

The University of Calgary

SHIFT

**A Structured Hierarchical Intermediate
Form for VLSI Design Tools**

by

Breen M. Liblong

A thesis

**submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements for the
degree of Master of Science**

Department of Computer Science

Calgary, Alberta

September, 1984

© Breen M. Liblong, 1984.

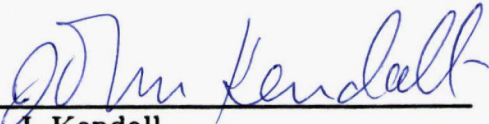
The University Of Calgary

Faculty Of Graduate Studies

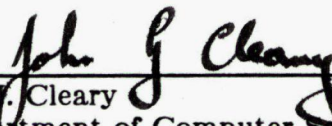
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "A Structured Hierarchical Intermediate Form for VLSI Design Tools" submitted by Breen M. Liblong in partial fulfillment of the requirements for the degree of Master of Science.



Supervisor,
Dr. G. M. Birtwistle
Department of Computer Science



Dr. J. Kendall
Department of Computer Science



Dr. J. Cleary
Department of Computer Science



Dr. K. Kaler
Department of Electrical Engineering

September 10, 1984

Abstract

This thesis examines the problem of increasing complexity in the design of integrated circuits. An analysis of methodologies used in managing this complexity is made and observations are drawn on the requirements for an intermediate form used to capture VLSI designs. While not providing a high level of abstraction directly, an intermediate form provides the framework on which to build tools which deal with designs at a higher level. SHIFT, a **Structured Hierarchical Intermediate Form for VLSI Design Tools**, has been defined and implemented. SHIFT uses a separated hierarchy of leaf cells and composition cells to obtain unified and consistent descriptions of the physical, structural, and behavioural attributes of a design. Leaf cells specify the actual artwork necessary to produce fabrication masks. Composition cells contain compositions of leaf cells and other (simpler) composition cells. Cells are composed by abutting together ports on adjoining walls, stretching them if necessary. Relationships between ports are defined in terms of minimum or exact distance constraints between them. A hierarchical method is used for solving the constraint graphs produced from composition. SHIFT is embedded in the Franz Lisp programming language. SHIFT is a keystone of EDICT [Birt84], a VLSI design tool environment under construction at the University of Calgary. SHIFT is also used in a primitive design library called *shiftlib* which is built on the JADE [Unge84] distributed environment using the *Jipth* [Libl84a] lisp interface to JADE. Shiftlib serves as a prototype for the library assistant envisaged in EDICT.

Acknowledgements

I would like to thank Dr. G. M. Birtwistle for his patience, encouragement and enthusiasm for this project. Thanks are also due to Dr. J. P. Gray, Dr. I. Buchanan, and Tom Melham for their discussions and encouragement.

In addition I would like to acknowledge the many useful discussions I have enjoyed with the graduate students and faculty in the Department of Computer Science at the University of Calgary.

Finally I would like to thank Evelyn Wolfe and our daughter Caitlin for their love and support.

Table of Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Tables	lx
List of Figures	x
 Chapter 1. Introduction	 1
1.1. The Problem	1
1.2. The Nature of VLSI as an Implementation Medium	2
1.3. Structured Design	4
1.4. The Need for an Intermediate Form	5
1.5. Scope and Structure of the Thesis	6
 Chapter 2. The Nature of VLSI	 8
2.1. A Characterization of VLSI Designs	8
2.2. Domains of Description of a VLSI Design	11
2.3. Current VLSI Design Tools	17
2.3.1. The Caltech Design Tools	17
2.3.2. Procedural Design Tools	22
2.4. Summary	28

Chapter 3. Design Methodologies	29
3.1. Structured Design Methodology	29
3.1.1. Regularity	31
3.1.2. Modularity	31
3.1.3. Hierarchy	32
3.1.4. Locality	32
3.2. Hierarchical Design Methodology	33
3.2.1. Leaf Cells	33
3.2.2. Composition Cells	34
3.2.3. Compatibility with Structured Design	35
3.3. Iterative Modelling	35
3.4. Simulation	36
3.5. Current Tools	38
3.6. Conclusion	40
Chapter 4. High Level Intermediate Forms	41
4.1. Intermediate Form Philosophy	42
4.1.1. Leaf and Composition Cells	43
4.1.2. Composition Rules	45
4.1.3. External versus Internal Information in a Cell	47
4.1.4. Design Systems Using a High Level Intermediate Form	48
4.2. SHIFT Design	49
4.2.1. SHIFT Cells	50

4.2.2. Leaf Cells	56
4.2.2.1. Physical Description	57
4.2.2.2. Structural Description	61
4.2.2.3. Behavioural Description	63
4.2.3. Composition Cells	64
4.2.4. Design Instantiation	66
4.2.5. The Composition Algorithm	67
4.2.6. Complexity of The Composition Algorithm	69
4.3. Summary	70
Chapter 5. SHIFT Implementation	71
5.1. Choice of Implementation Language	71
5.2. SHIFT Implementation	72
5.3. SHIFT and Current VLSI Tools	74
5.4. SHIFT and EDICT	75
Chapter 6. Conclusion	78
6.1. Summary	78
6.2. Observations on SHIFT	79
6.3. Future Research Directions	80
References	82
Appendix A. Syntax of SHIFT	88

1. Defining Cells	88
2. Geometry Primitives	90
2.1. Basic Geometry Primitives	90
2.2. Selectors and Predicates	91
2.3. Transformation Functions	91
2.4. nMOS Geometry Functions	92
2.5. CMOS Geometry Functions	92
2.6. Miscellaneous Functions	93
3. Points	93
3.1. Creation, Selection, and Relational Functions	93
3.2. Point Manipulations	93
3.3. Point Transformations	94
4. Instantiation and Selection	94
5. Other Functions	95
Appendix B. SHIFT Examples	96
1. The Shift Register Leaf Cell	96
2. A Shift Register Array	98

List of Tables

Table 1.1. A Comparison of IC Technology Complexity	3
Table 3.1. Tools vs Requirements	39

List of Figures

Figure 2.1. Moore's Law	9
Figure 2.2. NMOS Ramcell	13
Figure 2.3. The OM2 Floorplan	14
Figure 2.4. Representation of a Structural Description of a Selectively Loadable Dynamic Register Cell	15
Figure 2.5. The Scale System	24
Figure 3.1. A 4 by 4 Barrel Shifter	30
Figure 3.2. A Separated Hierarchy	34
Figure 4.1. A Cell Decomposition of the OM2 Datapath	44
Figure 4.2. The Behaviours of a Count Cell and Its Components	46
Figure 4.3(a). Overview of A Leaf Cell Definition	50
Figure 4.3(b). Overview of A Composition Cell Definition	50
Figure 4.4(a). The Ports Definition of a Shift Register Cell	51
Figure 4.4(b). The Ports of a Shift Register Cell	52
Figure 4.5(a). The Constraints Definition of a Shift Register Cell	53
Figure 4.5(b). The Constraints Graph of a Shift Register Cell	54
Figure 4.6(a). The Geometry Definition of a Shift Register Cell	55
Figure 4.6(b). The Geometry of a Shift Register Cell	56
Figure 4.7(a). Wire Connection - Curtailed	58
Figure 4.7(b). Wire Connection - Inflated	58
Figure 4.7(c). T Connection - Curtailed	59

Figure 4.7(d). T Connection - Inflated	59
Figure 4.7(e). Butting Contact - Curtalled	60
Figure 4.7(f). Butting Contact - Inflated	60
Figure 4.8(a). The Structure Definitlon of a Shift Register Cell	62
Figure 4.8(b). The Structure Diagram of a Shift Register Cell	62
Figure 4.9. The Behavlour of a Shift Register Cell	63
Figure 4.10(a). A 2 Element Shift Register Array Composition Cell	65
Figure 4.10(b). Geometry of a 2 Element Shift Register Array	65
Figure 4.11. An Example of Stretching	66
Figure 5.1. The EDICT Design Environment	76
Figure B.1. Geometry of a 4 x 4 Shift Register Array	99

CHAPTER 1

Introduction

1.1. The Problem

Over the past twenty years integrated circuit technology has grown exponentially from being capable of placing tens of devices on a single wafer of silicon to placing hundreds of thousands of devices on a single chip. As a result VLSI designers today are facing a crisis in complexity management not unlike the same crisis faced twenty years ago by software designers.

There is a widening gap between what VLSI technologies are capable of producing and what system designers can design. The designs that most fully utilise the potential of VLSI technologies are memory chips, and only as a result of the highly regular structure inherent in their design. With less regular structures such as microprocessors system designers are having trouble even *completing* designs.

The complexity scale implied by this technology can be visualised with the help of an analogy (see Table 1.1) presented by Charles Seltz of Caltech [Selt79]. Suppose we scale up a typical chip to make the spacing between conductors equal to one city block in size. In this way, the circuit can then be thought of as a multi-level road network carrying electrical signals instead of cars.

In the mid 60's the complexity of a chip was not much bigger than a small town. Most people can carry around a map in their heads of a town

and be able to find their way around without too much difficulty. Similarly designers could manage a design's detail in their heads.

A microprocessor built in the late 1970's using 5 micron technology is comparable in complexity to the entire Los Angeles basin. This would already tax our limits of memory in that only major freeways and avenues would be remembered; the rest would have to be negotiated using maps.

By the time a 1 micron technology is solidly in place (perhaps in as little as two years from now) designing a chip will be equivalent to planning a street network for all of Nevada and California at urban densities. At this point it is beyond our ability even to remember the major freeways; only the overall organisation of the design can be kept in our heads.

If this is extended to the ultimate limits of the technology (about 1/4 micron - [Mead80], Chapter 1), designing a chip will be comparable to designing a street network of urban densities that will cover the North American continent.

The only hope of dealing with such complexity is to find some method of managing it which does not increase in direct proportion to the size of the designs. Thus techniques for structuring designs and design aids that support these techniques must be introduced to realise the potential of this technology and avoid the same mistakes made in software design twenty years ago.

1.2. The Nature of VLSI as an Implementation Medium

VLSI is a new medium for the realisation of computations [Rem81]. Its power springs not from its ability to implement existing engines such as microprocessors, but to implement entirely new architectures directly. VLSI

A Comparison of IC Technology Complexity				
Year	Connector Separation	Scale Factor (for 800m block)	Chip Size (width)	Land Area (width)
1963	25 microns	4×10^6	1 mm	Caltech area 4 km
1978	5 microns	2×10^7	5 mm	Los Angeles 100 km
1985	1 micron	1×10^8	10 mm	California & Nevada 1000 km
19??	1/4 micron	4×10^8	20 mm	North America 8000 km

Table 1.1.

In effect is a highly concurrent realisation medium for computations and allows us to exploit parallelism on a massive scale.

In addition, VLSI is viewed by many as essential for fifth-generation computing efforts and VLSI design tools are required both to design the chips required in these computer systems and to support the experimental designs needed along the way [Wall83].

The domain of VLSI design is very large, spanning from extremely regular and highly space and time optimised designs such as memory chips, to highly irregular designs such as random logic circuits. This thesis will focus

on the design of an intermediate form that is well suited to a large subset of this domain that lies somewhere between the two extremes; that of semi-custom designs. Semi-custom designs can be characterised by the use of regular structures such as PLA's to implement random logic, and by repetition of elements in the design. The primary concern in semi-custom designs is to get a quick and correct implementation of a design.

With a proper design methodology a design can often be optimised after it is designed correctly. What is required is a method of analysing the performance bottlenecks, and then modifying the design to remove them. This requires that a design exhibit characteristics such as locality and modularity. By designing an intermediate form that incorporates these features, a firm foundation is laid for higher level design tools.

1.3. Structured Design

The Caltech "structured design methodology" as introduced by Carver Mead [Mead80] is one approach to system design. It deals with the problem of complex designs by introducing regularity into the system. Even random logic and irregular structure have a regular implementation using PLA and ROM structures. Hierarchical design techniques have been traditionally used to manage complex software systems, and Rowson has extended this design methodology into the IC domain [Rows80]. Specifically, Rowson introduces the concept of a separated hierarchy, where a design can be captured through its description in terms of leaf cells and the hierarchical structure that relates groups of these cells.

If structured design methodology is not incorporated into tools, then an increase in the complexity of the design as measured by the number of

components will dramatically increase the complexity of the overall design. Structured design methodology is a method of combating the combinatorial explosion of complexity of a design.

Further, the use of a structured design methodology in tools allows for consistent incremental design and modification of VLSI specifications. The result is tools which are simpler to develop and modify, since they can exploit the inherent structure of a design, instead of primarily focusing on the use of combinatorially optimal algorithms.

1.4. The Need for an Intermediate Form

Three domains can be recognised as being important in the specification of VLSI designs [Buch80]. These can be termed the *physical*, *structural*, and *behavioural* domains. Some tools are better suited to specifications in one domain over the others. As a result, no one tool exists that is ideal for designing chips; rather a suite of tools is necessary for the complete design of a VLSI circuit. Communication between these tools is greatly enhanced by a consistent representation of the design through the use of an intermediate form.

Further, designs are not arrived at in their totality; they are grown incrementally and modified many times before a satisfactory solution is reached. Some method is needed to force an incremental specification of a design which will remain consistent at each stage. The incorporation of a hierarchical design methodology into an intermediate form supports the incremental specification through stepwise refinement of the solution. It also restricts the effects of modification by localising such changes.

Many VLSI tools use CIF [Spro80] as their intermediate representation. The problem with CIF is that it was only intended to be used as a specification of the layout for the silicon foundry. As a result other descriptions (which were thrown away when the design was forced into CIF format) have to be synthesised from the geometric description by tools like circuit extractors. These descriptions are at best a canonical representation of the original description, and at worst a totally linear description of the original design, without any internal structure.

What is needed in VLSI design systems are tools which support a designer's flair and intuition about a solution to a design problem. They should also report back to the designer any inconsistencies and flaws in the design at the level in which they occur. In order to do this an intermediate form is needed that retains all of the original structure inherent in the design.

1.5. Scope and Structure of the Thesis

This thesis will examine current VLSI design tools and design methodologies in light of the current complexity crisis. An argument is made that an intermediate form capable of capturing the structure inherent in a design is crucial to the design process itself. This is further strengthened by the need for this intermediate form to tie together the various tools that are needed to deal with the design at the many different levels of abstraction. Finally the design of an intermediate form meeting these objectives will be outlined and its implementation and future use will be discussed.

This thesis will not address the more difficult aspect of automating the the design of VLSI circuits, nor will it investigate the modes of reasoning about highly complex IC designs. One must be able to walk before learning

to run.

Chapter 2 will elaborate on the nature of VLSI by characterising VLSI designs and analysing the domains of description inherent in them. It will also discuss how current tools deal with the design process.

Chapter 3 focuses on two design methodologies of interest; the structured design methodology of Carver Mead, and hierarchical design methodology as developed by Rowson. The nature of iterative modelling and the role of simulation in VLSI design are discussed. Finally an examination is made of current design tools and how they meet and fail to meet the criteria developed in chapters 2 and 3. The requirements for an intermediate form are drawn from this analysis.

Chapter 4 discusses the design philosophy behind SHIFT and then proceeds to describe the design of SHIFT in detail. The advantages and the consequences of implementing SHIFT as a distributed process in Lisp are examined.

Finally, chapter 5 focuses on a discussion of SHIFT in the light of experiences in implementing and using it. Some future extensions of this work are also discussed.

CHAPTER 2

The Nature of VLSI

2.1. A Characterization of VLSI Designs

We are now in the midst of a microelectronic revolution which has provided us with the ability to place 100,000 circuit elements on a single chip. Further, as each year goes by, manufacturers are able to put more and more devices on a single piece of silicon. This is shown by Moore's law in Figure 2.1. With the increasing complexity of a chip, the design time also increases at a rapid rate [Moor79].

Working further against the management of complexity is the fact that the current life cycle of a product is approximately five years, of which the design time average, which typically includes about two fabrication cycles, is two years. It now takes two years for a large team to complete a design with one hundred thousand devices; each year the number of devices will double, yet the life cycle and hence the elapsed average design time is expected to be the same. Thus the overwhelming problem that we will face once we scale down to sub-micron structures will be the management of complexity.

There are a number of methods of tackling increasing complexity. These include advances in design which reduce the number of components for a given function, the use of an increasing number of designers for a given project, and the exploitation of design methodologies which attempt to exploit certain properties of the medium and the design itself. These design methodologies include the use of standard parts from cell libraries, and mapping

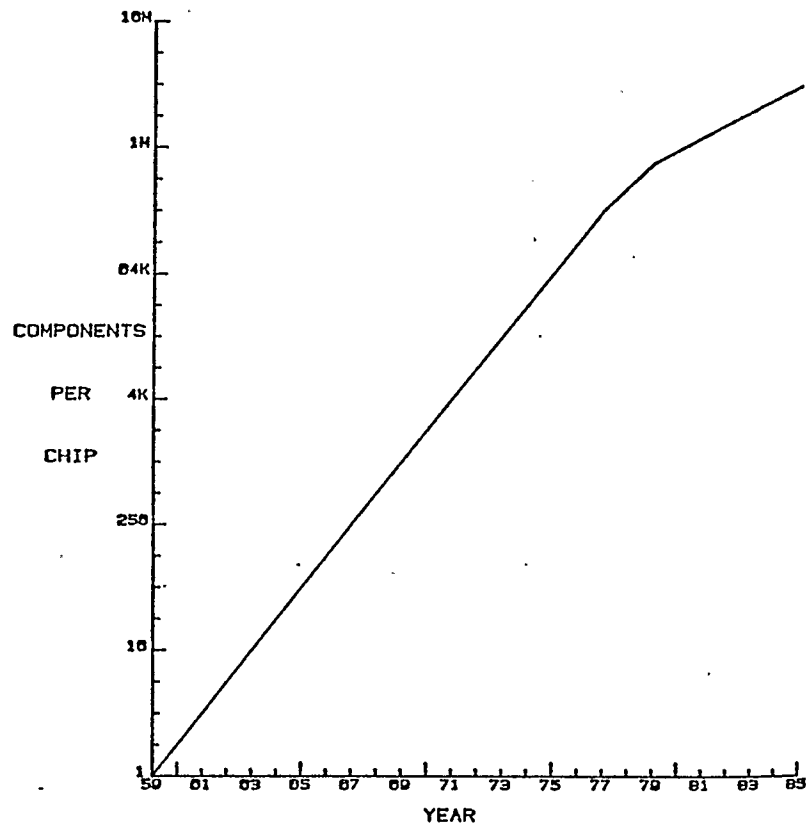


Figure 2.1. Moore's Law

problems into standard architectures, which may be best accomplished with the use of silicon compilers.

