for designing unstructured chips with today's technology, but they are only a short term

solution to a much larger problem.

Automatic Layout

The LTX system [Persky 1976] from Bell Laboratories and the Master Slice system from IBM [Chen 1977] are two examples of automatic layout systems. These systems represent two implementations of the same basic idea: an overall chip plan is used to implement all designs.

The LTX system, a "standard cell" system, builds chips that have parallel rows of cells alternating with "wiring channels." The interconnections between the cells in a standard cell layout is accomplished by routing wires in the wiring channels. The cells that are wired together are generally a silicon version of small and medium scale integrated circuits, such as TTL. By expressing the design in terms of NAND gates and storage registers, the designer is able to completely disregard the geometric problems of mask design. The system will produce all of that automatically. The cells are generally a fixed size in one dimension, variable in the other and can have connectors anywhere on a grid facing the wiring channels.

The Master Slice automatic layout system uses a different chip plan, the "chip image." This image is a fixed array of cells with wiring between cells placed in fixed sized wiring channels run parallel to both axes. Again, the design is expressed in terms of familiar components and the designer is free to disregard the layout process.

These systems both represent methodologies that are in some sense opposite from the structured design style. Automatic

layout systems attempt to hide the fact that VLSI chips are fabricated in a planar medium. The designer is shielded as much as possible from the messy details of mask layout. However, these automatic techniques do not scale well with VLSI technologies. The wire lengths, and corresponding delays, increase faster than the number of components in the design [Sutherland 1973]. The structured design style directly attacks this problem.

Silicon Compilers

A new type of automatic layout system has been developed at Caltech by Dave Johannsen, the Bristle Blocks "silicon compiler" [Johannsen 1979]. Like the other automatic layout systems discussed, bristle blocks presumes a basic chip plan. The plan used here is consistent with the structured design philosophy in that it is built from groups of abutting cells with included buses. The basic chip plan has three parts: a "data path" consisting of abutting cells containing two included buses, a structured control PLA that decodes the chip microcode, and the interconnections to the pads. The specification of a chip is done in a very high level, compact code that describes the data path and its associated microcode. The rest of the system automatically generates the precise geometry for the data path, adds drivers for the control lines, generates an optimized control PLA, and routes wires to the bonding pads.

One of the unique aspects of this work is the method used to build the geometry in the data path. Data path cells are described in a way that allows them to conform to other cells in the path. The key observation that drives this is

that the data path will be as wide as the widest cell. This means that no chip area will be lost if the other cells are adjusted to be the same "pitch", or height. Each cell's internal geometry is described with respect to important interface features, such as the position of a bus wire. The system can interrogate each cell, find the minimum position of that bus wire that satisfies the design rules, and the cells automatically adjust to fit.

By adhering to the structured design philosophy, the Bristle Blocks compiler actually manages to be more efficient than a hand layout. Relayout of a previously designed chip in Bristle Blocks proved to save area since the program was tuned to be very efficient for that class of chips. The chip plan used for Bristle Blocks will not generate area efficient layouts for any design, but it does do a remarkably good job for a wide class of chips.

4.2 The SLAP System

In this section, the data structures, parameters, and other properties of a geometric system will be described that will provide a hospitable environment for the geometry composition rule. The system must allow the specification of a separated hierarchy and must also support structured design style. Also described is the interface to the geometry composition algorithm and its general properties. The actual algorithm is described in the next section.

This geometry algorithm is predicated on a two dimensional parameterization for cells. The parameterization is particularly simple-minded in an attempt to make the attendant system simple. This algorithm is not intended to

be the optimal one, but rather an example of a geometric composition algorithm.

The system has been implemented in SIMULA [Birtwistle 1973] and runs on a DECsystem-20. It has been designed to be an embedded language to allow the designer the full power of a programming language. The system is called SLAP in acknowledgement of LAP [Locanthi 1978], the first such imbedded artwork language at Caltech. SLAP stands for "Stretchable LAP", but also has the advantage that the designer can use it to "SLAP chips together."

Overall Strategy

In order to be consistent with the simple hierarchy described in Chapter 2, the geometry system will deal with two distinct types of cells. The leaf cells contain the wires, transistors, and polygons to define the masks. All of these shapes are parameterized in such a way that an instance of the cell can be deformed for a particular environment. The parameterization is described in the next section. The other type of cell contains only instances of other cells. In particular, the hierarchy cells do not contain any shapes that appear on a mask. All of those data are in the leaf cells.

All cells in this system are rectangular and have connectors only on the boundary. These two restrictions are essential to the composition algorithm described later but may not be needed with a more general algorithm. The structured design style does not preclude the use of odd shaped cells. However, experience with the ICARUS [Fairbairn 1979] design system has shown that rectangular

cells are not a restriction. Similarly, connectors constrained to lie on the cell boundary have not turned out to be a severe restriction.

The geometry system, given a description of a non-leaf or hierarchy cell, solves for the instance parameters to produce a valid geometric layout. The description of a hierarchy cell is just a "netlist" that indicates what connectors on what instances should be interconnected. Interconnection is defined as superposition: two connectors that should be connected must physically lie on top of each other. The instances of lower level cells are parameterized so that they can be deformed to accomplish the connector superposition.

Notice that the geometry system never adds any mask data as it solves for the parameters in a cell. This is consistent with not adding any function during composition. Routing wires between instances to perform geometry composition could introduce appreciable function: delays from wire capacitance and other effects. The cell functions must depend, not only of the inputs to the cell, but also on the parameters used to effect cell customization. The delays introduced by stretching the cells are equivalent to those introduced by wiring. These delays are not dealt with here, although they are represent an important problem that needs examination.

Leaf Cells

In analogy to the Bristle Blocks system, all cells in our hierarchy will be parameterized in such a way that they can, in a design rule correct way, stretch to match a

neighbor. The parameterization is a simple one, all internal geometry features are drawn relative to the connector locations. The connectors have a fixed minimum distance from their neighbors, but the distances are free to increase any amount. The geometry solution computes the amount that these distances need to be increased to satisfy the logical constraints.

In the SIMULA implementation, each leaf cell is written as a CLASS definition. The geometry for the cell is written as a procedure that must be called PROCEDURE GEOM. Any legal language constructs can be used to produce that geometry, although there are quite a few useful procedures defined in the SLAP system. In addition to the GEOM procedure, the designer must specify the names and locations of the available connectors. These data are specified in the initialization code for the cell. There is no restriction preventing connector positions from changing, although the composition algorithm must be run again on any cell that contains an instance of the changed cell. The SIMULA code defining an nMOS shift register is shown in Figure 4.1. A graphic rendering of that shift register is shown in Figure 4.2 with no parameterization, i.e. with minimum separations between adjacent connectors. Figure 4.3 shows the same shift register with non-zero parameters. Note how the various elements in the cell have shifted to adjust for the movement of the connectors.

Although only geometry descriptions have been discussed, any number of different, interrelated representations could be included in the SLAP system. Each new representation must include leaf cell descriptions and an appropriate composition rule. An example might be a representation to

```
cell CLASS sreg;
BEGIN

    PROCEDURE geom(g); REF(geomout)g;
    INSPECT g DO BEGIN
        REAL invx;
        invx:=5;
        gb(g,invx,cy("gnd1"));
        gb(g,invx,cy("vdd1"));
        grN(g,invx,cy("in")+4);
        grS(g,cx("out")-3,cy("out")+5);
        wire(metal,4,npt(2,cy("gnd1"))).x(cx("gndr")-2).wend;
        wire(metal,4,npt(2,cy("vdd1"))).x(cx("vddr")-2).wend;
        wire(poly,2,npt(cx("clkb"),1)).y(cy("clkt")-1).wend;
        wire(diff,4,npt(invx,cy("gnd1"))).y(cy("in")+4).wend;
        wire(diff,2,npt(invx,cy("in")+4)).y(cy("vdd1")).wend;
        wire(diff,2,npt(invx,cy("in")+3)).x(invx+5).y(cy("out")+6)
        .x(cx("out")-3).wend;
        wire(poly,2,npt(1,cy("in"))).x(invx+4).wend;
        wire(poly,2,npt(cx("out")-1,cy("out"))).x(cx("out")-3).wend;
        wire(poly,6,npt(invx,cy("in")+8)).dy(1).wend;
    END of geom;

    nam:-copy("sreg");
    zero_level:=TRUE;
    size(21,26);
    clr("gnd",2);
    cl("in",6); cr("out",6);
    clr("vdd",21);
    cbt("clk",13);
END of sreg;
```

Figure 4.1 Code for Shift Register Leaf Cell

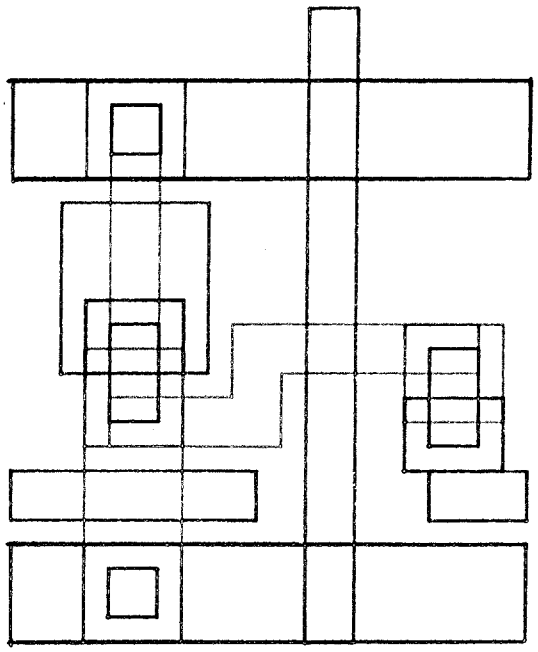


Figure 4.2 Plot of Minimum Size Shift Register

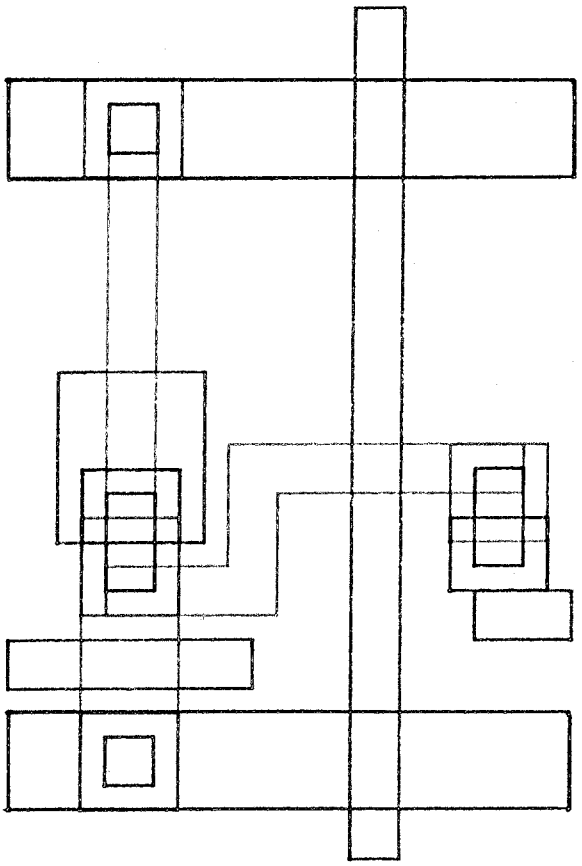


Figure 4.3 Shift Register Stretched

generate files for the SPICE circuit simulator. The leaf cell descriptions would be in terms of the transistors and their characteristics, resistors, capacitors and the interconnecting electrical nodes. The composition rule would be an algorithm that generates unique node names for each instance of a leaf cell, and makes sure that the logical interconnection between nodes implies that sharing of that node. Notice that the composition rule would not generate any more extra nodes than it does extra geometry on the mask.

Composition Cells

Cells that are not leaf cells are restricted to specifying only logical interconnections between instances of other cells. Other than that, the organization in SLAP is very similar. The connectors for the outside world are declared in the initializing code of the class, as they were for leaf cells, and the description of the implementation of this cell is done in a procedure.

A composition cell must be described in terms of instances and their interconnections. This description is provided algorithmically by PROCEDURE HIER. Again, the user gets the full power of the programming language along with the built in procedures of the SLAP system. The combinator representing this composition cell could be derived from a suitable notation, if not SIMULA, then LISP [Turner 1979].

