A Design by Example Regular Structure Generator

Cyrus S. Bamji

Technical Report 507

February 1985

Massachusetts Institute of Technology
Research Laboratory of Electronics
Cambridge, Massachusetts 02139

A Design by Example Regular Structure Generator

Cyrus S. Bamji

Technical Report 507

February 1985

Massachusetts Institute of Technology
Research Laboratory of Electronics
Cambridge, Massachusetts 02139

This work has been supported in part by the U.S. Air Force Office of Scientific Research Contract F49620-84-C-0004.

A DESIGN BY EXAMPLE
REGULAR STRUCTURE GENERATOR

by

Cyrus S. Bamji

Submitted to the
Department of Electrical Engineering and Computer Science
on February 28, 1985 in partial fulfillment of the requirements
for the degree of Master of Science.

Abstract

This thesis investigates technical issues concerning the automated generation of highly regular VLSI circuit layouts (e.g. RAMs, PLAs, systolic arrays) that are crucial to the designability and realizability of large VLSI systems. The key is to determine the most profitable level of abstraction, which is accomplished by the introduction of true *macro abstraction*, *interface inheritance*, *delayed binding*, and the *complete decoupling* of procedural and graphical design information. These abstraction mechanisms are implemented in the Regular Structure Generator, an operational layout generator with significant advantages over first generation layout tools. Its advantages are demonstrated by a pipelined array multiplier layout example. A *leaf cell* compactor that can make the RSG *technology transportable* is also investigated.

Thesis Supervisor: Jonathan Allen
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor Jonathan Allen whose insight and guidance have helped put this work in the right perspective and have given it the right direction. It has been my privilege to work with him.

I thank Charles Hauck for substantially contributing to the form as well as the content of this thesis. I have learned a great deal from our teamwork.

I would like to thank Robert Armstrong, Don Baltus, Paul Bassett and Steven McCormick for their many ideas and often needed help.

I especially want to thank my parents whose love and support have been the backbone of my education.

This work was supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Contract Number AFOSR F49620-84-C-0004

Contents

Appendices

List of Figures

Chapter 1

Introduction

1.1 Motivation

Circuit designs with highly regular and repetitive layouts are an effective solution to the VLSI design bottleneck, and therefore occur quite often in large VLSI systems. Familiar examples of regular circuit structures are RAMs, ROMs, PLAs, and array multipliers. In addition, recognition of the importance of regularity in VLSI systems has given rise to a large and continually growing collection of new regular structures for applications in signal processing, image processing, data structures, and CAD, to name a few. Since these designs are computationally powerful and widely applicable, there is a great demand for circuit design tools that make these structures generally accessible. This thesis describes a CAD tool, the Regular Structure Generator (RSG), that helps meet this demand by performing automatic generation of regular structure layouts and providing the means to efficiently capture, in all their richness and variety, most practical regular circuit designs.

Despite the uniform and repetitive appearance of their layouts, effective

regular structure circuits are not simply bland arrays of identical, abutting cells. In practice, there is always some degree of complexity along the edges of a regular array, and each design instance must be parametrically personalized with respect to problem size and functionality. This requires the placement of a variety of cell maskings that implement such options as transistor and bus sizing, cell interfacing, clock assignment, and functional encoding — a task which cannot be accomplished by the simple array generating commands found in graphics editors. Although regularity does permit most regular structures to be personalized in an algorithmic manner, a high degree of flexibility is still required in the placement and orientation of the cells and cell maskings. Insofar as first generation VLSI layout tools lack this high degree of flexibility, there is an opportunity for developing more advanced module generators that fulfill this need.

The RSG was developed with this approach to regular circuit layout in mind. The input language used for the procedural specification of circuit architecture is a subset of Lisp. Consequently, abstraction mechanisms are available to support a highly functional set of primitives for defining regular structures and evaluating the complex conditionals required by personalization and edge effects. Personalization is further supported by the ability to arbitrarily place and orient cells according to interfaces *defined-by-example* in the graphical domain. All design information is efficiently partitioned into procedural and graphical form.

A circuit layout is generated from the following inputs (Figure 1.1): a *design file*, which is a parameterized, procedural description of the architecture; a *layout file*, which is a graphical specification of cell layouts and interfaces; and a *parameter file*, which provides the size and functional speci-

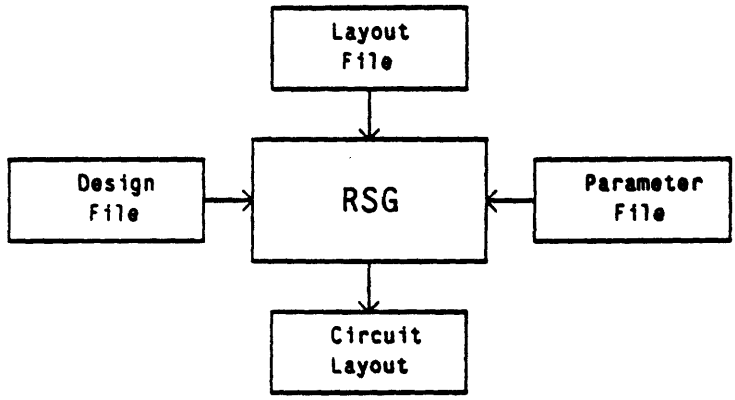


Figure 1.1: RSG Layout Generation

fications for the particular case. By completely decoupling the graphical and procedural domains, a level of modularity is obtained which achieves local efficiency in layout generation, and global efficiency in the management of new architectures, layouts, and interfaces to other CAD tools.

The RSG also supports *macro abstraction*, i.e. the specification of *macrocells* as interconnections of smaller cells whose *binding* to actual layouts can be *delayed* to any desired time. In addition, *interface inheritance* relations provide a procedural means for defining interfaces between any two *macrocells*: a new interface between two *macrocells* can be computed from any legal interface between a subcell in the first *macrocell* and a subcell in the second. As a result, *macrocells* can be used to specify even more complex cells in an entirely procedural manner with no need for additional layout.

At this stage of the discussion, all of the RSG's functionality appears to exist in other layout generators. For instance, procedural specification of circuit layouts is as old as silicon compilation itself, and essentially defines it. The novelty of the RSG is not its use of procedural specification, but rather

the level of abstraction at which it is used. Failure to choose an optimal level of abstraction complicates the user interface, and forces the designer to concentrate as much on the internal constraints of the generator as on the functionality of the circuit being designed. Examples of this are layout generators that require placement of cells by strict abutment, or that do not support true hierarchical *macro abstraction.*

The significant contribution of the RSG is efficiency, not computability, of design. That is, the RSG does not produce any circuit layouts which, given unlimited effort, could not be produced by other layout generators. The result of this efficiency, however, is a tool that performs well in practice, not just in principle, in a realistic VLSI design setting.

1.2 Comparison with other layout generators

1.2.1 Module generators and Silicon Compilers

Specialized VLSI module generators produce layouts of a particular architecture to implement a specific logic function such as PLAs, ROMs, or Weinberger arrays. These module generators produce layouts of a specific style of implementation in a specific technology. For example a PLA generator might generate PLA's with a standard NOR/NOR architecture, implemented with CMOS precharged gates. Such specialized module generators are capable of generating highly optimized layouts within the restricted class they are designed for. This is because these generators can incorporate specific knowledge about the details of their particular implementation. For instance a PLA generator which incorporates knowledge about the particu-

lar process technology and type of circuitry used can be made to size power busses and transistors according to some speed and power criteria. The disadvantage of these specialized module generators is that their scope is limited to the applicability of the specific function they implement and to the specific process technology they use. Other module generators such as HPLA also generate a single architecture but allow freedom in the implementation and choice of technology. All of these module generators take as their input a configuration specification (in the case of a PLA this would be the number of inputs, outputs, product terms and the truth table) and not a high level functional specification, or an architecture specification because functionality of the output layout is implicit in the single architecture they implement.

Silicon compilers start with a functional specification as their input. However current silicon compilers are not capable of determining and implementing the optimal architecture for a given functional specification and technology. These programs use a single canonical architecture into which most functional specifications can be compiled to implement all functional specifications. Their success depends on how well the canonical architecture they use is suited to the functional specification at hand. Macpitts[29] uses a data path implemented with registers, adders, and shifters, and a control path implemented with a Weinberger array as the canonical architecture. While such an architecture may be suited for some applications it clearly is not suited for applications in signal processing which require an efficient implementation for multiplications. Hence even if the program succeeds in keeping the transistor density high by packing a lot of circuitry in a small area, the functional density measured by how much silicon it takes to implement a given functionality is low. This is due to the inappropriate implementation

Generality			Efficiency
1 Canonical Architecture	Multiple Architectures 1 Framework	⇐	1 Architecture per Function
• Macpitts	• RSG		• HPLA[6]
• Bristle blocks[14]			• Multiplier Gen.[5]
			• F.P. ALU Gen.[4]

Figure 1.2: Comparison with other layout generators.

architecture where many more transistors are required than would be the case with a suitable architecture. Early versions of Macpitts required about 5 times the area than would be the case for layouts generated by hand.

Unlike specialized module generators and today's silicon compilers the RSG can generate many different architectures with just one framework. By matching the architecture to the functionality a level of generality greater than that of specialized generators can be achieved without the loss of efficiency incurred in current silicon compilers by a mismatched target architecture. Another big difference between silicon compilers and the RSG is that silicon compilers start with a function description of the problem whereas the RSG starts with user-defined primitive cells and cell connectivity information (as shown in Figure 1.1). Figure 1.2 shows how the RSG is moving toward greater generality than specialized compilers without the loss of efficiency incurred in todays silicon compilers.

1.2.2 RSG as a superset of HPLA

The RSG expands the scope of HPLA by allowing many different archi-

tectures to be generated with the same benefits as in the case of HPLA, but with just one framework. Though many of the features of the RSG can be explained and justified independently of HPLA, HPLA ideas have inspired and motivated the design of the RSG. HPLA does not support many of the key features of the RSG such as macro abstraction, inheritance and macro cell abstraction. Also the algorithms and software techniques used in the RSG are totally different from those used in HPLA. HPLA uses a cell relocation scheme whereas the RSG uses interfaces and an interface table. However both the RSG and HPLA use the idea that adjacent (primitive) cells in the final layout interface in the same way as they do in the sample layout. Hence in both programs the (primitive) cell definitions and spacing parameters are extracted from a sample layout.

In HPLA the sample layout was an actual assembled PLA and hence had the same architecture as the final layout. This constraint that the sample layout be a fully assembled PLA is actually superfluous. Using the same methods as those used in HPLA (i.e. relocation) it is possible to achieve the same results from a sample layout consisting of the PLA cells with the only other constraint being that all possible interfaces that might occur in the final layout be present in the sample layout. The fact that the sample layout was a *two input, two output, two product term* PLA was simply a way to ensure that all the required cells and interfaces between them be present in the sample layout because the architectural specification for PLAs is already hard coded in the HPLA program itself and is not extracted from the sample layout.

In the RSG this constraint is relaxed. This not only reduces the size and complexity of the sample layout, but it also allows the same sample layout

to be used in output layouts of various different architectures because the implicit architecture always present in the sample layout does not constrain the architecture of the final layout. The sample layout in HPLA was actually larger than necessary and contained redundant information. For example the sample layout for HPLA contained 2 (identical) instances of the and-sq connect-ao interface when only one was required. In so doing it increased the number of instances of and-sq and connect-ao making the sample layout larger than necessary. The cells in many PLA sample layouts can also be used to generate other layouts besides PLAs such as decoders and multiplexors (decoders can be built from an *AND* plane with appropriate output buffers). Hence requiring that the sample layout look like the finished product is not only an unnecessary restriction it also reduces the scope within which any given sample layout may be used.

The method (relocation) HPLA uses to generate new cells does not easily lend itself to cell hierarchy. This did not matter in HPLA because the architecture that HPLA generates (i.e. the architecture for standard PLAs) does not make use of cell hierarchy. Making use of cell hierarchy entails generating a macro cell from the primitive cells in the sample layout replicated according to some parameter, and then calling the new macro cell in an even higher order cell several times according to some other parameter. In the relocation scheme the cell definitions for subcells of a higher order cell are actually modified to suit the needs of the calling cell. This worked fine in HPLA because there was only one calling cell, i.e. the complete layout of the PLA. In a scheme which uses hierarchy there may be many higher order cells (which can possibly be called in even higher order cells), that call the same subcell. Each of these cells may request that the called subcell be modified in

some particular fashion to suit its specific needs. These modification requests can be conflicting. One way to solve the problem would be to create a copy of the subcell for each of the calling cells. Hence each calling cell can modify its copy of the subcell without conflicting with the modifications requested by the other calling cells. The RSG however uses a simpler and more powerful technique where this problem does not occur.

1.2.3 The description file verses the interface table.

Before HPLA can make a PLA from a sample layout it must first compile the sample into a special file called the description file. This description file contains the definition of all the key cells where the cell definitions have been modified as prescribed by the relocation scheme. It also contains the spacing parameters (pitches) for the various cells. In HPLA, for the users convenience, the process of making a PLA is divided into three parts each of which occur at different times in the design cycle. This division of the generation process allows delayed binding of the specifics of the PLA encoding until after the PLA is fully installed into the rest of a layout. The description file is accessed at each of these three phases, hence it makes sense to create the description file just once and refer to it in each of the three phases of the PLA design.

In the case of the RSG the data structure corresponding to the description file would be the interface table. However since the RSG produces the whole layout all at once, it does not make sense to store the data structure into a file and load it back immediately into the workspace and use it during just one session. Therefore no temporary file is created.

The RSG can generate any PLA that HPLA can. It can also generate

more complex PLAs such as PLAs with folded rows or columns. However in HPLA the division of the generation process into three parts facilitates recoding the PLA (or postponing its encoding) and speeds up the plotting of the chip by leaving out the PLA's crosspoints until required, making HPLA a little more convenient to use.

1.3 Thesis organization

- Chapter 2 lays down the mathematical foundations of *interfaces*, the method the RSG uses for local placement constraints.
- Chapter 3 gives the overall RSG algorithm .
- Chapter 4 Describes the Language for specifying design files and describes in more detail the specifics of the underlying data structures.
- Chapter 5 Describes the design of a class of pipelined multipliers using the RSG.
- Chapter 6 Is concerned with issues relating to building a special type of compactor for use with the RSG.

Each chapter is organized so that the first Sections lay down the concept and the foundations of the method and the last sections go into the details of some important facet of the problem.

Chapter 2

Interfaces

2.1 Cells and Instances

The RSG requires user-defined cells to hierarchically build larger cells. A cell A consists of objects whose locations in the cell are defined in terms of a local coordinate system C_a with origin S_a. The objects in A can be boxes of various layers, points, and instances of other cells. An instance of a cell B is the triplet $(L_b^r, O_b^r, \langle cell\ \ definition \rangle)$ where L_b^r is the point of call of the cell B, O_b^r is the orientation in the call of B and $\langle cell\ \ definition \rangle$ is a pointer to the cell definition of B (the superscript ^{r} means that the location or orientation is relative to a calling coordinate system). The effect of having an instance of B in A with point of call L_b^r and orientation O_b^r is that of performing the isometry[1] O_b^r on B (O_b^r is an isometry that leaves S_b^r, the origin of the coordinate system within B unchanged), placing the origin S_b^r of B at location L_b^r within the coordinate system of A, and finally adding to

[1] An isometry is either a rotation or a reflection.

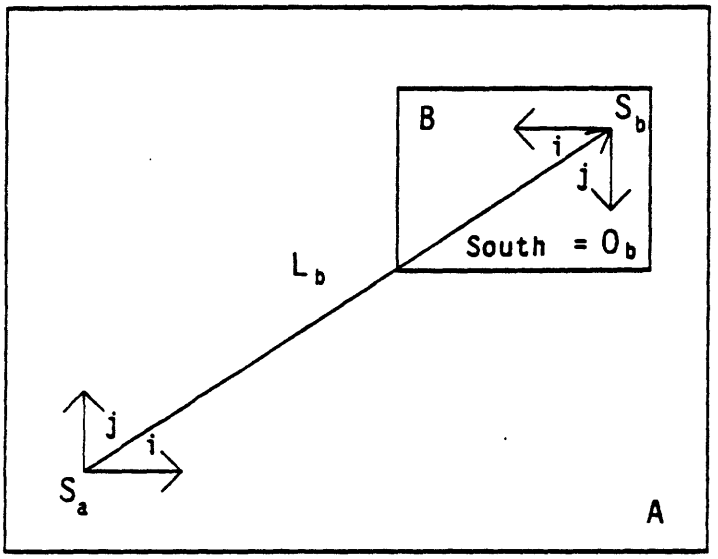


Figure 2.1: Instance of cell B in cell A.

A the collection of objects in B (see Figure 2.1).

2.2 Interface Definition

A key notion in the RSG is the *interface*. If instances of cells A and B (the cells A and B do not necessarily have to be distinct) are to be called within the same coordinate system, then cells A and B have an *interface* between them. The *interface* between two cells A and B is the ordered pair $I_{ab} = (V_{ab}, O_{ab})$ $(I_{ab} \neq I_{ba})$ where V_{ab} is the *interface vector* and O_{ab} is the *interface orientation.* V_{ab} is the vector whose starting point is the point of call of A and whose endpoint is the point of call of B, if the instance of A is held at orientation *north* (identity transform). O_{ab} is the orientation that B would have if the instance of A were held at orientation *north.*

Treating the orientations as operators with "$\circ$" being the operator com-

position rule we have[2]:

$$O_{ab} = (O_a^r)^{-1} \circ O_b^r \tag{2.1}$$

$$V_{ab} = (O_a^r)^{-1}(L_b^r - L_a^r) \tag{2.2}$$

The interface vector V_{ab} and interface orientation O_{ab} are obtained by deskewing the relative orientation of B i.e. O_b^r and the vector $(L_b^r - L_a^r)$ by the inverse orientation of A $(O_a^r)^{-1}$.