There are also various design styles which may be used to express a given function in silicon. Two of the most common in use in industry are the *gate array* technique and the use of *standard cells*.

The gate array approach is currently the most common approach to automated design of custom circuits. In this approach, a two-dimensional array of replicated cells composed of transistors is fabricated to a point just prior to the interconnection levels. A given circuit function is then implemented by customising the connections within each local group of transistors, to define its function as a basic cell, and then by customising the interconnections between cells in the array to define the function of the circuit. Gate arrays are most usefully employed when minimising design time is more important than minimising silicon area.

The problem with this approach is that the structure of the original design is flattened to a single level of interconnect at the silicon surface. This mapping can be both difficult and wasteful, but has the advantage of fast production turnaround. The ratio of circuit density between a structured design and a gate array has been investigated for a small set of chips, with the structured designs winning out by a factor of between 3 : 1 and 6 : 1 [Hell79]. Further, the mapping can only be expected to get worse as designs become more complex, since the management of interconnect becomes ever more important, yet the gate array approach robs us of the ability to manage the interconnect fully.

The standard cell approach refers to a design method where a library of custom-designed cells is used to implement a circuit design. The designer chooses the particular cells needed to implement the function, and specifies the interconnections between them. Thus the designer is freed of having to worry about the details involved in designing cells and can work at a higher level of abstraction. The actual placement of the cells may be manual or

automatic. A problem arises with this approach when no predesigned cells performing the exact function can be found in the library. The designer would then be forced to design a new standard cell that implements the required function, the very situation he hoped to avoid using this approach. Also, since regular interconnect cannot in general be achieved, this design style trades flexibility and silicon area for lower design times and correct design at the cell level.

Both gate arrays and standard cell approaches are seen to be unfit as a design methodology suitable for handling designs capable of fully exploiting sub-micron gate densities. For this reason only the structured design methodology is pursued any further; chapter 3 examines this design methodology in detail. The rest of this chapter will concentrate on the domains of description of a VLSI design and how current tools support these.

2.2. Domains of Description of a VLSI Design

Three domains have been identified to characterize a VLSI design: physical, structural, and behavioural. There exists a hierarchy of description in each of the three domains in a structured design, and it is important to ensure that the descriptions not only be consistent within each domain, but also across domains. Failure to do so can be catastrophic; for example the behavioural and physical descriptions of a design may each be consistent, but when the design comes back from the foundry the observed behaviour is not the desired or predicted behaviour because the specifications are not consistent with each other. One method of ensuring consistency across domains is to unify the description in each of the domains using a single hierarchy. In this section we will define each domain and attempt to show how current

tools address each of these domains of description.

The **physical** domain is concerned with the specification of the physical layout of the integrated circuit via patterns on fabrication masks. These patterns may be defined as boxes, polygons and wires. For example, an NMOS ramcell (see Figure 2.2) in LAP [Loca78] would be defined:

```
define("ramcell");
layer(green);
  wire(4,-1,29).x(ramlen+1);
  wire(2,3,5).x(9).y(15);
  pullup(path(8,14).y(26)).y(29);
  wire(4,11,10).x(16).w(2).y(19).w(4).x(22);
  wire(2,23,15).y(21).x(30);
  pullup(path(24,15).y(8).x(29)).xy(37,17).y(29);
  gb(2,6);
  gb(16,15);
  gb(31,22);
layer(red);
  wire(2,-1,2).x(ramlen+1);
  wire(2,6,2).y(7);
  wire(2,30,2).y(16).xy(27,19).y(23);
  wire(2,22,6).x(12).y(13);
  wire(2,10,23).x(20).y(16);
layer(implant);
  box(28,5,32,13);
layer(metal);
  wire(4,16,-1).y(ramhgt+1);
  wire(3,1.5,-1).y(ramhgt+1);
  wire(3,30.5,-1).y(ramhgt+1);
enddef;
```

where *ramlen* and *ramhgt* are the length and height, respectively, of the ramcell.

In LAP, all primitives are defined in terms of the current layer at the time of definition. *Wire(w,x,y).path* draws a wire of width *w* starting at point (x,y), and continues along the path traced out by successive movements in *x* and/or *y*. The wire is the locus of all points of half-width along the path. *Box(x1,y1,x2,y2)* places a box with diagonals at the corners (x1,y1) and (x2,y2). *Pullup(path)* plants a butting contact at the first point in the path,

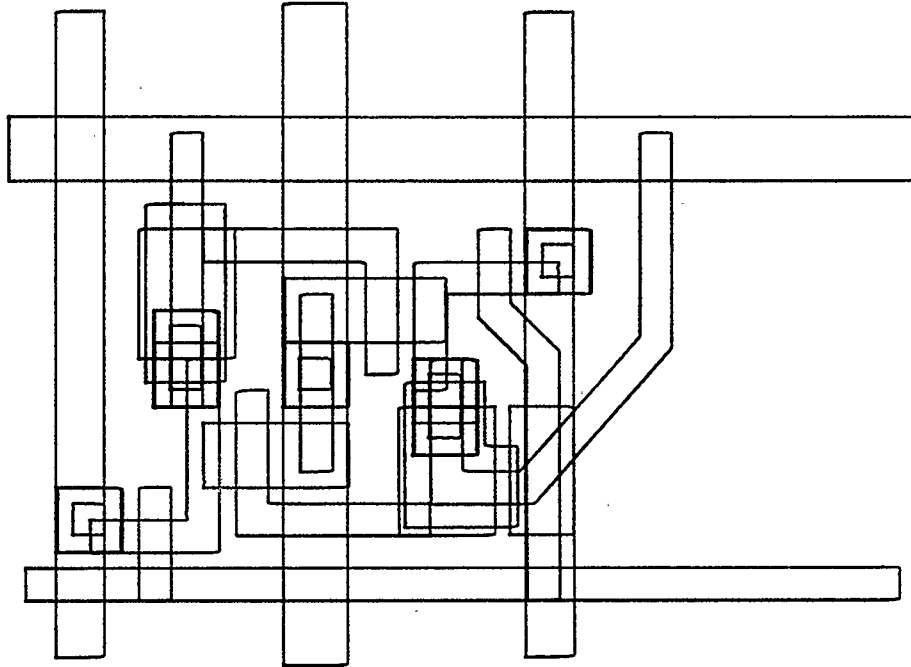


Figure 2.2. NMOS Ramcell

draws a 2-lambda wide depletion transistor along the points of the path, and returns a 2-lambda diffusion wire starting at the path's endpoint. Further, both starting and ending points are assumed to be 2-lambda away from the depletion transistor. $Gb(x,y)$ is a diffusion-metal (green-blue) feedthrough centred at x,y .

One could then define a $[1:x,1:y]$ array of ramcells by :


```

define("ramarray");
for i:= 0 step 1 until x do
  for j := 0 step 1 until y do
    draw("ramcell", i*ramlen, j*ramhgt);
  enddef;

```

At a higher level in the hierarchy of a design the physical description may be specified as abutting areas within a floorplan, with each of these areas enclosing a distinct module (see Figure 2.3).

The **structural** domain is concerned with describing a design in terms of components and connection nets. The components may be primitive components such as transistors or instances of other component blocks [vanC79]. The structural description can be visually represented as a series of boxes or special symbols with interconnecting lines representing the nets such as the selectively loadable dynamic register cell in Figure 2.4. Traditional forms of structural description have been logic diagrams, where the components are gates, multiplexors, etc. and circuit diagrams, where the components are

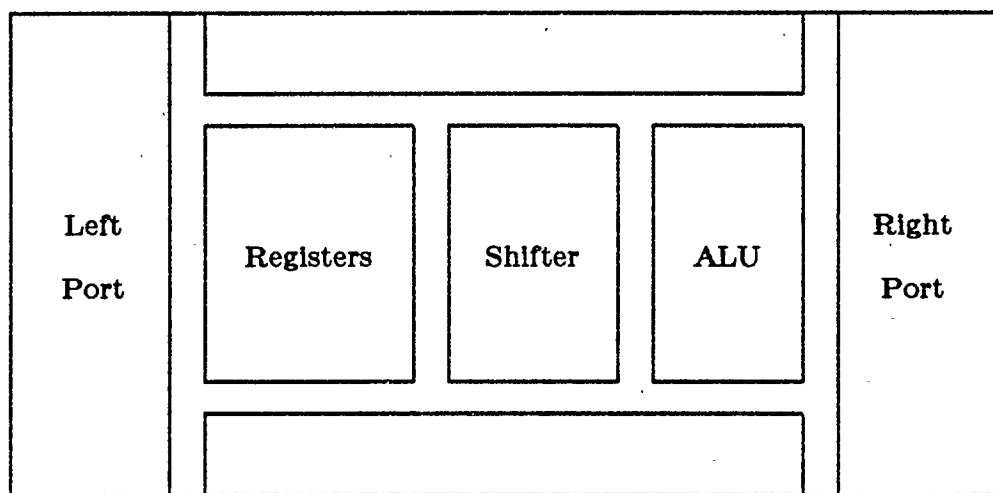


Figure 2.3. The OM2 Floorplan

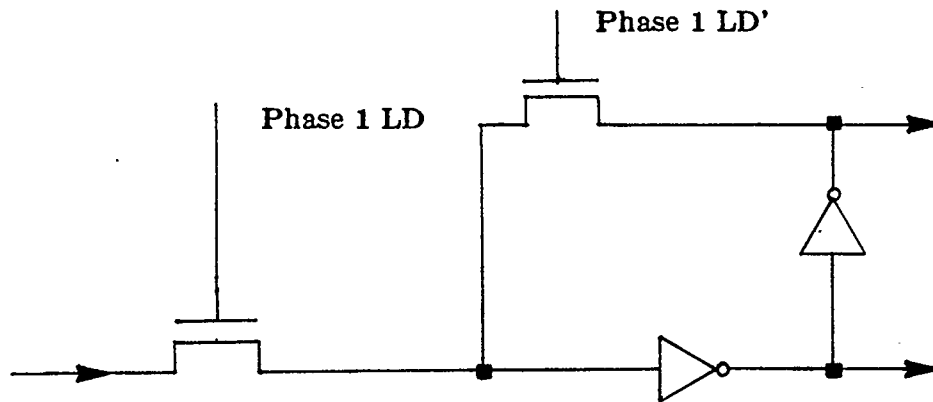


Figure 2.4. Representation of a Structural Description of a Selectively Loadable Dynamic Register Cell

transistors, capacitors, resistors, etc.

The **behavioural** domain describes a design in terms of its function. Some possible behavioural descriptions that have been used are ISPS [Barb81] at the register transfer (RT) level, electrical circuit parameters to be used with SPICE [Nage75], or a functional notation such as Gordon's LSM [Gord81]. An example of a behavioural description using Gordon's notation is that of a counting circuit COUNT, defined by

$$\begin{aligned} \text{COUNT}(n) &== \{sw, in, out\}. \\ &\quad \{out = n\}; \\ &\quad \text{COUNT}(\text{if } sw=1 \text{ then } in \text{ else } n+1) \end{aligned}$$

where COUNT is defined to be the behaviour of a sequential device with input and output lines $\{sw, in, out\}$, and a value for the output line $out = n$. The current state is given by the expression in the left hand side occurrence of COUNT and the next state is given by the expression in the right hand side occurrence of COUNT.

It is also common to describe the behaviour operationally in terms of a programming language such as Simula [Birt73], or through the use of timing diagrams, logic equations, etc. The advantage of a formal system such as Gordon's over a loosely defined operational approach is the ability to compute the composition of behaviours with the use of a composition rule, thereby allowing us to compare derived and specified behaviours.

The goal of computer aided design systems is to control the mapping between the hierarchies in each descriptive domain [Buch80]. An example is the REST system (Richard's Editor for STicks) [Most81], which maps from a structural (and partially physical) representation to a full physical description of a design automatically. Most tools attempt to describe one or at most two of these domains; the other domains being either ignored or specified separately. Unless these domains are specified in an integrated manner, however, inconsistencies among them can easily result in incorrect designs.

Also, since these domains partially overlap it is possible to deduce a description in one domain from the description in the other, and check the consistency between domains. The problem with doing so for any large design is that it is very difficult to map efficiently from one domain to another in an automatic manner, since there is only partial overlap and the mapping may be very complex. Thus REST encourages the user to place cuts in the wires where a wire may be jogged as hints to the compaction algorithm to produce a space-efficient layout.

In the following section we will survey a number of tools in use in both industry and research establishments, and examine them in relation to their descriptive power and consistency among domains.

2.3. Current VLSI Design Tools

Design tools must not only be easy to use; they must be able to handle the complexity of the design, and be able to do so in a consistent manner across all domains. In addition they must also allow for consistency throughout the design cycle. No single tool currently exists which satisfies these conditions. However, it is still instructive to examine a variety of tools in use in light of these criteria.

2.3.1. The Caltech Design Tools

A major influence in the design of VLSI circuits has been the Caltech structured design philosophy and its associated suite of tools [Trim81]. The Caltech structured design philosophy is discussed in greater detail in Chapter 3. It incorporates the idea of a special kind of hierarchy called a separated hierarchy which is composed of leaf cells and composition cells.

LAP. One of the most widely known leaf cell design tools has been LAP. LAP is a Simula package which has primitives for producing geometric specifications of cells. Its standard output is CIF for communication of information to the foundry and geometric design rule checking tools. Although LAP is embedded in Simula and allows the full features of the language to be used, it is still a geometric description design tool. Further, it is a *low level* geometry tool in that most of the LAP primitives have a one to one correspondence with CIF primitives.

REST. REST [Most81] is a leaf cell design system based on symbolic layout techniques of STICKS [Will77]. REST is a graphical design tool and runs on a high-resolution colour display using a mouse to input stick diagrams. Sticks diagrams are both a structural specification technique and a partial geometric

specification technique in that the relative position of wires and transistors are meaningful but it does not provide a full-blown layout.

REST provides consistency between geometric and structural descriptions of a design. It does not, however, provide a behavioural description. In addition it is also limited to the design of leaf cells, and as such, does not provide any means to express hierarchy in a large design. Its output is in Sticks Standard form [Trim80], which is used as input by other composition tools.

PAUL. PAUL is a tool which is similar to LAP. Like LAP, PAUL is embedded in Simula, and is used for designing leaf cells. Its main difference lies in the fact that it outputs Sticks Standard files rather than CIF files. Since Sticks are only a partial geometric specification (the actual size of the transistors can be specified, but the rest of the geometric specification is topological) as well as a structural specification, it becomes easier to design leaf cells that are process independent, using a program that fleshes out the Sticks to a full geometric specification using the appropriate design rules for a given process.

SAM. A fourth Caltech leaf cell tool is SAM [Trim81], which is a single interactive system written in Smalltalk which combines layout language and graphics as input. A user is given two windows which represent the state of the design, one containing a language representation, the other a graphical representation. The user can manipulate either view, and the change is displayed in both. It uses a single underlying representation of the design, thereby ensuring consistency amongst views. Although it allows parameterization and an algorithmic definition of cells, SAM is a geometric description tool for use in defining leaf cells only.

In addition to these leaf cell tools, researchers at Caltech have developed three primary composition tools; Bristle Blocks, SLAP/Earl, and SPAM.

Bristle Blocks. Bristle Blocks [Joha79] is a silicon compiler designed for the construction of datapath chips. A datapath chip consists of data processing elements such as register files, ALU's, and shifters connected by and communicating across data busses. The datapath chip is microcode controlled with each microcode word decoded on chip to drive the individual control lines of each of the processing elements. As an automatic layout system, Bristle Blocks imposes a generic (i.e. template) floorplan in return for ease in automating the layout. This results in the physical floorplan being the same as the structural floorplan.

Bristle Blocks cells are programs rather than data, thus in designing a cell one writes a program which generates the necessary physical description when executed. Bristle Blocks composes cells together by stretching so that cells connect by abutment, and allows the cells to perform computations and participate in the design of the chip. Since the actual mechanics of stretching is left to each cell, which makes local decisions (constrained by its neighbour), the result may be far from optimal.

The input to Bristle Blocks consists of parameterized cell definitions (as programs) and a high level description of the chip, which consists of calls to the cell programs. Bristle Blocks makes the chip by first executing the cell definitions calls, abutting the resulting stretched cells to form the datapath portion of the chip. Additional datapath timing and control information from the description of the chip is used to add control line buffers, parallel load shift registers and instruction decoder to drive the datapath. Finally, Bristle

Blocks adds pads and wiring to create the complete chip.

Bristle Blocks has been enhanced since its original design to allow the insertion of registers for testability, and a more general floorplan which allows multiple processor systems to be compiled. The systems compiled by Bristle Blocks can have circuit densities comparable to hand design.

A Bristle Blocks description does not form a functional description of the chip in that the required procedures only describe how a physical description is to be generated; it does not specify what its behaviour is. Further, Bristle Blocks is best suited to a two level hierarchy; a level composed of cell definitions, and a level composed of the description which calls the cells.

SLAP/Earl. SLAP and Earl [King82] are two implementations of a system closely connected to the separated hierarchy. Both compose geometric descriptions of rectangular leaf cells and other composition cells by superposition of connectors, stretching each cell when necessary. Constraints between connectors are introduced to accomplish minimum separation, producing a directed acyclic graph in each dimension which may be solved independently. Two cells are composed together by composing their graphs. The graph is then solved to produce the co-ordinates used in defining the physical instances of the cells. The graph solution technique used is similar to the one used for Sticks compaction and is based on finding a solution to the constraint graph by finding a topological sort of the nodes in the constraint graph. This algorithm is also used in SHIFT and is discussed in more detail in Chapter 4.

SLAP is embedded in Simula; Earl is an interpretive system with its own list manipulation language. Neither deals with the structural or behavioural domains of description.

SPAM. The final Caltech composition cell we mention is **Structure, Placement, And Modelling (SPAM)**. SPAM is a system that can be used to describe a hierarchical design which can then be simulated at any level of detail. SPAM provides a concise method for describing composition cells. SPAM deals with a structural description of the cells, from which a physical description might be produced using Earl.

The behaviour of a composition cell can also be described. The design can be simulated to any desired level of detail by SPAM by allowing the user to choose which cells are the lowest level of the simulation. The behavioural description of the cell is used instead of the behaviours of its parts.

The structural description is primarily concerned with the specification of the cell connectors. SPAM has typed connectors and these types are used for checking that valid compositions between connectors are performed, i.e. that the power connector of one cell is not connected to the clock connector of the adjoining cell.

SPAM is used to design in a top-down manner. Cells are specified, tested, and then decomposed into smaller cells. When a primitive enough level is reached, the cell is described as a leaf cell. Simulation in SPAM is accomplished by a built-in four-value event and clock driven functional simulator; and is interactive. Once a cell description is compiled, the user may request a documentation workbook consisting of a hierarchical map of the entire circuit, an interface specification diagram for each cell definition, and a floorplan diagram for each composition cell in the description.