An example of the use of the power of the host language in the design of a composition cell is a programmable PLA cell. The PLA cell that will be described later as an example of the composition algorithm is characterized by

data in a parameter file. These data include the number of inputs, feedback terms, outputs, and minterms. In addition, the code for each cell in the logic array is described. The PLA cell reads this information and generates the appropriate instance and interconnection description using many SIMULA constructs such as WHILE loops, FOR loops, procedure calls and so on.

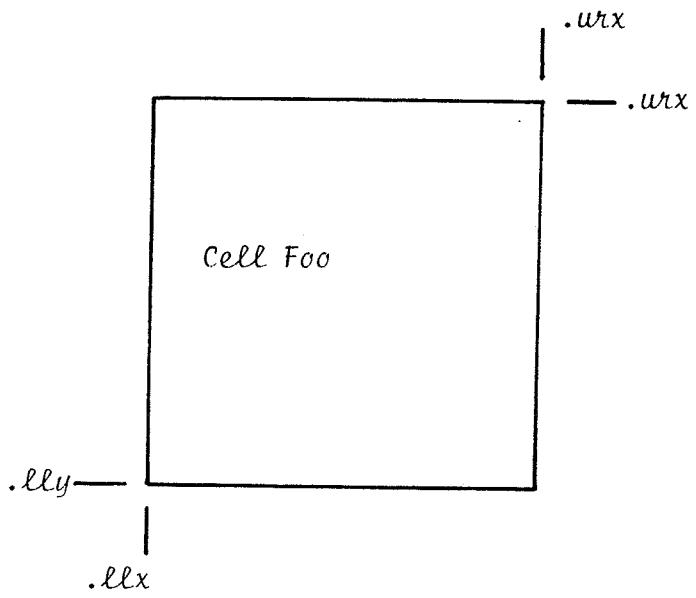
4.3 The Composition Algorithm

In this section, a geometry composition algorithm will be presented, some successfully run examples will be shown, and the limitations and needed extensions to the algorithm will be discussed. This algorithm is based on an idea originally suggested by Sally Browning.

The Inputs and Outputs

The SLAP geometry composition algorithm is invoked with a set of data describing a composition cell that needs to be correctly composed. The input and output needs of this algorithm are discussed in this section.

This algorithm works with a data base that describes the interfaces of cells. A cell's interface includes its name and the connectors split up into those on the horizontal and vertical edges. Each connector is named and lies at a specified position along one of the edges of the cell boundary. Along with the connectors declared by the designer are the automatically declared connectors used by the system to cause edges to abut. These four connectors are shown in Figure 4.4 along with the code declaring them.




```
Cell Class Foo;
Begin ----

        cl(".lly",0);
        cb(".llx",0);

        cr(".ury", ymax);
        ct(".urx", xmax);

END;
```

Figure 4.4 Automatically Generated Connectors

The SLAP algorithm expects as input two lists: the instances to be parameterized and a list of interconnections between connectors on instances. These two lists are generated by calling the HIER procedure of a composition cell.

The instances input to the SLAP algorithm contain, along with normal instance data including a pointer to its definition and a transformation, the stretching parameters that define how much each connector has been displaced from minimum. In addition to the instances defined by the user, each cell generates instances of fake cells for the left, bottom, right, and top boundaries. These are used to ensure that the connectors of this composition cell lie on its boundary.

Each interconnection represents a constraint in either the horizontal or vertical direction. The interconnection constrains two connectors to have a common coordinate, either an x or a y coordinate. A statement in the imbedded language that produces interconnections looks like:

```
net("inst1","conn1","inst2","conn2");
```

This would produce two interconnections constraints for the SLAP algorithm. Figure 4.5 illustrates this net statement and the constraints produced. The most obvious constraint is that the vertical positions, in this case, are constrained to be the same. Less obvious is that the edges, each represented by an automatically declared connector, must be constrained to have the same horizontal position. With this decomposition of logical constraint into two orthogonal physical constraints, the algorithm is separable into a horizontal part and a vertical part, each

of which is completely independent of the other. The independence of the two dimensions is made possible by the cells being rectangular. Since the connectors lie on the edges of a rectangle, the automatically declared connector that constrains instance edges to abut need not be connector specific. In this way, the set of horizontal connectors and constraints are completely separate from the vertical connectors and constraints.

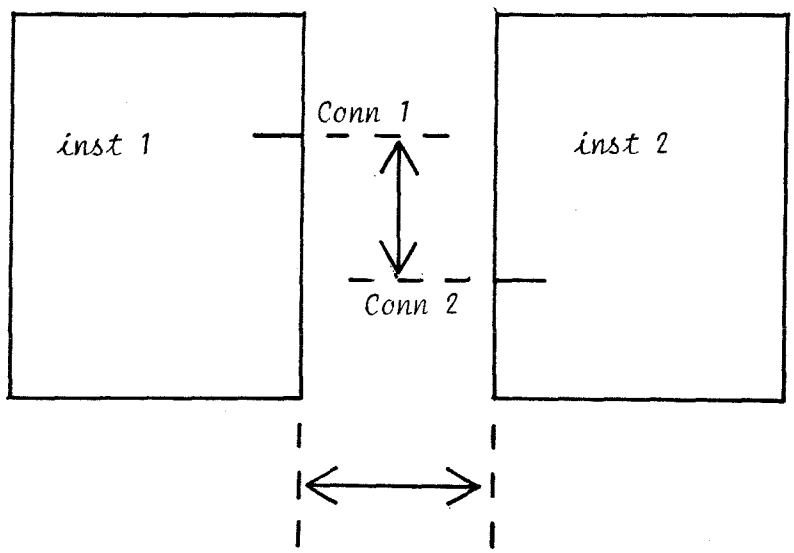
As set-up for the algorithm described below, each instance gets a pointer to all of its constraints. These are then sorted by increasing connector position.

As output, the SLAP algorithm produces a list of parameterized instances. There are two sets of parameters, one for the horizontal and one for the vertical connectors. The parameters are in the form of the amount a connector needs to be moved, along its edge away from the origin, in order to satisfy all its physical constraints.

Overall Strategy

The algorithm is basically axis independent: the horizontal constraints can be handled completely separately from the vertical. However, for purposes of discussion and ease of understanding, the two axes will be treated together. If they are treated together, then there is a nice image that describes exactly what happens as a new instance is placed.

The instances are placed, one at a time, into the plane. To place an instance, each constraint involving an instance already placed, an active constraint, is examined and an attempt to satisfy the constraint is made. Satisfaction is



Conn 1.y=Conn 2.y
Conn 1.x= Conn 2.x

Figure 4.5 Constraints Generated by
net("inst 1","conn 1","inst 2","conn 2");

achieved by setting the parameter for the constrained connector so that it moves to the right place. The cases where this cannot be done are discussed below. Since the instances need to be translated as well as stretched, the first active constraint is used to find the translation along the appropriate axis, rather than causing a parameter to be set.

Trying to stretch a connector by a negative distance means that the constraint is unsatisfiable. When this happens, the placement procedure is called recursively to re-place the other instance beginning at the offending connector. Trying to stretch a connector by a negative distance on one side of a constraint means that the other side of the constraint could be stretched a positive amount to achieve satisfaction. Of course, recursively re-placing the other instance involved may then cause other constraints to be unsatisfiable, and so on.

Each instance is placed by the top level of the algorithm once. Unsatisfiable constraints may cause an instance to be re-placed. The complexity of the algorithm is, at best, O(# of instances). At worst the instances are placed in just the backwards order, giving a worst case complexity of $O(n^2)$. Presorting the instances to discover as much ordering as possible in each dimension can reduce the expected time cost to near the best case.

Recursive Placement

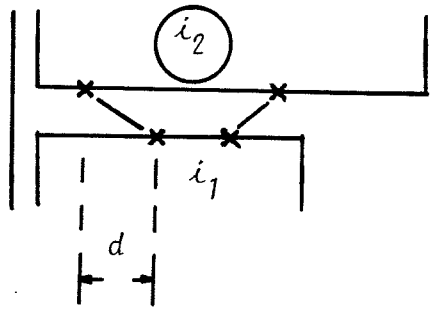
The discovery of an unsatisfiable constraint may cause the placement algorithm to recurse, moving or stretching a previously placed instance. This section discusses the

conditions that cause recursive calls to the placement routine.

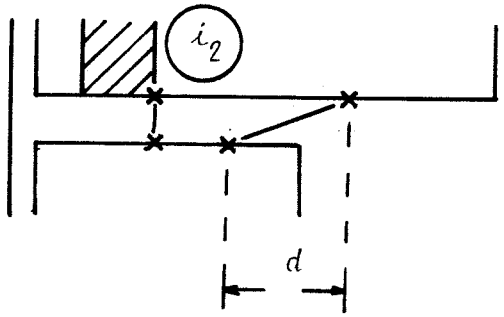
The example of recursive placement in Figure 4.6 shows that, as instance i2 is being placed, the constraint from i2.c2 to i1.c2 is unsatisfiable because the needed parameter, d, is negative. To satisfy the constraint, a recursive call is made on the placement routine, having it place i1 starting with connector c2. After re-placement, all the constraints are satisfied. Of course, the placement routine can recurse to an arbitrary depth depending on how far the constraints have to propagate. Figure 4.7 illustrates this propagation of constraints. As instance A is being placed, it recurses to stretch instance B, which has to move instance C, and so on down to instance E.

Another case that requires the recursive call of the placement routine is more than one constraint on a connector. The user is not allowed to connect more than two connectors; however the system generated edge connectors can be constrained to abut an arbitrary number of other edge connectors. Figure 4.8 illustrates the three instance case. Instances A and B have been placed and instance C is in the process of being placed vertically. After processing the constraint between the top edge of C and the bottom edge of A, we get the situation in Figure 4.8-B. The constraint between C and B causes a recursive call to re-place instance A, producing the final results shown in Figure 4.8-C.

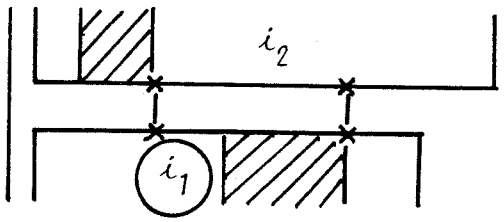
If the instances can be sorted so that there are never any unsatisfied constraints, then the recursive re-placement



Place i_2
$d > 0$
Stretch i_2

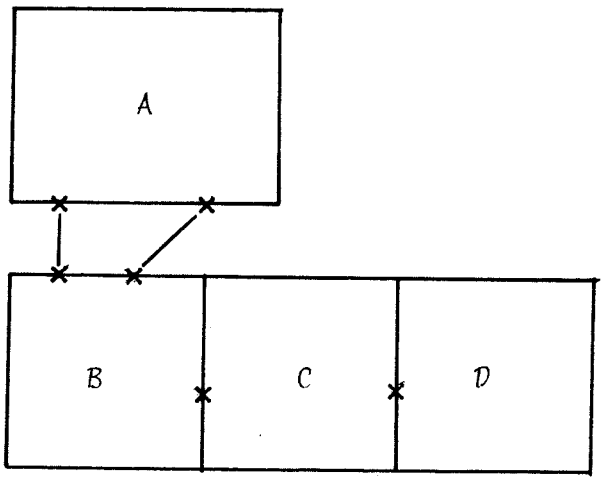


Place i_2
$d < 0$
Recurse with i_1



Final Placement

Figure 4.6 Recursive Constraint Propagation



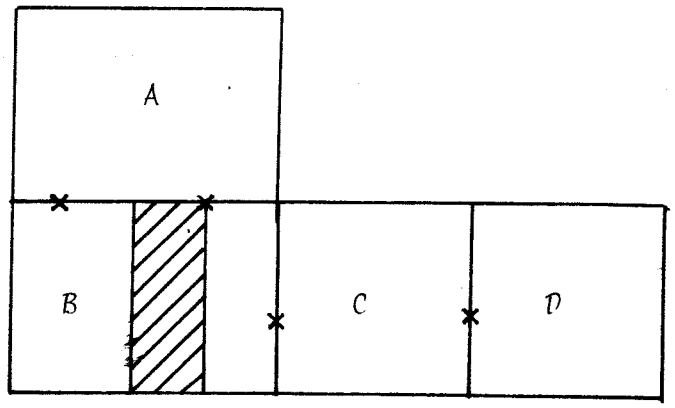
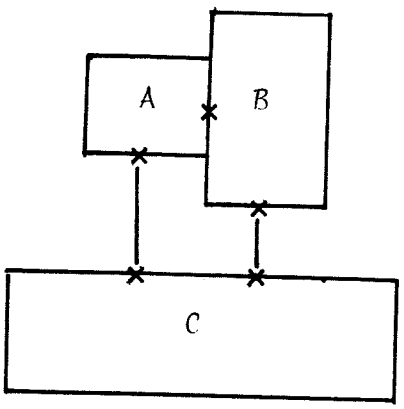
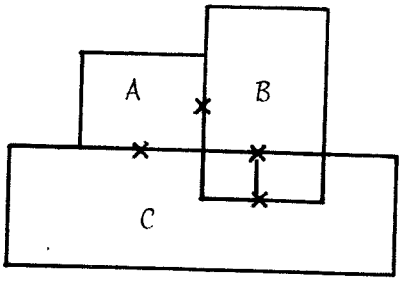