Figure 2.2(a) shows an instance of A and an instance of B called together in a same higher order cell (characterized in the Figure 2.2(a) by it's coordinate system (O, i, j)). The point of call L_a^r (respectively L_b^r) of A (respectively B) is the location where the origin of A (respectively B) is placed in the calling coordinate system (O, i, j). In order to obtain the interface I_{ab} between A and B we must first perform an isometry on the calling cell (the one with the (O, i, j) coordinate system in Figure 2.2(b)) such that the new orientation for the instance of A will be $North$. Since A is initially oriented $South$ the calling cell must be reoriented by $South^{-1} = South$ (because $180^o = -180^o$) so that A will ultimately be oriented $North$. Figure 2.2(b) shows the result of the transformation of the calling cell. The interface vector is now the vector whose starting point is at the new point of call A and whose endpoint is at the new point of call of B. The coordinates of the interface vector are computed in terms of the new basis (i', j') which is the same as the old basis (i, j) of the calling cell before the transformation was performed. The interface orientation is now the the new orientation of B after the transformation was performed.

The existence of an I_{ab} interface between A and B automatically gives

[2] O^{-1} is defined by $O^{-1} \circ O = O \circ O^{-1} = Identity$.

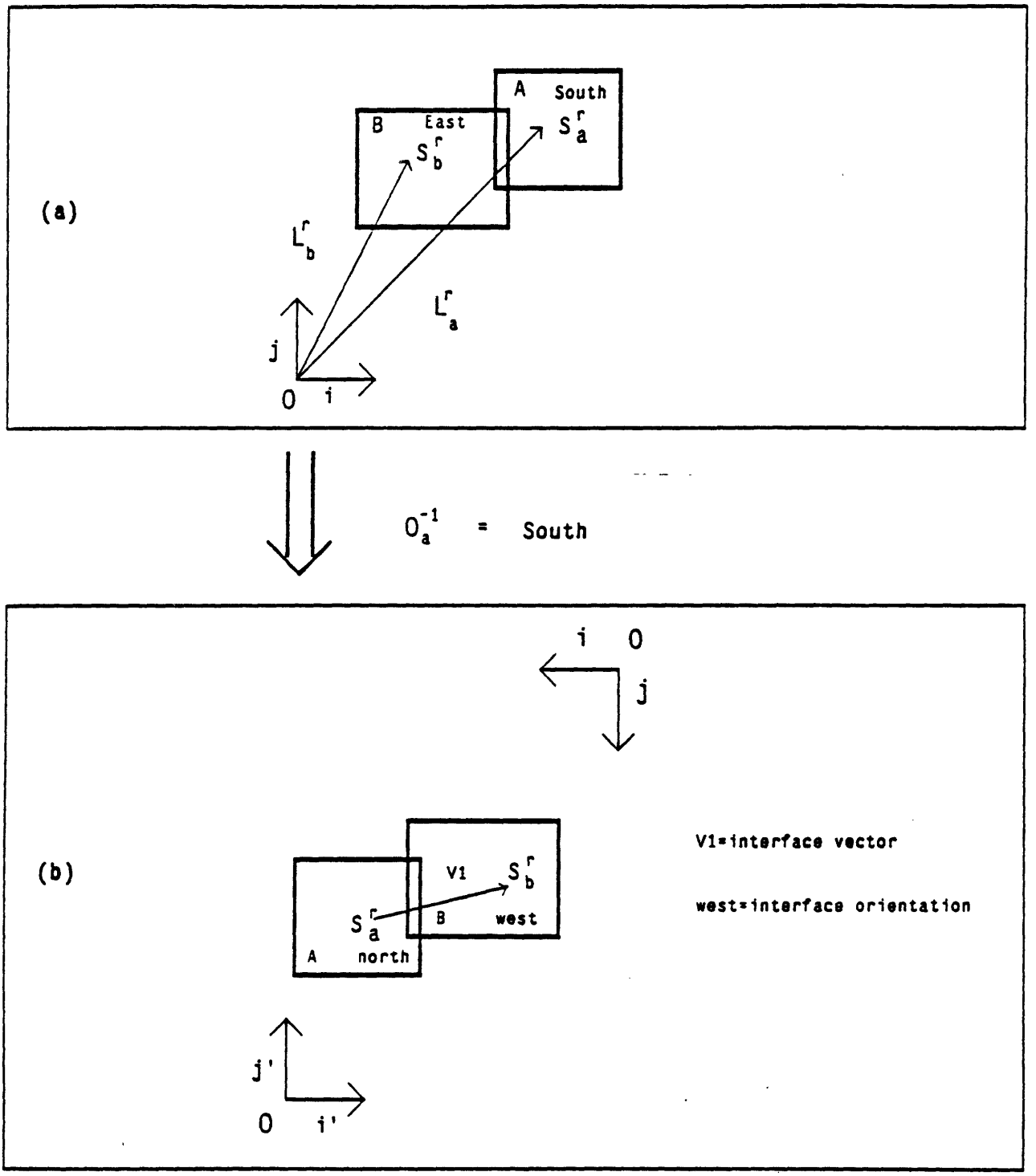


Figure 2.2: Interface between two cells.

rise to an interface I_{ba} between B and A. The expression for the I_{ba} interface can be obtained from equations 2.1 and 2.2.

$$\begin{aligned} O_{ba} &= O_b^{-1} \circ O_a \\ &= (O_a^{-1} \circ O_b)^{-1} \\ &= O_{ab}^{-1} \end{aligned} \tag{2.3}$$

$$\begin{aligned} V_{ba} &= O_b^{-1}(L_a - L_b) \\ &= (O_b^{-1} \circ (O_a \circ O_a^{-1}))(L_a - L_b) \\ &= ((O_b^{-1} \circ O_a) \circ O_a^{-1})(L_a - L_b) \\ &= (O_{ba} \circ O_a^{-1})(L_a - L_b) \\ &= (O_{ab}^{-1} \circ O_a^{-1})(L_a - L_b) \\ &= -O_{ab}^{-1}(O_a^{-1}(L_b - L_a)) \\ &= -O_{ab}^{-1} V_{ab} \end{aligned} \tag{2.4}$$

Therefore $I_{ba} = (V_{ba}, O_{ba}) = (-O_{ab}^{-1}V_{ab}, O_{ab}^{-1})$.

2.3 Advantages of using interfaces

Interfaces are a natural way of defining the relative placement and orientation between instances of cells. Hence knowing the calling information of a cell A in a cell C and knowing the *interface* between A and B it is possible to determine the calling information of B in C. The RSG allows the user to specify the primitive cells and *interfaces* between them graphically, by providing a *layout file* which will henceforth be referred to as the *sample layout*. The *sample layout* contains the definitions of all primitive cells as well as *interfaces* between them. An *interface* between cells A and B can be defined by calling A and B together in a higher order cell C with the appro-

priate relative placement and orientation between them. In practice when new cells are created by the layout designer they are assembled together in order to verify that the different new cells that have been designed, do in fact interface properly to each other. The simple fact of assembling the cells together requires calling them both in one cell (same coordinate system) and therefore automatically defines an interface between them. Hence interfaces can be designed at almost no extra cost to the designer.

By virtue of the *design-by-example* feature of the RSG, the relative placement of neighboring cells in the final layout is such that each *interface* in the final layout is an instance of an *interface* in the *sample layout.*

Since the relative placement of cells in the final layout is performed using *interfaces* between cells and not by using the sizes and shapes of the bounding boxes of those cells, the cells can be designed according to their functional boundary constraints and without regard to abutment constraints. Not only does this make cells easier to design and design rule check (because instances of cells can overlap, each cell can be made design rule correct[3]), the fact that cells are not cut at artificial boundaries helps reduce the proliferation of cells of essentially the same functionality but different abutment constraints. Using *interfaces* also allows cells to be easily encoded by superimposing several cells in order to modify the functionality of a basic cell. This too helps in reducing the proliferation of different cell types since the number of different encoding configurations is roughly exponential in the number of independent encoding decisions.

Cell encoding can also simplify the personalization process since instead of combining all the encoding decisions together to select a single cell of the

[3]Some hierarchical design rule checkers require that instances do not overlap.

appropriate type we can use each independent encoding decision to perform a simple encoding masking of one basic cell. An encoding cell may lie well within the bounding box of the cell it encodes and hence placement by abutment would be cumbersome since it would cause a proliferation of (spacing) cells that have nothing to do with functionality. By simply specifying an *interface* the relative orientation of the cells as well as whether the cells are side by side, one on top of the other, or one inside the other, is handled automatically.

2.4 The Interface Table

The RSG program maintains an *interface table* of all legal (user specified) *interfaces* between cells. This table is first initialized with *interfaces* from the *sample layout* and can be augmented as new cells are created by the system. Since there can be several different legal *interfaces* between two cells there can be a family of legal *interfaces* between two cells A and B. Figure 2.3 shows two different possible *interfaces* for a pair of cells A, B.

If the set of legal *interfaces* between any two cells is indexed over the integers then the *interface table* can be described as a mapping from triplets:

$$(\langle cellname1\rangle, \langle cellname2\rangle, \langle interface \quad index \quad number\rangle) \qquad (2.5)$$

to *interfaces:*

$$(\langle interface \quad vector\rangle, \langle interface \quad orientation\rangle) \qquad (2.6)$$

If I_{ab} is an interface in the *interface table*, then I_{ba}, the *corresponding* interface between B and A, is also loaded in the interface table. Hence knowing the

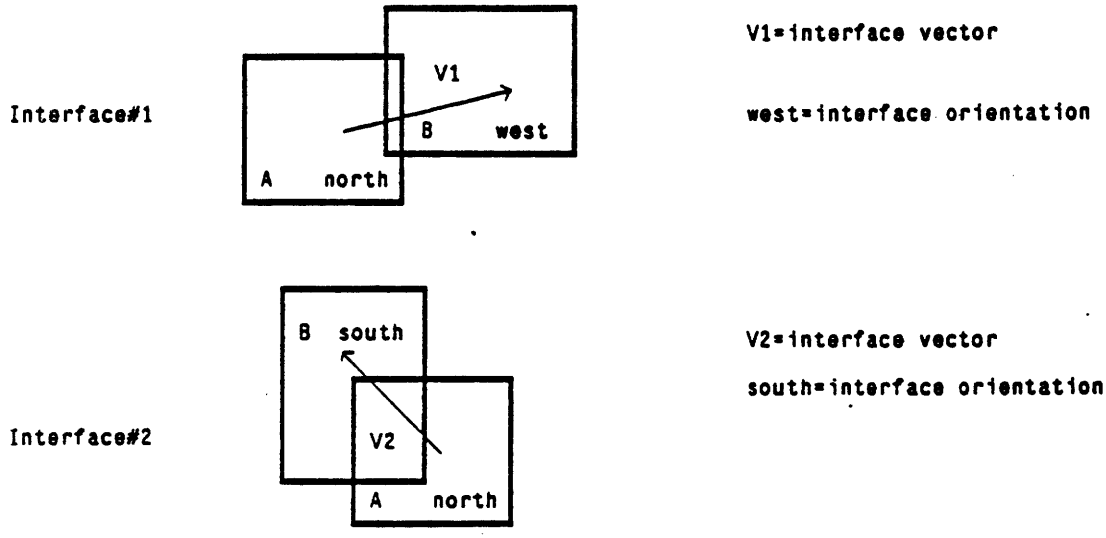


Figure 2.3: Different Interfaces between two cells.

placement of A one can determine the placement of B and vice versa. This bilaterality of the interface table is very important. We will see in section 3.4 that it may not be possible to determine in advance which of the two instances A or B has a known placement and which one will have its placement derived from the other.

2.5 Interface Inheritance Relations

In order for any cell to be used in the RSG it must have an *interface* with some other cell, otherwise there is no way to place it. When new cells are built up hierarchically by the system, in order to take full advantage of cell hierarchy, *interfaces* for new cells can be specified in terms of existing ones. In this way cells built up by the system can be used to build even larger cells

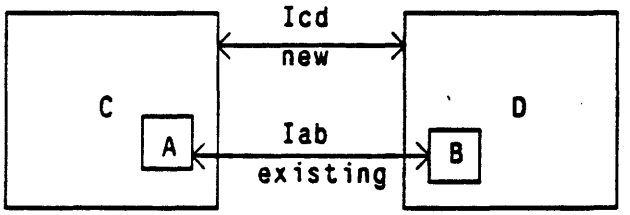


Figure 2.4: Interface Inheritance

in exactly the same fashion as were the primitive cells of the *sample layout.*

If A (respectively B) is a subcell of a new cell C (respectively D), it is then possible to define a new *interface* I_{cd} between C and D in terms of an existing *interface* I_{ab} between A and B. I_{cd} is the *interface* that C and D would *inherit* if the subcells A and B within C and D were placed and oriented with *interface* I_{ab} (see Figure 2.4). The RSG allows the user to define a new interface (and load it into the interface table) by specifying the two cells C and D, the instances of A and B in C and D, the *interface number* of the interface between A and B and an *interface number* for the newly defined interface between C and D.

The rest of this section is concerned with finding an algebraic expression for the interface vector and interface orientation of the new interface I_{cd} between C and D in terms of the existing interface I_{ab} between A and B and the calling parameters of the instances of A and B in C and D. Let[4] (L_a^{rc}, O_a^{rc}), (respectively (L_b^{rd}, O_b^{rd})) the calling information of A (respectively B) in C (respectively D) and (V_{ab}, O_{ab}) (respectively (V_{cd}, O_{cd})) be the *interface vector*

[4]The superscripts ^{rc} (respectively rd) mean that the locations and orientations are relative to the coordinate system of C (respectively D).

and *interface orientation* of I_{ab} (respectively I_{cd}). Also let L_a^r (respectively L_b^r, L_c^r, L_s^r) be the location of the origin of A (respectively B, C, D) in the implicit calling coordinate system (i.e. as they appear in Figure 2.4) and let O_a^r (respectively O_b^r, O_c^r, O_s^r) be the orientation of A (respectively B, C, D) in the implicit calling coordinate system (which can be for argument sake considered to be the absolute coordinate system) then:

$$O_a^r = O_c^r \circ O_a^{rc} \tag{2.7}$$

$$L_a^r = L_c^r + O_c^r L_a^{rc} \tag{2.8}$$

and

$$O_b^r = O_d^r \circ O_b^{rd} \tag{2.9}$$

$$L_b^r = L_d^r + O_d^r L_b^{rd} \tag{2.10}$$

Replacing 2.7 and 2.9 in 2.1 we get:

$$\begin{aligned}
O_{ab} &= (O_a^r)^{-1} \circ O_b^r \\
&= (O_c^r \circ O_a^{rc})^{-1} \circ O_d^r \circ O_b^{rd} \\
&= (O_a^{rc})^{-1} \circ (O_c^r)^{-1} \circ O_d^r \circ O_b^{rd} \\
O_{ab} \circ (O_b^{rd})^{-1} &= (O_a^{rc})^{-1} \circ (O_c^r)^{-1} \circ O_d^r \\
O_a^{rc} \circ O_{ab} \circ (O_b^{rd})^{-1} &= (O_c^r)^{-1} \circ O_d^r \\
&= O_{cd}
\end{aligned}$$

So

$$Ocd = O_a^{rc} \circ O_{ab} \circ (O_b^{rd})^{-1} \tag{2.11}$$

Replacing equations 2.8 and 2.10 in equation 2.2 we get:

$$
\begin{aligned}
V_{ab} &= (O_a^r)^{-1}(L_b^r - L_a^r) \\
&= (O_a^r)^{-1}(L_d^r + O_d^r L_b^{rb} - L_c^r - O_c^r L_a^{rc}) \\
L_d^r - L_c^r &= O_a^r V_{ab} - O_d^r L_b^{rb} + O_c^r L_a^{rc} \\
(O_c^r)^{-1}(L_d^r - L_c^r) &= ((O_c^r)^{-1} \circ O_a^r)V_{ab} - ((O_c^r)^{-1} \circ O_d^r)L_b^{rb} + (O_c^r)^{-1} \circ (O_c^r L_a^{rc})
\end{aligned}
$$

Using equations 2.2 and 2.1 with different subscripts, equation 2.7 and the previous result we get:

$$
\begin{aligned}
V_{cd} &= (O_c^r)^{-1}(L_d^r - L_c^d) \\
&= ((O_c^r)^{-1} \circ O_a^r)V_{ab} - ((O_c^r)^{-1} \circ O_d^r)L_b^{rb} + ((O_c^r)^{-1} \circ O_d^r)L_a^{rc} \qquad (2.12) \\
&= O_a^{rc} V_{ab} - (O_c^r)^{-1} L_b^{rb} + L_a^{rc}
\end{aligned}
$$

2.6 An efficient representation for orientations

Whereas *interface vectors* can be straightforwardly represented by a pair of real numbers, orientations require a slightly more complex data structure. The purpose of this section is to find an efficient representation for orientations in terms of memory, computation and ease of manipulation. Recall from Section 2.1 that calling an instance of B in A consists of performing an affine isometry to the objects in B and then adding the collection of objects in B to A. A layout editor needs to be able to perform affine isometries on the various cells. If A is called in a cell B which is in turn called in a higher order cell C then two affine isometries get applied to the objects in A. The first isometry I_1 corresponds to the calling parameters of A in B and the second isometry I_2 corresponds to the calling parameters of B in C. For an object Ob in A the corresponding component in C would be $I_2(I_1(Ob))$. I_1 is first performed on Ob and then I_2 is performed on the resulting object.