Although SPAM integrates the structural and behavioural domains of description, it still requires a separate tool (e.g. Earl) to describe and imple-

ment the physical domain.

2.3.2. Procedural Design Tools

ALI. ALI [Lipt83] is a procedural language for the description of layouts at a conceptual level at which neither sizes or positions (absolute or relative) of layout components may be specified. In ALI a layout is regarded as a collection of rectangular objects (oriented with their sides parallel to the Cartesian co-ordinate axes) and a set of relations that hold among these objects. The ALI programmer specifies a layout by declaring the rectangles and stating the relationships that hold among them.

When executed ALI generates a minimum-area layout that satisfies all the relations between the rectangles specified in the program. It does this by producing a set of linear inequalities involving the corners of the rectangles as variables. These inequalities are then solved to generate the positions and sizes of the boxes. The program also produces connectivity information about the rectangles in the layout, which may be used as input to a switch level simulator. This avoids the usual node extraction analysis.

ALI is built on top of PASCAL, thereby making full use of the programming constructs in that language. Since cells can be specified with the use of procedures, ALI can make use of a hierarchical design methodology to build large chips.

Although ALI is a procedural language, it is capable of describing only the layout of a chip. The behaviours of the design's components are not described in any manner. This makes ALI difficult to use in designing anything other than large leaf cells without an auxiliary tool to describe the behaviour of a design. ALI also suffers from its lack of connecting primitives

(such as contacts) making the programs hard to write and understand, and in the problems resulting from embedding it in PASCAL (i.e. no separate compilation facilities, lack of generic types and dynamic arrays, variant records, etc.).

Finally, ALI fails to exploit the hierarchic structure in generating and solving the set of linear inequalities. An ALI program is run through a filter to generate a standard PASCAL program, which when executed, produces the set of linear inequalities and connectivity relations for the entire design. Since a design can currently be 10 million rectangles (and is growing fast), the solution process, even when the relations are restricted to keep the placement algorithm linear, takes an inordinate amount of time and space. By exploiting the hierarchy of designs, it is possible to reduce the amount of effort in solving the graphs [Ullm84], [Gos183]. We shall examine this approach further in Chapter 4.

Scale. Scale [Buch82] is a procedural language that is considerably more flexible than ALI in describing designs. Scale is not a single language, but a range of special purpose languages covering different ranges of automatic layout generation (see Figure 2.5). Scale programs are written in terms of silicon structures such as wires, contacts and transistors, the primitive objects in the language and the basic building blocks of VLSI circuits.

Scale also provides separate mechanisms for defining separated hierarchy style leaf and composition cells, and procedural language constructs such as scoping and control structures.

All Scale compilers produce a description of a design in an Intermediate Design Language (IDL) format. This then may be used as input to a suite of

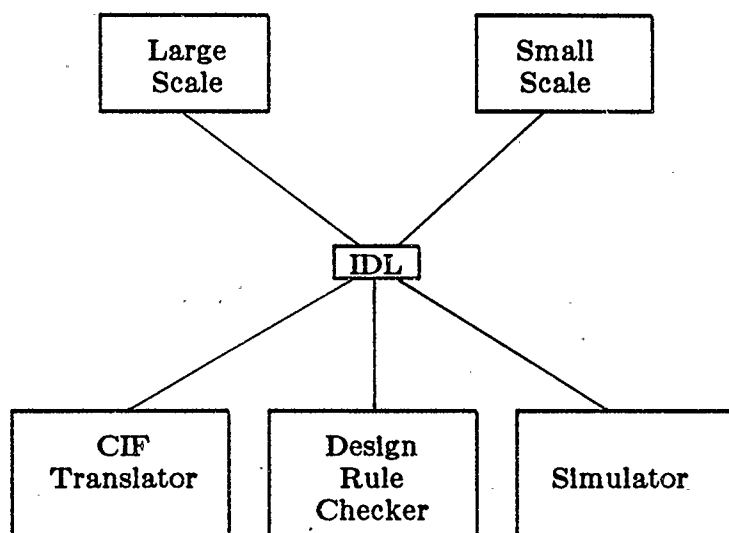


Figure 2.5. The Scale System

utilities such as foundry mask pattern generation translators, design rule checkers and simulators.

The IDL itself is a joint physical and structure representation based on the use of Buchanan's coordinodes [Buch80], which represents the circuit as a graph with paths running on different layers between coordinodes. Coordinodes represent all connection points: between cells, between layers, between components, and even at bends in wires. A complete hierarchy of cells is permitted in IDL, and all cells are stretchable.

In Scale there are three kinds of cells; leaf, composition and artwork. Composition cells are composed only of instances of leaf, artwork or smaller composition cells. All cells are joined together by abutment along adjoining edges. Leaf cells are pure geometric descriptions of designs in terms of contacts, wires, and transistors. Artwork cells allow the designer to work at the

mask level when necessary, for example when designing pads or analogue devices.

Although Scale is at a higher level than LAP or ALI in terms of its descriptive power of the geometric components, it still does not attempt to specify a functional description of a design.

MacPitts. MacPitts is a Lisp based silicon compiler for microprogram sequenced data path designs. MacPitts takes a high-level description of the design in a register transfer language which describes the control and data path parts of a processor. The target architecture for implementing the system is a combination of state machines, one for each of the parallel processes in the code, and a data path unit. MacPitts maps the control part of the specification into Weinberger NOR arrays, and the data part into a rectangular array of registers and logic elements.

The compiler consists of two levels of routines; a higher level which examines the source code and extracts a technology independent intermediate level description of the system in terms of data path specifications, control equations, and state assignments, and a lower level which binds the intermediate level description into an actual mask layout, specified in CIF.

MacPitts has several interesting features. First, it allows a design to be described algorithmically, and derives the physical layout from this, using a predefined target architecture. In our framework of descriptive domains, the functional description is mapped to a standard structural description and then a physical layout is generated from this. Importantly, the MacPitts design system also includes a functional simulator which operates directly on the intermediate level description output from the compiler's technology indepen-

dent component. This makes it possible to functionally simulate designs before the geometry is instantiated.

The MacPltts approach, like many silicon compilers, is only suited to a restricted class of problems, namely those which can easily be cast into the target architecture. Thus to cover the wide spectrum of VLSI design, one would like to have a range of silicon compilers at the designer's fingertips. Currently, MacPltts uses roughly ten times the area for layout compared with a good hand design. This area penalty will diminish for future silicon compilers just as the penalty for software compiler-generated code over hand-tailored code has decreased, and it will improve as software compilers have for similar reasons. Namely, as a result of new knowledge and experience in writing them, and as management of complexity becomes more important in relation to area minimization.

The DPL/Daedalus Design Environment. The DPL/Daedalus design environment is an interactive VLSI design system implemented at the MIT Artificial Intelligence Laboratory [Bata81]. The system consists of several components; a layout language called DPL, an interactive graphics facility (Daedalus), and several special purpose design procedures for constructing complex systems such as PLAs and microprocessor data paths. These tools are all organised around a hierarchical, object oriented database, written in LISP, which contains both the data representing the circuits (the INSTANCES) and the procedures for constructing them (the TYPES).

The Design Procedure Language (DPL) system is a layout language developed at MIT. A designer writes programs in DPL that create and manipulate the database. The user can then query the database to see the results.

A "design procedure" for the layout of a part is typically composed of;

- (1) a type name and parameters (which may have default values),
- (2) a set of constraints among the parameters,
- (3) a collection of other parts which are created as instances of other, previously defined, types,
- (4) and a series of statements which modify the instances in certain ways, such as aligning various parts.

Daedalus is an interactive, graphical interface to the DPL database, and may be thought of as an interactive, graphical programming environment for the DPL language. In Daedalus, the user is able to express any information either symbolically by typing an expression or DPL code, or graphically by pointing with a mouse. One may also make changes to a design either graphically or by editing the DPL code directly.

The DPL/Daedalus environment is concerned primarily with the physical and structural descriptions of a design. It is, however, a very good layout tool in that it supports an incremental design philosophy, and is embedded in a highly interactive Lisp programming environment.

Palladio. Palladio [Brow83] is a circuit design environment for experimenting with design methodologies and knowledge-based, expert-system design aids. Palladio includes facilities for defining models of circuit structure or behaviour, called perspectives. These perspectives are used to create and refine circuit specifications, and can include composition rules that constrain how circuit components may be combined to form more complex components.

Palladio's integrated design environment provides menu-driven, graphics interfaces for editing and displaying structural perspectives of circuits and a behavioural language with associated editor for specifying a design from a behavioural perspective. In addition, a generic, event driven behavioural simulator can simulate a circuit specified from any behavioural perspective and can also perform hierarchical and mixed-perspective simulation.

The design paradigm supported by Palladio is an incremental refinement of design specifications, with periodic validation of the specifications by simulation. Palladio allows multiple structural and behavioural perspectives, which do not necessarily follow the same partitions in the hierarchy of decomposition. While this gives flexibility in the freedom to explore different design strategies, it can lead to consistency problems between hierarchies. Since Palladio was designed more as an experimental tool for exploring the design process, and expert systems for circuit design, this flexibility is warranted. However most designers will benefit when a more rigid structure is imposed in a mature circuit design environment.

2.4. Summary

This chapter has presented a broad overview of the nature of VLSI as an implementation medium, and has examined current tools to show some approaches to designing custom silicon. In the next chapter the structured design and structured hierarchy methodologies are examined further, and an analysis is presented of how these tools incorporate these methodologies. Finally, from this the requirements for an intermediate form are drawn.

CHAPTER 3

Design Methodologies

VLSI designs are large and complex; current VLSI designs involve upward of 100,000 transistors and many display highly concurrent activities. Computer science has faced many of the same problems in the correct construction of large and complex software. Some of the lessons learned can be borrowed; the most important being the use of design methodologies to develop large and complex structures. In this chapter we examine the structured design methodology of Mead [Mead80] and Buchanan [Buch80]; the hierarchical design methodology of Rowson [Rows80]; and examine how iterative modelling and simulation fit in. Finally we shall show how current tools fare with respect to these requirements.

3.1. Structured Design Methodology

Structured design [Mead80] emphasizes the principles of top-down, hierarchical, modular design techniques. Unlike LSI, where circuit density is the major constraint in a design the major constraint in VLSI is the wiring between functional blocks. Mead states that a reasonable estimate of the size of a design in VLSI is just the area needed for routing control and data. Random wiring, like random city roads, consumes silicon area, and destroys the regularity and locality of a design. By destroying regularity, design modification is made extremely difficult and time consuming. Further, the loss of locality of function makes design verification much harder to achieve. Finally, the length of a wire determines how much energy and time is needed

to transmit a piece of data. Thus designs with lots of global wires either consume a lot of energy, or are slow, depending on what the designers critical constraints are.

It follows that a design should be optimized by placement of functions on the two dimensional surface based on the amount of intercommunication. Mead has shown that if wiring can be managed, the circuitry usually presents little or no additional cost. This means that the primary emphasis in design is on communication flow, rather than computation. An example is the barrel shifter of the OM-2 data path, where the logic fits completely under the wires needed to move the data and control. Thus data computation becomes incidental to data communication.

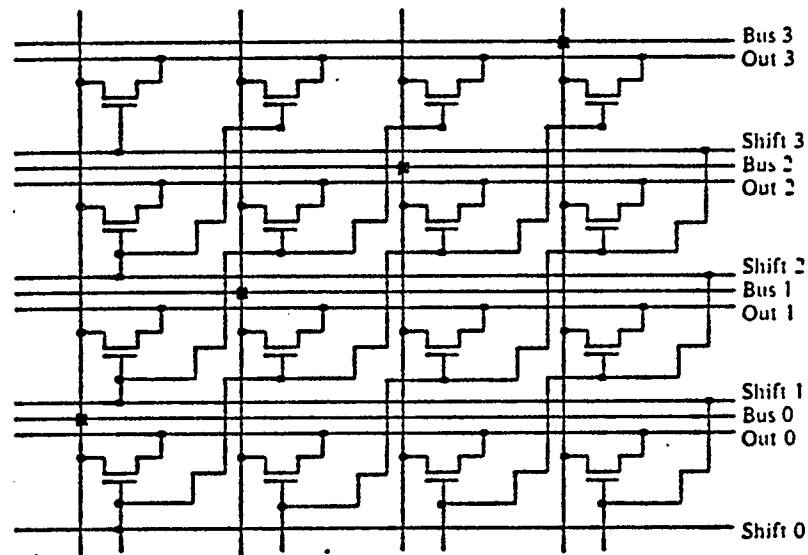


Figure 3.1. A 4 by 4 Barrel Shifter

3.1.1. Regularity

Regularity in design is desirable because it reduces the complexity of the problem. Regularity involves a number of factors. One of these is the ability of cells to tessellate in 2-space as a result of regular interconnection strategies (e.g. with two independent layers of interconnect we can run power and data orthogonal to control signals). Cells can then be connected by abutting together along their boundaries. Regularity in programming involves tackling similar problems with similar approaches. Regularity in a VLSI design may also be exploited by designing a data path in a bit slice approach and then replicating the slice.

In addition to the regularity of interconnect, specification of cells of identical pitch (i.e. same size along their interconnect boundary) promotes connection by stretching and abutting cells together. In contrast, the standard cell approach is to compose cells by placement followed by routing. Any wastage of area from stretching at the lower levels is made up by the removal of random interconnect paths. Informal estimates of area gain using this approach is around 20% over small areas [Buch80].

3.1.2. Modularity

Modularity makes it easier to partition a design among a group of designers by presenting each module with a well-defined function and interface. This enables designers to work on a design in a more independent manner, which will tend to decrease design time. It also is a powerful tool in the control of complexity of the design. Buchanan [Buch80] makes an analogy between the restriction in structured programming to the three flow control constructs of concatenation, conditional selection and iteration, and to the

restriction in structured design to the use of the constructs of cell abutment, PLAs, ROMs and other conditional control structures, and one and two dimensional arrays of cells. Design verification and simulation is also made easier by the use of modularity.

3.1.3. Hierarchy

Different levels of the hierarchy correspond to different levels of granularity of function. By partitioning a design in a modular and hierarchical manner, the designer is able to abstract to the level of detail desired. In bottom-up structured programming, larger structures are built from smaller structures by the use of the control structures described above. These then in turn may be used to build even larger structures, and so on up the hierarchy, until a complete design is realised.

Alternatively, in a top-down structured programming approach, the hierarchical strategy maps functional modules onto predetermined partitions on the chip (the floor-plan). In a like manner, these modules then may be decomposed into their components until some point is reached where the primitive components may be directly mapped onto their portion of the surface.

3.1.4. Locality

At any level, a design can be modularized such that the module must communicate through a well-defined external interface and internal components are hidden from the outside. As a result, the internal functionality of the module is localized and is not affected by changes to other modules. This allows the designer to abstract the details of the design at any level desired

without having to carry with him some detailed knowledge from the global level. An example of this is the structured design guideline that buses should be distributed (i.e. run through the module) and not global. This rule also works to increase design density, since global wiring is an expensive process in terms of the amount of silicon surface used. In software global information detracts from locality because it is always subject to misinterpretation by different code modules.

The principle of locality also aids verification in that any properties that a module has is shared by all instances of it. Therefore only one instance needs to be verified.

3.2. Hierarchical Design Methodology

A design can always have some hierarchical structure imposed on it. There are many hierarchies of description of a design. An extreme form discussed by Rowson [Rows80] that forms the basis of hierarchical design methodology is the *separated hierarchy*.

A separated hierarchy consists of two kinds of cells; *leaf* cells and *composition* cells. A leaf cell is atomic, it has no internal hierarchical structure. A composition cell is composed purely of instances of other leaf and composition cells interconnected in some manner. The separated hierarchy completely separates the leaf cells from the composition cells.

3.2.1. Leaf Cells

Leaf cells may be instantiated at any level in the hierarchy. A leaf cell may have multiple representations. For example, it may have a geometric representation consisting of polygonal shapes on mask layers, or a logic circuit

representation of logic elements and their interconnections. Typical sizes for geometric representations of leaf cells range up to 100 transistors. Only instances of leaf cells have any "data" (i.e. functionality) in them.

Leaf cells are important for their function, not their implementation. In analogy to software, leaf cells are the basic semantic units which may be used by the designer. Alternatively, if a hierarchical design can be thought of as a theorem in an axiomatic system, leaf cells form the axioms used in the derivation of the theorem.

3.2.2. Composition Cells

In contrast, composition cells are implementation independent and describe only the functionless logical interconnection of instances of leaf cells and other composition cells (i.e. composition cells merely structure the "data"). The composition cells may be thought of as theorems in an axiomatic system. Composition rules (i.e. the interconnection mechanism) are implementation dependent and are analogues of the rules used to construct

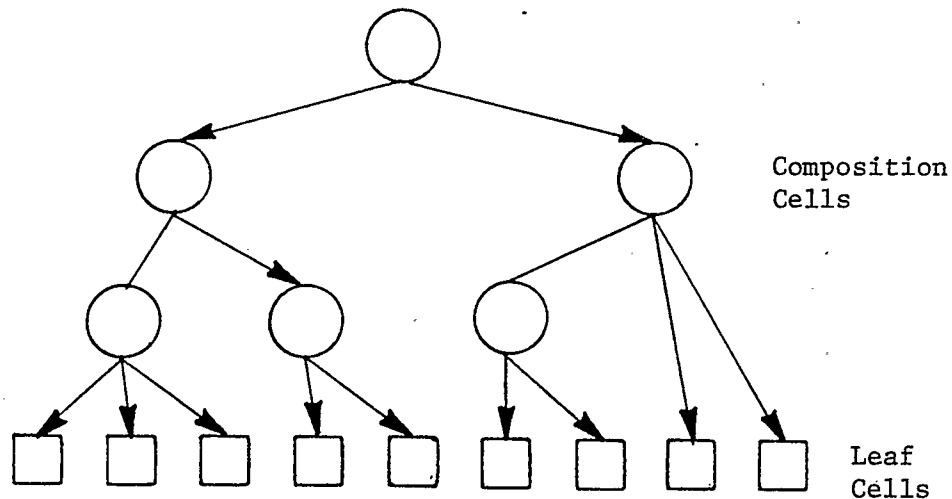


Figure 3.2. A Separated Hierarchy

theorems. There is one composition rule for each leaf cell representation domain.

The separated hierarchy allows a designer to produce a design that is correct by construction. That is, given a property P and a composition rule R that preserves the property P , if submodules having P are composed according to R , the composition will have property P . This makes consistency checking easier, since only the composition rule and leaf cell representation need to be checked for each domain of description. By separating out the hierarchical structure of a design from its actual representation, it is possible to make statements about the structure alone. Rowson introduces a formal method of proving the equivalence of hierarchies, given composition from the same leaf cells.