Figure 4.7 Constraint Propogation

A



B



C

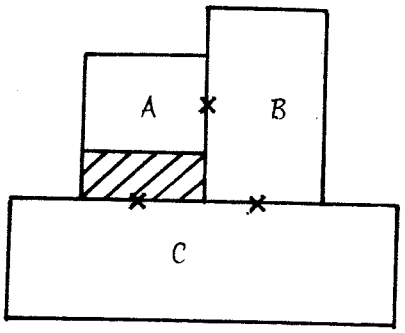


Figure 4.8 Multiple Constraints

needn't be included. Doing that sort also reduces the time complexity of the algorithm to its minimum. However, finding the ordering that minimizes recursive re-placement is a not an easy thing to do, and was the basis of the first few cuts at the SLAP algorithm. The present algorithm is very straightforward and easy to code, making it clean and attractive.

Extensions to the Algorithm

The present algorithm has two major deficiencies: unconstrained instance edges may overlap other instances and instances do not automatically mirror or rotate to satisfy constraints.

Any instance with any interconnection to another instance has both horizontal and vertical constraints; however, any particular edge may not be constrained. This lack of constraint might result in one instance overlapping another. The illegal overlap is easily found, but not so easily fixed. The solution to this problem is not as straightforward as it might first seem, since there may be many different ways to guarantee non-overlap. The trick is to find the "best" one.

Mirroring and rotating instances to satisfy the constraints is, I believe, a fairly trivial extension to the overall system. Mirroring requires logic to notice that, when trying to satisfy the second constraint, the two connectors involved are on opposite sides of the already satisfied constraint. Rotating can be discovered from the examination of the interconnections: if it involves a connector on a horizontal edge and one on a vertical edge

then a rotation is required.

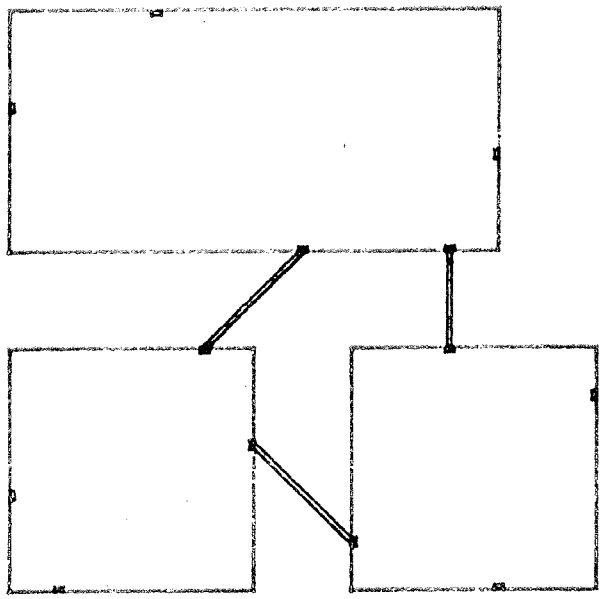
4.4 Examples and Conclusions

In this section, some small examples are discussed to illustrate features of the SLAP system. A fairly complex example is presented as some evidence that the system is usable to design VLSI. Some conclusions about the algorithm and future extensions are also discussed.

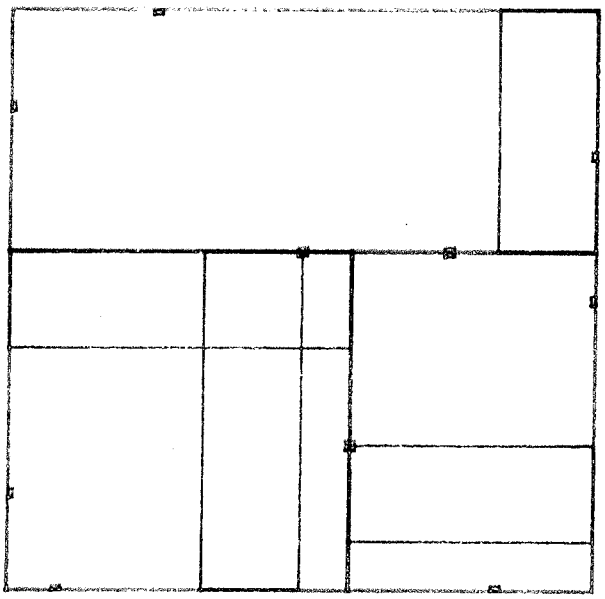
Small Examples

Three views of a very simple, three instance cell are shown in Figure 4.9. The top view shows a bounding box representation of each of the three instances showing their relative sizes, connector locations, and logical interconnections. The center view shows a bounding box representation that is produced by the SLAP system. This is a view of the solved cell, with the connectors shown abutting. The red boxes indicate how much the different connectors have had to be stretched to satisfy the constraints. The bottom view shows how the instantiated layout has the internal geometries stretched along with the connectors.

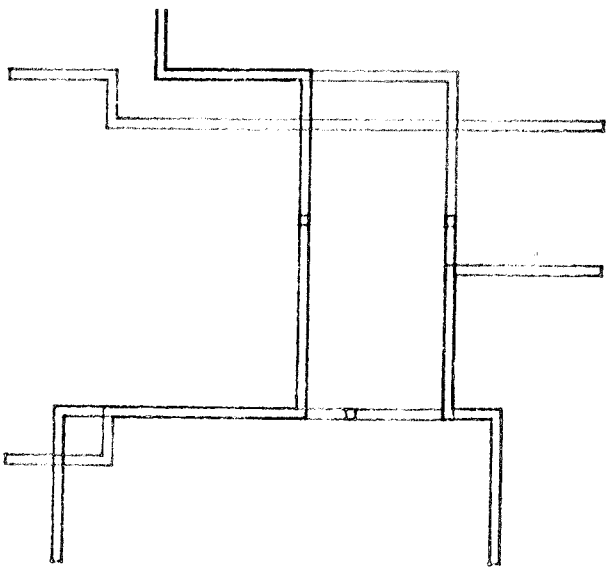
To illustrate what happens with a multi-level hierarchy, two instances of the cell solved in Figure 4.9 are made into another cell, as shown in Figure 4.10. The bounding box representation of the solved cell is shown at the top, one level deeper in the hierarchy is shown in the center. The bottom view is the instantiated layout.



Logical Interconnection



Bounding Box Representation



Instantiated Layout

Figure 4.9 Three Instance Cell

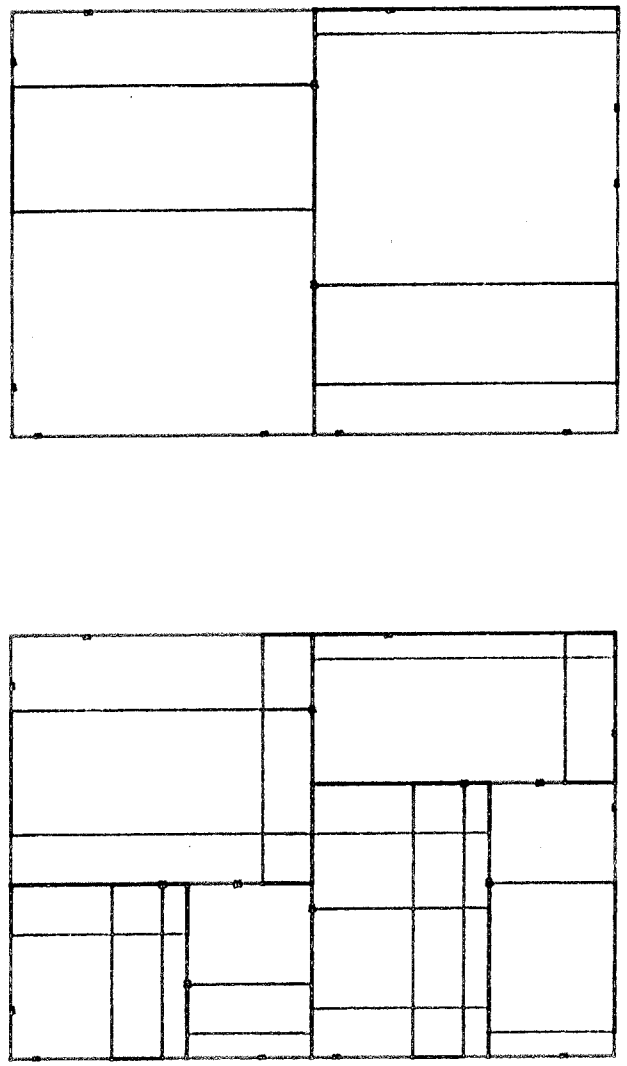
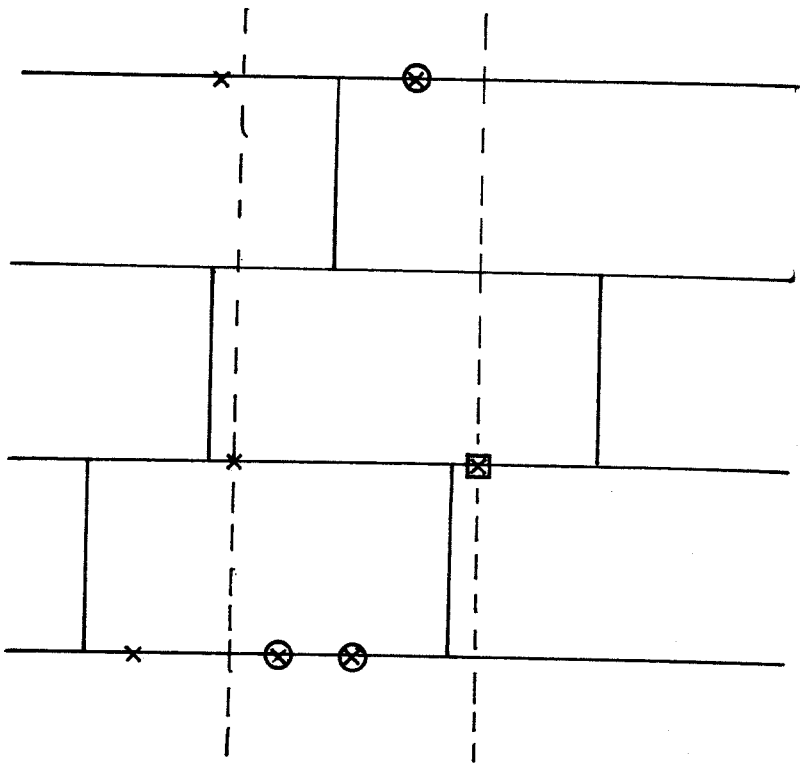


Figure 4.10 Multi-Level Hierarchy Example

In a multilevel hierarchy, a composition cell can be instanced in another composition cell. When the composition algorithm is run, the composition cell instance is parameterized. Inside that instanced composition cell are other instances with their own parameterization. When instantiating the hierarchy tree, the parameters must be propagated down the hierarchy by a reverse composition rule. Composing the parameters is a straightforward process. As illustrated in Figure 4.11, some of the parameters passed into the parent cell need to be added into the parameter for a particular instance. Basically the algorithm adds up all the parameters from connectors on the parent cell that are to the left of, or below, the instance's connector under consideration. Not all of the parent connectors count however, because some of them were already used for connectors to the left or below on this instance. In effect, all of the parameters for the connectors in the range shown in Figure 4.11 are added into the parameter for that instance's connector. This algorithm is recursively applied down to the leaves of the hierarchy, where the parameters are used to generate geometry.