Another way to perform isometry composition is to first compose the two operators and then apply the resulting operator to the object. Since

$I_2(I_1(Ob)) = (I_2 \circ I_1)(Ob)$ it is possible to first compute $(I_2 \circ I_1)$ and then apply this new transformation to Ob. This method of first computing the resulting isometry and then applying it to the object can be computationally more efficient as the resulting isometry is computed only once and hence effort is not duplicated over the various objects on which this transformation is to be performed.

In layout editors the preferred way of composing operators could be $I_2(I_1(Ob))$ because this method is easier to implement [5]. If there is already a method for performing isometry on objects then, since the result of applying an isometry to an object is an object of the same type no extra mechanism is needed to successively perform several isometries on the object. In the case where only a finite set of legal isometries are implemented this method can lead to more efficient methods for applying single isometries to objects. For example one could index the set of available isometries over the integers. In that case, in order to apply a isometry known by its index number to a given object, one could use the index number to lookup a table of procedures (there is one procedure per isometry) to get the procedure that implements that particular isometry and then apply it to the object[6]. This method eliminates the interpretive overhead associated with the decoding of the isometry representation. For example isometries can be represented as matrices, and a program that can apply any matrix transform to an object would be slower than one that performs an unique fixed linear operation. However this indexed representation does not lend itself to symbolic composition. If the number of implemented indexes is n then (assuming that the set of imple-

[5]However HPEDIT uses the $I_2 \circ I_1$ method.

[6]HPEDIT uses this method.

mented isometries is closed under isometry composition rules) knowing the index of I_2 and the index of I_1 in order to compute the index of $(I_2 \circ I_1)$ a mapping table from $n * n$ to n integers is required. Another table from n to n integers is also required to invert the isometries (assuming the set is closed under inversion). Hence this method becomes cumbersome in the case where there is a large number of implemented isometries. It also requires a large number of procedures; one for each implemented isometry.

In the RSG at times it is necessary to obtain expressions for new transformations and therefore operations for symbolic composition and inversion of transformations are required. Recall equations 2.11 and 2.12 from Section 2.5. In order to compute the new inherited interface vector and interface orientation, we need to obtain expressions for the composition and inversion of orientations. It is therefore necessary to have a representation for orientations that allows them to be easily applied as operators and also allows them to be easily composed and inverted.

One possible way to implement all orientations is to use $2 * 2$ matrices of real numbers. $2 * 2$ matrices of real numbers can however represent all the different linear transformations in the vectorial plane out of which isometries (which are orientations) are only a very small subset. As a result they require storage and manipulation of much more information than is needed. Matrix composition and inversions are also relatively costly computationally.

There are more compact representations for orientations. We can represent all the vectorial rotations in the plane with a real number between $[0, 2\pi[$. The rotation can be expressed by the complex number e^{ij} where j belongs to $[0, 2\pi[$ and $i^2 = -1$. Orientations are either rotations about the origin or reflections about an axis passing through the origin. All the reflec-

tions about an axis passing though the origin can, however, be generated by composing the reflection about the y axis (or any other axis passing though the origin) with a rotation about the origin. If M is the interval $[0, 2\pi[$ and B is the set of Booleans, it is then possible to represent an orientation by the pair $(j, k) \in M * B$ where j represents the rotation, and k indicates whether or not a rotation about the y axis is to be performed before the rotation (the composition of rotations and reflections is not commutative). If $+$ (respectively $-$) is the induced addition (respectively subtraction) modulo 2π from M to M and if R is the rotation about the y axis. Than any orientation can be written as: $e^{ij} \circ R^k$ where $(j, k) \in M * B$ and $i^2 = -1$.

2.6.1 Inverting two orientations.

Let $O = e^{ij} \circ R^k$

and $O^{-1} = e^{ij'} \circ R^{k'}$

• If $k = 1$, then O is a reflection. Therefore $O \circ O = I$ where I is the Identity transform and hence

$$\begin{aligned} O^{-1} &= O \\ &= e^{ij} \circ M \\ &= e^{ij'} \circ M \end{aligned} \tag{2.13}$$

so $j = j'$ and $k = k'$

• If $k = 0$, then O is a rotation and hence

$$\begin{aligned} O^{-1} &= e^{ij'} \\ &= \frac{1}{e^{ij}} \\ &= e^{i(-j)} \end{aligned} \tag{2.14}$$

so $j' = -j$ and $k = k'$

Hence If $k = 1$ then $j = j'$, $k' = k$ otherwise $j = -j'$, $k' = k$

2.6.2 Composing two orientations

Let

$$\begin{aligned} O_1 &= e^{ij_1} \circ R^{k_1} \\ O_2 &= e^{ij_2} \circ R^{k_2} \\ O &= O_2 \circ O_1 \\ &= e^{ij} \circ R^{k} \end{aligned} \tag{2.15}$$

Then

$$\begin{aligned} O &= (e^{ij_2} \circ R^{k_2}) \circ (e^{ij_1} \circ R^{k_1}) \\ &= e^{ij_2} \circ (R^{k_2} \circ e^{ij_1}) \circ R^{k_1} \end{aligned} \tag{2.16}$$

because of the associativity of linear operators.

• If $k_2 = 1$ then

$R^{k_2} \circ e^{ij_1}$ is a reflection and hence $(R^{k_2} \circ e^{ij_1}) \circ (R^{k_2} \circ e^{ij_1}) = I$ therefore

$$\begin{aligned} R^{k_2} \circ e^{ij_1} &= (R^{k_2} \circ e^{ij_1})^{-1} \\ &= (e^{ij_1})^{-1} \circ (R^{k_2})^{-1} \\ &= e^{i(-j_1)} \circ R^{k_2} \\ &= (e^{i(-j_1)}) \circ (R^{k_2}) \end{aligned} \tag{2.17}$$

because R^{k_2} is a reflection (or identity) and e^{ij_1} is a rotation.

therefore

$$\begin{aligned} O &= e^{ij_2} \circ (R^{k_2} \circ e^{ij_1}) o R^{k_1} \\ &= e^{ij_2} \circ (e^{i(-j_1)} \circ (R^{k_2})) \circ R^{k_1} \\ &= (e^{ij_2} \circ e^{i(-j_1)}) \circ (R^{k_2} \circ R^{k_1}) \\ &= (e^{i(j_2 - j_1)}) \circ (R^{k_2 \oplus k_1}) \\ &= e^{i(j_2 - j_1)} \circ (R^{\overline{k_1}}) \end{aligned} \tag{2.18}$$

where $\oplus$ is the XOR operator.

hence $j = j_2 - j_1$ and $k = \overline{k_1}$

• If $k_2 = 0$ then

$$\begin{aligned} O &= e^{ij_2} \circ (R^{k_2} \circ e^{ij_1}) \circ R^{k_1} \\ &= e^{ij_2} \circ (e^{ij_1}) \circ R^{k_1} \\ &= (e^{ij_2} \circ e^{ij_1}) \circ R^{k_1} \\ &= e^{i(j_1+j_2)} \circ R^{k_1} \end{aligned} \tag{2.19}$$

hence $j = j_2 + j_1$ and $k = k_1$

So Hence If $k_2 = 1$ then $j = j_2 - j_1$, $k = \overline{k_1}$ otherwise $j = j_1 + j_2$, $k = k_1$

We have seen that we can represent an arbitrary orientation (isometry) by the pair $(j, k) \in M * B$ and using the associativity of linear operators we can compute any expression involving composition and inversion of orientations. It is computationally expensive however to apply an operator represented in this form to actual objects, because a *sin* an a *cos* must be computed. Due to numerical inaccuracies an object (say a box) with vertical and horizontal edges can be transformed by a quarter turn rotation into a object whose edges are not precisely aligned with the axis. Adding and subtracting elements of M can also lead to numerical inaccuracy as elements of M are represented in the computer by real numbers and a modulo 2π operation has to be performed on the result of every real addition (or subtraction) to ensure that the result is an element of M.

In the RSG the choice therefore was made not to support arbitrary rotations and reflections. Most VLSI circuit layouts are built using boxes of various layers where the boundaries of the boxes are vertical or horizontal lines i.e. parallel to one of the coordinate axis. Hence in most cases it is

sufficient to support all orientations that transform vertical and horizontal lines into vertical and horizontal lines.

The four multiples of the quarter turn rotation are the only rotations that have this property. The only reflections that can have this property are those that transform vertical edges into vertical edges and horizontal edges into horizontal edges which are the two reflections about the axis. And reflections that transform vertical edges into horizontal edges and vice versa which are the reflections about 45 degree lines passing through the origin. These 4 reflections can be generated by first reflecting about the y axis and then applying one of the four quarter turn rotations.[7]

Just as arbitrary orientations can be represented by an element of $M * B$, these eight basic orientations can be represented by z, an element of $\frac{Z}{4Z}$ ($\frac{Z}{4Z} = \{0, 1, 2, 3\}$), and a boolean k, hence by an element of $\frac{Z}{4Z} * B$. This would correspond to the orientation $e^{\frac{\pi}{2}iz} \circ R^k$ in the previous notation. Using the induced addition and subtraction on $\frac{Z}{4Z}$ the rules for composing and inverting orientations are the same as previously described using the $M * B$ representation. Orientations can now easily be applied to vectors and boxes since performing a reflection about the y axis corresponds to changing the x coordinate of an object to $-x$. The four quarter turn rotations require only permutations and negations of the two coordinates. For instance the one quarter turn rotation maps the x coordinate into the y coordinate and the y coordinate into the $-x$ coordinate. The Figure 2.5 shows the mapping of coordinates for each of the four basic rotations.

[7]these are the 8 orientations also supported by HPEDIT.

Orientation	x coordinate	y coordinate
North	x	y
South	$-x$	$-y$
East	y	$-x$
West	$-x$	y

Figure 2.5: Coordinate mapping for the 4 basic rotations

Chapter 3

The Algorithm

3.1 Algorithm Overview

The RSG algorithm (see Figure 3.1) consists of first reading in the *sample layout* in order to define the primitive cells and build up the initial *interface table.*

New cells are then created in a two step sub-algorithm. The first step in the sub-algorithm consists of building a *connectivity graph* for the new cell. The *connectivity graph* for the new cell is a graph whose vertices represent *partial instances* whose cell type is known but whose location and orientation are as yet unspecified.

The edges between vertices represent *interfaces* between instances and the weights assigned to them are the *interface index numbers.* The *connectivity graph* need only be a spanning tree since cycles in the graph contain redundant information. For a given *sample layout*, each *connectivity graph* gives rise to a unique layout (see Figure 3.2). Interfaces provide the local placement constraints between (two) cells. The connectivity graph provides information

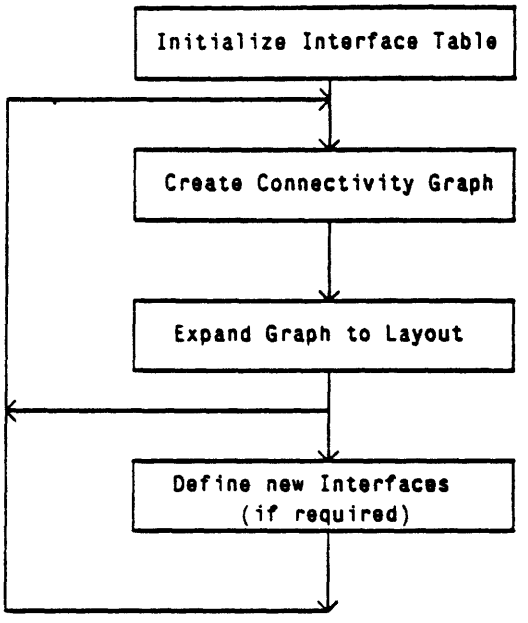


Figure 3.1: RSG algorithm

about the global placement of all the subcells in a macrocell. The graph sets up an implicit system of linear equations whose unknowns are the placements and orientations of the (pseudo) instances in the graph and where the given parameters are the interfaces between the various cells.

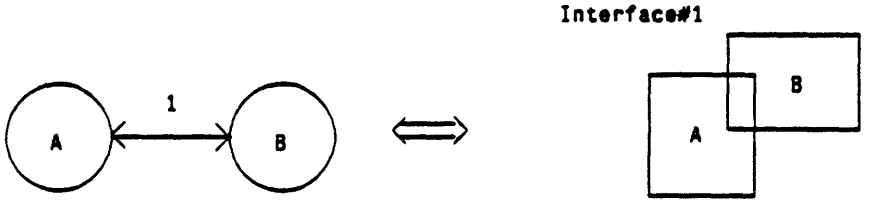


Figure 3.2: Graph and Layout Equivalents

The second step consists of converting the *connectivity graph* into a layout. This is done by first selecting a root node in the graph and arbitrarily placing and orienting the corresponding instance. The graph is then traversed, and each of the nodes in the graph (which initially are all partial instances) gets expanded into a *complete instance* with a location and an orientation. The location and orientation L_b and O_b of a partial instance B can be computed from the location and orientation L_a and O_a of one of its already traversed neighboring nodes A using the formula,

$$O_b = O_a \circ O_{ab} \tag{3.1}$$

$$L_b = O_a V_{ab} + L_a \tag{3.2}$$

where (V_{ab}, O_{ab}) is the *interface* between A and B. Finally once a new cell is created, if it is to be used in a larger cell, it is necessary to define new *interfaces* between it and the already existing cells.

Since the connectivity graph need only be a *spanning tree* many of the interfaces that occur in the final layout need not be present in the sample layout. Figure 3.3 shows a cluster of instances of A, B, C and D assembled together. The corresponding connectivity graph is also shown. The labels inside the nodes of the connectivity graph correspond to the nodes a well as the instances they are contained in. Since the connectivity graph need only be a spanning tree, it does not have to contain edges between A and D, A and C, or B and D. This is because with or without those edges the graph remains a *single connected component* (i.e. one can reach any node starting from any node by *walking* along edges in the graph). Since the three described edges are not present in the graph the I_{ad} (or I_{da}), I_{ac} (or I_{ca}), and I_{bd} (or I_{db}) are never accessed by the RSG, and therefore need not

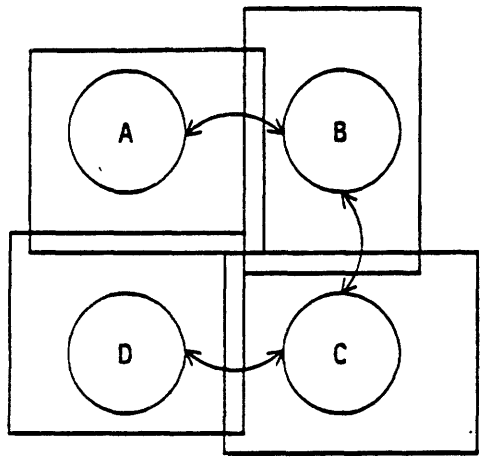


Figure 3.3: Graph Connectivity Requirements

be present in the sample layout. Hence the creation of both *design file* and *sample file* is simplified by requiring that the graph be only a *spanning tree.*

3.2 Advantages of the method

This (augmented) two step process of first determining connectivity and then using the connectivity information along with cell definition and cell *interface* information to build a layout, provides a clean separation between the graphical and procedural information. The procedural information in the *design file* is used to build the *connectivity graph* and remains constant over different implementations of the design as given by the *sample layout.* The graphical information from the *sample layout* is used to transform the *connectivity graph* into a physical layout of a particular implementation of the design. Cell spacing parameters which relate to the graphical information are never accessed or manipulated in the *design file.* This *delayed binding* on the

location and orientation of instances allows for clean *macro abstraction* in the *design file*. Since in the *design file*, partial instances are connected together without assigning actual locations and orientations to them, it is possible to build subgraphs without prior knowledge of where and with which orientation the instances in the subgraph will be used. It is easier and cleaner to write and compose macros for sub-graphs, because the state of a calling macro does not side-effect the called macro by imposing a starting location and a starting orientation at which to start assembling the subcells (i.e. the called macro returns the same subgraph regardless of how the calling macro will choose to connect the subgraph and regardless of the final calling parameters of the instances of the subgraph). Macro abstraction suppresses details of how and where a macro for generating a subgraph gets called and allows the designer to concentrate only on the connectivity of the subgraph.

3.3 Limitations

The two step process as described in the previous section provides a high level of separation between the graphical and procedural part of the layout process. Since geometrical parameters are not accessed in the *design file*, however, decisions based on the size and shape of the final layout such as placement and routing are difficult to make. For example the choice between the two routing configurations in Figure 3.4 requires knowledge of the sizes and shapes of the two cells A and B as well as the size of the routing channels.

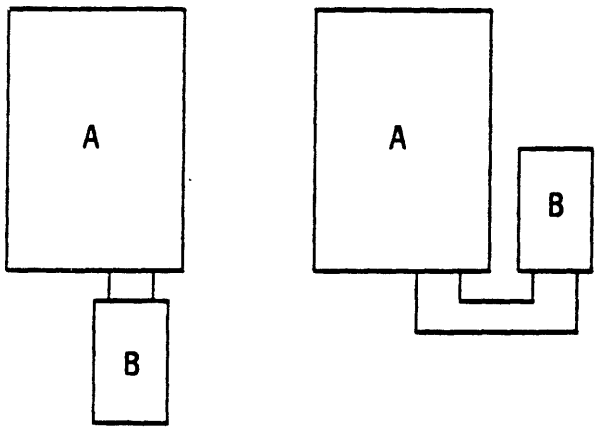


Figure 3.4: Different routing configurations

3.4 Connectivity Graphs in Greater Detail

The purpose of this section is to investigate some of the properties of connectivity graphs both in terms of data structures as well as in terms of their mathematical properties. The previous section described an equivalence between connectivity graphs and physical layouts. Actually (for a given sample layout) to each connectivity graph there corresponds a whole equivalence class of layouts. All the layouts in an equivalence class are such that any element in the class can be transformed into any other element in the class by an affine isometry i.e. all elements in an equivalence class are identical modulo an affine isometry. By selecting a root node in the graph and by placing and orienting the corresponding instance a particular element in the equivalence class is identified, namely the one where the instance corresponding to the root node has the chosen placement and orientation.