3.2.3. Compatibility with Structured Design

Being a special case of hierarchies, separated hierarchies preserve the properties of regularity, modularity, and locality. Regularity is enhanced because the necessary property of functional abstraction is embodied in the composition rules. Modularity is promoted since interconnection of cells by composition means two cells can only compose if they have common interconnections. Locality is guaranteed by the composition rule, since it may not introduce new functionality into the resultant cell.

3.3. Iterative Modelling

A structured design is not concocted out of thin air; it needs to grow from an initial idea to a fully specified design. Often the designer has very little insight into the implications of a decision made early on in the gross

decomposition of a design. As in software design, the experience and intuition of a designer can often guide the development of a design through these critical choices in a manner that no fully automated approach can now equal. The designer's intuition can be further enhanced by the use of a design methodology such as top-down, bottom-up or structured growth (a methodology familiar to Lisp programmers) [Sand79].

Experience with software has shown that it is difficult to get a design right the first time. In fact, many iterations are often needed over the design before it is correct. If a design is not structured in a hierarchical and modular fashion it is extremely difficult to isolate the error at the appropriate level of abstraction and correct it. A design that contains global wires or random logic may require an large amount of rearrangement to accommodate the fix which in turn may lead to new errors.

Even after producing a correct design, a good designer will want to optimize the design over time or space where prudent. Modularity makes it easier to isolate and remove bottlenecks from a structured design without introducing new errors.

With structured hierarchical designs the designer can propose modifications and evaluate the consequences without actually having to implement the changes in detail. This "what if.." approach allows the designer to heavily restrict the domain of all possible alternative designs. It also requires that structures be modular at any level of abstraction.

3.4. Simulation

The most important use of simulation is in verifying the correctness of a design according to some specification of its input and output behaviour.

This is more necessary in VLSI design than in software design since it is time consuming and expensive to test the design directly. It is also a valuable tool in the development of a structured design since it can be used to verify the correctness of any component without worrying about side effects.

Just as there are many levels of abstraction of a structured design, there are many levels of granularity of a simulation of a design's behaviour. The most common simulators used are circuit-level simulators like SPICE [Nage75] and switch-level simulators like MOSSIM [Brya81]. Both of these simulators operate at only one level of the hierarchy. While capable of giving an extremely accurate description of a device's behaviour, SPICE is very impractical to operate for a circuit containing more than, say, a hundred transistors. It is also very difficult to reason about the cause of the behaviour exhibited by a SPICE simulation because of the low level of abstraction (or high level of detail).

MOSSIM is a level of abstraction above SPICE since it simulates a switch level model of a circuit. This allows more complex designs to be practically simulated, at the cost of discretizing the signal behaviour. MOSSIM can also be difficult to interpret, since it still deals with a design at the switch level of abstraction.

Ideally we should be able to simulate a design at any level of abstraction. This would allow the designer to draw conclusions about the correctness of a module at the same level that it is simulated. The result is that errors are more quickly and easily pinpointed.

A hierarchical design methodology supports the latter approach by allowing behaviours at a given level of the hierarchy to be simulated from the

behaviours of its components. A structured design methodology also encourages this approach since the modularity of structures allows simulated behaviours to be easily composed.

The use of a separated hierarchy also allows behaviours to be composed at different levels. Thus, once a module has been verified as correct, a simulation of a composition cell using that component only deals with the component's behaviour at that level of abstraction, and not in terms of any sub-component behaviours. Like design rule checking, this is another example of hierarchies making consistency checking easier.

3.5. Current Tools

Many design tools currently in use implicitly recognize the necessity of a design methodology. Most support some of the criteria laid down for these methodologies; few support all. Table 3.1 shows a variety of tools and which requirements each meet. Tools that actively support a discipline are indicated by a "+," those allowing a discipline to be exercised in conjunction with them are indicated by a "*", and those which do not allow such a discipline to be exercised are indicated by a "-." The domains each tool deals with are indicated by **Geometric**, **Structural**, and **Behavioural**.

The earlier tools which deal with a single domain, such as Caesar and Spice, are inherently global and only deal with a single domain of description. The more recent tools have recognized the complexity problem, and have attempted to deal with it by either allowing or directly supporting design methodologies that deal with complexity. Examples of this are Mossim, which allows the user to black box any portion of the design by writing a module which exhibits the desired behaviour, and DRCFIL [Whit81], a hierarchical

Tools vs Requirements						
Tools	Regul- arity	Modul- arity	Hier- archy	Local- ity	Iterative Modeling	Domains
Lap	+	*	+	*	*	G
Rlap	+	+	+	+	*	G
CIF	+	*	+	*	*	G
REST	*	+	-	+	+	G,S
SPAM	+	+	+	+	+	S,B
Caesar	*	-	-	-	-	G
Moslim	*	*	-	*	*	S,B
Spice	-	*	-	-	-	S
Slap	+	+	+	+	*	G
Earl	+	+	+	+	*	G
MacPitts	+	+	+	+	+	G,B
Bristle-Blocks	+	+	*	+	*	G,B
Scale	+	+	+	+	+	S,G
DRCFIL	+	+	+	+	*	G
ALI	+	*	*	*	*	G
DPL	+	*	+	*	*	G

Table 3.1.

design rule checker, which uses hierarchic information to minimize the amount of checking needed for a design.

The success of silicon compilers like MacPitts and Bristle Blocks¹ is also partly due to using a design strategy (i.e. the datapath design style) that is inherently hierarchical and modular in its organization. Finally, the most recent tools have tended to support not only hierarchy, but also modularity and locality, through the use of boundaries on cells with connections allowed

¹ Bristle Blocks was recently used by a three-man team to generate the 37,000-transistor datapath chip for the MicroVAX in only seven months [John84].

only on the boundary wall. These include such tools as REST and Scale. SHIFT continues in this tradition.

3.6. Conclusion

This chapter has examined the structured design methodology of Mead and Buchanan, paying attention to the key aspects of regularity, modularity, hierarchy, and locality. We have also examined the hierarchical design methodology of Rowson, showing how a separated hierarchy can deal with complexity, and how it fits in with structured design methodology. In addition, the role of iterative modelling in design evolution was investigated. Finally we examined how current tools fared with respect to these requirements.

In the remainder of the thesis we focus on the requirements of an intermediate form for VLSI design tools which work in an integrated environment, and support both a separated hierarchy and a structured design methodology. A structured intermediate form for VLSI design tools called SHIFT will be defined, and the algorithm it uses for composing cells in a stretchable manner along adjoining ports will be detailed. Finally, we show how SHIFT fits into future VLSI design environments.

CHAPTER 4

High Level Intermediate Forms

An intermediate form is a half-way house between analysis and synthesis. As a result a good intermediate form must carefully balance both of these objectives. This requires a thorough knowledge of the sources that use (map to) the intermediate form, and the targets that are mapped from the intermediate form. Ideally, what the designer produces at a high level of abstraction should be clearly reflected in the layout of the design. It is also important that any unintended effects of the specification be clearly traceable to its cause at the designer's level of abstraction.

Therefore, if an intermediate form is used it should clearly reflect any constraints the designer imposes on the resulting layout. It is just as important that the intermediate form should allow any errors detected in the layout (such as design rule violations) to be expressed to the designer at the level of abstraction that he was using. The degree of success in meeting both of these objectives in the intermediate form determines both the amount and ease of control the designer has over the mapping of the design to the target domain.

If the sources and targets are predetermined, as for example in a silicon compiler with a restricted target architecture and a restricted application range, we may design the intermediate form to be complete. However, if the sources and targets are not predetermined, then a reasonable guess about the design criteria for the intermediate form has to be made.

In the previous chapter, we examined the design methodologies of Mead and Buchanan, and Rowson, and showed how they promote iterative design and support specification, verification and simulation. Various tools were examined in terms of these criteria, and found wanting. In this chapter we will elaborate on the essential requirements for an intermediate form, and specify the design of SHIFT (a **Structured Hierarchical Intermediate Form for VLSI Design Tools**) in terms of these. The implementation of SHIFT is discussed in chapter 5.

4.1. Intermediate Form Philosophy

The purpose of an intermediate form is to act as a vehicle for the specification of an IC design in a design system composed of a number of tools. These tools are of a diverse nature, consisting of graphical and procedural tools for specifying a design in the intermediate form, and tools which use the intermediate form as input, such as circuit extractors, design rule checkers, logic simulators, artwork plotters, and CIF generators.

If the intermediate form is targeted at too low a level, any structure inherent at different levels of the design will be thrown away, and this will make it hard to report back information such as timing errors or design rule errors in the designer's terms. If the intermediate form is at too high a level, then we lose the ability to have some control over the implementation of the design from its specification. This especially applies to silicon compilers, where the mapping from a behavioural domain to a physical domain is initially not very transparent.

What is needed is a representation in the middle ground; something which allows us to specify the implementation at the lowest level, and yet

retains the structure of the design at differing levels of abstraction.

SHIFT supports the design methodologies examined in the previous chapter in the following ways.

4.1.1. Leaf and Composition Cells

A SHIFT specification consists of a hierarchy of leaf cells and composition cells. Leaf cells are the lowest cells in the hierarchy, and describe basic components which will be laid out such as shift-registers, I/O pads, and wiring cells. Leaf cells contain descriptions of designs in the physical, structural, and behavioural domains. Composition cells are used to capture the design hierarchy in terms of instances of leaf cells and simpler composition cells. Composition cells have no inherent functionality; they contain only a description of constituent cells and their interconnections. This is important for the management of increasingly complex designs, since today's design as a composition cell is tomorrow's component of a design. Notation in the use of cells must *not* distinguish between the two, and this is reflected in the design of SHIFT. Thus the designs and sub-designs expressed in the intermediate form can be left in a library of standard components (leaf cells). The library grows as more designs are created.

IC designs can then be described in terms of a hierarchy of cells, where each level of the hierarchy is simpler in terms of its functionality than the level above, and the lowest level cell being an implementation of the function. An example of this is the OM2 datapath chip described in [Mead80], where the chip at the top level is viewed as a functional black box with external wires (see Figure 4.1).

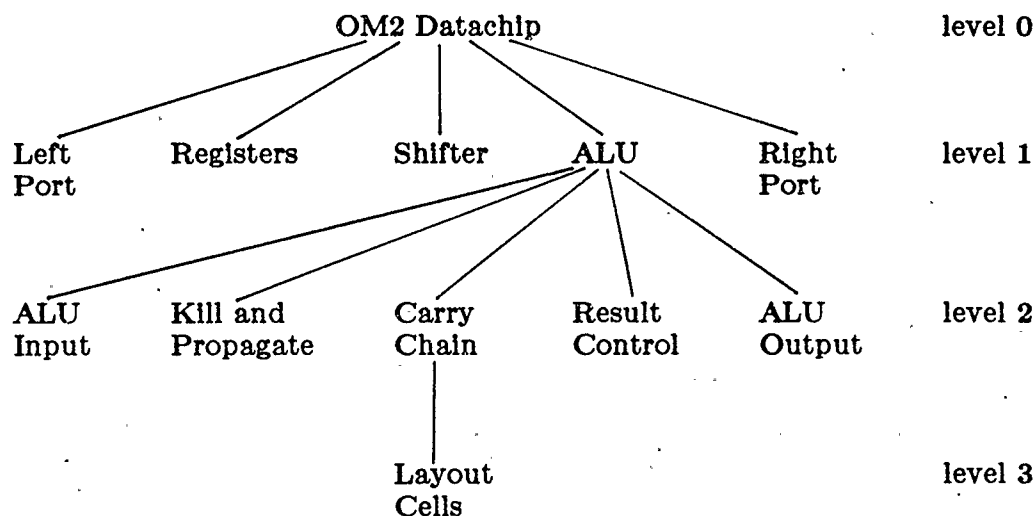


Figure 4.1. A Cell Decomposition of the OM2 Datapath

At the next level in the hierarchy, the chip is composed of cells representing a left port, a register block, a barrel-shifter, an ALU, and a right port. If we traverse down a level of the hierarchy, from left to right we can decompose the ALU block into an ALU input register, a kill and propagate control cell, a carry chain block, a result control cell, and an ALU output register. Finally we could decompose the carry chain into leaf cells that implement the function of the carry chain by specifying the layout.

A SHIFT description forms a separated hierarchy. This allows a degree of technology independence, since the hierarchy of a description will be the same over a range of technologies used to implement the leaves.

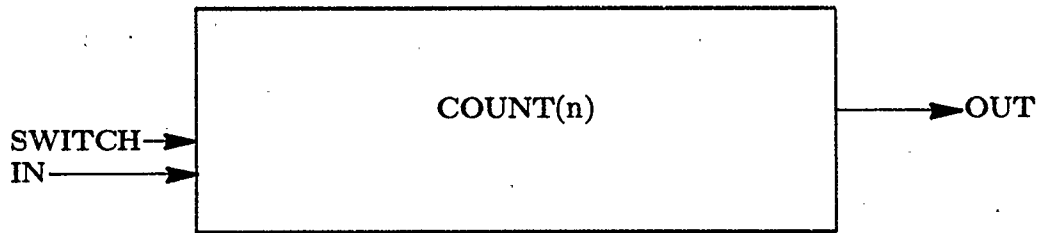
All cells are rectangular with a boundary on which ports are placed. Cells are composed by abutment in horizontal and vertical directions. The composed cells are stretched to connect adjacent ports. Composition cells are also built from other cells by abutment. The single operation of abutment serves as a composition rule in all three domains. Note that any necessary routing between cells can be incorporated as a routing cell sandwiched between the two cells to be connected.

SHIFT cells support the design principle of modularity, since a cell has a well defined boundary, and components may only be connected through adjacent ports. SHIFT cells also support design regularity, since cells connect along their rectangular boundaries in horizontal and vertical directions only, allowing cells to be easily composed.

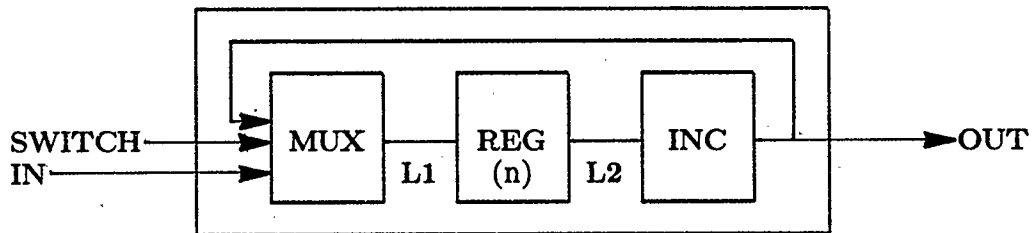
Composition cells may be augmented with a description in any of the domains. In effect, this augmented description is like a leaf cell description at this level of the hierarchy. For example, an ALU composition cell may have a behavioural description specified in addition to its behaviour derived by composition from constituent cells. Gordon [Gord81] shows an example of a counter that is decomposed into constituent behaviours (see Figure 4.2). These descriptions may be used to specify a design at the given level of abstraction. In this manner, both top down and bottom up design methodology are supported.

4.1.2. Composition Rules

A composition rule simply specifies how modules are to be built from components in order to guarantee the preservation of properties. The composition rule itself adds nothing to the composition cell. All functionality comes



$$\text{COUNT}(n) = \{\text{SWITCH}, \text{IN}, \text{OUT}\} \cdot \{\text{OUT} = n + 1\}, \\ \text{COUNT}(\text{SWITCH} \rightarrow \text{IN}, n + 1)$$



$$\text{MUX} = \{\text{SWITCH}, \text{IN}, \text{L1}, \text{OUT}\} \cdot \{\text{L1} = (\text{SWITCH} \rightarrow \text{IN}, \text{OUT})\}, \text{MUX}$$

$$\text{REG}(n) = \{\text{L1}, \text{L2}\} \cdot \{\text{L2} = n\}, \text{REG}(\text{L1})$$

$$\text{INC} = \{\text{OUT}, \text{L2}\} \cdot \{\text{OUT} = \text{L2} + 1\}, \text{INC}$$

Composition of Component Behaviours

$$\begin{aligned} & \llbracket \text{MUX} \mid \text{REG}(n) \mid \text{INC} \rrbracket \setminus L1 \, L2 = \\ & \quad \{\text{SWITCH}, \text{IN}, \text{OUT}\} \cdot \{\text{L1} = (\text{SWITCH} \rightarrow \text{IN}, \text{OUT}), \\ & \quad \quad \quad \text{L2} = n, \text{OUT} = \text{L2} + 1\}, \\ & \quad \llbracket \text{MUX} \mid \text{REG}(\text{L1}) \mid \text{INC} \rrbracket \setminus L1 \, L2 \end{aligned}$$

Figure 4.2. The Behaviours of a Count Cell and Its Components

from the components. The composition rule used in a composition cell has a specific meaning in each of the three domains of description of its component cells. For example, the application of the composition rule in the physical

domain of description simply forms the union of the component's physical descriptions, whereas the application of the composition rule in the behavioural domain identifies the signals connected through adjacent ports. This preserves the locality of function that is present in component cells. Finally, the application of the composition rule to the structural domain results in a merging of the graphs of the components structural descriptions.

Since the leaf cell has multiple representations, each representation must be checked for consistency with the others. However, since leaf cells are small (typically less than 50 gates) this is tractable. Even isomorphism problems of consistency checking (which are NP-complete) are still manageable for cells of this size. Also, however often it is used, a leaf cell has to be checked for consistency between domains only once.

The other advantage of the composition rule is that once a leaf cell has been checked for correctness, then the composition rule will preserve that correctness in all composition cells that contain that leaf cell. This hierarchical approach considerably reduces the amount of work required to verify the correctness of a design. For example, once a leaf cell has been checked for design rule violations, then it only remains to check the interconnection ports to show all future uses of that composition cell to be free of design rule errors.

4.1.3. External versus Internal Information in a Cell

By placing a boundary around all cells such that information can only flow through ports we hide the internal details of a cell at a given level. This provides us with a powerful tool in designing circuits; namely that we can abstract out the detail that we want to consider at any level in the design. The internal information in a cell is only accessible within the cell. The

external information of a cell is limited to the ports through which a cell communicates with its neighbours. This approach actively discourages a non-functional approach to designs. As in software, the process of specification and verification are much easier when designs become modelled with a functional approach rather than with a von Neumann approach [Back78]. By restricting communication through the ports, we outlaw the hardware analogues of software's "gotos" and "side-effects".

This is not the penalty that it first seems. As chip designs become larger the cost of communicating global information becomes much higher than the cost of computing it locally. Global information also restricts the amount of concurrency exhibited in a design. As circuits become larger, self-timing schemes become more attractive, with the result that cells become truly self-contained.