A Programmable PLA

To try and lend credence to this whole algorithm, a fairly complex cell was designed: a programmable PLA. This PLA is characterized by a simple file format that contains the number of inputs, outputs, feedback state terms, and minterms. The file also contains the code for the AND and OR planes for the PLA. The layouts for the cells were stolen directly from Mead and Conway, although they had to be parameterized for the SLAP system.



× external connectors

⊠ connector whose parameter is being computed

⊗ external connectors whose parameters contribure

Figure 4.11 Composing Parameters

The total number of instances in the cell, whose layout is shown in Figure 4.12, was 75, with 529 constraints. The system produced the two graphic representations shown in Figure 4.12, the top view being the bounding box with visible stretch-marks, and the bottom view being the mask layout. The file that generated this PLA is the following:

```
2 2 4 4          -- #in #feedback #out #minterms
100101 111111    -- and-plane code/or-plane code
011010 000000
1010X0 101101
0X1X11 010010
```

The SIMULA implementation that produced these drawings took slightly over 2 cpu-minutes on a DECsystem-20.

Conclusions and Extensions

The current algorithm does not automatically rotate and mirror instance to satisfy the logical constraints of the composition cells. This is a major lack in a practical design system, but should be a straightforward addition to the algorithm presented.

An improvement suggested by Martin Newell would allow the connectors to move a certain distance before they affected their neighbor. In this way, minor mismatches in an interface could be adjusted without the need to increase the size of the cell.

Probably the biggest deficiency of the geometry algorithm is its inability to handle incompletely specified systems. Each instance must have a specified relationship with any instance that might interfere. Not only does this put a burden on the designer, but it precludes using the

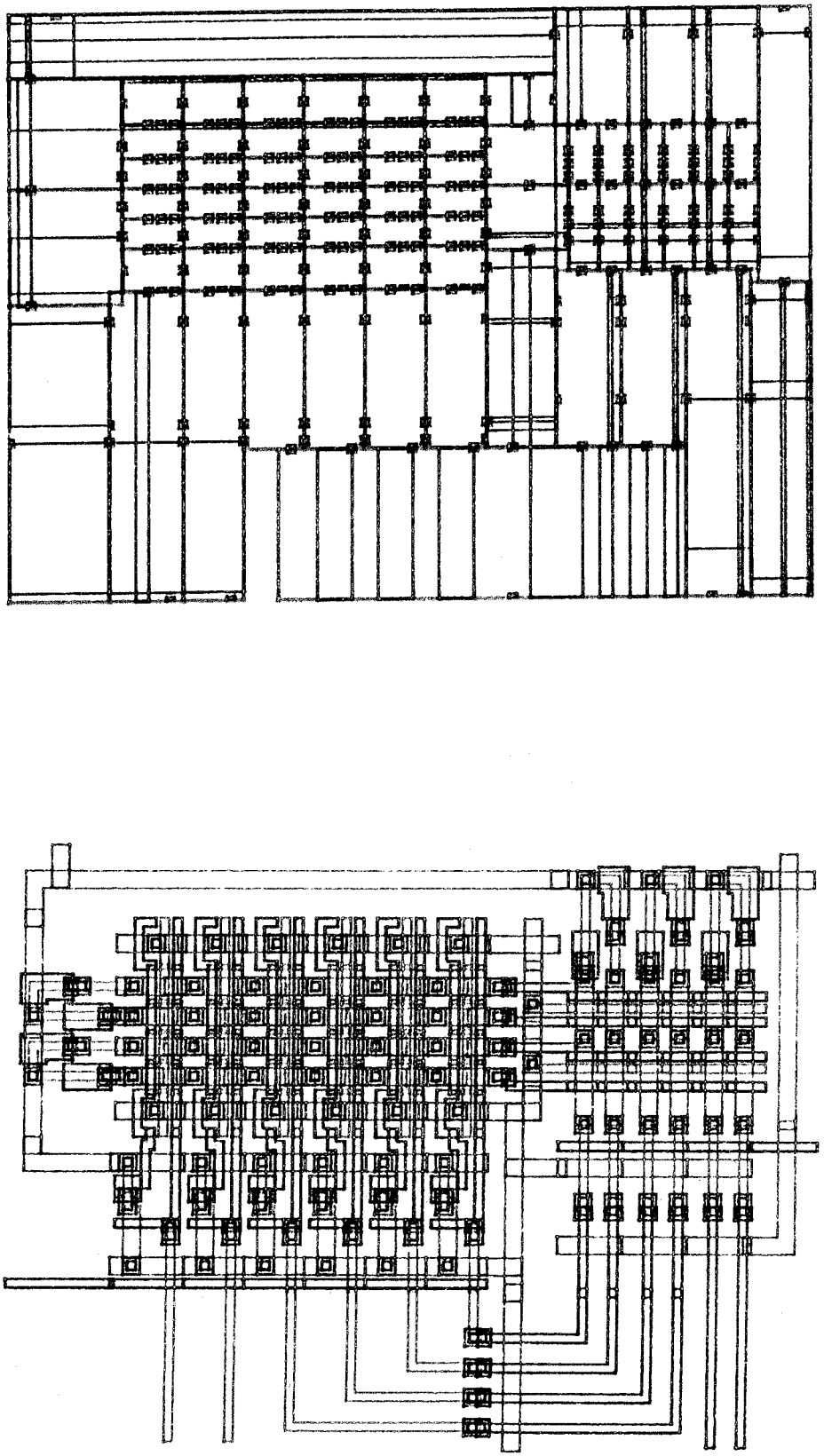


Figure 4.12 A Programmable PLA

algorithm on a very interesting class of cells. An intriguing application of the geometry algorithm is in the area of wiring cells. The automatic generation of wiring cells could be a fairly straightforward adaptation of classic routing techniques. Wiring cells could be generated using a small set of basic wiring cells: cross overs, contacts, corners and so on. These primitives could then be just logically interconnected to satisfy the logical constraints. The geometry composition algorithm would then place and stretch the basic wiring cells to fit together.

Chapter 5

Functional Abstraction

The previous chapter looked into guaranteeing the geometric correctness of a design. In this chapter, the major subject is that of guaranteeing some sort of functional correctness or at least consistency throughout the design.

Guaranteeing geometric correctness is a large step toward guaranteeing the functional correctness. Having a layout that is inconsistent with the logical interconnections does not usually help to improve the functionality. However, there need to be more aids to help the designer be sure that he is building a logically consistent hierarchy.

5.1 What Is "Functional Abstraction"?

Functional abstraction is a buzzword used to describe TYPEing in a hierarchical design system. It is an attempt to capture the inherently recursive nature of hierarchies by abstracting the semantics of a design. Abstracting a function means to pick out exactly those features, i.e. the drive capability of a particular output, which are important to someone trying to deal with the function. Thus, functional abstraction in a hierarchy is accomplished through an abstraction of what is happening at the level

below. Functional abstraction presupposes that details of the implementation are not necessary when dealing with a system, resulting in a reduction of information.

Functional abstraction allows us to interact with a cell without having to delve into the guts of its implementation.

Ideas from TYPEing in Software

TYPEing is a fairly widespread software technique that attempts to eliminate certain classes of common programming errors. Some of the errors caught by typical TYPE systems include misspellings, missing parameters, parameters of the wrong TYPE, and sometimes even array bound errors. These errors can be identified merely by forcing the programmer to associate every variable with a TYPE.

In some languages, TYPEing is also used to cause computation, i.e. coercions between equivalent TYPEs. Some coercions are almost always implicitly included in a TYPEd language. The most common examples are the compiler coercions between INTEGER and REAL. Some languages, notably ICL [Ayres 1978], allow the user to declare an algorithmic way to coerce one TYPE into another.

Coercions are basically a method used to "relax" a TYPEing system a little. As an example, let's consider writing a sort program. The logic, and 90 percent of the program, is completely independent of what TYPE of objects are being sorted. The only TYPE dependent operation is an order relation between the objects. With coercions, the entire sort program can be written in terms of a TYPE optimized

for sorting. When the program is called to sort a group of some other TYPE, a coercion can provide automatic translation to the sorting TYPE. In a typical fixed TYPEing system, either a different sort program must be written for each TYPE of data, or the TYPE system has to be breached in some way. (As a sidelight, this particular example works very well in an object oriented environment: each object that can be sorted is declared a subclass of SORTABLE. The new subclass provides a procedure to define its peculiar ordering relation. CLASS SORTABLE would contain the code to do the actual sorting.)

TYPEs in a Hierarchy

In common with all properties in our hierarchy, TYPEs originate at the leaves and are propagated up the hierarchy according to some composition rule. This is somewhat different than the usual kind of TYPEing in programming languages. TYPEing in a hierarchy specifically governs the set of legal compositions.

TYPEs will be used in our hierarchy to catch illegal compositions, those where the connectors on two instances are interconnected but are not compatible. Two connectors that can be legally connected are said to conform. Checking for connector conformance is not as simple as a simple equality test between their TYPEs. As an example, consider a TYPE system that has inputs and outputs. Connecting an input to an input is allowed, as is an output to an input, but connecting two outputs is illegal. As the number of TYPEs increases, these conformance rules get more complex.

Each TYPE must have a propagation rule to guarantee that the functional constraints it represents are carried up the hierarchy. As has been mentioned before, all good hierarchical algorithms consist of leaf definitions and a composition rule. Part of the composition rule is the generation of the data needed by that composition rule at the next level in the hierarchy. This generation of data, or abstraction of function, is handled by a propagation rule for TYPEs.

Satisfying the Mathematics

When discussing the mathematics of hierarchical systems in chapter 4, we assumed a strict separation between inputs and outputs. In particular, outputs could not be connected. A simple fixed typing system would allow us to enforce such strict separations. In addition, the separation of VDD from GND could be maintained.

Remember that buses, or shared variables, imply that some constraints on the inputs and outputs of a module be propagated up the hierarchy. Not only do these constraints have a fixed aspect, i.e. that an input must be mutually exclusive, they also have connector specific aspects, i.e. that an input is mutually exclusive with respect to a particular bus.

The addition of a simple typing system to check for these kinds of rules allows the assumptions necessary for mathematical analysis to be enforced during design. This enforcement makes the mathematics a bit more realistic because the designer is forced to rigidly adhere to his original assumptions.

5.2 Example Type Systems

This section will present two hierarchical TYPEs: the Rem and Mead restored logic types [Rem 1980] and types to guarantee mutual exclusion between bus writers. The objective of the restored logic, RL, type system is to guarantee that all signals in a CMOS VLSI system are restored at some point, i.e. that there is a short path between a signal and one of the "rails", either power or ground. The RL system also happens to support the separation of inputs and outputs that is needed for the mathematics of hierarchies from Chapter 3. The other type system that will be discussed is one that attempts to guarantee that multiple writers on a bus will be mutually exclusive, i.e. only one will write on the bus at a time. The mutual exclusion type system enforces the constraints discovered in section 3.4.

Restoring Logic

The original formulation of the Restored Logic rules dealt with a slightly different hierarchy than the one used in this thesis. The restoring logic hierarchy is built out of RLM's, or restoring logic modules. Each RLM can be composed of instances of other RLM's hooked together with a network of switches. The switches are, in general, built out of two parallel CMOS transistors, although one of the transistors can be removed in many cases by a simple optimizer.

The connectors on an RLM are split into inputs and outputs. Bidirectional buses are handled by making the entire bus,

including drivers, receivers, and the bus-writer selection circuitry an RLM. The designer must verify that the bus RLM satisfies the restored logic requirements, detailed below.

All RLM's must obey two rules, a "no-fighting" rule and a "close-to-an-output" rule. The no-fighting rule guarantees that no two outputs will be connected together. The close-to-an-output rule guarantees that all inputs are driven by restored outputs. Since an input can be connected to some output through a network of switches, the switch network must be examined to make sure the input is properly connected independent of the state of the module's inputs. With the additional definition that an RLM's outputs to the outside world look like inputs inside, and similarly with the inputs from outside, these two rules guarantee that all strings of switches will be no longer than a constant multiple of the depth of the hierarchy.