Connectivity graph data structures must have *bilateral* edges. If there is an edge between nodes A and B then in the data structure of A there must be a pointer to the data structure of B and in the data structure of B there must be a pointer to the data structure of A. This is because when a connectivity

graph is being created, the root node of the graph (which is arbitrarily chosen, placed and oriented) which is the starting point for traversing the graph (in order to convert the graph into a layout) may not be known. Macros for generating subgraphs of a layout have no knowledge of how the subgraphs they generate will be connected together by their calling macros in order to make larger graphs. For example if a macro M for creating graphs were to return the subgraph of Figure 3.2, either node B or node A could be a leaf node in the graph (i.e. a node with only one connection to it) depending on whether node A or node B was connected to the rest of the connectivity graph by the macro that called M. Hence even if the graph is a spanning tree the parent-son relationship between directly connected nodes in the graph is not known until the graph is traversed. This is why during the graph traversal one must be able to get to node B from node A and also get to node A from node B because we do not know which of the two nodes will be visited first.

The bidirectionality of the graph is essentially a data structure problem that is constrained only by the graph traversal requirements and not by the abstract mathematical properties of the graph. This requirement does not constrain whether or not the graph is *directed* or not. A graph $G = (N, E)$ where N is a nonempty set of nodes and E is the set of edges is said to be *directed* if the edges are *ordered* pairs (v, w) where $(v, w) \in N * N$. That is to say there is a privileged direction for the edges of the graph. A graph $G = (\{A, B\}, (A, B))$ (a graph with nodes A and B and an edge from A to B) can have a bilateral data structure which means that from node A we can go to node B and vice versa, and can at the same time be directed which means that the (A, B) edge has a privileged direction (i.e. the (B, A) edge may not exist).

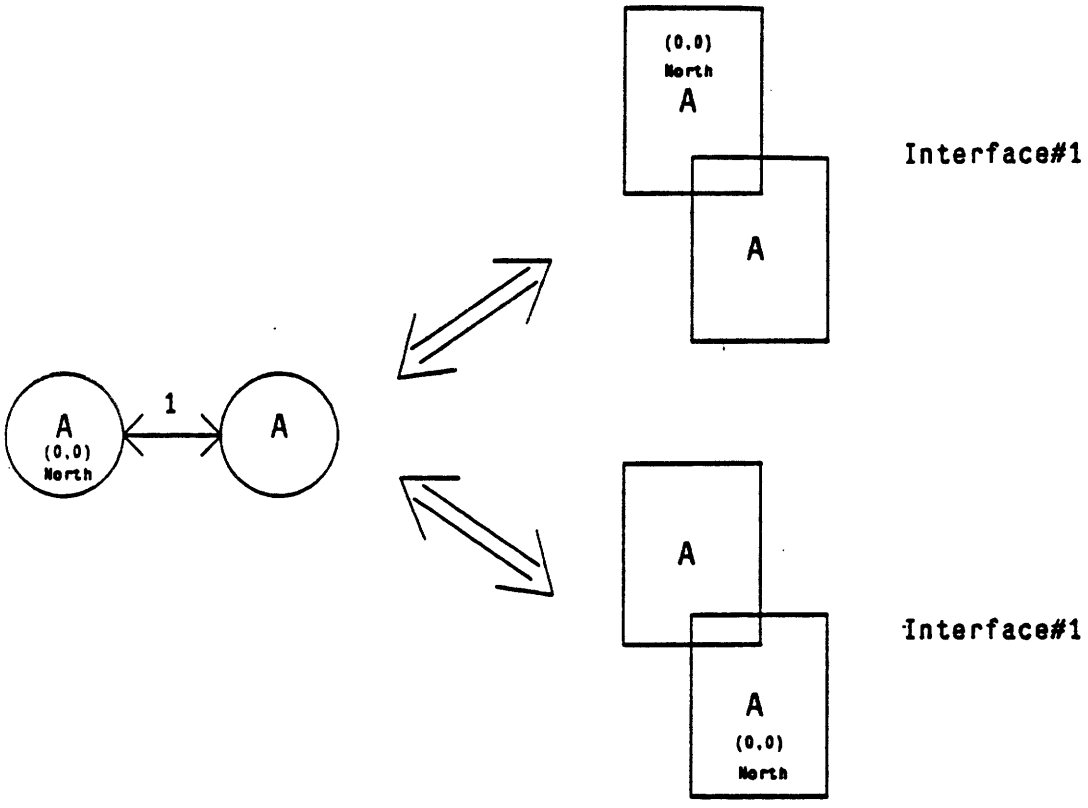


Figure 3.5: Interface ambiguity in undirected graphs.

We now need to decide whether or not connectivity graphs for the RSG should be directed graphs or non-directed graphs. What is needed is a graph that for a given sample layout uniquely defines an output layout (modulo an affine isometry). If the celltypes of nodes A and B are distinct then knowing the locations and orientations of node A it is always possible to determine the placement and orientation of node B because the right hand side of equations 2.1 and 2.2 are well defined. Hence at first it would seem that an undirected graph would suffice. However, in the case of Figure 3.5, if we know the location and orientation of the left node, there are two possibilities for the placement and orientation of the right node.

If $I_{ab} = (V_{ab}, O_{ab})$ is an interface between A and B then using equations 2.1 and 2.2

$$\begin{aligned} I_{ba} &= (V_{ba}, O_{ba}) \\ &= (I_{ab})^{-1} \\ &= (-(O_{ab})^{-1}V_{ab}, (O_{ab})^{-1}) \end{aligned} \tag{3.3}$$

is an interface between B and A.

Therefore if $I_{aa} = (V_{aa}, O_{aa})$ is an interface between A and A then

$$\begin{aligned} I'_{aa} &= (V'_{aa}, O'_{aa}) \\ &= (I_{aa})^{-1} \\ &= (-(O_{aa})^{-1}V_{aa}, (O_{aa})^{-1}) \end{aligned} \tag{3.4}$$

is also an interface between A and A. In equation 2.1 and 2.2 it is not clear whether V_{aa} and O_{aa} or V'_{aa} and O'_{aa} should appear on the right hand side of those equations. The problem here is not that of determining the right interface index (interface number) so as to choose the right interface from the interface table. The real problem is determining which instance the left node in Figure 3.5 refers to. Another problem which we will deal with later is that we do not know which of the two interfaces I_{aa} or I'_{aa} gets loaded into the interface table. The two interpretations of Figure 3.5 can lead to non equivalent layouts as shown in Figure 3.6. If the edges are undirected then there is no way to discriminate between these two cases. In the first versions of the RSG this problem caused the final layout to depend on how the graph was actually traversed. What is needed is a way of discriminating between the two nodes of Figure 3.5 which are directly connected together and have the same celltype. This can be done by giving privileged directions to the edges in the graph (making the graph a directed graph).

If we are able to characterize interfaces according to some criteria so as to discriminate between the two possible interfaces I_{aa} and I'_{aa} and select one

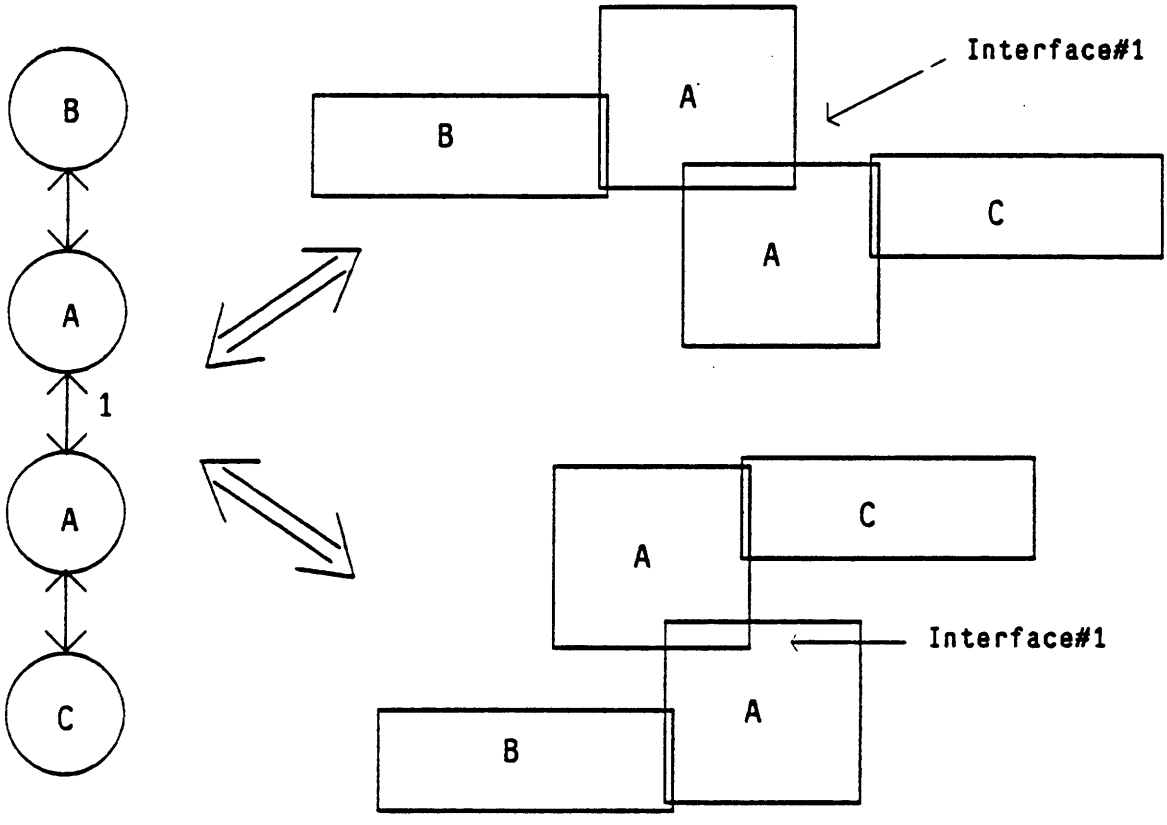


Figure 3.6: Layout ambiguity for undirected Graphs.

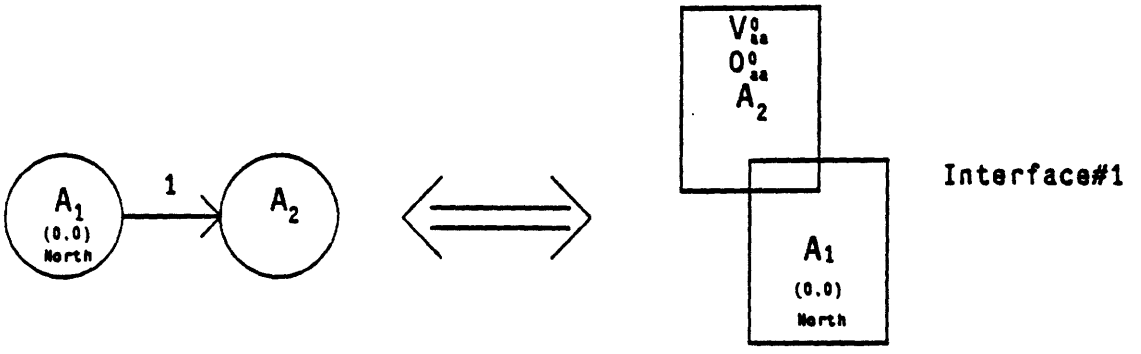


Figure 3.7: Resolving layout ambiguity with a directed graph.

of them (which I will refer to as I^0_{aa}) then with the convention that if there is a directed edge in Figure 3.7 from A_1 to A_2 (A_1 and A_2 have the same celltype: the indices are just to distinguish between the two of them) then it is A_1 that serves as the reference instance i.e. A_1 refers to the instance in the interface (see Figure 3.7) that is deskewed to orientation North and at whose point of call the interface vector begins. Knowing the placement and orientation of A_1 we can determine the placement and orientation of A_2 using equations 2.1 and 2.2 where the interface I^0_{aa} and knowing the placement and orientation of A_2 we can determine the placement and orientation of A_1 using the interface $(I^0_{aa})^{-1}$. The main problem has been to determine when to use (I^0_{aa}) and when to use $(I^0_{aa})^{-1}$ and this problem has been solved by making the edges of the graph directed.

The problem that now remains to be solved is that of selecting I^0_{aa} from I_{aa} and I'_{aa}. One possible way to perform the selection process is to mathematically characterize a property that is possessed by only one of the two interfaces I_{aa} or I'_{aa}. This property cannot depend on the interface vec-

tors alone because it is possible to have $I_{aa} \neq I'_{aa}$ with $V_{aa} = V'_{aa}$ making the selection between I_{aa} and I'_{aa} using V_{aa} and V'_{aa} impossible. Foe example if $I_{aa} = (0, East)$ then $I'_{aa} = (I_{aa})^{-1} = (0, West)$ hence $V_{aa} = V'_{aa}$ and $I_{aa} \neq I'_{aa}$. Similarly the property cannot depend on the interface orientation alone because it is possible to have $I_{aa} \neq I'_{aa}$ with $O_{aa} = O'_{aa}$. As an example Let $I_{aa} = (V_{aa}, North)$. Then $I'_{aa} = (-V_{aa}, North)$. Hence $O_{aa} = O'_{aa}$ and $I_{aa} \neq I'_{aa}$.

Since any reasonable mathematical criterion for selecting between I_{aa} and I'_{aa} depends on both the interface vector and the interface orientation, chances for finding a simple user understandable selection criteria are seriously jeopardized. The user does in fact need to know which of the two interfaces gets loaded into the interface table , because the effect of loading $(I^0_{aa})^{-1}$ in the table instead of I^0_{aa} is that of inverting the direction of all the edges (with the appropriate interface number) between nodes of celltype A.

The RSG solves this problem by allowing the user to specify (in the sample file) the right interface by graphically discriminating between the two instances of Figure 3.7 (which might occur in the sample file). If it is possible to graphically identify A_1 in the sample file then it is possible to force $I^0_{aa} = (V^0_{aa}, O^0_{aa})$ (see Figure 3.7) to be the interface that gets loaded into the interface table by forcing A_1 to be the reference instance at whose point of call the interface vector begins and whose orientation is deskewed to North.

We have seen that the connectivity graph data structure must have bilateral edges but that the graph itself must be directed. Only the edges between nodes of the same celltype need to be directed as direction information on edges between nodes of different celltype is not used.

Chapter 4

The Language

In order to make efficient use of the framework of the RSG we must be able to build large and complex *connectivity graphs* easily and efficiently. It is therefore imperative that the language for specifying *design files* supports good abstraction and powerful decision making. The *design file* interpreter has been embedded inside a Lisp interpreter so that the full power of a structured programming language is available to the designer. The interpreter provides a variant of the Lisp Programming Language (a subset of it) with a few special primitives for building and manipulating *connectivity graphs* as well as for converting *connectivity graphs* into layouts (a BNF grammar for the language can be found in Appendix A). Primitives for manipulating encoding tables (such as PLA truth tables) have also been added.

The design of the language was instrumental in defining the underlying mechanisms in the RSG. It allowed me to get a users perspective on what should be the right abstraction mechanisms even before I had an understanding of how these mechanisms could be implemented. Besides the fact that the language contains special features specific to the RSG, the language dif-

fers from standard LISP (for example MACLISP [27]) in two ways. First the Language does not support LIST structures. Instead it provides primitive facilities for arrays because arrays are more suited to array-like regular structures. Lists are not used (see Section 3.4) to implement connectivity graphs since these graphs are more than simple linked lists. The second difference is that procedures are not *first class* objects. I.e. it is not possible to pass a procedure as a parameter to another procedure. This decision was made to simplify the design of an efficient parser and interpreter.

4.1 Interfacing the parameter file to the design file

The parameter file to design file interfacing is done through variable scoping rules. The parameter file sets up parameters values in the global environment of the design file interpreter. Theses parameters can be accessed through variable scoping rules. A form of *lexical scoping* proves to be the simplest and most efficient way to do the scoping. A variable lookup during execution of the design file first causes that variable to be searched for in the environment of the procedure being executed. If the search fails a new search is then performed in the global environment of the interpreter. Should this search fail too it is assumed that the variable is a cell name and a search is performed on the table of available cells.

For example if the variable *corecell* in Figure 5.4(a) is meant to refer to a cell, since *corecell* is not assigned in the environment (it is not a formal or a local variable of the macro). The interpreter knows that it is either a variable

defined in the global environment or a cell name and initiates a search in the global environment and then in the cell table. This scoping methodology allows variables to be handled uniformly whether they are calling parameters of the macro, parameters set up in the parameter file, or just cells. Hence a powerful coupling between the parameter file and the design file is achieved by *immersing* the design file evaluation in a (global) environment set up by the parameter file.

Personalization of the variable names in the design file according to the cell names used in a sample file can also be achieved using the parameter file and scoping rules. A statement of the form *corecell* = *basiccell* in the parameter file would cause the variable named *corecell* in Figure 5.4 to now refer to the cell named *basicell* in the sample layout (or to be more general the cell named *basicell* in the current cell definition table which contains new cells as well as the primitive cells in the sample layout).