4.1.4. Design Systems Using a High Level Intermediate Form

Producing an IC design requires a number of tools to assist the designer in the design synthesis, test and validation process. A number of these have been encountered in chapter 2. What is required is to tie these tools together into an integrated design environment by using a consistent intermediate representation. Some of these tools create or modify the design description; other tools like simulators, need only extract information from it. A real design system, like a real programming environment must have its constituent parts work in unison. This requires a degree of intercommunication which is difficult to achieve without a consistent view of the data they operate on.

It also has been shown in chapter 3 that these designs are developed in an incremental fashion, with many iterations over the design. With a

structured consistent specification of the design the development and modification of the design is accomplished more easily.

Finally, by making the design modular, and specifying the behaviour of the modules and how they interact, it allows many designers to work on different parts of the system with some assurance that the design will work as specified.

4.2. SHIFT Design

In the previous chapters we have examined the nature of VLSI design and the kinds of tools needed in the development of a design. A set of requirements for an intermediate form was then specified and elaborated in the previous section. This section concentrates on the specification of SHIFT.

A design in SHIFT involves the definition and subsequent instantiation of cells. The three domains of description in SHIFT are the physical, structural, and behavioural domains. As seen by Table 3.1 SHIFT is the only tool thus far that allows a design description in all three domains.

There are three different representations or stages of cells. These are *archetypes*, *prototypes*, and *instances*. The act of defining a cell in SHIFT creates an archetype. Evaluating an archetype cell with parameters creates a prototype by fully defining the set of constraints. Finally, an instance is created by applying a solution of a constraint graph to a prototype. Thus the values bound to the parameters are known by both prototypes and instances, and the value of the ports are known only by the instances.

There is a monotonicity of requirements between archetypes, prototypes, and instances for valid composition. Two archetypes may be composed pro-

viding they have the same number of ports. Two prototypes may be composed providing they have the same number of ports and there exists a solution to the constraint graph. Two instances may be composed providing they have the same number of ports, a solution to the constraint graph exists, and one such solution is specified.

4.2.1. SHIFT Cells

There are two kinds of cells that may be defined in SHIFT, leaf cells and composition cells. All cells have a ports, a constraint, a behaviour, and a structure definition component. Where leaf and composition cells differ is that leaf cells have a geometry component, and composition cells have a

```
(defleaf leafname (parameter-list)
  (ports ..... )
  (const ..... )
  (struct ..... )
  (beh ..... )
  (geom ..... )
)
```

Figure 4.3(a). Overview of A Leaf Cell Definition

```
(defcomp compname (parameter-list)
  (ports ..... )
  (const ..... )
  (struct ..... )
  (beh ..... )
  (composition-expression ..... )
)
```

Figure 4.3(b). Overview of A Composition Cell Definition

composition component (Figure 4.3). This is because the leaf cell contains artwork, and the composition cell contains only other cells. Once a sub-design is fully specified as a hierarchy of cells it can be stored away in a design library, and subsequent designs need not know whether it is a leaf or composition cell. There are clear advantages to storing the sub-design in its hierarchical form rather than flattening it out to a fully detailed leaf cell, for to do so takes up far more space and makes variations on a design much more difficult to accomplish.

In the case of composition cells any of the common cell fields may be null, in which case the ports will have names constructed from their component cells port names, and the constraints will be synthesized from their components constraints.

The ports are a list of names in ascending order along the north south east, or west boundaries. The ascending order is necessary for the identification of ports along the adjoining wall when composed with another

```
(ports
  (north n:clock) (south s:clock)
  (east e:gnd out e:vdd) (west w:gnd in w:vdd)
  (interior
    gc pc last          ; ground, power and butting contacts
    pd.gtin pd.gtout pd.src pd.drn  ; pulldown nodes
    pu.gtin pu.gtout pu.src pu.drn  ; pullup nodes
    pt.gtin pt.gtout pt.src pt.drn  ; passtran nodes
    middle)              ; inverter output
  )
```

Figure 4.4(a). The Ports Definition of a Shift Register Cell

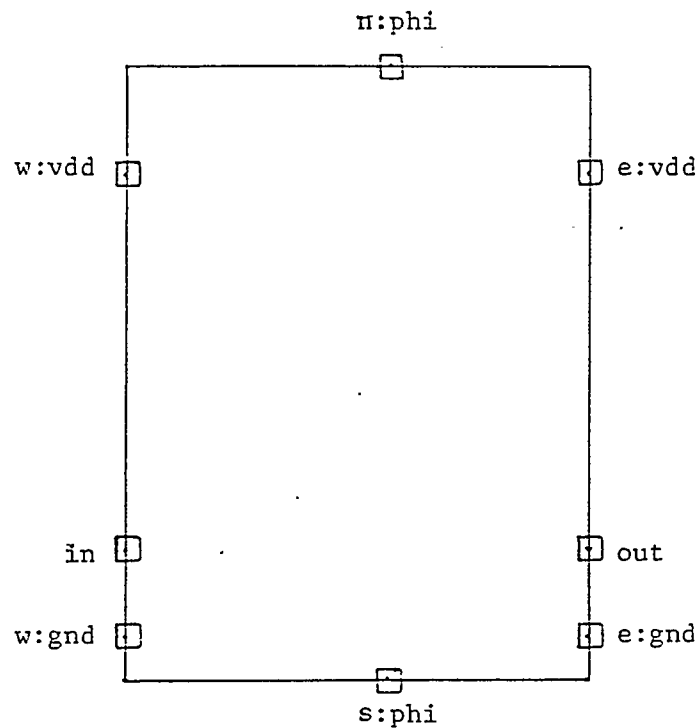


Figure 4.4(b). The Ports of a Shift Register Cell

cell. Ports may also be defined as interior ports in leaf cells. Unlike other ports, interior ports do not lie on one of the cell walls, nor need they be listed in any order. Rather, the relationship between interior ports is defined solely by the horizontal and vertical constraints specified. An example of a leaf cell ports definition is shown in Figure 4.4.

Constraints may be specified between any two ports, or between ports and a wall, where these constraints are meaningful. Horizontal constraints between any two ports A and B may be of the form

$$A = B + c$$

or

$$A \geq B + c, \text{ where } c \text{ is a numeric expression.}$$

with the meaning A lies to the east of B by exactly c, and A lies at least c to the east of B. Vertical constraints take the form

$$\begin{array}{l} \text{or} \\ A !! B + c \\ A \hat{!!} B + c \end{array}$$

with an analogous meaning in the y direction.¹ The distinction between verti-

```
(const
  (w:gnd ^!! south + 2) (in ^!! w:gnd + 4)      ; west wall
  (north ^!! w:vdd + 5) (w:vdd ^!! pu.drn + 1)
  (out >= last + 3)
  (e:gnd ^!! south + 2) (out ^!! e:gnd + 4)      ; east wall
  (e:vdd ^!! out + 2) (e:vdd !! w:vdd)
  (s:clock >= pt.src + 3) (n:clock = s:clock) ; south & north walls
  (gc = west + 5) (gc ^!! south + 2)             ; ground contact
  (pd.src = gc) (pd.drn = pd.src)                 ; pulldown
  (pd.gtln >= in + 1) (pd.gtout >= pd.gtln + 8)
  (pd.src ^!! gc + 1) (pd.gtln ^!! pd.src + 3)
  (pd.gtln !! in) (pd.gtout !! pd.gtln)
  (pd.drn ^!! pd.gtln + 3)
  (middle ^!! pd.drn + 1) (pu.src ^!! middle + 1); inverter output
  (middle = pd.drn)
  (pu.src = middle) (pu.gtln = pu.src) ; pullup
  (pu.gtout = pu.gtln) (pu.drn = pu.gtout)
  (pu.gtln ^!! pu.src + 2) (pu.gtout !! pu.gtln + 7)
  (pu.drn ^!! pu.gtout + 2)
  (pc = pu.drn) (pc !! w:vdd)                     ; power contact
  (pt.gtout = s:clock) (pt.src >= pd.gtout) ; passtran
  (pt.drn >= pt.gtln + 3) (pt.gtout ^!! pd.gtout + 1)
  (pt.gtln ^!! pt.src + 3) (pt.src !! middle)
  (pt.gtln = s:clock) (pt.drn !! pt.src)
  (last !! pt.drn) (last >= pt.drn + 3) ; last contact
)
```

Figure 4.5(a). The Constraints Definition of a Shift Register Cell

¹These operators look similar to the horizontal operators turned on their side.

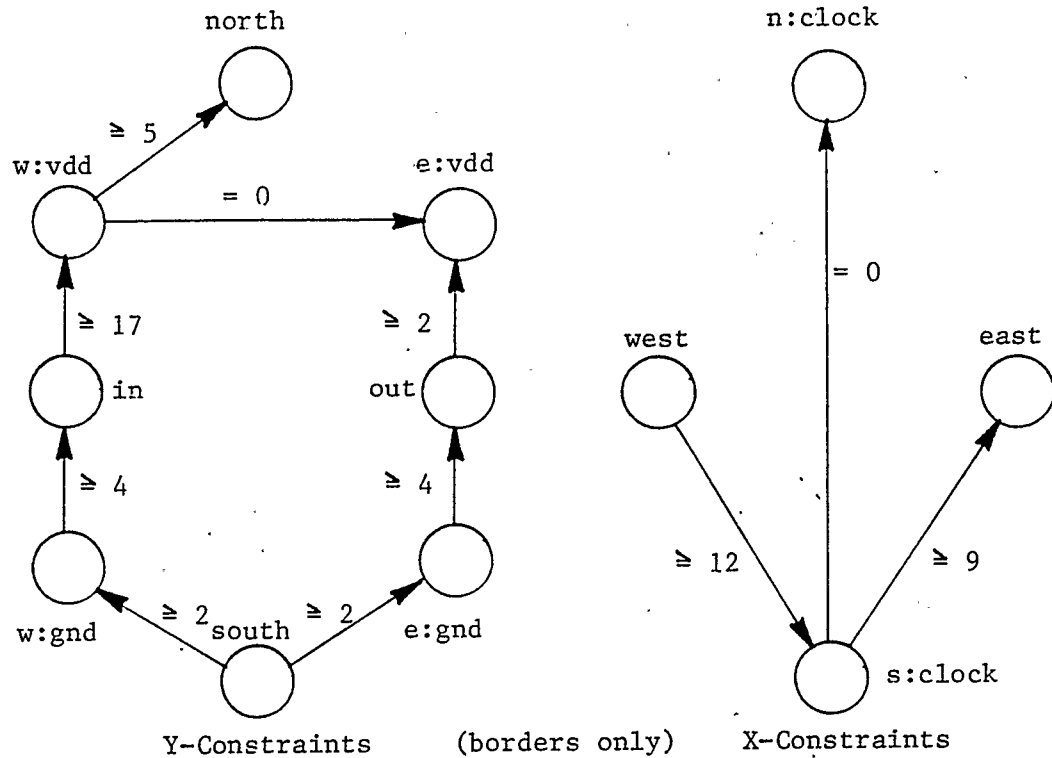


Figure 4.5(b). The Constraints Graph of a Shift Register Cell

cal and horizontal constraint relations serves to disambiguate constraints with respect to interior ports. It also enforces a notational distinction which makes the constraints easier to read and specify. An example of a leaf cell constraints definition is shown in Figure 4.5. This example is rather elaborate since all points interior to the cell used in constructing the geometry are represented as interior ports with constraints used to define their final values. In practice, one might specify rigid components in the interior, each being anchored to a single node, with constraints relating to these nodes used to define minimum distances between ports on the outer walls. However, this example shows that constraints may be used to build the entire cell.

```

 geom
  (dm-at gc)           ; contact between pulldown and ground
  (wire metal 4 w:gnd gc e:gnd); ground wire
  (pulldown           ; pulldown has two parts
    4 (path pd.src pd.drn) ; diffusion path from source to drain
    2 (path pd.gtin pd.gtout)); a poly path from gtin to gtout
  (wire poly in pd.gtin) ; connect gate to input port
  (wire diffusion gc pd.src) ; connect pulldown to ground
  (pullup           ; pullup has four parts
    2 (path pu.src pu.drn) ; a diffusion path from source to drain
    6 (path pu.gtin pu.gtout)); a poly path from gtin to gtout
                                ; an implant layer is automatically drawn
                                ; over the poly layer, extended by 2 lambda
                                ; on either end, and a butting contact at the
                                ; gate input connecting the gate to the source
  (wire diffusion pd.drn middle pu.src); connect the pullup and pulldown
  (wire metal 4 w:vdd pc e:vdd); power wire
  (wire diffusion pu.drn pc) ; connect pullup to power
  (dm-at pc)           ; contact between pullup and power
  (passtran           ; passtran is equivalent to the pulldown
    2 (path pt.drn pt.src)
    2 (path pt.gtin pt.gtout))
  (wire poly n:clock pt.gtin) ; wire up clock to one end of the gate
  (wire poly s:clock pt.gtout); wire up other end of gate to clock
  (wire diffusion middle pt.src); connect inverter output to passtran's
                                ; source
  (be-at last)           ; butting contact for passtran to out
  (wire diffusion pt.drn last); connect passtran to contact and
  (wire poly (pt-dx last 1) ; contact to out
    (then-y (:y out)) out)
 )

```

Figure 4.6(a). The Geometry Definition of a Shift Register Cell

Only constraints which lie vertically or horizontally a minimum distance greater than zero from the south and west walls, respectively, need to be specified. All ports are automatically constrained to lie at least on or to the east and north of the west and south walls, respectively.

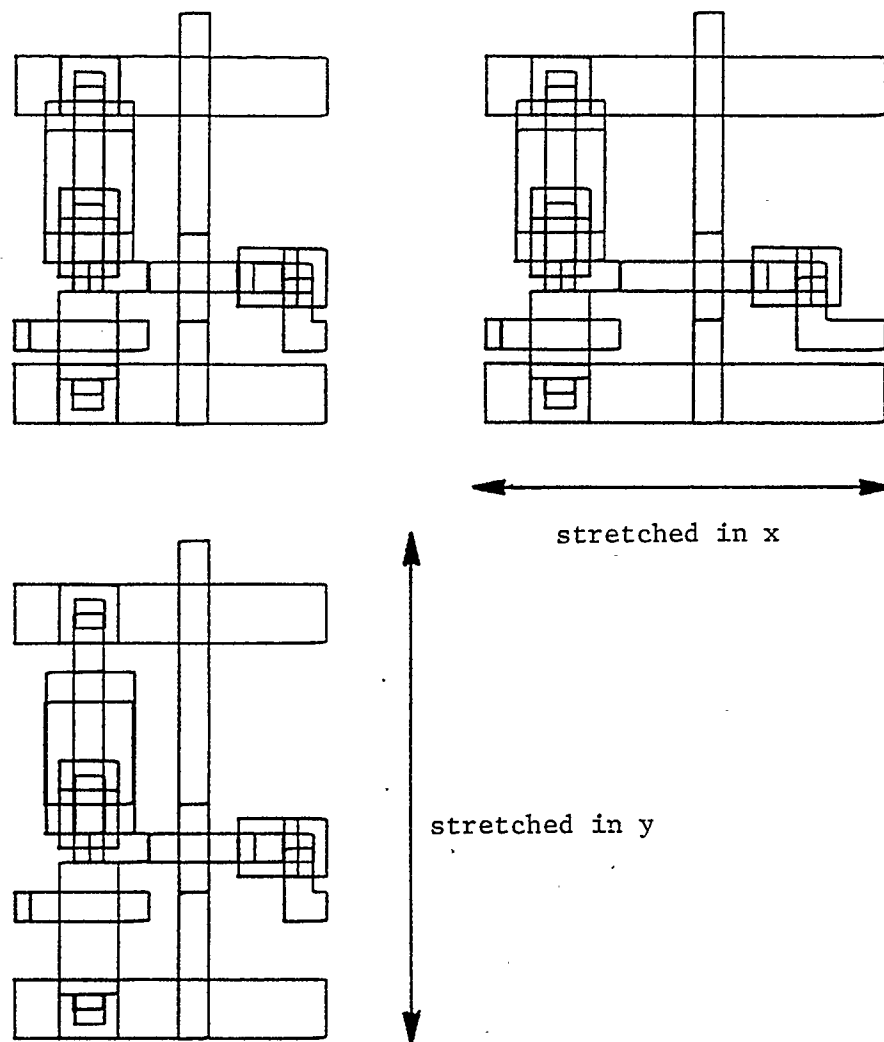


Figure 4.6(b). The Geometry of a Shift Register Cell

4.2.2. Leaf Cells

Leaf cells are defined with the *defleaf* operator and contain a parameter list (with optional defaults), a sequence of ports, a set of constraints among the ports, and a description of a cell in one or more of the physical, struc-

tural, and behavioural domains of interest.

4.2.2.1. Physical Description

A physical description of a leaf cell consists of a list of geometric functions, which expand to lists of geometric primitives. These geometric primitives describe the layout of the design on mask layers that are used in the fabrication process to manufacture the integrated circuits. See Figure 4.6 for an example of the geometry of a shift register. In this example, we see the use of both primitive functions, and several higher level nMOS-specific functions which map to lists of primitives, to create the geometry.

The basic geometric types are box, polygon, and wire. A primitive is a list of the primitive type, a layer, a width (if the primitive is a wire), and a path. A path is a list of points, where each point may be either absolute or relative to the previous point in the list. Obviously, the first point in the list must be absolute. Various operators exist for creating and manipulating points and their x and y components.

In the case of the box primitive, the two points define the adjacent corners. The path of the polygon primitive represents the ordered list of vertices defining the boundary of a closed polygon. Finally the wire's path defines a centre line of a long uniform width run along a layer. However, unlike the CIF-style wire, defined as the locus of points within one-half width of the path, the SHIFT wire has curtailed endpoints i.e. the endpoints of the wire lie on the perimeter of the path.