One of the most important facts to notice is that the rules can be completely verified within 1 level of the hierarchy. This fact in conjuction with the "closed" nature of RLM's, i.e. that any proper composition of RLM's makes another RLM, means the restoring logic type system will make a proper composition rule.

Since the hierarchy used in this thesis does not have any interconnection through switches, the RL rules simplify to the "no-fighting" rule and a "no-floating-input" rule. Since there are no switch networks interconnecting RLM's, the rule requiring inputs to be driven by outputs reduces to a rule requiring that all inputs be connected to an output.

A complication presented by the nature of the hierarchy is that of wiring cells. In order to eliminate the necessity of defining a new wiring cell for every combination of inputs and outputs that are hooked together, some way of representing the untyped nature of wires is needed. A wire by itself has no type, it merely represents the electrical equivalence of the connectors it touches.

A type system that will check that a composition is a legal RLM has been implemented using three basic types. The input type represents a connector that expected information to be passed into the cell. The output represents a restored logic signal leaving the cell. A wireThru type represents the uncommitted nature of the wiring cells. Connectors of type input are allowed to connect to other inputs and outputs. They propagate up the hierarchy just as an input with no changes. Ouputs can connect only to inputs and propagate unchanged up the hierarchy. The wireThru type has the most interesting conformance and propagation rules. These rules will be described in detail in the next two major sections.

Mutual Exclusion

In a design style that allows multiple writer bus communication, a formal way of guaranteeing exclusive use of the bus is needed. The problem is amplified in a hierarchical system, where the bus will tend to be split into little independent pieces. The guarantee of mutual exclusion between bus writers is the aim of the MEX type system.

The system discussed in this chapter has three types: type bus, type MexIn (mutual exclusive input), and type MexOut (mutual exclusive output). Type bus is used to identify those connectors that represent a shared bus. Connectors of type MexIn represent signals that can enable a write on a bus. MexOut connectors are guaranteed to be mutually exclusive and suitable for enabling a bus write.

The conformance rules for these types are all simple. Type bus can conform only with other connectors of type bus, MexOuts can conform only with MexIns, and MexIns only with MexOuts. The propagation rules are more complex and will be described in the next major section.

Equivalence Classes

A common property between the more interesting of the types discussed above is that connectors with a given type are somehow grouped. The wireThru type in the RL typing system has to have some way of grouping all of those connectors that are electrically equivalent. The bus from the MEX scheme should be grouped with all of the other connectors that also represent that bus. Similarly, the MexOuts have to be grouped with the other connectors that are mutually exclusive.

These groupings will be called equivalence classes. An equivalence class is a set of objects that share a common property. For example, the connectors of type wireThru grouped in the same equivalence class are all electrically equivalent.

For these two typing systems, several equivalence classes will be defined. Each class will relate some of the connectors in a cell, but the property that relates these connectors will vary with the type. The equivalence class will be part of the data that is needed at the leaves or abstracted for each cell.

5.3 Propagation

The rules that abstract the appropriate functional aspects for RLMs and mutual exclusion cells are the subject of this section.

Restoring Logic

As mentioned when presenting the RL types, the propagation rule that is of interest is the rule for type wireThru. Types input and output just trivially propagate themselves up the hierarchy. The interesting feature of type wireThru is that it requires the use of an equivalence class.

Type wireThru is meant to represent the behavior of wiring cells in an RL type system. As such, all of the connectors that are wired together in a leaf cell must be declared to be a part of an equivalence class to represent their electrical equivalence.

Given that the leaf cells are correctly specified, the connectors for a composition cell containing wiring cells must be correctly typed. The Equivalence classes relating those composition cell connectors that are wired together through instances of wiring cells must be constructed. Once these equivalence classes exist, the composition cell

connectors must be typed appropriately.

Figure 5.1 shows some pidgin Simula code describing an algorithm that abstracts the equivalence classes from an RL composition cell. The algorithm visits each connector of the composition cell once. If the connector is already in an equivalence class, then it has already been typed correctly. The connector is interconnected with only one internal connector, one that is on an instance contained by this composition cell. Thus there are three cases depending on the internal connector type. Two of the cases are easy: if the internal connector is of type input or output, then the composition cell connector merely copies that type. The third case is interesting. When the internal connector is of type wireThru, a recursive algorithm is called to build the equivalence class and return the correct type for the composition cell connector.

The recursive algorithm that builds the equivalence class may not end up specifying that the composition cell connector be a wireThru. In the process of building the equivalence class, it also touches all of the "ends" of the wiring network, i.e. all of the connectors in the composition that are not of type wireThru. While doing that it keeps track of the kinds of connectors wired together. The rule for determining the type of the externally available connector has three cases. If there are no inputs or outputs, then the external connector is of type wireThru. If there are inputs and no outputs, then the external connector is of type input. If there are outputs, then the external connector is an output. Having more than one output wired together is a composition error, which will be identified during the conformance check.

```
PROCEDURE propogateTypes( compositionCell ) =
    FOR c <- each compositionsCell connector DO
        IF NOT c.beenTyped THEN c.type <- typeOf(c)
        ENDIF;
    ENDDO;

PROCEDURE typeOf( connector ) =
    IF connector.beenTyped THEN RETURN(connector.type)
    ELSE
        IF connector.toType = input THEN RETURN(input)
        ELSEIF connector.toType = output THEN RETURN(output)
        ELSEIF connector.toType = wireThru THEN
            RETURN(recursiveTypeOf(connector.toConnector))
        ENDIF;
    ENDIF;

PROCEDURE recursiveTypeOf( connector ) =
    eq: equivalenceClass
    FOR c <- each connector in my equivalence class,equiv DO
        IF externalConnector(c) THEN append(eq,c)
        ELSEIF typeOf(c) = input THEN RETURN(input)
        ELSEIF typeOf(c) = output THEN RETURN(output)
        ELSEIF typeOf(c) = wireThru THEN
            eq <- union(eq,typeOf(c).equiv)
        ENDIF;
    ENDDO;
    RETURN(wireThru with eq as equivalence class);
```

Figure 5.1 Code for Propagating TYPE Wire Thru

After the recursive algorithm builds up the equivalence class and determines what type those equivalent connectors will have, it assigns a type to each external connector in the set.

Figure 5.2 shows an example of how type wireThru propagates up the hierarchy. The composition cell, GRID, is made up of 9 cells laid out in a grid. The four corner instances are of another composition cell called ELEN because of the cells it composes. The other five instances, of cell TEE, have four connectors, all of which are wired together in a tee. ELEN is made up of four instances, two of which instance a DASH cell containing a single horizontal wire, another instances a cell containing an EL shaped wire, and the last instances a cell with an EN shaped wire. Cell GRID is shown in Figure 5.2, and its equivalence classes are shown in Figure 5.3. A plot of the GRID cell is shown in Figure 5.4.

Mutual Exclusion

Similar to the wireThru type, the MEX types will be grouped in equivalence classes. However, the MEX equivalence classes are a bit different since the connectors in the class can be of different types. The equivalence that holds together a set of MEX connectors is that they all deal with a particular bus, the bus whose access is being controlled.

Leaf cells are declared in a similar way to RL leaf cells, any connector that is to be typed as a bus, MexIn, or MexOut is made a part of an equivalence class. All of the MEX type connectors that have any connection at all are