The sequence of steps taken by the interpreter to evaluate the variable *corecell* during execution of the design file is summarized in table 4.1. Dynamic scoping was considered and rejected because many of the variables in a macro refer to cell names defined in the cell table or variables defined in the global environment and often the whole current chain of environments would have to be searched needlessly.

4.2 Macros and Functions

In Lisp and other languages that support procedural abstraction a procedure can return a single object (or a pointer to it). Connectivity graphs used in the RSG have several nodes in them and what can be returned by a

Action Taken	Result
Lookup *corecell* in the environment of *mcell*	Failed
Lookup *corecell* in the global environment	A variable named *basiccell*
Lookup *basiccell* in the environment of *mcell*	Failed
Lookup *basiccell* in the global environment	Failed
Lookup *corecell* in the cell table.	⟨celldefinition of basiccell⟩.

Figure 4.1: Environment lookup.

procedure is a pointer to one of them. A pointer to a single node in a subgraph, however, may not be sufficient to efficiently manipulate the subgraph. In the process of building graphs from subgraphs a calling macro may need to identify several key nodes in the subgraph returned by the called macro in order to connect these key nodes to nodes in other subgraphs. Since all nodes *look alike* except for their celltype (a subgraph may even contain only one celltype) it is extremely difficult to determine the nodes of interest (the ones which are to be connected to other nodes) by performing a tree walk through the graph (starting from the node for which we have a pointer to). In the case where the calling macro was in fact sufficiently smart to identify the nodes of interest in a subgraph that macro probably contains a large part of the information needed to build the subgraph, defeating the spirit of macro abstraction and information hiding.

A mechanism is needed whereby a macro can return several objects at a time. To further enhance information hiding and at the same time enhance generality the calling macro should not know how many objects and how the

objects (in what order) are returned by a called macro. The calling macro should be able to pick from a menu of available objects the nodes of interest to it. The way this is achieved in the RSG is by making macros return the whole environment frame that was used during their execution. This method provides great flexibility since any variable bound during the execution of the called macro can be accessed using the *subcell* command. The *subcell* command provided by the interpreter allows the selection of a particular variable in a user-specified environment. If E is an environment (returned by a macro) and V is a variable bound in that environment then *(subcell E V)* returns the value to which V is bound in the environment E.

As an example, in Figure 5.4(b) the 4^{th} statement of macro `mall` assigns the variable `tregs` to the object returned by the macro call to `mtopregs`. Macro `mtopregs` is assumed to create a cell named `topregistername` and returns an environment in which one of the instances of `topregistername` (one for which it useful to get a handle on) is bound to the variable `ref`. Statement 5 of `mall` which defines a new interface between cells `topregistername` and `arrayname` requires the instance (of `topregistername`) bound to the variable `ref` in the environment `tregs`. The `(subcell tregs ref)` expression in statement 5 returns the appropriate instance.

The RSG has two classes of procedure types. The first type are *functions* which operate just as in LISP and return a single value which is the value of the last statement executed in the body of the function. Their syntax is almost identical to that of MACLISP (a variant of LISP).

The second class of procedure *macros*, are identical to functions in every respect except that they return their evaluation environment instead of the value of the last statement executed. Their syntax is the same as for func-

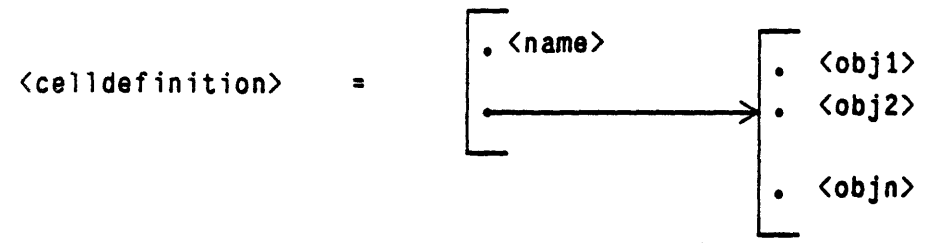


Figure 4.2: Celldefinition Data Structure.

tions except that the LISP function header *defun* is replaced by *macro*. The interpreter also requires to know ahead of time whether a statement of the form (⟨ function or macro name⟩ ⟨arg1⟩ ..⟨argn⟩) is a function call or a macro call and hence the interpreter requires that the macro name begin with an *m*.

4.3 Data Structures

This sections describes in detail the data structures used in the RSG by *spelling out* each of them. Its purpose is to give the reader a concrete feel for implementation issues of the abstract data types described in the previous chapters and serves as an introduction to the next section. Three data structures; the *cell definition*, the *instance* and the *node* will be examined.

Figure 4.2 shows the *cell definition data structure* which consists of a name (the name of the cell) and list of objects in the cell.

Figure 4.3 shows how the *instance* data structure builds on the *cell definition* data structure by adding calling parameters (a location and an orientation) to it.

Figure 4.4 shows how the *node* data structure is in turn built from the instance and a list of edges to other nodes. The location and orientation

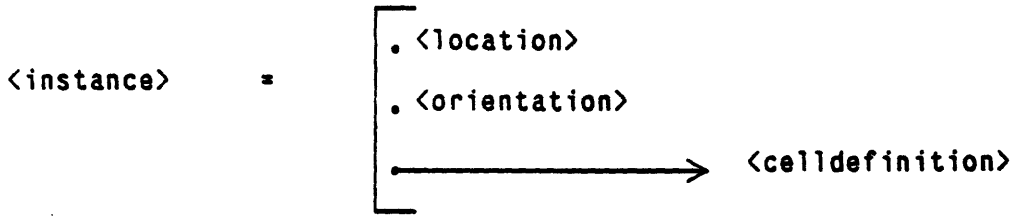


Figure 4.3: Instance Data Structure.

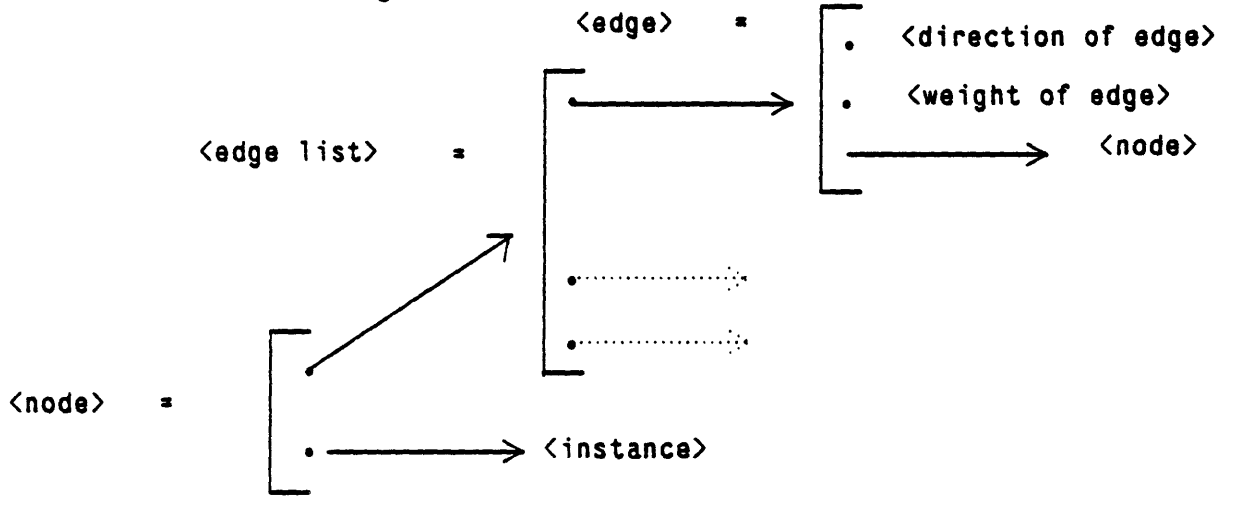


Figure 4.4: Node Data Structure.

fields of the corresponding instance data structure may or may not be blank depending on whether or not the graph (which contains the node) has been traversed. Each edge in the edge list of the node has a bit to indicate whether the edge is emanating or terminating at the current node, an integer for the weight of the edge, and a pointer to the other node attached to the edge[1].

[1]Recall from Section 3.4 that the graph must be directed and that the data structure must be bilateral.

4.4 Primitive operators for connectivity graphs

This section describes *mk_instance*, *connect* and *mk_cell* the three primitive operators provided in the RSG for building and manipulating connectivity graphs. Mutation of the data structures described in the previous section under these operators is also shown.

4.4.1 mk_instance operator

The basic *create* operator for creating connectivity graphs is the *mk_instance* operator. The purpose of this operator is to create a *pseudo instance* connectivity graph node (the node data structure of the previous section). Figure 4.5 shows in large font (the top line) a call to the *mk_instance* operator as it would appear in the design file. The data structures before the operator is executed appear in unbroken line and in normal font. The data structures created or modified after the operator is executed appear in broken line and in italics. The edge list of the created node is the empty set and the fields for the calling parameters of the corresponding instance are blank. $\langle$*return value*$\rangle$ is the value for the calling expression (the top line in Figure 4.5) that is returned by the design file interpreter.

4.4.2 connect operator

The primitive operator for connecting two nodes together by an edge is the *connect operator*. Figure 4.6 shows the effect of the *connect* statement with the same conventions as in Figure 4.5. Notice that the edge of the node corresponding to $\langle arg1 \rangle$ (pointing to $\langle arg2 \rangle$) has a 1 as its direction bit which means that the edge emanates from $\langle arg1 \rangle$. Similarly the corresponding edge

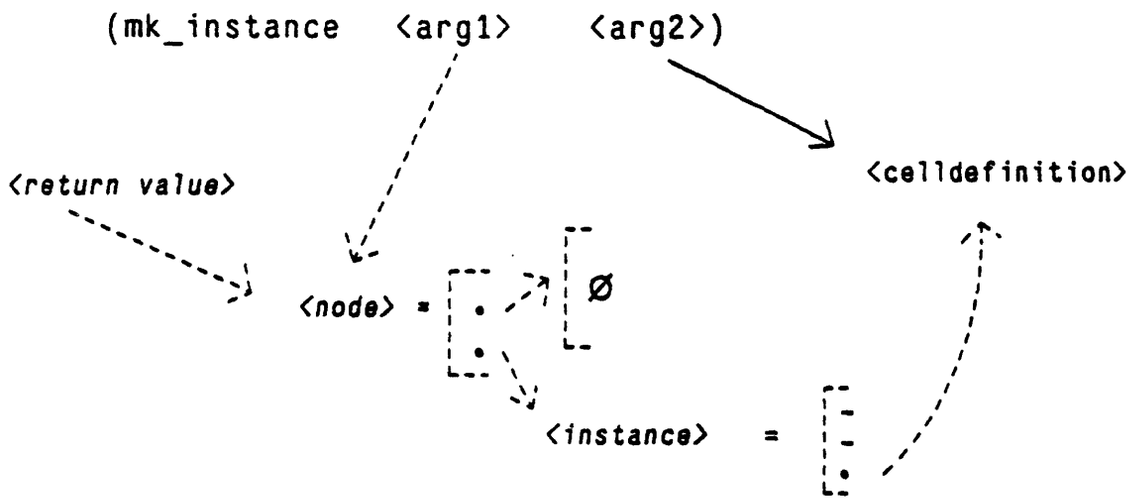


Figure 4.5: mk_instance operator.

in $\langle arg2 \rangle$ has 0 as its direction bit which means that the edge terminates at $\langle arg2 \rangle$.

4.4.3 mk_cell operator

The primitive operator for traversing and transforming a connectivity graph into a layout is *mk_cell*. Figure 4.7 shows the effect of calling the *mk_cell* operator in a design file. For simplicity sake nodes have been represented by circles instead of expanding their internal data structures. Each of the nodes has a pointer to the instance to which they correspond to. The calling parameters of the instances are initially blank and are filled in as the graph is traversed. The root of the graph is the node $< arg2 >$ and its instance is called at $((0,0), North)$. As each new node is visited and its instance's calling parameters are filled in, a pointer to the completed instance is pushed on the list of objects of the new cell being built. When the graph traversal is complete the object list of the cell definition of the new

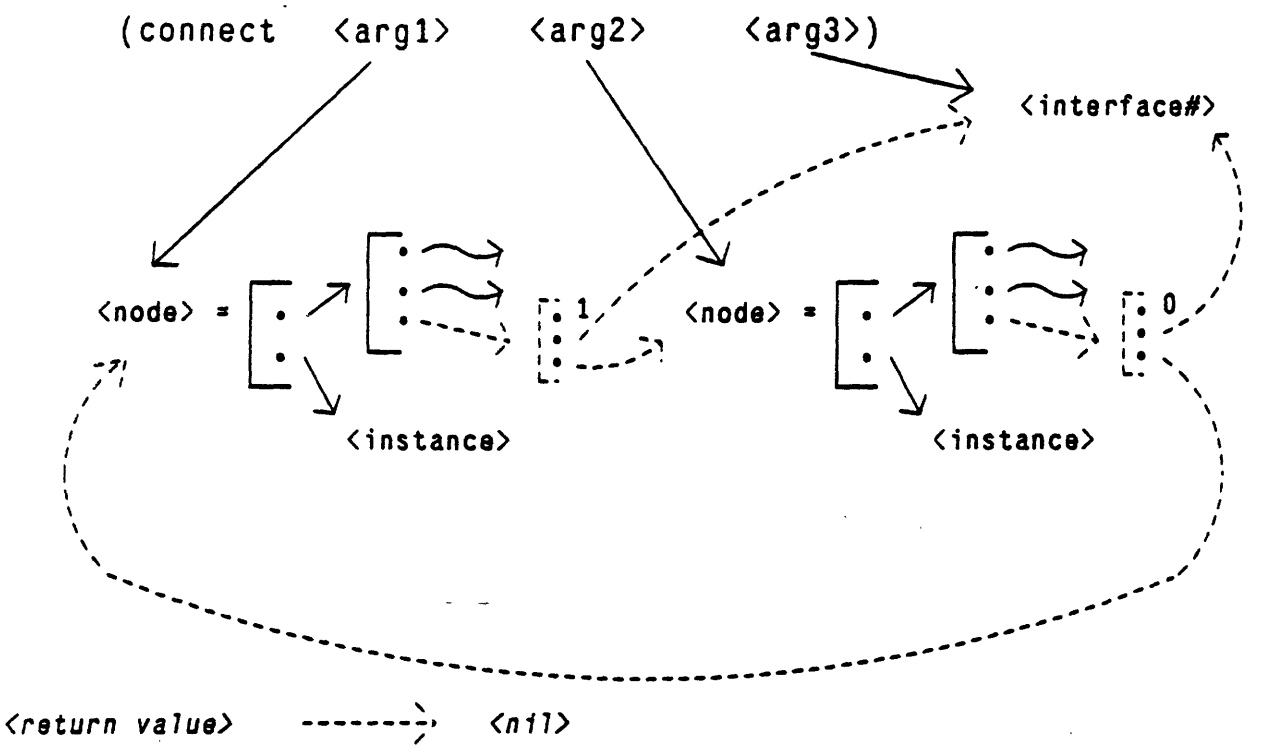


Figure 4.6: **connect** operator.

cell contains a pointer to all the instances. Not shown in the figure is the update of the *cell definition* table which after execution contains the binding $(\langle new\ \ cell\ \ name\rangle, \langle new\ \ cell\ \ definition\rangle)$.