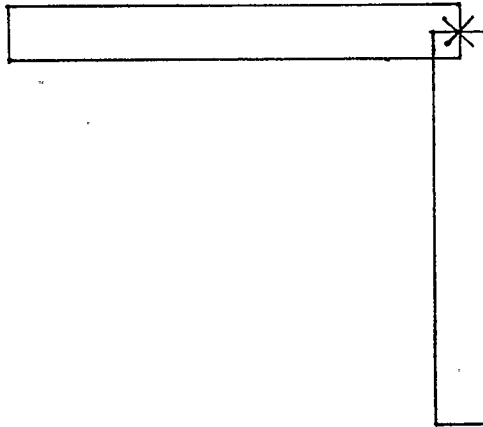


Figure 4.7(a). Wire Connection - Curtalled

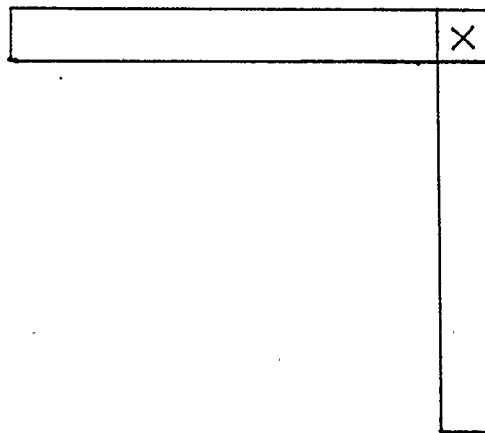


Figure 4.7(b). Wire Connection - Inflated

The CIF-style wire has the advantage over the SHIFT-style wire in that any two wires connected together at a common endpoint will form a proper connection regardless of the angle (Figure 4.7(a-b)). However, in practice this type of connection is not likely to occur in SHIFT.

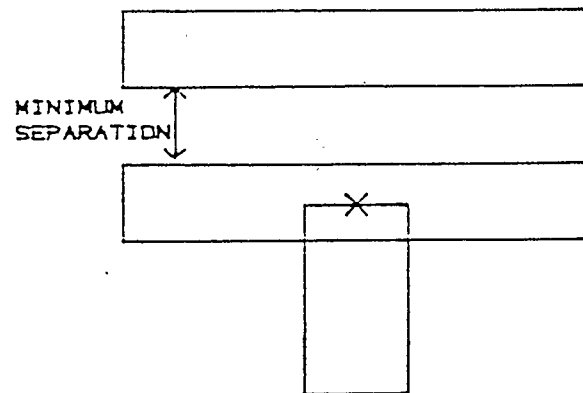


Figure 4.7(c). T Connection - Curtailed

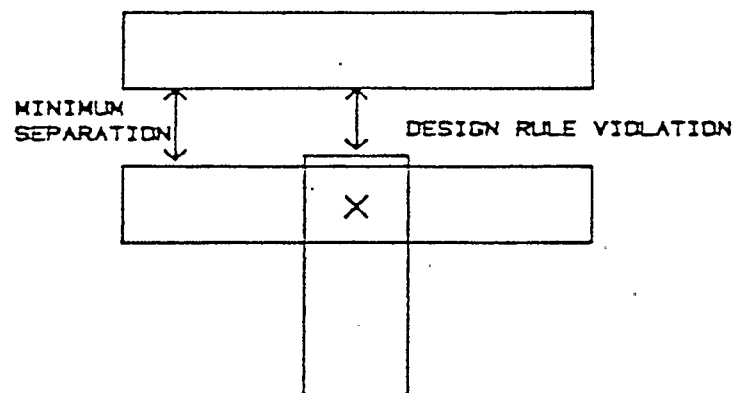


Figure 4.7(d). T Connection - Inflated

In addition, a number of undesirable effects occur when using CIF-style wires in constructing circuits. One example is a T connection of wires on the same layer with another wire above and parallel to the cross piece and separated by the minimum design rule distance (Figure 4.7(c-d)).

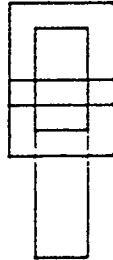


Figure 4.7(e). Butting Contact - Curtailed

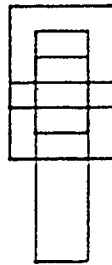


Figure 4.7(f). Butting Contact - Inflated

If the vertical piece is wider than the horizontal piece, then a design rule violation occurs with the inflated wire.

Another example is the connection of a polysilicon wire to the centre of a butting contact (Figure 4.7(e-f)). The CIF-style polysilicon wire extends too

far under the contact hole, resulting in another design rule violation. These reasons make the curtailed style wire more attractive than the inflated style wire. If we want to make a connection at right angles between two wires at their common endpoints, a "contact" box of the same layer may be placed at the point, thus ensuring proper connectedness.

The layer and the width may be omitted for the primitives, in which case a technology defined default is used.

4.2.2.2. Structural Description

A structural description of a leaf cell is a lumped circuit model of the cell. The structural description serves to describe the performance of the design, that is, both its power and speed. While the structural description of a cell may be extracted from the geometry of a cell, and is therefore not strictly necessary, it is used often enough in the design of chips to provide a place specifically for it, so that it may not need to be repeatedly extracted. Further, this may be generated automatically by a circuit extractor on the leaf cells as they are defined.

The structural description consists of a list of named components, and a netlist of connections between the components and the ports. Components may be resistors, capacitors, or n-type and p-type enhancement and depletion mode transistors, with various attributes supplied either by explicit declaration or defaulted to a process/technology dependent value.

Resistors and capacitors have two ends to which one may connect, denoted 'one-end' and 'the-other-end'. Resistors may take a specified resistance in ohms, and capacitors may take a specified capacitance in pico-farads.


```

(struct
  (nodes
    pullup (n-type-dep len 6 wid 2)
    pulldown (n-type-enh len 2 wid 6)
    pass (n-type-enh))
  (connect
    (e:vdd w:vdd)
    (e:gnd w:gnd)
    (n:phi s:phi)
    ((:source pullup) e:gnd)
    ((:source pulldown) e:vdd)
    ((:drain pullup) (:drain pulldown))
    ((:drain pullup) (:source pass))
    ((:gate pullup) (:drain pullup))
    ((:drain pass) out)
    ((:gate pulldown) in)
    ((:gate pass) n:phi))
)

```

Figure 4.8(a). The Structure Definition of a Shift Register Cell

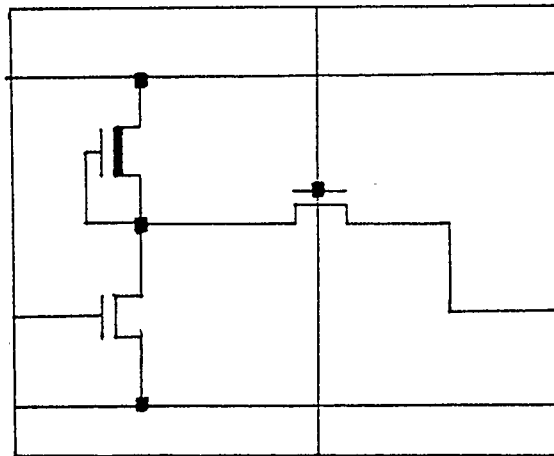


Figure 4.8(b). The Structure Diagram of a Shift Register Cell

Transistors have three nodes, labelled 'drain', 'source', and 'gate', and may take optional parameter values specifying their length and width, which

may be used in performance evaluation. An example of a structural description of a shift register is shown in Figure 4.8.

Here we see two n-type enhancement-mode transistors being declared, one with a declared length and width, and the other defaulting to a technology dependent value. These components are then connected to each other, and to the ports.

4.2.2.3. Behavioural Description

The behavioural domain is specified using an approach similar to that used in denotational semantics in which the behaviour of a device is modelled by a function which is an element of a domain of "sequential behaviours" [Gord81]. This domain is a unit delay model of behaviours. The domain used is defined to be the least solution to the domain equation:

$$\text{BEH} = \text{IN} \rightarrow (\text{OUT} \times \text{BEH})$$

where IN and OUT model input and output signals of the device.

The behavioural specification of our example is seen in Figure 4.9. In this example, the first element is the current state of the device, the second

```
(beh (s)
  ((s:phl = n:phl)
   (out = if n:phl then (not s) else @))
  ((if (in = @) then s else in))
)
```

Figure 4.9. The Behaviour of a Shift Register Cell

element is a set of equations in terms of the current state and inputs mapping signals to lines, and the last element specifies the next behaviour, which is a function of the current inputs and state. Float is represented by '@'.

4.2.3. Composition Cells

Composition cells are defined with the *defcomp* operator and contain a parameter list (with optional defaults), an optional sequence of ports, an optional set of constraints among the ports, and an optional description of the cell in the structural and behavioural domains. Finally, a composition cell definition contains a composition expression, where each element is a leaf or prototype cell, or a composition expression.

Cells may be composed by means of one of the four composition operators '>', '<', '^', and 'v'. These correspond to horizontal composition, east to west and west to east, and vertical composition from south to north and north to south, respectively. An example of a 2 element array of shift register cells is shown in Figure 4.10.

The composition between cells is performed by stretching the ports on the adjoining walls until they align; there is no generation of river routing between the cells.² An example of this is shown in Figure 4.11. The subject of composition by stretching is discussed in further detail in the section entitled "Composition Algorithm".

²If routing is desired it can be captured with a routing cell sandwiched between the cells to be routed together.

```

(defcomp shift2 ()
  (ports (north n:phi1 n:phi2)
        (south s:phi1 s:phi2)
        (west w:gnd in w:vdd)
        (east e:gnd out e:vdd))
  (> (shiftreg) (shiftreg))
  (beh (s1 s2)
    ((out = if n:phi2 then (not s2) else @))
    ((if (in = @) then s1 else in)
     (if n:phi1 then (not s1) else s2)))
  )

```

Figure 4.10(a). A 2 Element Shift Register Array Composition Cell

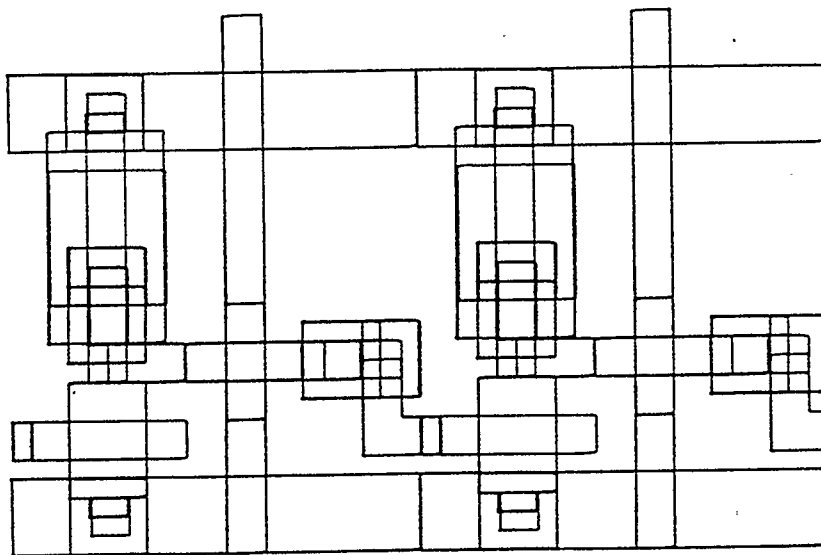


Figure 4.10(b). Geometry of a 2 Element Shift Register Array

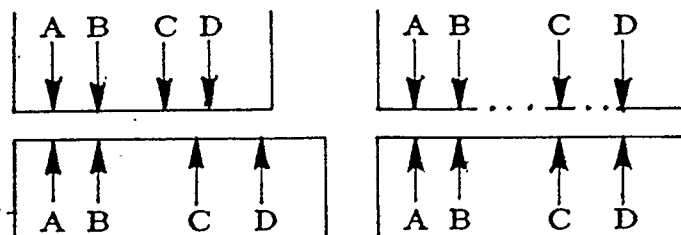


Figure 4.11. An Example of Stretching

4.2.4. Design Instantiation

In order to produce a description of the design in any of the three domains we must produce an instance of the design from its description in SHIFT. A design is instantiated in the following manner. The archetype specification is traversed from the root, inheriting down (i.e. binding) the parameters at each level to the cell and evaluating the expressions in the constraints to produce a fully determined set of constraints that defines the prototype. These parameter values also may be used in expressions in the composition definitions. This result of this traversal is a rooted tree that comprises the prototype of the design.

To produce an instance from the prototypes we first synthesize up the constraint graphs from the leaves to the root. We then solve the graph at the top level, and inherit down the solution to the leaves. These values are then bound to the ports and each of the domains is evaluated to produce the leaf instances, which are then synthesized up the tree to produce an instance of the design.

Expanding an entire design to get all the constraints in one direction, and then solving them could be quite time consuming, as there could be tens

of thousands of ports, and hundreds of thousands of constraints. Instead, we can use the hierarchy of SHIFT to exploit the locality of constraints and considerably reduce the work.

4.2.5. The Composition Algorithm

The approach used in composing cells follows that used in the composition of a sticks languages called LAVA [Ullm84]. The basic idea is that we can eliminate the interior constraints of a cell at any level in the hierarchy by using them to produce a new set of constraints which only involve the border ports of the cell. Further, these new sets of constraints are produced in such a manner that any given solution for the border ports of the cell will not violate the interior constraints which had been previously eliminated. The algorithm is a two stage process.

In the first stage we reduce the constraints for the bottom-level cells, so that we are left only with constraints involving ports on the borders of the cell. These are synthesized up the next level of the hierarchy, and the process repeats until we reach the root of the hierarchy.

In the second stage we find a solution to the constraint graph at the top level of the hierarchy, and recursively inherit down the values for the border points of the component cells, solving at each step in the hierarchy until we reach the leaf cell at the bottom of the hierarchy. By making the coordinates for the internal points of a cell as low as possible, consistent with the constraints of the cell and the values of the border points inherited down, we may solve for the internal points of the cell at each level of the hierarchy.

The key steps in this process are:

- (1) The elimination of those constraints within a cell that do not involve the border points.
- (2) The combination of constraints from several cells into one set of constraints.

The general algorithm for the solution of the constraint graph for the entire design is as follows;

The topological sort³ is used to determine in what order values are assigned to the nodes in the constraint graph. Every node with no predecessors in the topological sort may be taken to have the value of 0. When reaching node u with some predecessors, we have already assigned values for those predecessors. We then assign a value for u by evaluating the constraints connecting u to its predecessors and taking the lowest value consistent with these.

To eliminate the interior constraints of a cell we consider each border point a , in turn from the bottom of the cell. Note that the bottom of the cell

SYNTHESIS

```

FOR cells at level 0, 1,...,level of root DO BEGIN
    eliminate interior nodes from constraints;
    IF level > 0 (i.e. a composition cell) THEN
        combine constraints involving border points of subcells
END;

```

INHERITANCE

```

solve constraints for the root cell, by finding a
topological sort of the nodes in the constraint graph;
FOR all instances of cells at level of root - 1 down to 0 DO
    solve constraints for interior points, given values for border points.

```

³ A topological sort of an acyclic graph is the reverse of a depth first ordering of the graph. This ordering has the important property that if there is an arc from a to b , then a precedes b in the ordering.

is regarded as a border point. We perform a depth-first search on the constraint graph from a but we stop when we reach another border point, b (i.e. we do not follow any arc out of it). Thus we reach all and only the nodes accessible from the border point. By visiting the nodes in topological order, we can derive for each such border node b , the length of the longest path l , from a to it. Thus the constraint $b \geq a + l$ is the minimum constraint implied by the given constraints. By repeating this for all border points, we derive the set of constraints involving only the border points.

4.2.6. Complexity of The Composition Algorithm

Assuming that the constraint graph is acyclic, we may topologically sort the nodes in time proportional to the number of arcs, i.e. the number of constraints. The assignment of values to nodes given the topological sort of the cell is also proportional in time to the number of constraints. Thus complete solution of a cell is proportional in time to the number of constraints.

The partial solution of constraints is actually more complex than completely solving for the constraints. However, we cannot simply solve the constraints for several cells independently since they may be connected at a higher level and if we have found incompatible values for the corresponding ports, then the cells cannot be abutted as intended. Since the depth-first search from each border point may involve visiting all, or almost all, of the points in the cell, in worst case the time to eliminate the interior points is on the order of the product of the number of border points and the number of constraints.

In practice, however, the elimination of the interior points is much less time-consuming, as many of the interior points will be sandwiched between

two border points, resulting in their appearing only in the depth-first search of the border point immediately below it. In addition, the extra complexity is offset since a new constraint graph only has to be produced once for a given cell. The same constraint graph can be subsequently used wherever else the cell appears, so the overall cost is significantly reduced when the cell is used more than once (i.e. regularity), which is characteristic of increasing trends in integrated circuit design. Thus we can exploit the locality of the constraints at every level of the hierarchy.

4.3. Summary

This chapter has focused on the requirements of an intermediate form for VLSI design tools in an integrated environment. Also the requirements for an intermediate form that supports a separated hierarchy and a structured design methodology have been presented. A structured intermediate form for VLSI design tools called SHIFT has been outlined, together with an algorithm for composing cells in a stretchable manner along adjoining ports. In the next chapter we focus on the implementation-dependent aspects of SHIFT, and how it fits within a proposed design environment called EDICT [Birt84].

CHAPTER 5

SHIFT Implementation

In chapter 4 we focused on the requirements of an intermediate form for VLSI design tools, and gave an overview of an intermediate form (SHIFT) designed with these requirements in mind. This chapter focuses on the current implementation of SHIFT and factors influencing decisions made in the course of implementation. Finally, we discuss where SHIFT fits in with current VLSI tools being developed at the University of Calgary.

5.1. Choice of Implementation Language

The choice of language was generally motivated by the fact that SHIFT was designed to mix in with both existing and developing tools. Since most of the current tools exist under the Berkeley Unix† 4.2 operating system this meant that the implementation language should also exist on the same system. Further, since most of the tools are written in a variety of languages, (e.g. SPICE in Fortran, LAP in Simula), the language chosen to implement SHIFT should provide as flexible an interface as possible to other languages. A consideration of these and the following reasons led to the choice of Franz Lisp [Wille84] as the language of implementation.

First, since SHIFT is a procedural as well as declarative intermediate form, it was desirable that SHIFT be embedded in some general purpose language, rather than re-inventing the wheel. Second, it was felt that SHIFT should also be extensible, thus the language in which it was to be embedded

† UNIX is a Trademark of Bell Laboratories.

should also be extensible. Third, we much preferred to cast the design in an object-oriented language, since a lot of the manipulation was symbolic in nature. Fourth, SHIFT should be both human readable and portable to many systems. Fifth, SHIFT should execute efficiently, since VLSI designs tend to be extremely large, and are continually growing in size. Lisp was the only language available which fulfilled all these objectives.

Franz Lisp was chosen because of several additional features it has that many other languages on Berkeley Unix do not have. Franz Lisp allows the user to load in foreign functions dynamically, thus allowing it to make use of software already written. In addition, Franz Lisp was chosen because of its capability to run within a distributed environment as well as in a stand-alone configuration.

5.2. SHIFT Implementation

SHIFT retains much of the flavour of Lisp's syntax. This decision was made to minimize the effort involved in building SHIFT; any syntactic "sugaring" could be done later as an interface sitting on top of SHIFT. Further, the mapping into SHIFT would be made easier by the fact that Lisp is both easy to parse and to produce in an automatic manner.