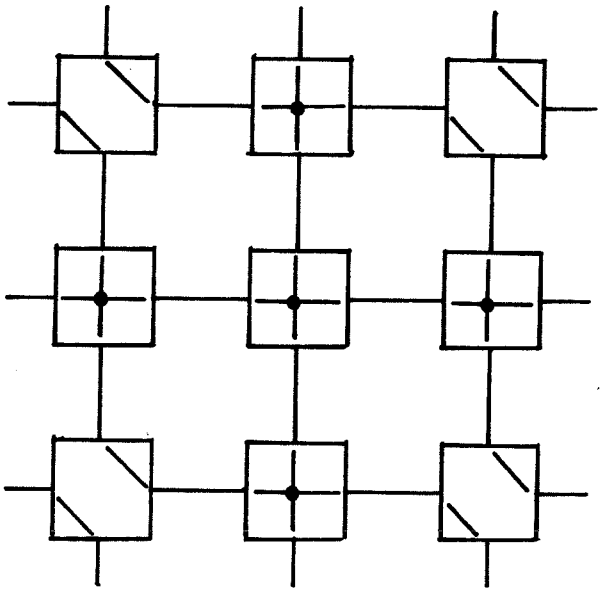


Figure 5.2 Cell Grid

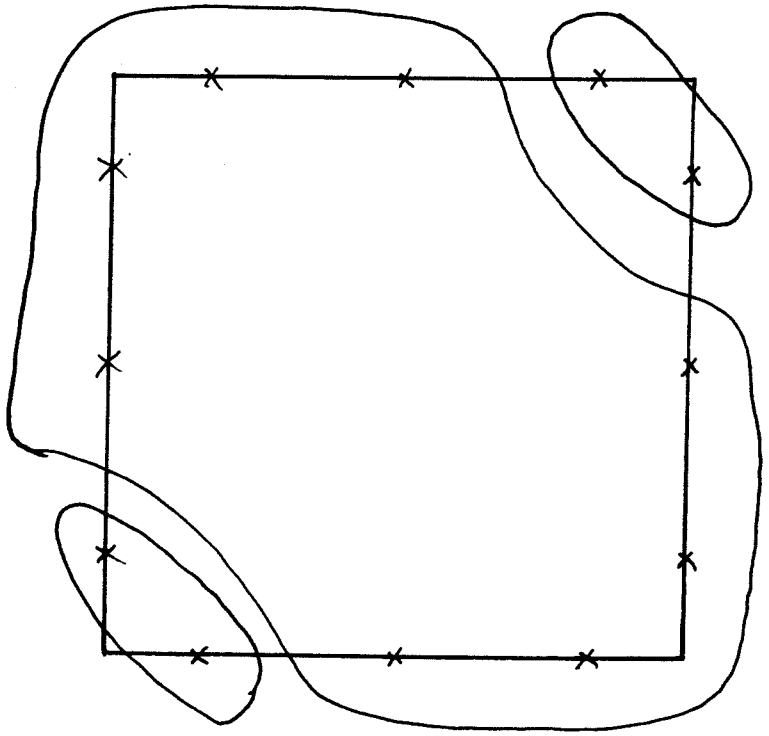


Figure 5.3 Equivalence Classes in Cell GRID

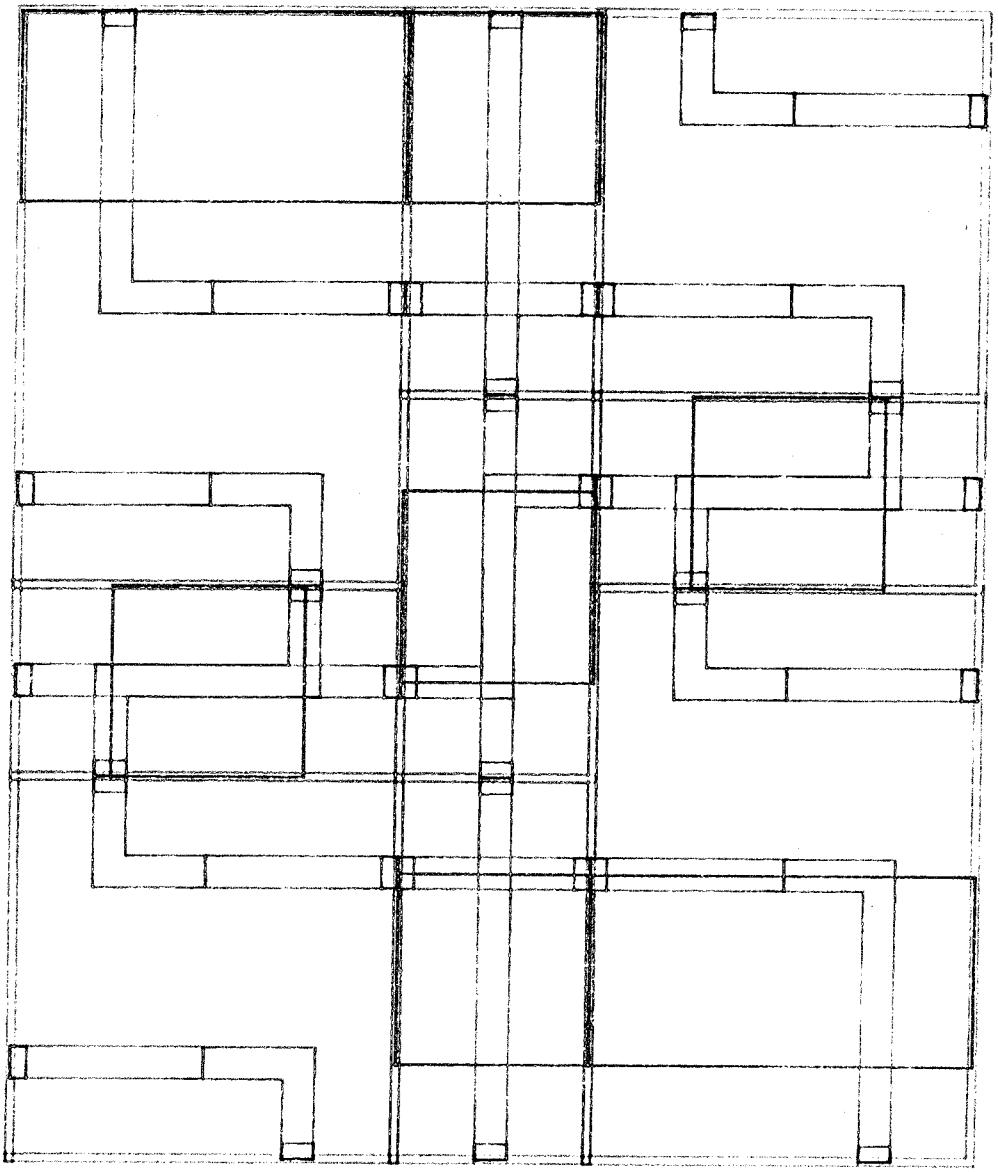


Figure 5.4 Plot of GRID

made a part of the same class. "Any connection at all" means all connectors representing a particular bus, any MexIns that enable drivers to write on that bus, and any MexOuts representing enables that could drive that bus. If there are no buses in the cell, then the equivalence class include those MexOuts enabled by specific MexIns.

As an example of a leaf cell, consider a two output de-multiplexor, or selector. This leaf cell could be the basis for mutually exclusive bus control. The outputs of the cell, the connectors named 0 and 1, are mutually exclusive independent of the inputs to the cell. Independent of the state of the selector, s, only one of the two outputs can be high. The two outputs will not be high at all unless the enable, e, is high. This means that for the outputs to be mutually exclusive with other outputs that might drive the same bus, the enable must be mutually exclusive with those other outputs as well. In short, the MEX equivalence class for this leaf cell consists of 0, 1, and e. The selector, s, is not included because the MEX property is completely independent of its state. Figure 5.5 illustrates the schematic of the selector and indicates the appropriate equivalence class.

Another sample leaf cell is the bus writer itself. Figure 5.6 shows its schematic along with the MEX equivalence class. Except for power and ground, most of the connectors of this cell are MEX equivalent.

The composition rule for propagating equivalence classes up the hierarchy is very similar to that of type wireThru. In fact, with minor modifications the algorithm in Figure 5.1 could be used to find the composition cell equivalence

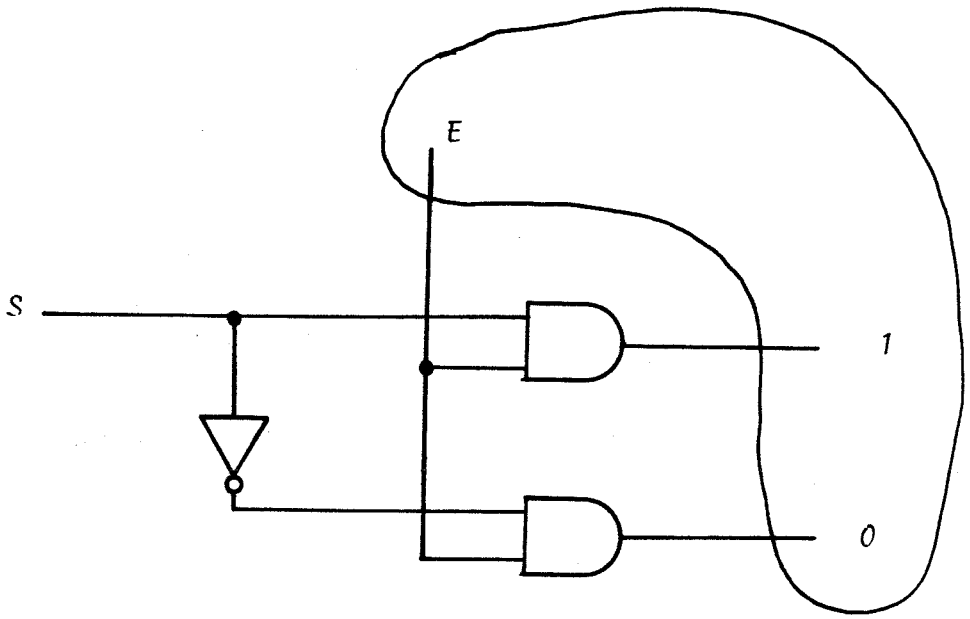


Figure 5.5 Selector Equivalence Class

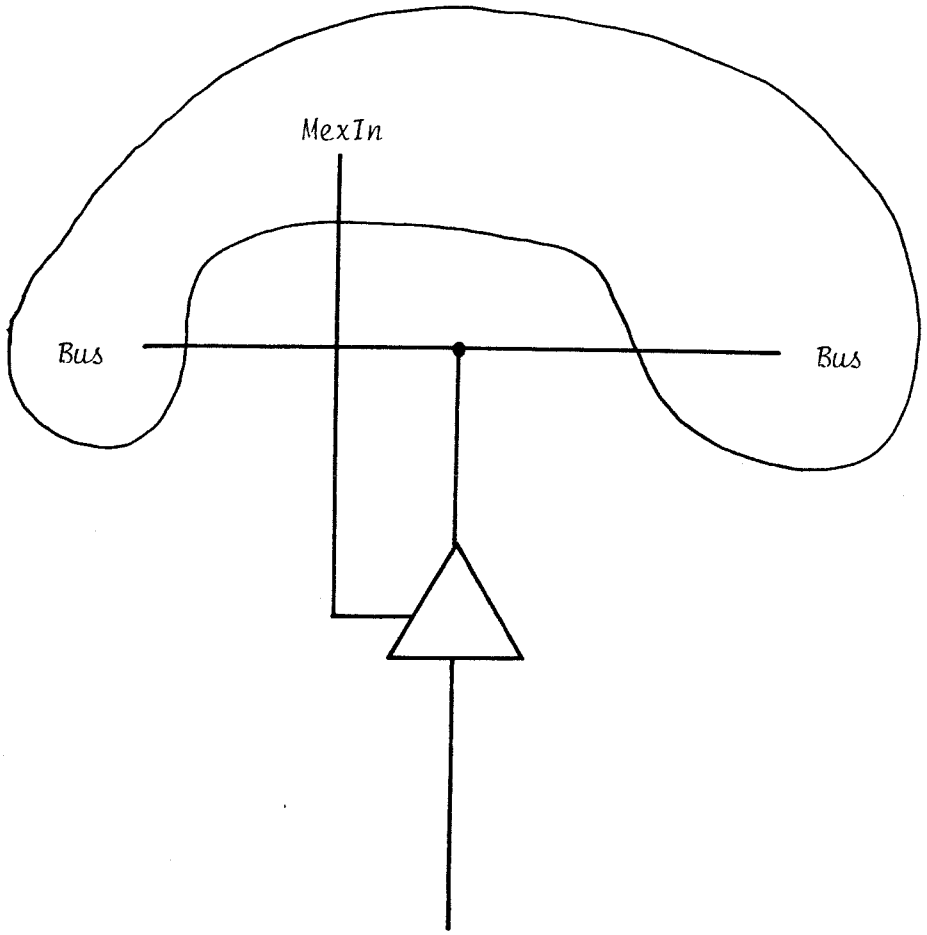


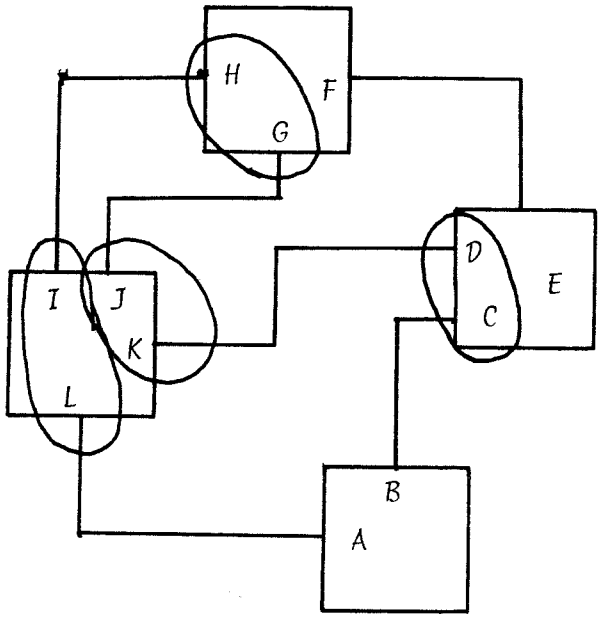
Figure 5.6 Bus Driver

classes. A more elegant view of the composition algorithm is suggested by a graph theory model.

Consider the graph formed by the logical interconnections between equivalence classes on instances of a composition cell. As illustrated in Figure 5.7, this is an undirected, possibly unconnected graph. The transitive closure of a graph is formed by adding an edge between two nodes if there is any other path connecting them. In the transitive closure of the graph of Figure 5.7, an edge would be added between the nodes A and B. The composition algorithm for the MEX typing system builds the equivalence classes that contain the externally visible nodes of the result of the transitive closure of the equivalence class interconnection graph. (Building the equivalence classes in the RL type system can be described in exactly the same way. The algorithm given there illustrates how this might be implemented in a typical programming language.)

Not only do the connectors in a composition cell belong to an equivalence class, but they have to have a type as well. Instead of the funny way the types abstract in the wireThru cell, the type of a composition cell connector is just copied directly from the level below. The only thing that changes is its equivalence class.

An example of a MEX composition cell is a 4 output selector built out of three 2 output selectors. The cell is simply built by hooking the 0 and 1 outputs from one selector into the enables of the other two selectors. This is illustrated in Figure 5.8. All of the other connectors are available externally. The four outputs, labeled 00, 01, 10, and 11 and the enable are all part of the composition



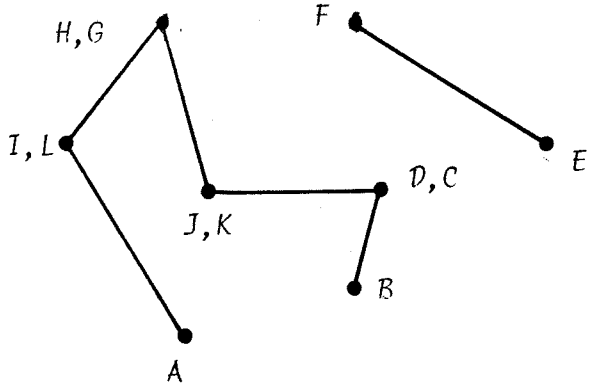


Figure 5.7 Equivalence Graph

cell's equivalence class. Again, the selectors aren't involved. Notice that if the instance A, the selector that enables the other two selectors, is not included in the cell and instead the enables from the remaining two instances are externally available, then the connectors form two equivalence classes. These two classes reflect the independent nature of the internal structure.

To illustrate what happens when a bus is introduced, the second composition example will hook together the 4 output selector and 4 bus drivers. The interconnections of the instances is shown in Figure 5.9. Notice how, despite the large number of MEX equivalent connectors inside the composition cell, only three of the externally available connectors are MEX equivalent. This illustrates the simplifications that functional abstractions can provide, even at higher levels in the hierarchy.

5.4 Conformance and Coercion

In this section, the "horizontal" aspects of typing are discussed. Those algorithms that work entirely within one level of the hierarchy might be termed "horizontal", as opposed to those that propagate information vertically in the tree. The horizontal algorithm in a typing system is the one that verifies that a composition cell represents a legal composition of lower level cells. An illegal composition is one that tries to interconnect non-conforming connectors. The check for conformance between connectors is the central part of this horizontal algorithm.