4.5 Implementation

Implementation of the RSG was rather straightforward. Roughly two thirds of the code was overhead. Building and maintaining the layout database represents a sizable portion of the code. The single largest part of the code however is the design file interpreter which parses the design file (and parameter file) and then executes the commands in it. Writing a reasonable design file parser and interpreter was also the most time consuming task as

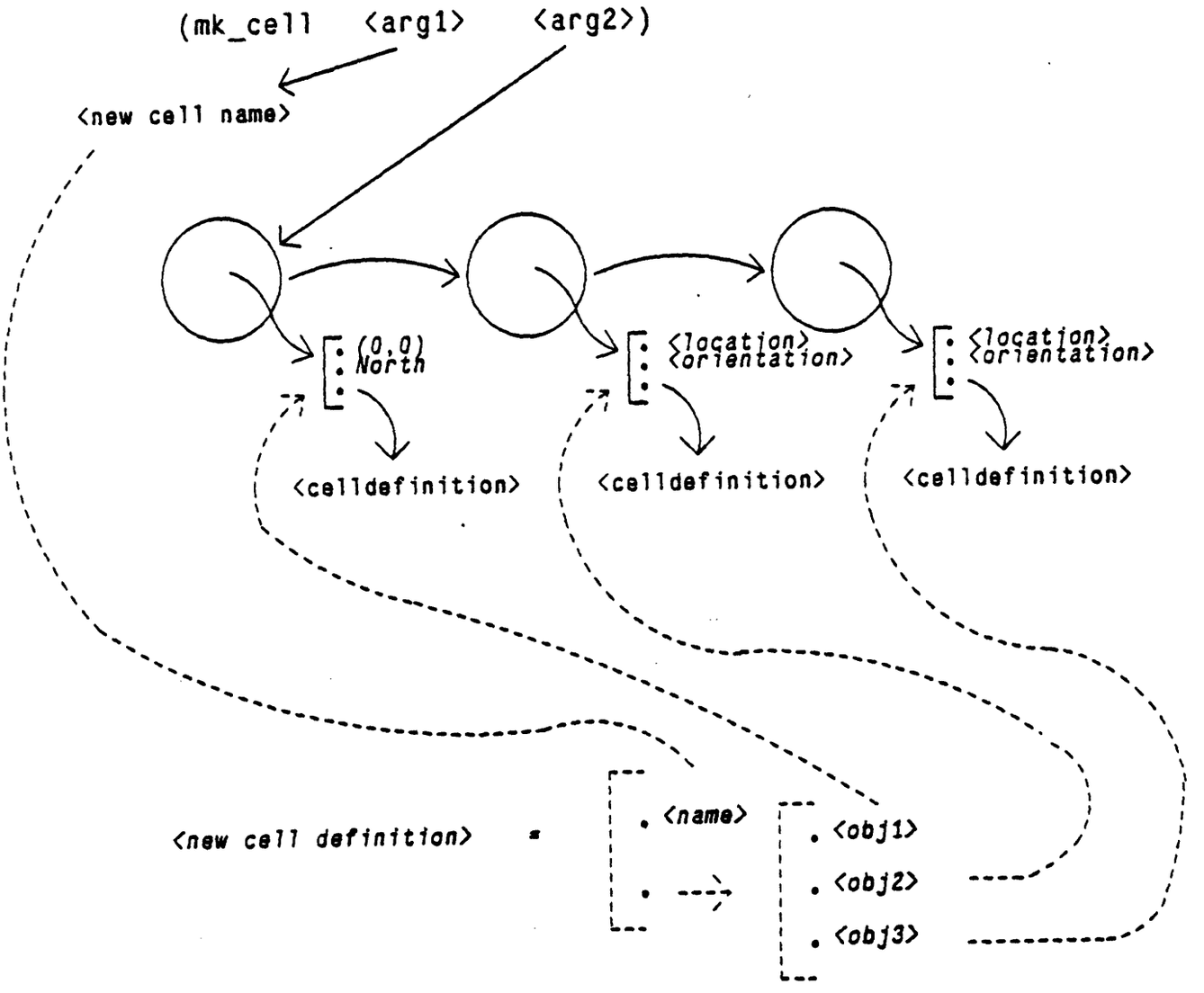


Figure 4.7: mk_cell operator.

the language supports full recursion, reasonable error handling and high execution speed. Embedding the RSG in a VLSI database type system such as Magic [26] or Schema [32] would have drastically reduced this overhead. Furthermore the availability of a suitable parser and interpreter which could support macros and functions (as they are described in Section 4.2) would have reduced the code by perhaps one half. In order to embed the RSG in a VLSI database type scheme, such as the two systems described above, facilities must be provided to create the design file language by performing minor alterations to a standard programming language such as LISP from where the whole layout database could be accessible.

The RSG program is written in CLU [21] and consists of approximately 6000 lines of source code. The program is highly modularized and consists of roughly a dozen major parts (CLU clusters), one for each major data type. The code trades memory for greater execution speed. The interpreter makes extensive use of CLU variants[2] and hence reduces the design file instruction decode overhead. The *interface table*, the *cell definition* table and even the interpreter *environment frames* are all implemented with `hash tables` [1] which makes lookup extremely fast. While walking though a connectivity graph the system accesses the interface table once for each node hence it is imperative that interface lookup be fast. While building large array structures the graph may be built by a tight loop in one of the design file macros. At each loop all the variables have to be resolved by the interpreter. Also due to the scoping rules described in Section 4.1 several environments (and the cell definition table) may have to be looked up to resolve a variable binding

[2]A variant is an object which has a special tag. Program flow can be dispatched according to this tag.

(especially since variables often refer to cells like in Table 4.1). It is therefore imperative that variable lookup also be extremely fast. Hash tables have the unfortunate property of consuming a lot of memory (memory concerns will become clearer in the next paragraph) and becoming inefficient as the number of bindings grows beyond their individual capacity which is fixed at the time they are created. Care must be taken while creating these tables to make them large enough to handle the required number of bindings but not too large in order not to waste too much memory.

The design file interpreter which uses hash tables to implement environments pays particular attention to this by first computing the number of formal and local parameters in a called procedure and then accordingly allocating a hash table of the right size for the environment. Unlike a classical LISP interpreter which disposes of the environment frame when a procedure is exited, environments in design files may have a much greater lifetime. Macros return their calling environment. This environment may in turn be held on to by the calling macro in its own environment. This environment may in turn be retained by an even higher order macro. It is possible to write a design file which holds on to too many environments (several thousand) at a time and exhausts the memory of a DEC-20. On the VAX this problem shows up in the form of a substantial decrease in speed due to excessive page faults. However it is almost always possible to decrease the memory requirements (by orders of magnitude) to within manageable limits by writing the design file in such a way so as not to hold on to many unneeded environments.

The RSG maintains it's own database and as such it is layout file format independent. The RSG can be made to accept any file format by providing an appropriate parser for the file format (this procedure requires that the

code be recompiled). The user can in the parameter file select the layout file format from a list of available file formats. Two layout file formats (CIF [25] and DEF [2]) are supported. Plans for supporting HPDRAW [3] files are also under way. Primitive functions can easily be added to the design file interpreter provided they fulfill some input output requirements.

The execution time is divided into roughly three equal parts: reading in the source file and building up the initial *interface table*, parsing and executing the *design* and *parameter file*, and writing the output file. A 32 x 32 Baugh-Wooley multiplier as discussed in Chapter5 is generated in 5 seconds on a DEC-2060.

The basic RSG mechanisms can be easily implemented in any language that supports good primitives for manipulating *pointers* and *heaps* (Pascal, C and Lisp would be suitable candidates). Memory management for the design file interpreter (a variant of Lisp) which supports heap storage and garbage collection is automatically handled by the underlying CLU[3] runtime system. Implementing the interpreter in a language which does not support automatic garbage collection might require restricting the power of the design file interpreter or implementing some form of automatic garbage collection. Lexically scoped Lisp with some primitive mechanisms for manipulating arrays would be very suitable as many of the primitive operators provided by the design file interpreter are also Lisp primitives. The Lisp *closure* mechanism could perhaps be used to implement the macro[4] mechanism in the RSG.

[3]CLU supports heap storage and garbage collection.

[4]Recall from Section 4.2 that macros return their environment.

Chapter 5

Example: Pipelined Array Multipliers

A pipelined array multiplier provides a good illustration of the RSG's ability to generate layouts for the kind of nontrivial regular structures that typically arise in practice. Figure 5.1 shows a purely combinational 6x6 signed two's complement multiplier based on the Baugh-Wooley algorithm [13]. The multiplier consists of an array of two types of carry-save adders that reduce the product to the sum of two words, which are then added in a final row of cells connected as a carry-propagate adder. (The two diagonal connections have been condensed to one for clarity). Each cell type contains an AND gate and a full adder: cell type I adds the bit-product $a_i b_i$ to its sum and carry inputs; and cell type II adds $\overline{a_i b_i}$ to its sum and carry inputs. The carry-propagate adder consists of type I cells which are drawn as polygons to distinguish them from the carry-save cells.

Using retiming transformations [18], the multiplier can be pipelined to any degree in a manner that preserves the regularity of the inner array, but

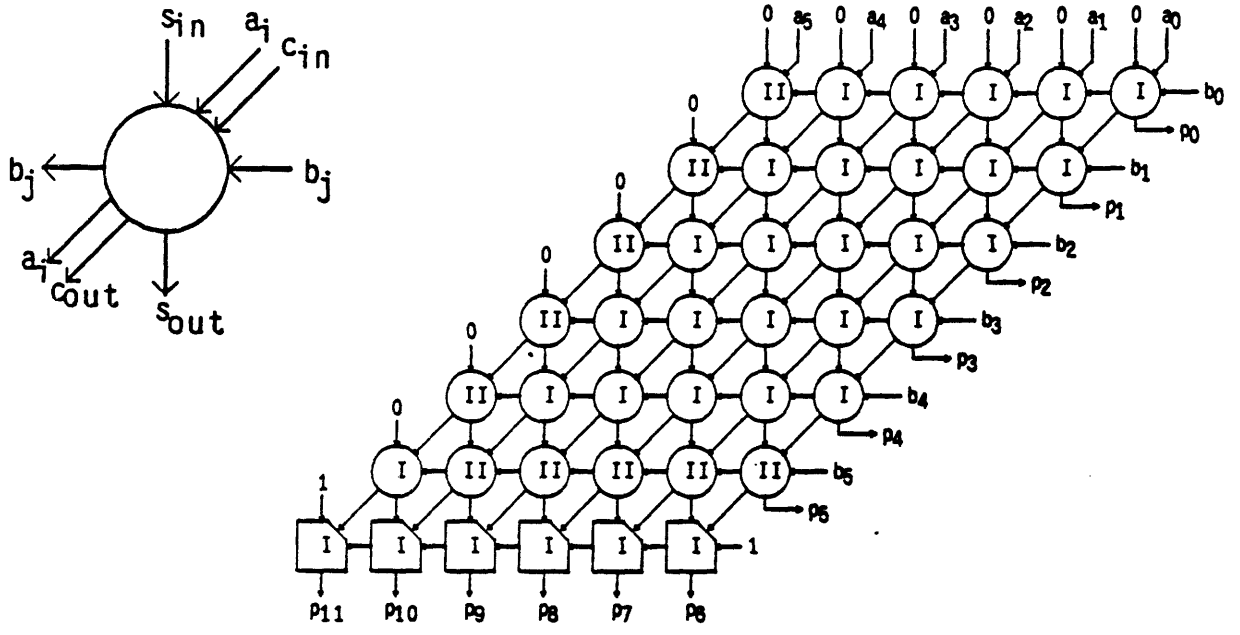


Figure 5.1: Combinational Baugh-Wooley Multiplier

adds irregularity to the periphery of the array in the form of input and output register stacks. Figure 5.2 illustrates two pipelined versions of the multiplier. (An integer near a dot represents the number of registers on the corresponding connection). The first version (2a) is a bit-systolic multiplier that has at most one full adder combinational delay between any two registers, and represents the highest possible degree of pipelining given the choice of the full adder as the largest indivisible cell. The second version (2b) implements a lower degree of pipelining, allowing at most two combinational delays between any pair of registers. From a circuit perspective, the optimal degree of pipelining is application and technology dependent, so it is necessary to be able to automatically generate any degree of pipelining.

A pipelined multiplier of given size and level of pipelining can be constructed by personalizing an array of basic cells which has been sized accord-

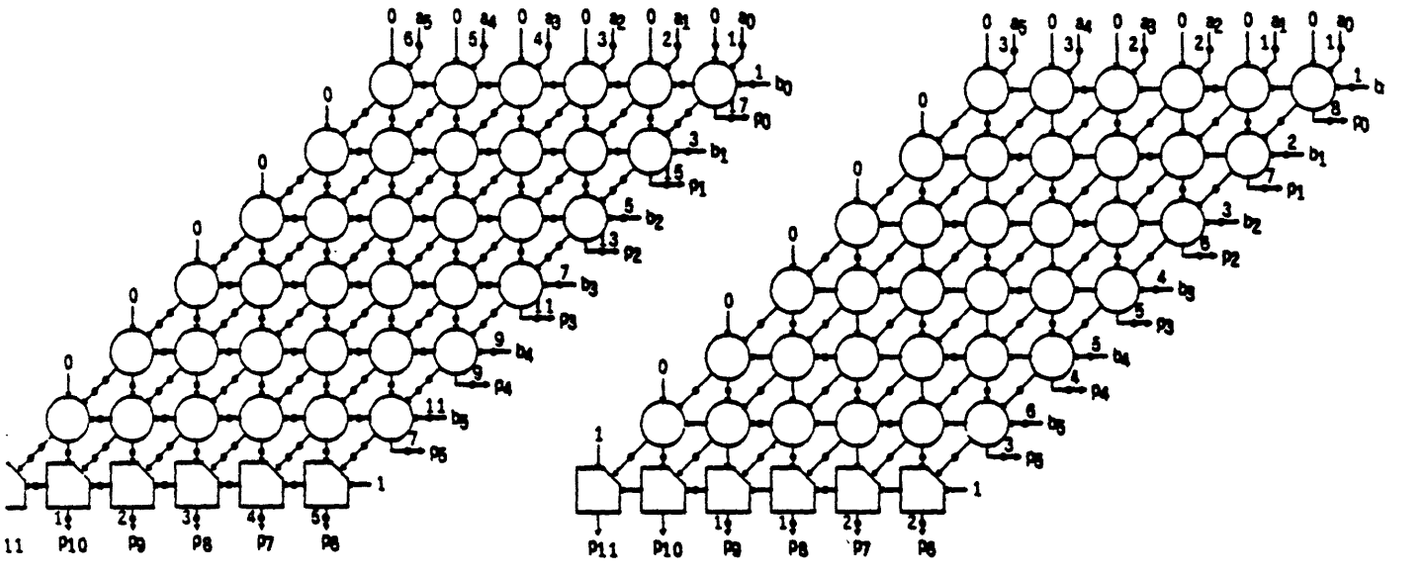


Figure 5.2: (a) Bit-Systolic Multiplier; (b) Pipelined Multiplier

ing to the number of bits in the multiplier and multiplicand. Each cell in the array must be personalized with respect to each of the following options depicted in Figures 5.1 and 5.2:

1. Cell type: Each cell must be programmed as either type I or type II to correctly implement the signed two's complement algorithm. Type II cells occur on the left and bottom edges of the carry-save array, except for the cell at the lower left corner. All remaining locations require cell type I.

2. Cell interface: To obtain nearly identical circuit topologies, cell types I and II use different active input levels. Furthermore, active output levels are affected by the amount of pipelining. Therefore, each cell interface is determined by the type of cells being connected and the number of registers on the connection.

3. Register assignment: The placement of registers on connections between cells depends on the degree of pipelining and the locations of the cells being connected.

4. Clock assignment: Pipelined systems generally require several clocks which must be assigned to registers according to their location in the array. Clock assignment is further complicated by the need to employ such circuit techniques as precharging to reduce area and power requirements.

In addition to the internal array configuration, there are "edge effects" to consider as well:

1. Peripheral registers: In order to properly skew the inputs and deskew the outputs, registers must be placed along the periphery as determined by the retiming transformations.

2. Input assignment: Ones and zeros must be assigned to the unused inputs along the top and left edges as prescribed by the Baugh-Wooley algorithm.

Cell masking is used extensively to convert an array personalization to actual layout. A basic cell is created which contains the layout features common to all cell personalities and which can accommodate the variations in layout necessary to implement all design options. Mask cells are instantiated on the basic cell to activate particular options by adding objects to the various layers. Figure 5.3 illustrates this with a basic cell designed to specifically optimize the electrical performance of the bit-systolic multiplier of Figure 5.2a. This cell contains input inverters, full adder circuitry, and six output

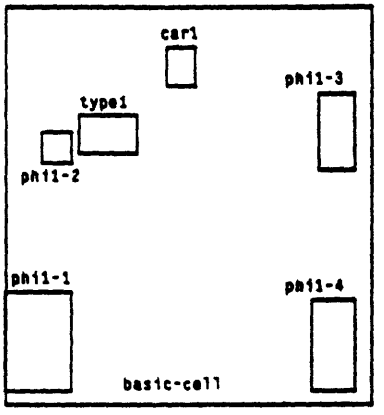


Figure 5.3: Multiplier Cell Maskings

registers. In this example, the `basic-cell` is programmed to type I by the mask-cell `typeI`, its carry input inverter is programmed by mask-cell `carl` to interface with a type II cell, and it is assigned the clock $\phi 1$ by mask-cells `phi1-1`, `phi1-2`, `phi1-3`, and `phi1-4`. The inner array of the multiplier is built up one cell at a time by first personalizing a copy of `basic-cell`, and then adding it to the array. Then the multiplier is completed by adding registers to the periphery of the array.

Figure 5.4 shows two sections of the *design* file written to generate a bit-systolic multiplier for any m-by-n case, and demonstrates the use of *macro abstraction*, *delayed binding*, and *interface inheritance*. The `mcell` macro of Figure 5.4a executes the personalization of `basic-cell` as a function of array size and cell index, and is used to hierarchically build the *macrocell* `innerarray` (the inner array of the multiplier). *Delayed binding* on the absolute location of each personalized cell greatly simplifies the definition and use of `mcell` in the creation of larger *macrocells* like `innerarray`. The code in Figure 5.4b constructs the complete multiplier from `innerarray` and three boundary *macrocells*, `tregs`, `rregs`, and `bregs`, which are constructed from

a single register cell. The three boundary cells are connected to `innerarray` using interfaces that are *inherited* from an interface between the basic cell and register cell. This example is cited to emphasize that *macrocells* can be manipulated with absolutely no need to enter the graphics domain and manually define *interfaces* or add spacing cells, as required by layout generators with restricted powers of abstraction.

The input *layout file* in Figure 5.5 demonstrates the ease and generality with which cell interfaces are specified in the RSG. One merely provides an example of the interface, and places a numerical label in the overlapping region, as for example, interface number 1 (the only interface) between `basic-cell` and `typeI`. The RSG then creates an *interface vector* and *orientation* from this graphical specification, and uses it to implement all instances of this interface that occur in the final circuit layout. The *layout file* provides a natural means for the user specification of cell layouts and interfaces and greatly reduces the amount of redundant information needed to characterize regular circuit layouts. This can be appreciated by comparing Figure 5.5 with the 6x6 systolic multiplier layout shown in Figure 5.6. This layout also illustrates the amount of complexity that exists in practical regular structures, even though this design has been simplified by omitting the register masking option. Register placement can be easily achieved by requiring that the user provide a register configuration table in the parameter file. Ultimately a subprogram to perform the retiming can be embedded in the multiplier design file. The program would take as input the parameter β which specifies the degree of pipelining and produce as output a register configuration table consistent with the multiplier size.