SHIFT in its current state consists of approximately 3,000 lines of sparsely documented code. The Franz Lisp code conforms wherever possible to the Macclisp dialect, making SHIFT easily transportable to many other Lisp systems which use Macclisp or a similar dialect. This makes it easy to develop tools in Lisp that are built on SHIFT. For the purposes of efficiency in execution as well as code clarity, an early decision was made to use the MIT structures package to build prototypes and instances. This, however, should not

decrease the portability of SHIFT, since the structures package is supported for almost all major dialects of Lisp.

As stated before, the basic SHIFT geometric primitives are the wire, the box, and the polygon. While these are sufficient to capture all designs of interest, it was felt that most tools using SHIFT would want to work at a slightly higher level. For example, a Sticks-based editor manipulates wires, contacts, and transistors. Also, users designing leaf cells procedurally, even using SHIFT, would find their task greatly simplified if they could specify basic units like transistors.

As a result, SHIFT fully supports the geometric domain with a variety of routines for designing layouts in both nMOS and CMOS technologies. NMOS technology routines allow the user to specify pullup, pulldown, pass, and enhancement mode transistors. CMOS technology routines consist of pmos and nmos transistor functions, as well as a precursor gate transistor routine for specifying a variety of simple logic configurations such as nand, nor, and pla's, etc., as well as transmission gates.

While it is not central to SHIFT, it was felt that a geometry composer should be provided. As stated before, this would provide immediate feedback to users building tools, and allow users to try out SHIFT. It was also useful for providing the examples in this thesis. Since the geometry of a composition is simply the union of the geometry of its component parts, the geometry composer was extremely easy to implement.

The structural and behavioural domains are not as fully fleshed out, and their composer functions have not yet been implemented. These would be implemented in the following manner. The structural composer would require

a function which would compose graphs together by descending the hierarchy and merging the leaf cell structural graphs by merging the nodes of adjacent ports on abutting cells.

The most difficult composer to implement would be the behavioural composer. Fortunately, this has already been provided as a set of functions implemented in LCF-LSM [Gord83]. It is intended that the existing software be used in parallel with SHIFT, either as communicating processes through the Jade [Unge84] distributed system, or in a more intimate manner, as LCF-LSM is implemented in Franz Lisp, and provides hooks to load Lisp, and therefore SHIFT, functions.

5.3. SHIFT and Current VLSI Tools

There are three ways in which design tools can use SHIFT. The first method is to use SHIFT as a textual interface, in a manner similar to CIF. This will eliminate the nasty problem of "user extensions" to CIF that contain information obtained by circuit extraction and used in circuit simulation. In addition, "user extensions" are allowed in SHIFT definitions. They are simply placed in a special slot of the prototype as an association list in an unevaluated form. The first element of the form simply becomes the name of the user extension, and if the form is a symbol, then the value of the association is nil.

The second method is to use SHIFT in future Lisp-based tools simply by incorporating it into the Lisp environment. This would also allow tools such as the Lisp-based SPICE interface (with a minimum of modification to it) to use performance simulation information provided by SHIFT from the structural domain. A variant form of this would be to use SHIFT interactively,

since it is embedded in Lisp. This would allow a designer to design small cells interactively, and in the process, become familiar with SHIFT.

The final use of SHIFT would be to use SHIFT within the context of a distributed environment such as Jade. A Lisp-Jade Interprocess Communication Interface called Jipth [Libl84a] was developed in order to use SHIFT as a library process that would contain the information of an evolving design. A primitive version of the SHIFT library manager called *shiftlib* currently exists, and will allow a user process to pass messages defining cells, (in fact, any Lisp s-expression), instantiating them, and querying the library for any information. Further, SHIFT contains version control information that is used during instantiation to limit the necessary modifications to only those parts of the design dependent on the changes.

A symbolic layout editor is currently being developed based on SHIFT that would allow the designer to build leaf cells using a graphical interface, similar to REST [Most81]. However, once the cell is laid out it would be compacted in a manner that would preserve the inherent constraints for subsequent stretching when composed.

5.4. SHIFT and EDICT

EDICT is a VLSI design tool environment under construction at the University of Calgary. It will guarantee that designs meet their specifications; allow specifications to be composed from verified sub-modules (bottom-up), or be refinements of rougher specifications (top-down); and cater for the incorporation of previously validated building blocks, large or small. The first experimental versions of EDICT will be extensions of current tools, and will be written as applications of the JADE distributed environment.

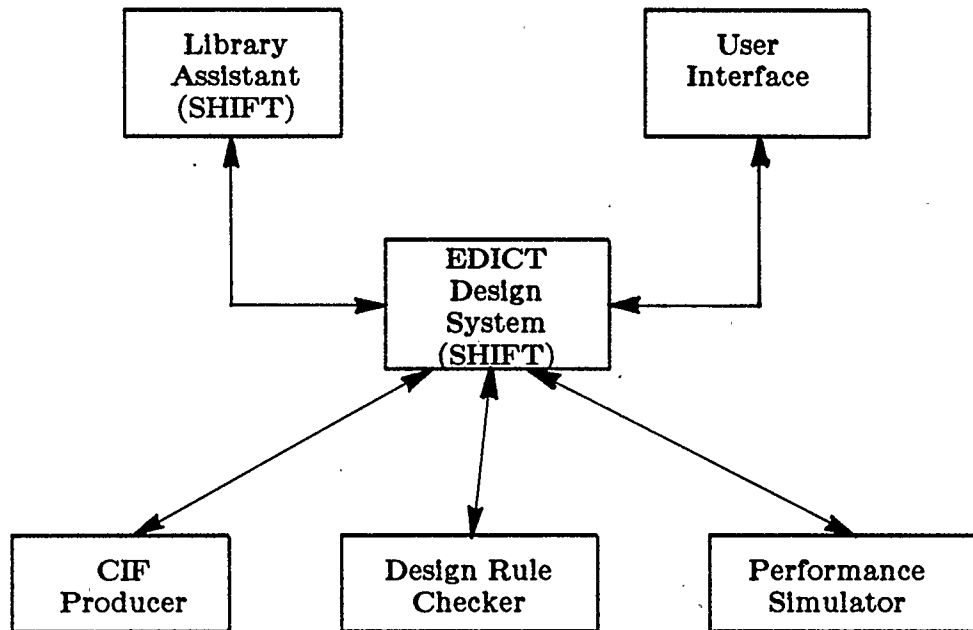


Figure 5.1. The EDICT Design Environment

A critical sub-system of EDICT will be a *library assistant* which remembers leaf cells and composition cells and stores them in a library for future reference (see Figure 5.1). When later designs require elements with the same specification, cells will be suggested by a library assistant working on the fly. The library assistant will grow ever more knowledgeable as verified designs are automatically added to the library. Since today's design will be tomorrow's component, using SHIFT to represent designs in the library means that we will be able to build a collection of tried and tested parts which will slot into future designs with a minimum of modification. Thus a consideration in the design and implementation of SHIFT was that SHIFT should form the kernel of the library assistant.

Since the preliminary version of EDICT will be built within JADE, Jipth will allow any of the EDICT components implemented in Franz Lisp to operate as complete entities within the Jade distributed environment. When combined with SHIFT to create the library assistant, Jipth will allow processes in other languages to query and modify the design database. This means that current tools such as LAP may co-exist with EDICT, and future tools such as layout editors may be built to use SHIFT as their intermediate form of choice.

This chapter has focused on the implementation-dependent aspects of SHIFT, and how SHIFT is expected to fit in with both current and future VLSI design tools at the University of Calgary. Chapter 6 will conclude with an overview of the research work in this thesis, and draw some observations about the future of SHIFT in the state of VLSI design.

CHAPTER 6

Conclusion

6.1. Summary

This thesis has focused on the problem of increasing complexity in the design of integrated circuits. An analysis of methodologies used in managing this complexity has been made and observations have been drawn on the requirements for an intermediate form used to capture VLSI designs. While not providing a high level of abstraction directly, such as a silicon compiler which maps from a behavioural description to a physical layout, an intermediate form provides the framework on which to build tools dealing with designs at a higher level. SHIFT, a structured hierarchical intermediate form for VLSI design tools, has been defined and partially implemented.

SHIFT uses a separated hierarchy of leaf cells and composition cells. Leaf cells specify the actual artwork necessary to produce fabrication masks. Composition cells contain compositions of leaf cells and other (simpler) composition cells. Cells are composed by abutting together ports on adjoining walls, stretching them if necessary. Relationships between ports are defined in terms of minimum or exact distance constraints between them. A hierarchical method is used for solving the constraint graphs produced from composition. SHIFT is embedded in Lisp and consists of approximately 3000 lines of Franz Lisp code.

SHIFT is a keystone of EDICT, a VLSI design tool environment under construction at the University of Calgary. It is also the intermediate form

used in a symbolic layout editor being developed, and will be the intermediate language for future work by VLSI groups in the Computer Science Department at Calgary. Finally, it is used in a primitive design library called *shiftlib* which is built on the JADE distributed environment using the *Jipth* lisp interface to JADE. Shiftlib serves as a prototype for the library assistant envisaged in EDICT.

6.2. Observations on SHIFT

One observation that has been made is that SHIFT is devoid of the syntactic sugar that makes a language easy to program. However, SHIFT was designed as an *intermediate* form, and its syntax makes it easy for tools to generate SHIFT code in an automatic manner. Thus one tool which should be built on SHIFT would be a procedural interface which would allow designs to be specified as programs. Another observation made by the author was that designing leaf cells by hand using SHIFT for the examples generated some unexpected constraint solutions, primarily because the constraints between ports on the walls went through interior ports. It is not easy when laying out sizable leaf cells procedurally to think in terms of constraints. It is much easier to think of them graphically. It is expected that a symbolic layout editor will greatly facilitate the use of SHIFT for designing leaf cells.

Composing cells procedurally using SHIFT will not give the same problems, since the chief concern is the abutment of cells, and any additional constraints must only be between border ports. This observation, however, must await confirmation by others using SHIFT. The lack of design experience using SHIFT has hindered making many observations about its effectiveness as an intermediate form, but early work such as a shift-to-lap filter and

shiftilib is satisfactory. SHIFT is expected to be thoroughly tested by EDICT.

6.3. Future Research Directions

Future work to be done on SHIFT includes the incorporation of electrical properties in the design using constraints in the form of local maxima. This would allow the designer to specify that a given cell not exceed a certain size, and therefore some critical power level or time delay. This is necessary as designers must meet certain global constraints with regard to power consumption and speed that are present in any real design. It would also allow the designer to get feedback on the critical paths present in a design at a very early stage in the development process.

Also, a proof system for behaviours needs to be connected to SHIFT. One possibility, as mentioned in the previous chapter, would be to use the existing LCF-LSM system as developed by Gordon [Gord83] by coupling it to SHIFT using Jade. Another possibility would be to use an alternative system such as VERIFY [Barr84], which is written in Prolog, and has the advantage of being fully automated.

Finally, methods need to be developed to specify interior constraints of leaf cells in such a manner that stretching would never result in design rule violations. One possibility would be to incorporate the technique of rift lines [Widd84] into a layout editor. The user would layout the cell symbolically, and then draw rift lines where he wanted the cell to stretch.

Since SHIFT is embedded in Lisp it has the advantage of being an open-ended intermediate form, and as such, it is easy to incorporate new ideas into it. This will insure its use over a longer period of time than other intermediate forms. It will also have beneficial effects for tools using it, since there will

be a strong desire to integrate these tools into a solid and workable VLSI design environment that can change continuously to use new ideas as they are developed.

References

[Back78]

Backus, J.W. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *Comm ACM*, August 1978.

[Barb81]

Barbacci, M.R. "Instruction Set Processor Specifications (ISPS): The Notation and Its Application." *IEEE Transactions on Computers* C-30(1):24-40, January, 1981.

[Barr84]

Barrow, Harry. "Proving the Correctness of Digital Hardware Designs", *VLSI Design*, Vol. 5, No. 7, July 1984, pp. 64-77.

[Bata81]

Battall, J., Mayle, N., Shrobe, H., Sussman, G., and Weise, D. "The DPL/Daedalus Design Environment", *VLSI 81*, The Proceedings of the First International Conference on Very Large Scale Integration, August 1981, J. Gray (editor).

[Birt73]

Birtwistle, G., Dahl, O-J., Myhrhaug, B. and Nygaard K. *SIMULA begin*, Studentlitteratur, Lund, Sweden, 2nd ed., 1979.

[Birt84]

Birtwistle, G., Hill, D., Kendall, J., Coates, B., Esau, R., Kroeker, W., Liblong, B., Liu, E., Melham, T. and Schedlwy, R. *EDICT - An*

Environment for Design using Integrated Circuit Tools, University of Calgary Computer Science Research Report No. 84/155/13, June 1984.

[Brow83]

Brown, H., Tong, C., and Foyster, G. "Palladio: An Exploratory Environment of Circuit Design". *IEEE Computer*, December 1983, pp. 41-56.

[Brya81]

Bryant, Randal. *A Switch Level Simulation Model for Integrated Circuits*. MIT Laboratory for Computer Science Technical Report-259.

[Buch80]

Buchanan, Irene. *Modelling and Verification in Structured Integrated Circuit Design*, PhD Thesis, Department of Computer Science, University of Edinburgh, 1980.

[Buch82]

Buchanan, Irene. *Scale - A VLSI Design Language*. Technical Report CSR-117-82, University of Edinburgh, Department of Computer Science, May 1982.

[Gord81]

Gordon, Mike. *A Model of Register Transfer Systems with applications to Microcode and VLSI correctness*. Department of Computer Science Internal Report CSR-82-81, University of Edinburgh, March 1981.

[Gord83]

Gordon, Mike. *LCF-LSM*. University of Cambridge Computer Laboratory Technical Report No. 42, 1983.

[Gos183]

Gosling, J. *Algebraic Constraints*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, May, 1983.

[Hell79]

Heller, W.R. *An Algorithm for Chip Planning*, Caltech Silicon Structures Project File #2806, 1979.

[Joha79]

Johannsen, D. "Bristle Blocks -- A Silicon Compiler", *Proc. of The 16th Design Automation Conference*, 1979.

[John84]

Johnson, Stephen C. "Top-down system design through silicon compilation", *Electronics*, Vol. 57, No. 9, pp. 121-128. May 3, 1984.

[King82]

Kingsley, Chris. *Earl: An Integrated Circuit Design Language*. Caltech Technical Report 5021, June 1982.

[Lib183]

Liblong, B. M., Birtwistle, G. M. "A VLSI Design System Based Upon a High Level Intermediate Form", *1983 Canadian Conference on Very Large Scale Integration*, Waterloo Ont., 1983, pp. 150-153.

[Lib184a]

Liblong, B.M., and Bonham, M. *Jipth - The Lisp - Jipc Interface*, University of Calgary Computer Science Technical Report in preparation.

[Lib184b]

Liblong, B.M. *The SHIFT Users Manual*, in preparation.

[Libl84c]

Liblong, B., Melham, T., Birtwistle, G., Kendall, J. "Towards A VLSI Design Tool System", *Proceedings of CIPS Session 84*, Calgary, Alta., 1984.

[Lipt83]

Lipton, R.J., Valdes, J., Vijayan, S.C., North, S.C., and Sedgewick, R. "VLSI Layout as Programming", *ACM Transactions on Programming Languages and Systems*, Volume 5, Number 3, July, 1983.

[Loca78]

Locanthi, B. *LAP: A Simula Package for IC Layout*. Caltech Technical Report Display File #1862, July, 1978

[Mead80]

Mead, Carver and Conway, Lynn. *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[Moor79]

Moore, G.E. "Are We Really Ready For VLSI?", *Proceedings of the Caltech Conference on VLSI*, January, 1979, C. Seltz (editor).

[Most81]

Mosteller, R.C. *REST - A Leaf Cell Design System*. M.Sc. Thesis, Silicon Structures Project Technical Report 4317, Caltech, December, 1981.

[Nage75]

Nagel, L.W. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. ERL Memo ERL-M520, University of California, Berkeley, May 1975.

[Rem81]

Rem, Martin. "The VLSI Challenge: Complexity Bridling", *VLSI 81*, The Proceedings of the First International Conference on Very Large Scale Integration, August 1981, J. Gray (editor).

[Rows80]

Rowson, James Allely. *Understanding Hierarchical Design*, PhD thesis, Caltech Technical Report 3710, April 1980.

[Sand79]

Sandewall, E. "Programming in the Interactive Environment: The LISP Experience", *ACM Computing Surveys*, Vol. 10, No. 1, March, 1978, pp. 35-72.

[Selt79]

Seltz, C. "Self-Timed VLSI Systems", *Proceedings of Caltech Conference on VLSI*, January 1979.

[Spro80]

Sproull, R. F., and Lyon, R. F. "The Caltech Intermediate Form for LSI Layout Description", from [Mead80], 1980.

[Trim80]

Trimberger, S. *The Proposed Sticks Standard*, Caltech Computer Science Department. Technical Report #3380, 1980.

[Trim81]

Trimberger, S., Rowson, J., Lang, C. and Gray, J. "A Structured Design Methodology and Associated Software Tools", *IEEE Trans. on Circuits and Systems*, Vol. CAS-28, No. 7, July 1981, pp.618-633.

[Ullm84]

Ullman, J.D. *Computational Aspects of VLSI*, Computer Science Press, 1984.

[Unge84]

B.W.Unger et al. "JADE: a software simulation and prototyping environment". *Proceedings of the Conference on Simulation in Strongly Typed Languages*, San Diego, 1984.

[vanC79]

vanCleemput, W. M. "Hierarchical Design for VLSI: Problems and Advantages", *Proceedings of Caltech Conference on VLSI*, January 1979.

[Wall83]

Wallich, P. "Tomorrow's Computers - The Challenges", *IEEE Spectrum*, November 1983, pp. 73-77.

[Whit81]

Whitney, T. *A Hierarchical Design Rule Checker*, Caltech Computer Science Department. Technical Report #4320, 1981.

[Widd84]

Widdowson, Rod. *An Investigation of Stretchable Cells in SCALE*. To appear as a University of Edinburgh Computer Science Technical Report in late 1984.

[Wille84]

Wilensky, R. *LISPCraft*, W.W. Norton & Company, New York, 1984.

[Will77]

Williams, J.D. *STICKS - A New Approach to LSI Design*, MIT MSEE Thesis, 1977.