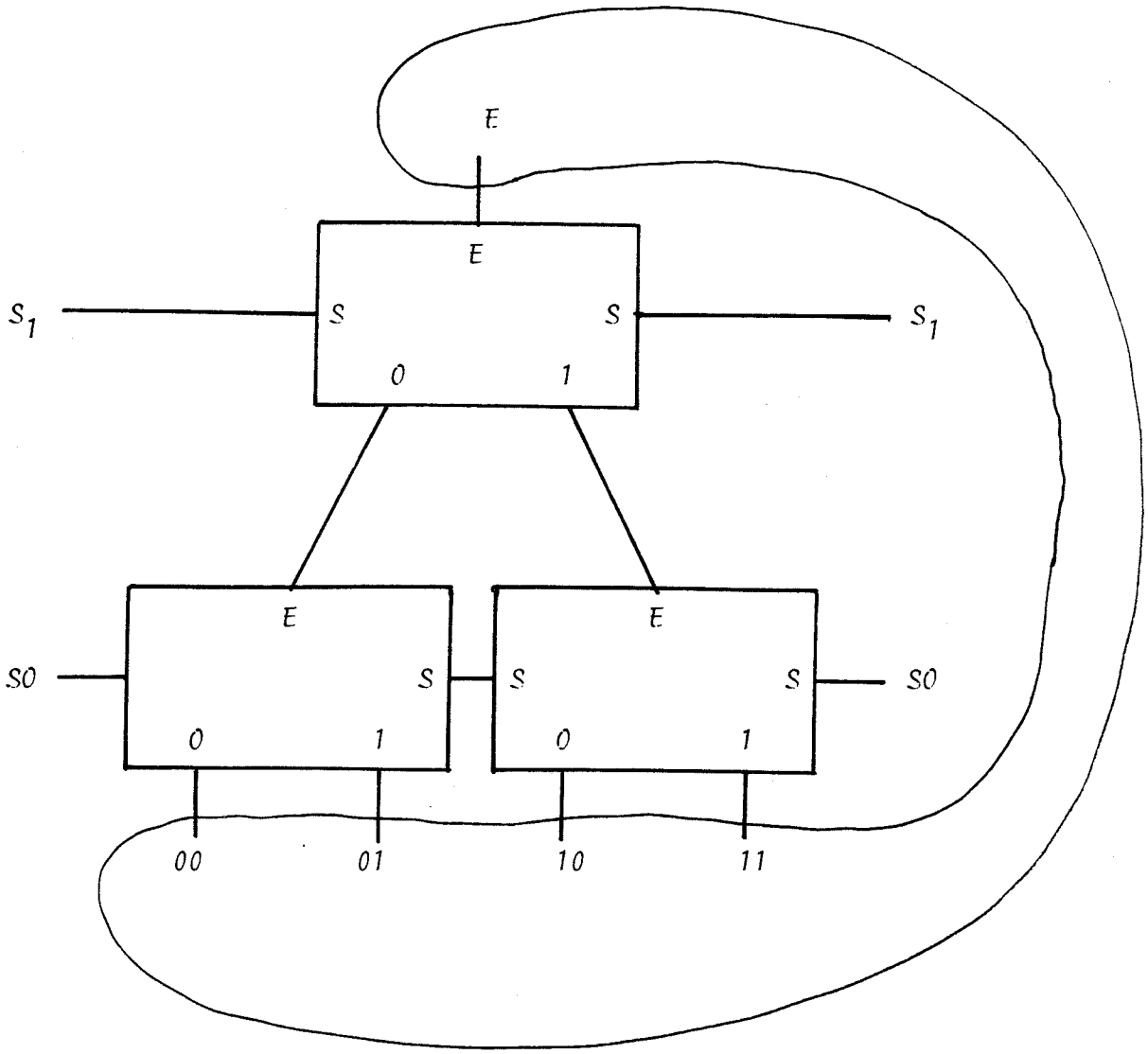


Figure 5.8 4-Output Selector

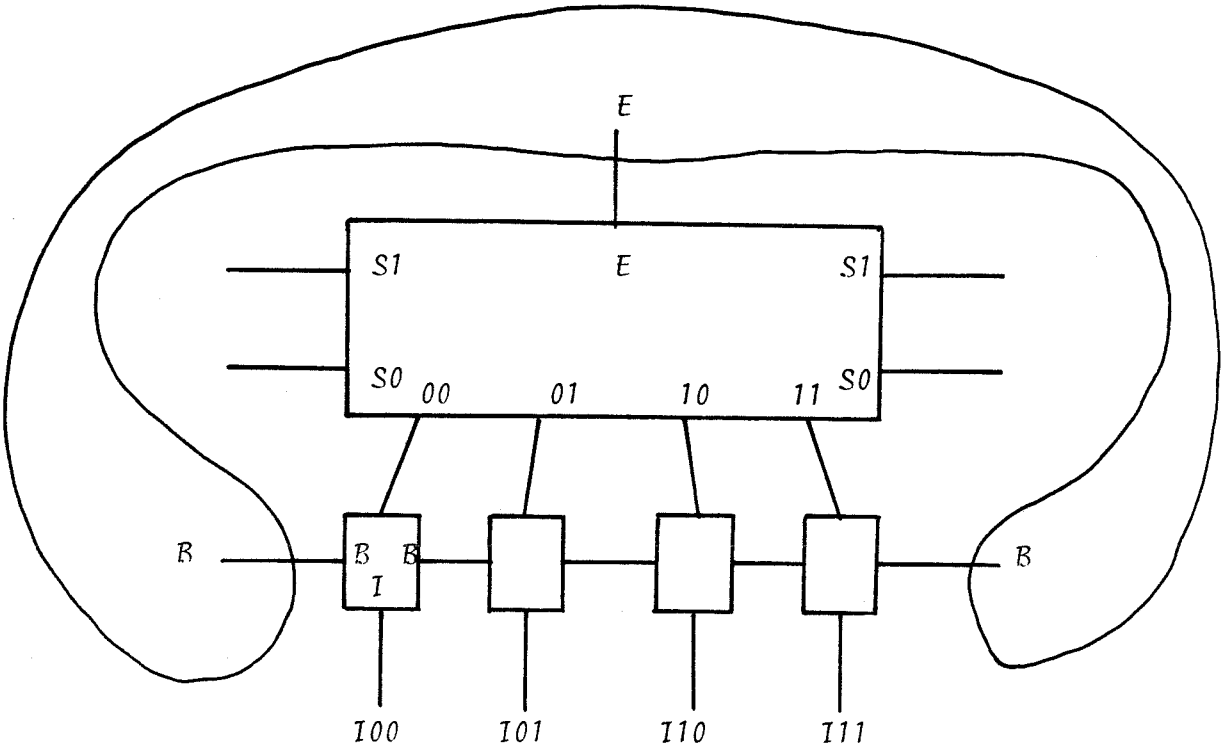


Figure 5.9 Bus and Selector

Restoring Logic

As was mentioned when the RL typing system was presented, two of the three types are easy to check for conformance. Inputs conform to outputs or other inputs, but outputs do not conform to other outputs. The trick with restoring logic is handling the type wireThru.

Fortunately, the transitive closure metaphor is directly applicable here. To check that a wiring cell doesn't violate any of the RL rules is simply a matter of building an internal equivalence class and examining it. The equivalence class should contain all of the internal connectors that transitively connect. This transitive property can be found using a classical transitive closure algorithm [Warshall 1962], or a recursive one as described before. The legality of the interconnection can be checked by simply counting the number of outputs included in that equivalence class. If there is more than one, then it is an illegal composition.

Mutual Exclusion

The MEX typing system is easy to check for legal composition as it stands, since nothing need be done except local checks. Bus type connectors conform only to others of type bus. MEXinputs conform only to MEXoutputs and vice versa.

To simplify this discussion, wiring cells, and thus type wireThru, were not included in the MEX type system. The addition of a MexWireThru type would impact both the vertical propagation and the horizontal checking algorithms

in a way very similar to the way the RL type system handles type wireThru.

Coercions

The topic of coercions is an appropriate one for a section about type conformance since type mismatches are the reason to coerce. Coercions are used to automatically fix up a minor type mismatch. The classic example from programming languages is INTEGER versus REAL. Most programming languages will coerce between these two virtually at will. Some language designers consider this a virtue, some do not.

In a hierarchical system, coercions are used to increase the number of legal compositions by fixing an illegal one. Coercions can often help to make predesigned cells "fit" easily into a larger design, perhaps with different coercions in different places. Coercions in effect make the specification of interfaces a less exacting process. Small, easily fixed misunderstandings between designers on different sides of an interface can be automatically swept under the proverbial rug and allow the larger problems to be attacked.

Just by adding coercions, the locality of a design system can be increased. Even if the design system restricts all propagation effects to a straight line path to the root of the hierarchy, coercions can help by stopping the propagation prematurely. Take a type change at a leaf, for example. In a typical system, the user might have to manually touch all of the composition cells that use that leaf cell, even indirectly. By introducing a coercion that

adjusts for the type change, rippling a change up the hierarchy can be stopped well before the root, perhaps even at the first level above the leaf.

5.5 Some Conclusions

The example typing systems that were presented in this chapter are by no means a complete set of VLSI types. Rather, the abundance of ways of using a type system in VLSI presents a problem of which to do next. Some of the obvious candidates are power, ground, clock phases, various types of control lines, and dynamic versus static storage nodes.

With the realization that there are a lot of different types comes the fear that a type system with many types might become unmanageable. How, with 100 different signal types, do you write the rules for conformance? Do all n^2 possible cases have to be addressed?

Perhaps the type system can be managed through some kind of orthogonalization. Many of the different types of connectors that were mentioned are representative of independent properties. A particular connector might represent a control line that is active on phase 2 of the clock. The fact that it is a control line has little to do with its timing information. The ability to orthogonalize a given type system is going to be important, especially if user defined types are allowed. It may be that types fall into a natural hierarchy, or that they group naturally the way layout geometry orthogonalizes to circuit diagrams. Whatever the method, this seems like a fruitful area for more work.

Chapter 6

The Future with Hierarchies

Previous chapters have presented a separated hierarchy, a mathematic model, and several composition algorithms. The adoption of a separated hierarchy led to a precise mathematical notation of the hierarchy. A geometry composition algorithm was presented that automatically produces the mask descriptions of a system given the mask descriptions of the leaf cells. Two TYPEing systems, and their associated composition algorithms, were presented that identify illegal compositions.

This chapter will draw some conclusions from the preceding chapters and will try to suggest some areas for future research.

6.1 Some Conclusions

Perhaps the most innovative part of this dissertation is the organization of the hierarchy. By separating the composition from the implementation, whole problem areas are eliminated, and simple algorithms exist for at least some of the problems that remain. Since the separated hierarchy lends itself to mathematical analysis, there is some hope that design may become a more formalized

procedure.

The simplification of consistency checks in the separated hierarchy may be its biggest advantage. Each representation of a leaf cell must still be checked for consistency with the others, but analysis of small cells has never been a problem. The problem with consistency checks has always been the sheer amount of computation time required to check an entire chip. With the complexity of VLSI systems rising as it is, no full chip algorithms will be able to survive. Simple computations show that programs like design rule checkers would end up running for years on what will soon be routine designs.

Mathematically Speaking

The mathematical analysis of hierarchies using combinators lends one immediate result: equality between hierarchies is, in general, an undecidable question. Because of the recursive nature of the Y operator, a combinator may represent an infinite system.

However, by reducing the combinator to its "normal form," if any, hierarchical equivalence reduces to a trivial syntactic check. A combinator in normal form can be used to describe finite objects, such as any realizable chip, and yet is easily checked for equivalence to another normal form combinator.

Geometry Algorithms

The conclusions drawn from the geometry composition algorithm presented in Chapter 4 suggest many improvements

needed for its practical application. These improvements include some necessary additions such as rotation and mirroring. One of the "luxury" improvements suggested was to allow connectors to move a certain distance without affecting their neighbors. This freedom of movement will produce smaller layouts without too much effect on the algorithm. Another luxury would be unordered connectors, allowing for "pin swapping" and other automatic optimizations.

Functional Abstraction

The area of functional abstractions seems to be the area needing the most work. An unbiased language for abstracting functional constraints is needed for a generally useful hierarchical design system. No such language is available although specific cases have been analyzed.