The optimum β for circuit performance within this class of pipelined mul-

```
(macro mcell (xsize ysize xloc yloc)
       (locals c temp)
       (mk_instance c basiccell)
       (cond ((= (+ ysize 1) yloc) (connect c (mk_instance temp type1) t1inum))
             ((= xsize xloc) (cond ((= ysize yloc) (connect c (mk_instance temp type1) t1inum))
                                   (true (connect c (mk_instance temp type2) t2inum))))
             (true (cond ((= ysize yloc) (connect c (mk_instance temp type2) t2inum))
                         (true  (connect c (mk_instance temp type1) t1inum)))))
       (cond ((= (mod xloc 2) 0)
              (prog (connect c (mk_instance temp phi1_1) clk1inum)
                    (connect c (mk_instance temp phi1_2) clk1inum)
                    (connect c (mk_instance temp phi1_3) clk1inum)
                    (connect c (mk_instance temp phi1_4) clk1inum)))
             (true
              (prog (connect c (mk_instance temp phi2_1) clk2inum)
                    (connect c (mk_instance temp phi2_2) clk2inum)
                    (connect c (mk_instance temp phi2_3) clk2inum)
                    (connect c (mk_instance temp phi2_4) clk2inum))))
       (cond ((= yloc ysize) (connect c (mk_instance temp car2) car2inum))
             ((= yloc (+ ysize 1))
              (cond ((= xloc xsize) (connect c (mk_instance temp car1) car1inum))
                    (true  (connect c (mk_instance temp car2) car2inum)))
              (true (connect c (mk_instance temp car1) car1inum)))))
```

(a) Cell personalization

```
(macro mall (xsize ysize)
       (locals innerarray tregs bregs rregs tri arrayi bri rri)
       (setq rregs (mrightregs ysize))
       (setq bregs (mbottomregs xsize))
       (setq innerarray  (marray xsize ysize))
       (setq tregs  (mtopregs xsize))
       (declare_interface topregistername arrayname 1
                          (subcell tregs ref) (subcell innerarray topright)
                          cell_to_topreginum)
       (connect (mk_instance tri topregistername) (mk_instance arrayi arrayname) 1)
       (declare_interface  arrayname bottomregistername 1
                           (subcell innerarray bottomright) (subcell bregs ref)
                           cell_to_bottomreginum)
       (connect (mk_instance bri bottomregistername) arrayi 1)
       (declare_interface  arrayname rightregistername 1
                           (subcell innerarray topright)  (subcell rregs ref)
                           cell_to_rightreginum)
       (connect (mk_instance rri rightregistername) arrayi 1)
       (mk_cell "the_whole_thing" arrayi))
```

(b) Multiplier Construction

Figure 5.4: Design File for a Systolic Multiplier

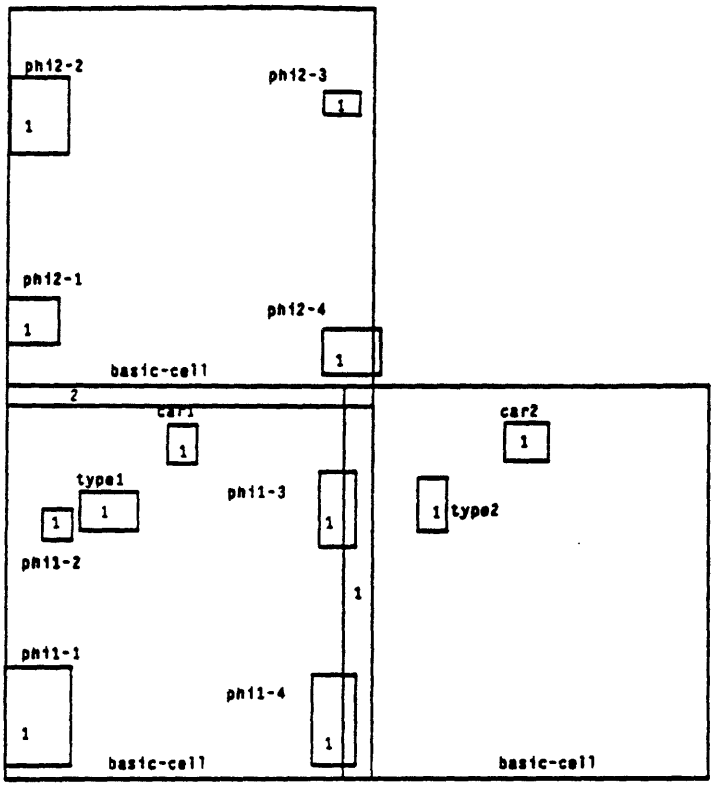


Figure 5.5: Layout File for a Systolic Multiplier

tipliers must be determined empirically through repeated iterations of multiplier layout generation, circuit extraction, and electrical simulation. The structure of these pipelined multipliers facilitates such an empirical investigation by admitting very regular layouts that can be generated quickly and interactively by the RSG. A study of the circuit issues determining pipelined array multiplier performance[12] is now underway using the RSG for layout generation, EXCL [23] for circuit extraction, and SPICE [30] for circuit simulation. Preliminary simulations suggest that clock drive, clock skew, and I/O pad drive — all of which vary with the level of pipelining and multiplier size — will be the primary limitations to throughput. For large multiplier sizes, macromodeling of critical paths can be used to alleviate the computational requirements of SPICE.

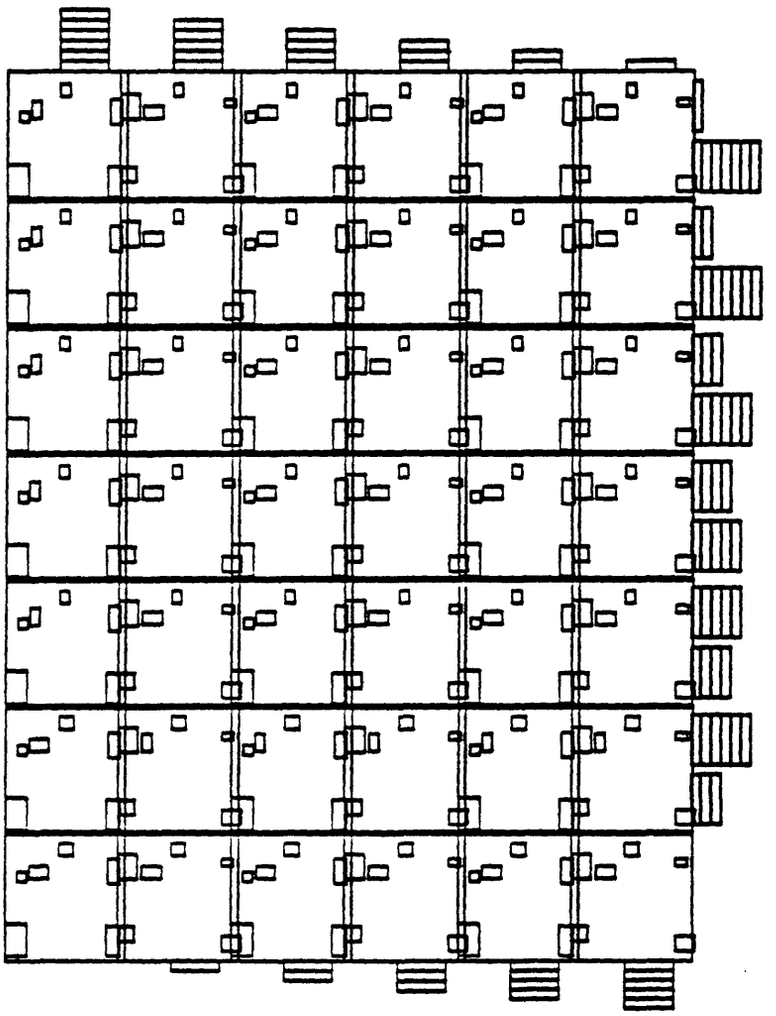


Figure 5.6: Bit-Systolic Multiplier Layout

Chapter 6

Compaction

6.1 Motivation

Despite the fact that the RSG is technology, implementation and architecture independent, the RSG by itself is not technology transportable (The RSG cannot be made to produce designs in a new technology simply by providing a new *design rule* file). A library of cells for the RSG designed in an older technology can quickly become obsolete as new process technologies with smaller geometries become available. Another problem with the RSG is that highly electrically optimized layouts require fine tuned optimization of the bus and device sizes. These optimizations depend on the particular configuration (size) of the final layout. Therefore cells designed for small configurations may not be suited for larger ones which might require larger buses and larger transistors to drive them. Since the RSG cannot modify the primitive cells specified in the sample file one solution to the layout optimization problem would be to design several cells for each functionality where each cell is designed for a different configuration range. For example one might

design three different input buffers for a PLA. One type of buffer would be designed for use in PLAs with a large number of product terms, another for use in PLAs with an average number of product terms and one for use in PLAs with a small number of product terms. This method of choosing the right set of primitive cells according to the replication factors , requires the substantial layout investment of having to design a large number of cells. Also the method lends itself to only a *coarse grained* optimization due to the approximation of the electrical optimization requirements by one of the cells already defined in the library. The appropriate device sizes given some speed and power constraints could be derived from Macromodeling Optimization techniques[22].

The problem of making the RSG technology transportable and allowing generation of electrically optimized layouts could be achieved by using a special kind of compactor which I will refer to as a *leaf cell* compactor. I believe that this kind of compactor has not yet been seriously investigated because of the significant difficulties encountered in straightforward compaction, and also because the usefulness of this kind of compactor is closely related to an RSG type design methodology whose benefits have only recently been established.

A *leaf cell* compactor is a compactor capable of compacting cells from a library while taking into account how the cells in the library may potentially interface together. For example if cells A and B can potentially interface as in Figure 2.3 then while compacting cell A we have to take into account the constraints generated by its connection to B. If cell B cannot be compacted further then it is possible that due to the constraints between A and B, A cannot be compacted further although A if compacted by itself on a

classical compactor could stand to be further compacted. Context sensitive compaction is different (probably simpler) than hierarchical compaction [8] which starts with a complete final layout but does the compaction hierarchically.

The advantages of a *leaf cell* compactor are that by compacting only the primitive cells in a library instead of fully assembled structures the compaction effort is not duplicated over the various replication factors in the layout. For example if a cell A appears a hundreds time in a layout, a compactor operating on the final layout (where A appears one hundred times) would be more computationally expensive than one which cleverly compacts the cell A only once. Also the compaction may only be performed once for a given set of design rules (and other constraints such as bus and device sizing) instead of running the compactor on each new structure created (by the RSG). These two factors (i.e. the compaction effort not being duplicated over the various replication factors and also the compaction being performed only once and not on each structure generated) can lead to orders of magnitude improvements in computation costs, perhaps allowing implementations previously thought of as too computationally costly (such as for instance simulated annealing[16]).

The costs associated with a *leaf cell* compactor are:

1) Perhaps a more complex compactor.

2) After compaction all instances of a cell A in the final layout have exactly the same geometry. In the case of a classical compactor which first flattens the layout (gets rid of the cell hierarchy) before compacting it, circuitry which used to belong to instances of A may end up having different layout geometries.

The relaxation of the constraint that all *instances* of A have the same geometry can potentially lead to more optimal layouts. Howeyer in the case of highly regular structures with large replication factors, what goes on along the boundary of arrays of cells has a negligeable impact on the total size of the layout. Most of the cells in a large structure are far away from the boundaries of the array (assumed for simplicity sake to be an array of identical cells) anyway and hence geometrical constraints on each of them can be nearly identical since the constraints caused by the boundary of the array can be attenuated. Hence the constraint that the layout of all the instances of A be identical after compaction may not be too restrictive. Furthermore assuming that compactors are not perfect and do from time to time produce legal but electrically poor layout, quality control of the compactor output can more easily be performed on a library of a few cell than on each of the large layouts generated by an RSG type generator.

At this point let us take a step back and examine the real motivation behind a *leaf cell* compactor and the motivation behind a classical compactor, since they differ in essence. A good classical compactor should be able to start with a stick diagram or a very poorly designed starting layout. From this poor starting point the compactor should be able to investigate different compaction options in order to find an optimal (or satisfactory) layout. Unfortunately for a given electrical functionality, the space of legal layouts is not convex. This means that if we use a model where we continuously deform the starting layout in search of a more optimal one (while keeping the layout legal at all times) we might have to shrink as well as expand the layout as we move along a path leading to an optimal solution. Therefore a *greedy* algorithm which looks only for a local minima can fail to find very profitable

optimizations which require *hill climbing* (moving temporarily in a direction leading to to a less optimal layout). One dimensional compactors which compact in one dimension at a time are an example of *greedy* optimizations which do not lead to the optimal solution. A one dimensional compaction algorithm tries to *greedily* optimize one dimension at a time and misses out on the optimizations that require a more careful analysis of the interaction between the two dimensions. Besides the fact that the space of legal layout may not be convex it may also not be connected. In order to reach an optimum by a continuous deformation from the initial layout one might have to deform the layout along a path parts of which do not correspond to legal layouts.

The motivation behind a *leaf cell* compactor is to be able to transform cells from one technology to another and also to be able to size busses and devices. The cells already existing in the library can be assumed to be highly optimized for the technology in which they are designed and there is a good chance that the topology of the initial layout can be used as a good starting point for the target technology into which we are going to compact the cells. Under these assumptions the minima (of the objective function) has a better chance to be reached by a greedy type algorithm that searches for a local minima. Hence some of the inherent difficulties in *leaf cell* compaction can be offset by the previous simplifying assumptions on the initial starting layout (namely that the cells in the library can be assumed to be designed carefully and the easier quality control of the output) making the task of designing such a compactor a more manageable one.

6.2 Defining a cost function

The purpose of this section is to show the importance and raise some of the issues related to defining a layout cost function for a *leaf cell* compactor. The cost function is an evaluation of the *goodness* of the layout and the compactor's goal is to produce the layout with the lowest cost subject to a set of constraints. Defining a cost function for a *leaf cell* compaction scheme is not as straightforward as it is in the case of a simple compactor. Also the impact of the chosen cost function on the final layout (variations in the final layouts produced using different cost functions) may be greater than would be the case in simple compaction.

Figure 6.1 shows a structure consisting of a linear array of cells. The m rightmost cells are of type A and have pitch λ_a, the n leftmost cells are of type B and have pitch λ_b. It can be shown that in the general case (if there are constraints between A and B other than those shown in Figure 6.1) there are tradeoffs between minimizing λ_a and λ_b. λ_a can be minimized to a greater extent at the cost of increasing λ_b and vice versa. Let us consider an extremely simple cost function for simple compaction and try to find a corresponding cost function in the case of *leaf cell* compaction. Let the cost function be X, the *x dimension* size of the layout (for simplicity sake assume that the y coordinates are fixed). Finding an optimal λ_a and λ_b (given the geometric constraints) so as to minimize X, depends on the replication parameters n and m. However in a *leaf cell* compactor n and m are not known at compaction time. Hence the user has to explicitly provide a cost function in terms of λ_a and λ_b (as well as other parameters) based on empirical estimates of what n and m are expected to be. In the case where n and m are large numbers

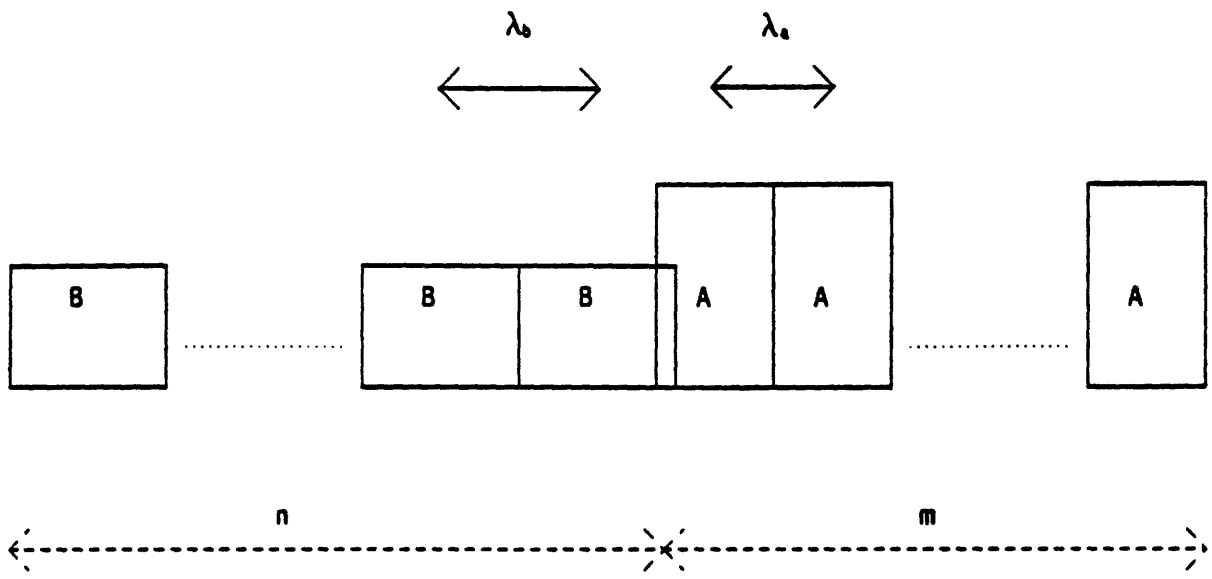


Figure 6.1: Defining a cost function.

$X \approx n\lambda_a + m\lambda_b$, therefore minimizing λ_a and λ_b is much more important than minimizing the sizes of the cells themselves. For a given λ_a and λ_b (assume for simplicity sake that the I_{ab} interface is fixed) reducing the size of A and B has only a marginal impact on X because it effects only the extremities of the array, since its impact is independent of the replication factors n and m. Hence the cost function should depend essentially on λ_a and λ_b and to a much lesser extent on the physical sizes of the cells themselves.