APPENDIX A

Syntax of SHIFT

The following comprises a user-level syntactic description of SHIFT. The description method was chosen for readability and because it gives some flavour of the semantics of the functions. The syntax used is a modified form of BNF, where constructs enclosed in brackets (`[]`) are optional, constructs enclosed in braces (`{ }`) in conjunction with the vertical bar (`|`) mean choose one of, and both forms may be modified with a repetition factor. The repetition factor may be `"*"`, meaning 0 or more times, `"+"` meaning 1 or more times, and `" +x"`, meaning x or more times. Non-terminals are denoted by names beginning with `"l_"`, denoting a list expression, `"s_"`, denoting a symbol, `"n_"`, denoting a number, `"h_"`, denoting a cell structure, and `"p_"`, denoting a point expression. For more information, consult the SHIFT Users Manual [Libl84b].

1. Defining Cells

Cell definitions have the following syntax.

```
s_leaf-cell_definition ::=
    (defleaf s_cell-name
      [l_leaf_ports_expr]
      [l_const_expr]
      [l_struct_expr]
      [l_beh_expr]
      [l_geom_expr])
```

```

s_composition-cell_definition ::=
    (defcomp s_cell_name
      [l_comp_ports_expr]
      [l_const_expr]
      [l_struct_expr]
      [l_beh_expr]
      [l_comp_expr])

l_leaf_ports_expr ::=
    (ports [(north [s_port]*)]
      [(south [s_port]*)]
      [(east [s_port]*)]
      [(west [s_port]*)]
      [(interior [s_port]*)])

l_comp_ports_expr ::=
    (ports [(north [s_port]*)]
      [(south [s_port]*)]
      [(east [s_port]*)]
      [(west [s_port]*)])

l_const_expr ::=
    (const [(s_port
      { >= | = | ^!! | !! }
      s_port
      [ { + | - } n_value])])*)

l_struct_expr ::=
    (struct [(nodes
      [s_node_name
      (s_component [s_attribute n_val]*)*)]
      [(connect
      [( { s_port | (s_component_node s_node_name) }
      { s_port | (s_component_node s_node_name) })])])*)

l_beh_expr ::=
    (beh ([s_state]*)
      ([ (s_port = e_val) ]*)
      ([e_next_state]*))

l_geom_expr ::=
    (geom l_geom_primitives)

l_comp_expression ::=
    ({ > | < | ^ | v } {s_cell_name | l_comp_expression}+)

```

```

l_geom_primitives ::=
    ({l_geom_primitive | l_geom_primitives}*)

```

2. Geometry Primitives

Since SHIFT is a procedural form, one can write and use functions which return lists of geometry primitives. In particular, there are basic functions for specifying relative paths in the path primitive, for applying transformations to primitives and lists of primitives, predicates and selectors which can be used to write user geometry functions, and cmos and nmos functions which take higher-level concepts such as transistors and map them into lists of geometric primitives.

2.1. Basic Geometry Primitives

```

l_geom_primitive ::=
    l_box_primitive | l_polygon_primitive |
    l_wire_primitive | l_geometry_function |
    l_geometry_function

```

```

l_box_primitive ::=
    (box [s_layer] { l_point l_point | l_path })

```

```

l_polygon_primitive ::=
    (polygon [s_layer] { [l_point]+3 | l_path })

```

```

l_wire_primitive ::=
    (wire [s_layer] [s_width] { [l_point]+2 | l_path })

```

```

l_path ::=
    (path l_point { l_point | l_abs-motion | l_rel-motion }*) |
    (pmerge { l_point | l_path }+ ) |
    (perim l_box_primitive) |
    (lengthen-path l_path n_first n_last)

```

```

l_point_primitive ::=
    l_geom_primitive | l_path

```

```
l_abs-motion ::=
    (then-x n_expr) | (then-y n_expr) | (then-xy n_expr n_exp)
```

```
l_rel-motion ::=
    (by-x n_expr) | (by-y n_expr) | (by-xy n_expr n_epxr)
```

```
l_geometry_function ::=
    l_cmos_geometry_function | l_nmos_geometry_function |
    l_user_defined_function | l_compound_geometry_function
```

2.2. Selectors and Predicates

```
l_selector ::=
    (:type l_point_primitive) |
    (:layer l_geom_primitive) |
    (:width l_geom_primitive) |
    (:low l_box_primitive) |
    (:high l_box_primitive) |
    (:path l_point_primitive) |
    (:nth l_point_primitive)
```

```
l_geom_predicate ::=
    (layerp s_layer) |
    (pathp l_geom_primitive) |
    (widthp n_expr)
```

2.3. Transformation Functions

```
l_transform ::=
    (apply-fcn f_function l_point_primitive) |
    (trans-pt l_point l_point_primitive) |
    (trans-x n_expr l_point_primitive) |
    (trans-y n_expr l_point_primitive) |
    (trans-xy n_expr n_expr l_point_primitive) |
    (scale-pt l_point l_point_primitive) |
    (scale-x n_expr l_point_primitive) |
    (scale-y n_expr l_point_primitive) |
    (scale-xy n_expr n_expr l_point_primitive) |
    (mr-x l_point_primitive) |
    (mr-y l_point_primitive) |
    (mr-xy l_point_primitive) |
    (rot-pt l_pt n_expr l_point_primitive) |
    (rot n_expr l_point_primitive) |
    (apply-tx l_geom_primitives l_trans_mat) |
    (apply-rot l_geom_primitives l_point)
```

2.4. nMOS Geometry Functions

```

l_geom_primitives ::=
  (dm) |
  (dm-at l_point) |
  (pm) |
  (pm-at l_point) |
  (dpeast) |
  (bn-at l_point) |
  (be-at l_point) |
  (bs-at l_point) |
  (bw-at l_point) |
  (but-rot-at l_point l_point) |
  (pulldown n_diff-width l_diff-path n_poly-width l_poly-path) |
  (passtran n_diff-width l_diff-path n_poly-width l_poly-path) |
  (pullup n_diff-width l_diff-path n_poly-width l_poly-path) |
  (enhtran (diff-width diff-path poly-width poly-path))

```

2.5. CMOS Geometry Functions

```

l_geom_primitives ::=
  (am) |
  (am-at l_point) |
  (pm) |
  (pm-at l_point) |
  (apeast) |
  (bn-at l_point) |
  (be-at l_point) |
  (bs-at l_point) |
  (bw-at l_point) |
  (ameast) |
  (sn-at l_point) |
  (se-at l_point) |
  (ss-at l_point) |
  (sw-at l_point) |
  (n&p+-box l_box_primitive) |
  (pwell&guards-box l_box_primitive) |
  (gate n_active-width l_active-path n_poly-width l_poly-path) |
  (pmos n_active-width l_active-path n_poly-width l_poly-path) |
  (nmos n_active-width l_active-path n_poly-width l_poly-path) |
  (split-rot-at l_point l_point) |
  (split-&-n&pplus-rot-at l_point l_point)

```

2.6. Miscellaneous Functions

```

l_misc_geom_functions ::=
  (union-box l_box_primitive l_box_primitive) |
  (inflate-box l_box_primitive n_value) |
  (cbox [s_layer] l_point n_horiz n_vert) |
  (box-to-polygon l_box_primitive) |
  (mbb l_point_primitive) |
  (shift-to-cif-wire l_wire_primitive) |
  (cif-to-shift-wire l_wire_primitive) |
  (mbb-wire l_wire_primitive)

```

3. Points

Points are simply a structure of two numbers. Operations are provided for the creation, selection, manipulation, and transformation of points.

3.1. Creation, Selection, and Relational Functions

```

l_point ::=
  s_port | (point n_expr n_expr)

```

```

l_point_selection ::=
  (:x l_point) | (:y l_point)

```

```

l_point_ops ::=
  (pointp l_point)
  (pt= l_point l_point) |
  (pt/= l_point l_point) |
  (pt< l_point l_point) |
  (pt> l_point l_point) |
  (pt>= l_point l_point) |
  (pt<= l_point l_point)

```

3.2. Point Manipulations

```

l_point ::=
  (pt+ l_point l_point) |
  (pt- l_point l_point) |
  (pt* l_point l_point) |
  (pt/ l_point l_point) |
  (pt-scale n_expr l_point) |
  (pt-rot n_expr l_point) |
  (pt-dx l_point n_expr) |

```



```

(pt-dy l_point n_expr) |
(pt-dxy l_point n_expr n_expr) |
(pt-minus l_point) |
(sq-pt l_point) |
(xy-sum l_point) |
(xy-difference l_point) |
(xy-times l_point) |
(xy-quotient l_point) |
(dist l_point l_point) |
(pt-max l_point l_point) |
(pt-min l_point l_point) |
(pt-round l_point) |
(pt-trunc l_point)

```

3.3. Point Transformations

```

l_point ::=
  (normalize l_point) |
  (pt-mult l_point l_trans_mat) |
  (identity) |
  (trans-mat l_point) |
  (scale-mat l_point) |
  (rot-mat l_point) |
  (pre-mult l_trans_mat l_trans_mat)

```

4. Instantiation and Selection

Instantiation of a design returns the instance-name of the design.

```

s_instantiation ::=
  (instantiate 's_cell_name)

```

An instance's fields may be selected with the following functions.

```

sl_prototype_selectors ::=
  (cell-prototype-source h_leaf_or_comp_prototype) |
  (cell-prototype-date-created h_leaf_or_comp_prototype) |
  (cell-prototype-version h_leaf_or_comp_prototype) |
  (cell-prototype-name h_leaf_or_comp_prototype) |
  (cell-prototype-ports h_leaf_or_comp_prototype) |
  (cell-prototype-constraints h_leaf_or_comp_prototype) |
  (cell-prototype-reduced-constraints h_leaf_or_comp_prototype) |
  (cell-prototype-behaviour h_leaf_or_comp_prototype) |
  (cell-prototype-structure h_leaf_or_comp_prototype) |
  (cell-prototype-struct-fcn h_leaf_or_comp_prototype) |
  (cell-prototype-user-extensions h_leaf_or_comp_prototype) |

```

```

(leaf-prototype-geometry h_leaf_prototype) |
(leaf-prototype-geom-fcn h_leaf_prototype) |
(comp-prototype-composed-of h_comp_prototype) |
(comp-prototype-merged-constraints h_comp_prototype) |
(comp-prototype-composed-ports h_comp_prototype) |
(comp-prototype-composed-interior h_comp_prototype)

```

```

sl_instance_selectors ::=
  (cell-instance-name h_cell_instance) |
  (cell-instance-class h_cell_instance) |
  (cell-instance-border-values h_cell_instance) |
  (cell-instance-wall-values h_cell_instance) |
  (cell-instance-port-values h_cell_instance) |
  (cell-instance-structure h_cell_instance) |
  (leaf-instance-geometry h_leaf_instance) |
  (comp-instance-composed-of h_comp_instance)

```

5. Other Functions

These include a geometry extractor and functions for version control.

```

s_version_fcns ::=
  (get-version s_cell_name) |
  (get-creation-date s_cell_name)

l_geometry_extractor ::=
  (get-geometry s_cell_name s_instance)

```

APPENDIX B

SHIFT Examples

1. The Shift Register Leaf Cell

The shift register in chapter 4 is reproduced here in its entirety.

```
(defleaf shiftreg
  (ports
    (north n:clock) (south s:clock)
    (east e:gnd out e:vdd) (west w:gnd in w:vdd)
    (interior
      gc pc last                ; ground, power and butting contacts
      pd.gtin pd.gtout pd.src pd.drn ; pulldown nodes
      pu.gtin pu.gtout pu.src pu.drn ; pullup nodes
      pt.gtin pt.gtout pt.src pt.drn ; passtran nodes
      middle))                ; inverter output
  (const
    (w:gnd ^!! south + 2) (in ^!! w:gnd + 4) ; west wall
    (north ^!! w:vdd + 5) (w:vdd ^!! pu.drn + 1)
    (out >= last + 3)
    (e:gnd ^!! south + 2) (out ^!! e:gnd + 4) ; east wall
    (e:vdd ^!! out + 2) (e:vdd !! w:vdd)
    (s:clock >= pt.src + 3) (n:clock = s:clock) ; south & north walls
    (gc = west + 5) (gc ^!! south + 2) ; ground contact
    (pd.src = gc) (pd.drn = pd.src) ; pulldown
    (pd.gtin >= in + 1) (pd.gtout >= pd.gtin + 8)
    (pd.src ^!! gc + 1) (pd.gtin ^!! pd.src + 3)
    (pd.gtin !! in) (pd.gtout !! pd.gtin)
    (pd.drn ^!! pd.gtin + 3)
    (middle ^!! pd.drn + 1) (pu.src ^!! middle + 1) ; inverter output
    (middle = pd.drn)
    (pu.src = middle) (pu.gtin = pu.src) ; pullup
    (pu.gtout = pu.gtin) (pu.drn = pu.gtout)
    (pu.gtin ^!! pu.src + 2) (pu.gtout !! pu.gtin + 7)
    (pu.drn ^!! pu.gtout + 2)
    (pc = pu.drn) (pc !! w:vdd) ; power contact
    (pt.gtout = s:clock) (pt.src >= pd.gtout) ; passtran
    (pt.drn >= pt.gtin + 3) (pt.gtout ^!! pd.gtout + 1)
    (pt.gtin ^!! pt.src + 3) (pt.src !! middle)
    (pt.gtin = s:clock) (pt.drn !! pt.src)
    (last !! pt.drn) (last >= pt.drn + 3)) ; last contact
  (geom
    (dm-at gc) ; contact between pulldown and ground
```

```

(wire metal 4 w:gnd gc e:gnd) ; ground wire
(pulldown ; pulldown has two parts
  4 (path pd.src pd.drn) ; diffusion path from source to drain
  2 (path pd.gtin pd.gtout)) ; a poly path from gtin to gtout
(wire poly in pd.gtin) ; connect gate to input port
(wire diffusion gc pd.src) ; connect pulldown to ground
(pullup ; pullup has four parts
  2 (path pu.src pu.drn) ; a diffusion path from source to drain
  6 (path pu.gtin pu.gtout)) ; a poly path from gtin to gtout
                                ; an implant layer is automatically drawn
                                ; over the poly layer, extended by 2 lambda
                                ; on either end, and a butting contact at the
                                ; gate input connecting the gate to the source
(wire diffusion pd.drn middle pu.src); connect the pullup and pulldown
(wire metal 4 w:vdd pc e:vdd) ; power wire
(wire diffusion pu.drn pc) ; connect pullup to power
(dm-at pc) ; contact between pullup and power
(passtran ; passtran is equivalent to the pulldown
  2 (path pt.drn pt.src)
  2 (path pt.gtin pt.gtout))
(wire poly n:clock pt.gtin) ; wire up clock to one end of the gate
(wire poly s:clock pt.gtout); wire up other end of gate to clock
(wire diffusion middle pt.src) ; connect inverter output to passtran's
                                ; source
(be-at last) ; butting contact for passtran to out
(wire diffusion pt.drn last) ; connect passtran to contact and
(wire poly (pt-dx last 1) ; contact to out
  (then-y (:y out)) out))
(struct
  (nodes
    pullup (n-type-dep len 6 wid 2)
    pulldown (n-type-enh len 2 wid 6)
    pass (n-type-enh))
  (connect
    (e:vdd w:vdd) (e:gnd w:gnd) (n:phi s:phi)
    ((:source pullup) e:gnd) ((:source pulldown) e:vdd)
    ((:drain pullup) (:drain pulldown))
    ((:drain pullup) (:source pass))
    ((:gate pullup) (:drain pulldown))
    ((:drain pass) out) ((:gate pulldown) in)
    ((:gate pass) n:phi)))
  (beh (s)
    ((s:phi = n:phi)
      (out = if n:phi then (not s) else @))
    ((if (in = @) then s else in)))
)

```

2. A Shift Register Array

The definition of a 4 element bit-slice looks like:

```
(defcomp shiftslice
  (> (> shiftreg shiftreg)
    (> shiftreg shiftreg)))
```

and the definition of a 4 bit wide bit-slice (Figure B.1.) is simply:

```
(defcomp shiftarray
  (^ (^ shiftslice shiftslice)
    (^ shiftslice shiftslice)))
```

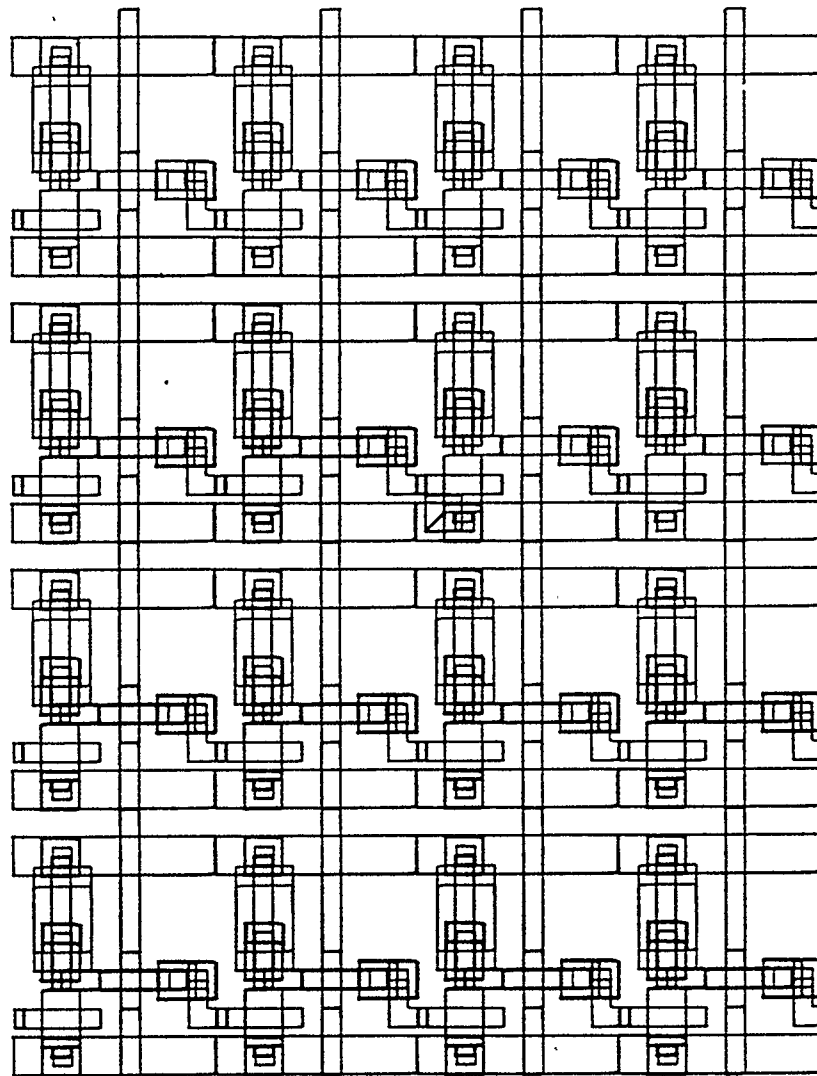


Figure B.1. Geometry of a 4 x 4 Shift Register Array