It may be that there is no general method for abstracting functions. Each hierarchical algorithm needs something slightly different abstracted from the level below. If there is no general method of abstraction, then that needs to be shown, and an easy way of adding new abstractions needs to be developed. The system described in this thesis does not provide any method for adding new hierarchical algorithms. It should.

Compilation

The ideal symbolized by the term "silicon compilation" is to automatically implement a system, represented in some programming language, in silicon. As discussed in Chapter

4, the Bristle Blocks system does just that for a particular chip architecture. A key question is: What is the relationship between the work presented here and "silicon compilation?"

There are some encouraging signs that suggest that the separated hierarchy approach may lead to a more general silicon compiler. The close relationship between the combinator notation and the lambda calculus suggests a close relationship between combinators and LISP [McCarthy 1965]. A LISP system consists of two types of objects: a pointer pair, and the primitive objects called "atoms." This is very suggestive of the separated hierarchy in that there is a strict separation between objects that "compose," the pointer pair and the composition cell, and objects that just "are," the atoms and leaf cells. A potentially rewarding research area might be to push on this analogy by trying to "compile" LISP programs into silicon.

6.2 Other Problems -- testability, speed, power, etc.

The hierarchical algorithms presented in this thesis represent a very small subset of the algorithms needed for a design system. The geometry and typing algorithms discussed might be the most basic, but may not be the most important to producing a functionally correct design.

One badly needed class of algorithm would produce output compatible with simulators. A wide range of algorithms would be useful. Ouput for a functional simulator would be very useful in the early portions of the design cycle. With a behavioral description included in each leaf cell, a

hierarchical algorithm could produce code that would run in some functional simulator. The algorithm might even generate code in your favorite language which, when compiled and executed, simulated the chip. Another output form might be a switch level simulator like MOSSIM from MIT or a similar simulator developed at Caltech by John Wipfli. These simulators provide the designer with a way to quickly check the logical functional of a small design. No timing or electrical characteristics are involved. Output to a SPICE or MSINC type of circuit simulator would be another useful output for checking out critical paths in a design.

The hierarchical algorithms and structure presented in this thesis have no provisions to increase design testability. By the same token, there is certainly nothing to prevent producing testable designs. Some research on testability in hierarchical designs could well be fruitful. Having a formalism for discussing hierarchies may be an aid to this research.

There is a whole raft of performance measures that are needed in a useful design system. These measures include timing and delay information, power dissipation, and so on. None of these are terribly interesting technical problems, but they are essential to effective design.

The whole issue of performance was ignored in this thesis. The structured design methodology includes a method for structuring the use of time in a system. The use of "semi-static" logic [Mead 1979] or "self-timed" logic [Seitz 1979] separates considerations of system timing from system function. By disallowing time-dependent circuitry, a system can be guaranteed to give the correct result,

albeit somewhat slowly. The time performance of the system can then be optimized independently, gaining 10 to 20 percent. Conversely, a whole new architecture can be explored that may give 10 times the performance.

An interesting research area would be developing a representation, and associated composition rule, that would help designers to guarantee a certain level of performance.

6.3 Optimizing Correct Designs

One of the implications of true hierarchical design as preached in this thesis is a slightly different way of optimization. Independent of the performance characteristic being optimized (speed, power, area, or whatever), traditional design schemes have tended to make the process of optimizing a chip be a long one. In fact, many times the process of optimization is one of complete redesign. When that is the case, the optimizing cycle may be on the order of years with little guarantee that the new version is functionally the same as the old.

With the hierarchy developed in Chapter 2, and the geometry algorithm from Chapter 4, a different optimization strategy presents itself. First of all, the chip is designed in a rather traditional method: a floor plan is made of the top level and the design is partitioned among the design team. Each portion of the chip has a guestimate as to size and position of connectors. Immediately, the geometry algorithm can be run to estimate chip size. Any other hierarchical algorithms can be run to estimate speed, power, and whatever other performance characteristics are needed. As the first cut at each portion of the chip is

finished, it is plugged into the high level description and better performance figures are obtained. When all the first cuts are done, a finished chip is available. That first cut at a chip will usually be terrible, since none of the designers have consulted their neighbors as to cells pitches or connector locations. However, the chip will be functionally correct.

From this point on, the process of chip design will be one of optimization. The design can be checked, automatically, for the worst physical mismatches: those instances that have been stretched the most. Select cells can be redefined to reduce the mismatches. At any point, the design can be declared to be finished. This allows the chip to be released to the marketplace at the earliest possible moment. As the designers continue to optimize, smaller, faster versions can be released.

One of the optimizations that should be available to the designer is the ability to insert an automatically routed wiring cell between instances in a composition cell. As an example of the usefulness of this optimization technique consider a composition cell containing two instances. The cells that are instances each have a number of connectors spaced irregularly along one edge. When the stretching parameters for the two cells are computed by the SLAP algorithm from chapter 6, the top level composition cell could be as much as twice the area it should be. By inserting a cell that does a "river route" between the two instances, all of the parameters revert to zero. Of course, that savings only happens in one dimension, while the other dimension is increased in length.

Key to this whole process is the observation that each optimization can be a fairly quick procedure. Locating the next thing to optimize will be a simple process, whether manual or automatic. Making local redesigns that do not affect functionality but do improve either area or performance are generally simple provided the redesigns are truly local. Making changes that require people to crawl all over the hierarchy are only cost effective if they produce either huge area savings or large speed improvements. Percentage improvements can probably be obtained quickly through local optimizations that require little designer time with a true hierarchical system.

6.4 Small Today, Fast Tomorrow

An option that becomes available in a true hierarchical system allows the designer to "tune" his designs in a nice, intuitive way. Any design is a delicate trade-off involving a number of different cost functions. In VLSI, the cost functions include speed, power, and chip area. A true hierarchical design system would allow changes that affect the trade-offs to be made easily.

An interesting method of designing chips would be to enable each cell to adjust to different trade-off strategies. Then, when the designer decides that the chip is just too big, and that to compensate for it, he is willing to lower the speed, he just changes the trade-off and gets a new chip.

One way of implementing this might be to have each cell parameterized based on a vector of priorities. The

priorities represent a vector in the N-space represented by the N different cost functions. Any design is at a point in this N-space that represents the trade-offs made. Each leaf cell is designed to conform as best it can to the vector. Thus an inverter cell, when given a vector slanted toward performance, would beef up its transistors. The same inverter, given a vector slanted toward low power, might raise the resistance of its pullup. A given cell's ability to conform to the priority vector might be poor, but being able to conform at all will help to customize a design.

The composition cells would also be parameterized with the same priorities, and might pick between several implementations or just pass the vector down the tree.

The ability to investigate different points in the trade-off space would enable designs better tuned to their requirements as well as a single design to be competitive in different marketplaces.

With design aids like this, designing chips might even turn out to be fun!

References

[Applicon 1979]
Applicon Users Manual
Applicon, Inc. Burlington, MA.

[Ayres 1978]
Ayres, R.F.
A Language Processor and a Sample Language
Ph.D. Thesis (#2276)
California Institute of Technology

[Birtwistle 1973]
Birtwistle, G.M., Dahl, O-J, Myhrhaug, B.
and Nygaard, K.
Simula Begin
Petrocelli, New York

[Blum 1967]
Blum, M.
"On the Size of Machines"
Information and Control 11, p. 257-265

[Brooks 1975]
Brooks, F.P., Jr.
The Mythical Man-Month
Addison-Wesley Publishing, Reading MA.

[Burge 1975]
Burge, W.H.
Recursive Programming Techniques
Addison-Wesley Publishing, Reading MA.

[Calma 1979]
GDS II Product Specification
Calma Interactive Graphics Systems
Sunnyvale, CA

[Chen 1977]
Chen, K.A., Feuer, M., Khokhani, K.H.,
Nan, N., and Schmidt, S.
"The Chip Layout Problem: An Automatic Wiring Procedure"
Proceedings of the 14th Design Automation Conference

[Church 1941]
Church, A.
The Calculi of Lambda-Conversion
Princeton University Press, Princeton RI

[Curry 1958]
Curry, H.B. and Feys, R.
Combinatory Logic, Volume I
North-Holland Publishing, Amsterdam

[Dunlop 1978]
Dunlop, A.E.
"SLIP: Symbolic Layout of Integrated Circuits
with Compaction"
Computer-Aided Design, vol. 10, number 6, p. 387-391.

[Fairbairn 1979]
Fairbairn, D.G. and Rowson, J.A. 1979
"Interactive Integrated Circuit Design
on a Small Computer"
Proceedings of 1st Conference on Computer Graphics
in CAD/CAM Systems.

[Gibson 1976]
Gibson, D. and Nance, S.
"SLIC - Symbolic Layout of Integrated Circuits"
Proceedings of the 13th Design Automation Conference

[Hsueh 1979]
Hsueh, M-Y
Symbolic Layout and Compaction of Integrated Circuits
PhD Thesis
University of California at Berkeley
UCB/ERL M 79/80 Memo

[Intel 1974]
Intel Corporation
Santa Clara, CA

[Johannsen 1979]
Johannsen, D. 1979
"Bristle Blocks: A Silicon Compiler"
Proceedings of the 16th Design Automation Conference

[Koestler 1967]
Koestler, A.
The Ghost in the Machine
Chicago, Henry Regency Co.

[Lattin 1979]
Lattin, W
"VLSI Design Methodology: The Problem of the 80's for Microprocessor Design"
Caltech Conference on VLSI, January 1979

[Locanthi 1978]
Locanthi, B.N. Jr. 1978
"LAP: A SIMULA Package for IC Layout"
Computer Science Department Display file #1862
California Institute of Technology

[McCarthy 1965]
McCarthy, J.
LISP 1.5 Programmer's Manual
MIT Press, 1965

[Mead 1979]
Mead, C.A. and Conway, L.
Introduction to VLSI Systems
Addison-Wesley Publishing, Reading MA.

[Miller 1956]
Miller, G.A.
"The magical number seven, plus or minus two: some limits on our capacity for processing information"
Psychology Review, 1956 63, 81-97.

[Naur 1963]
Naur, P.
"The Revised Report on the Algorithmic Language ALGOL 60"
CACM 1/63, p. 1-17

[Nerode 1958]
Nerode, A.
"Linear Automaton Transformations"
Proc. Amer. Math. Soc. 9, p. 541-544

[Persky 1976]
Persky, G., Deutsch, D.N., and Schweikert, D.G. 1976
"LTX - A System for the Directed Automatic Design of LSI Circuits"
Proceedings of the 13th Design Automation Conference

[Rem 1980]
Rem, M. and Mead, C.A.
"A Notation for Designing Restoring Logic Circuitry in CMOS"
(in preparation)

[Rogers 1967]
Rogers, H., Jr.
Theory of Recursive Functions and Effective Computability
McGraw-Hill, New York, NY

[Scott 1975]
Scott, D.
"Data Types as Lattices"
Logic Conference, Kiel 1974
Springer-Verlag Lecture Notes, vol. 499

[Seitz 1979]
Seitz, C.L.
"System Timing"
Chapter 7 of Introduction to VLSI Systems
Addison-Wesley Publishing, Reading MA. [Mead 1979]

[Seitz 1979a]
Seitz, C.L.
"Self-timed VLSI Systems"
Procedure of the Caltech Conference on VLSI
January, 1979

[Simon 1962]
Simon, H.J.
"The Architecture of Complexity"
Procedure of the American Philosophical Society,
vol. 106, no. 6 December 1962.

[Sutherland 1973]
Sutherland, I.E., and Oestreicher, D.R. 1973
"How Big Should a Printed Circuit Board Be?"
IEEE Transactions on Computers, Vol C-22, pp. 537-542.

[Sutherland 1977]
Sutherland, I.E. and Mead, C.A.
"Microelectronics and Computer Science"
Scientific American 237:3, September 1977, p. 210-228

[Turner 1979]
Turner, D.A.
"A New Implementation Technique for
Applicative Languages"
Software-Practice and Experience, vol. 9, p. 31-49

[Warshall 1962]
Warshall, S.
"A Theorem on Boolean Matrices"
J. ACM 9:1, January, 1962, p. 11-12

[Williams 1977]
Williams, J.D.
Sticks -- A New Approach to LSI Design
Masters Thesis
Massachusetts Institute of Technology

[Zilog 1978]
Zilog, Inc.
Cupertino, CA