The remainder of this section describes a layout example where the pitches λ_i between the cells do in fact have to be traded off. Figure 6.2(a) shows three instances of a same cell A. The cell A consists of two horizontal bars. Since the three instances are all of the same celltype the pitch between them is the x distance between the left edges of their bounding boxes. This is because the x distance between their respective points of call and the left edges of their bounding boxes is the same and hence cancels out in the pitch calculation. One can reduce the λ_1 pitch by moving the top bar of the top instance toward the left. This causes the layout to deform to the configuration of Figure 6.2

(b). Moving the top bar of the topmost instance to the left causes the bottom bar of the middle instance to move to the right increasing the pitch λ_2 in so doing.

Choosing an appropriate cost function can be facilitated by the knowledge of the replication parameters in the structure to be built from the leaf cells. An optimal cost function for a given set of replication parameters may not be optimal for another set of parameters. In practice, however, tradeoffs between the pitches may not be as extreme as in Figure 6.2. Experimental results are needed to determine just how much interaction there is between the pitches of leaf cells that occur in practice. Making the cost function linear in the λ_i and the box edge locations can substantially simplify the problem of solving the constraint system i.e. finding a minimum for the cost function subject to the constraints.

6.3 Constraint Representation

The purpose of this section is to propose a representation of the constraint system in *leaf cell* compaction. It is assumed that the reader is somewhat familiar with graph based constraint systems. We will restrict ourselves to *one dimensional* compaction in the *x dimension*. Compacting in the *x dimension* entails determining the abscissas of all the vertical edges of the boxes in a layout. Horizontal edges play no role in the constraint representation and are assumed to shrink or expand in response to the displacement of the vertical edges. In the case of *leaf cell* compaction the unknowns of the problem are the abscissa of the vertical edges of boxes in the leaf cells, as well as the λ_i which are the *x dimension* pitches between the various cells. The known

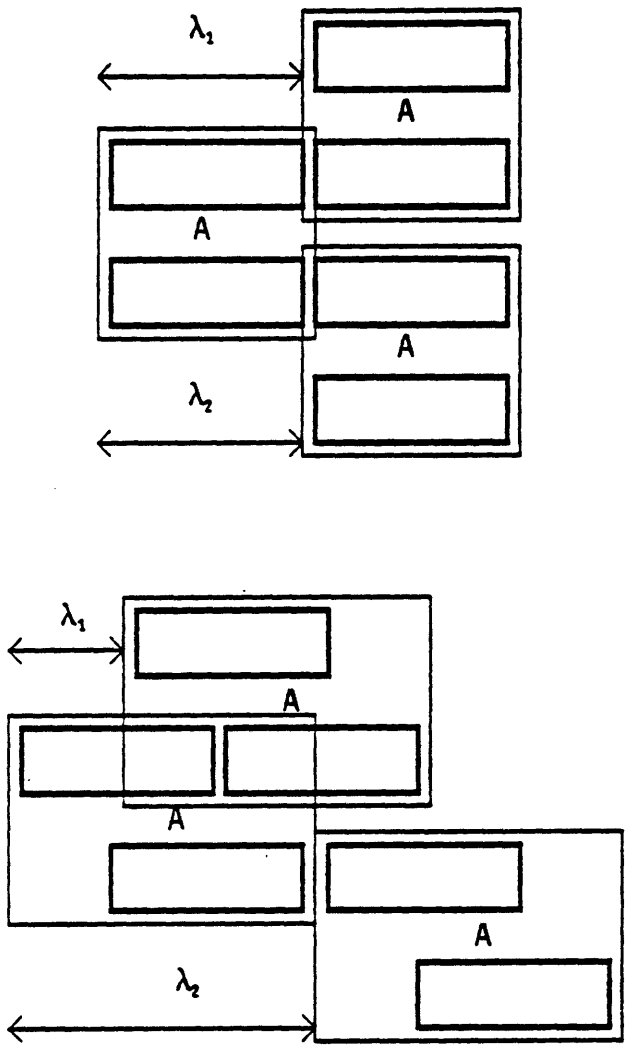


Figure 6.2: Tradeoff between pitches.

parameters are the design rules of the process, the sizing constraints that arise from electrical considerations and the electrical network implicit in the initial layout. The constraints that arise from the interaction of the parameters can be represented by a constraint graph whose vertices correspond to vertical edges of boxes in the layout. The edges between the vertices in the graph correspond to minimum spacing constraints between the objects represented by the vertices. The weights on the edges of the graph are the actual values of the minimum permissible distances between the vertices.

A possible strategy for leaf cell compaction is to build a constraint graph for each of the leaf cells and then include the constraints arising from the interaction of the cells by adding new edges between the graphs. The resulting graph (formed by the union of the leaf cell constraint graphs and the new edges) has 2 kinds of constraints: *intra cell* constraints (constraints within a cell) and *inter cell* constraints (constraints from the interaction between cells). Both *intra cell* and *inter cell* constraints can be extracted from an RSG sample layout. The *intra cell* constraints can be extracted from the cell definitions of the leaf cells in the sample layout. *Inter cell* constraints can be determined from the various *cell interfaces* present in the sample layout. After the compaction is completed, it is possible to build a new sample layout for the new technology and electrical constraints, from the new cell definitions of the leaf cells and the new pitch parameters (both of which were the unknowns of the initial compaction problem). Recall from Section 3.1 that the sample layout does not necessarily have to contain all the possible interfaces that might occur in a final layout (because the RSG connectivity graph need only be a spanning tree). However if a sample layout is to be used for leaf cell compaction, then in order for the compactor to generate all the required

inter cell constraints it is imperative that all possible interfaces that might arise in the final layout be present in the sample layout. The next paragraph describes how these constraints can be generated in the very simple case where the sample layout contains 1 cell and 1 interface.

Figure 6.3 shows two instances of A interfaced together. A is a cell containing four vertical (box) edges. The left (respectively right) instance of A as well as the corresponding $1, 2, 3, 4$ (respectively $1', 2', 3', 4'$) constraint graph and the edges in the graph are shown in solid (respectively dotted) line. *Inter cell* constraints between the two instances arising from the existence of the I_{aa} interface are shown in broken line. If compaction was performed on the $1, 2, 3, 4, 1', 2', 3', 4'$ graph, the compacted layouts of the two instances of A may not be identical. The unknowns of the problem are the abscissa of the four vertical edges in the cell (and not the instances of) A and the pitch λ_a after compaction. We must express the constraint system in terms of a graph where the vertices are the vertical box edges of A and the weights are functions of λ_a. This will ensure that both instances of A in the compacted layout have the same geometries. Since the pitch between the two instances is λ_a the distance between the 1 and the $1'$ node is necessarily λ_a. Hence since node 4 must be x_4 to the left of node $1'$ it must be $x_4 - \lambda_a$ to the left of 1. Therefore we can replace the dashed edge weighted by x_4 by an edge from node 4 to node 1 weighted by $x_4 - \lambda_a$. Similarly we can replace the edge between node 4 and node $3'$ weighted by x_5 by an edge between node 4 and node 3 weighted by $x_5 - \lambda_a$. Once this edge replacement is complete we can discard the $1', 2', 3', 4'$ graph and all edges terminating on vertices of that graph. We are then left with the $1, 2, 3, 4$ graph where the edges drawn with straight lines are *intra cell* constraints and edges drawn with arcs are the

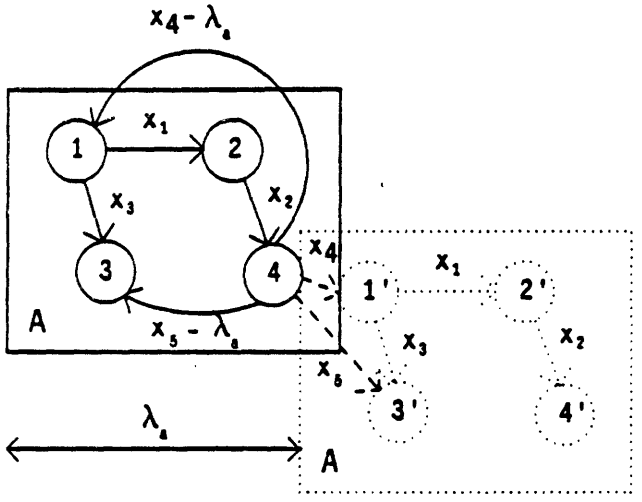


Figure 6.3: Constraint representation.

inter cell constraints. The new constraint system ensures that both instances of A will have the same geometries and at the same time reduces the number of unknowns from 8 (the abscissas of $1, 2, 3, 4, 1', 2', 3', 4'$) to 5 (the abscissas of $1, 2, 3, 4$ and λ_a). In the case of larger cells and multiple interfaces, the reduction in the number of unknowns can be be much more substantial since only one new unknown (a λ_i pitch parameter) is added for each new interface.

This graph constraint system cannot be solved by shortest path algorithms such as Bellman Ford[17] because the weights on the edges are not all constants. Some of the weights depend on the λ_i which must also be determined. Algorithms such as the Bellman Ford algorithm are used to solve a system of linear equations where there are only (at most) two unknowns per equation. Such systems can be represented by a constraint graph with constant weight edges. However (if the abscissas of the vertices $1, 2, 3, 4$ are X_1, X_2, X_3, X_4) in the resulting graph of Figure 6.3 the edge between node 4

and node 1 represents the equation $X_1 - X_4 \leq x_4 - \lambda_a$ where X_1, X_4 and λ_a are unknowns. A simple minded way to solve the system would be to convert the graph to a system of linear equations and solve the system of equations using a linear programming algorithm like Simplex [10]. Since we know that there are tradeoffs between the λ_i we will have to define a cost function that is to be minimized subject to the above set of constraints.

6.4 Experiments in compaction

Over one hundred and thirty kilobytes of code have been written in order to build an experimental compactor with the intent of modifying it to ultimately do *leaf cell compaction*. One third of the compactor code deals with maintaining and manipulating the data structures (such as scan lines sorted lists etc..) required by the constraint generation process. This is where most of the CPU time is spent. One fourth of the code embeds the decision making process of determining what type of constraint is appropriate between a pair of box edges. This part of the code proved to be the most convoluted, the hardest to write and debug and also the most error prone. The actual constraint solving routine (a modified Bellman Ford Algorithm: see Subsection 6.4.2) is only slightly over a page in length. The rest of the code is overhead and consists of layout manipulating routines, design rule tables etc.. The speed of the compactor compares favorably with other compactors and the output quality can, depending on the input layout, be reasonably good. However for a large complex layout the compactor will often produce a legal layout where small regions of the layout are electrically poor, making hand checking (and minor modifications) of the result a necessity.

While the general methods and mathematical foundations of the compaction problem are well understood they seem inadequate to deal with the myriad of special cases encountered in practice. Whether commercial compactors function properly in a realistic VLSI setting is still an open question for me as I did not have a compactor with which to compare results readily available to me. However I believe that my compactor would compare favorably on many of the examples found in compaction papers. Rather than laboriously go through the quagmire of designing and implementing a reasonable compactor, I will skim through some of the salient difficulties and in some cases propose solutions to the problems I encountered. Many of the classical difficulties of compaction are explained in [31].

The rest of this section is for the benefit of whomever continues the compactor project. it describes three major difficulties (encountered during the compactor project) which can be corrected by a more appropriate choice of strategy. Its intent is not to give an overview of the compaction problem. The compactor used a one dimensional graph-based constraint method where the vertices in the graph represent layout box edges[1]. Other one dimensional techniques include shear line compaction [9].

6.4.1 Constraint generation

One of the purposes of the compactor is to perform device and bus sizing. Device and bus sizing requires the ability to tag (identify) the particular devices (or buses) to be sized in the layout. This can be accomplished by making the bus (or the gate and channel of the device) to be sized, a cell.

[1]the edges are vertical since it is assumed throughout this section that compaction is being performed in the x dimension.

The compactor can then size all instances of that cell according to some user defined specification. In some processes transistor gates must be wider than the minimum poly width. This can be achieved by making the gates of transistors instances of a particular cell. The compactor must then make all instances of that cell a certain minimum size. Finally there may be critical parts of the layout (such as sense amplifiers) which must be left unchanged by the compactor. This also can be achieved by making those portions of the layout (to be kept *frozen*), out of cells which the compactor will know how to handle.

Many compactors first perform a preprocessing phase on the layout. During this preprocessing phase boxes of the same layer are merged together. For example EXCL uses a merging technique (although not for compaction) which gets rid of redundant vertical edges of boxes. After the merging process is complete each layer of the layout consists of nonoverlapping boxes such that each box has the largest possible x dimension size (as a result of this there are no hidden[2] or partially hidden vertical edges).

Merging boxes considerably reduces the constraint generation problem. Figure 6.4 shows two boxes of a same layer (in solid line). The existence of a minimum spacing constraint between the right edge of the left box and the left edge of the right box depends on the presence of the middle box (shown in broken line) whose presence masks the two previous edges. Always generating the constraint between those two edges (regardless of the presence of the middle box) can substantially overconstrain the system. Consider a piece of diffusion fragmented into n abbuting boxes as in Figure 6.5. Indiscriminately

[2]A hidden box edge is an edge that does not actually correspond to an actual boundary of a layer since material from the layer is present on both sides of the edge.

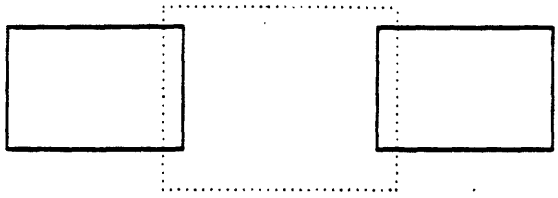


Figure 6.4: Constraint for hidden edges

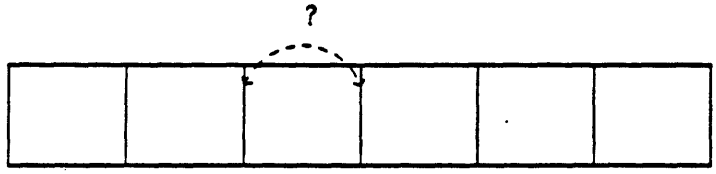


Figure 6.5: Fragmented Layout

generating constraints between left edges end right edges would force the x size of the final layout be at least $n\lambda$ where λ is the minimum spacing for diffusion. Merging the boxes into one box would get rid of the fragmentation and allow the layout to shrink to the minimum width for diffusion.

Unfortunately, due to the device and bus sizing mechanism in the compactor, it is not possible to perform merging on the boxes. Merging boxes causes loss of information relating to which cells the boxes came from. A long bus might require to be wider in certain regions. These regions can be identified by the compactor as being part of certain cells. Merging the boxes in the bus of Figure 6.5. would cause the loss of that information since after the merging process there is only one box for the whole bus. This constraint (i.e. merging being unacceptable) combined with the wrong constraint generation technique made constraint generation an extremely hard problem. The main problem is to generate enough constraints so that the result is a legal layout without overconstraining the system, which degrades the quality of

the result.

The minimal constraint set is not unique (A minimal constraint set is such that removing any constraint from it may cause the resulting layout to become illegal) and therefore it is not possible to reach the optimal constraint set simply by removing overconstraining constraints. Generating a good constraint set is a particularly hard problem. Substantial gain in output quality can be made by simply making the constraint generator smarter without having to go to a more complex compaction strategy as in two dimensional compaction [15].

Most graph based compactors use a scan line technique for the generation of constraints. Other reasonable ways of generating constraints include walking through a layout database as in MAGIC where each box (tile) has pointers to its neighbors (corner stitching). There are essentially two possible ways to perform scanning. The way it was performed in the compactor was using a scan line which represents a slice through the layout[3]. Constraints in the x dimension are generated with a horizontal scan line that moves vertically. At any given time the scan line holds the part of the layout that intersects its current y position[4]. Only objects that were in the scan line at the same time can have a constraint between them. If the current scan line location intersects the piece of diffusion in Figure 6.5 then all the boxes in the Figure are simultaneously present in the scan line. The constraint generator must then examine each pair of vertical edges and determine what constraint to put between them. In order to determine the appropriate constraint between

[3]EXCL uses this method.

[4]In practice the scan line is actually a band. It contains objects that intersect a band centered at it's current y location.

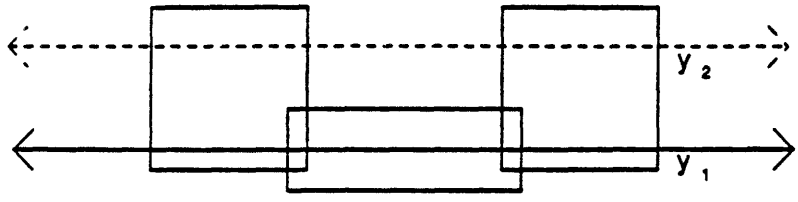


Figure 6.6: Constraint between partially hidden edge

a pair of edges, the constraint generator has to shuffle through the objects in the scan line to examine the relevant neighboring objects. This turns out to be one of the most difficult and critical parts of the compactor. A smart compactor must at least notice that some of the edges might be hidden and that it may not be appropriate to put a constraint between them. Deciding on an appropriate constraint is not a straightforward task. In Figure 6.6 the right edge of the leftmost box and the left edge of the rightmost box are hidden when the scan line is at location y_1. However when the scan line reaches y_2 the edges are no longer hidden and therefore the constraint generator must place a constraint between the two edges.

By selecting a more appropriate scanning technique it is possible to eliminate part of the hidden edge problems. The scan line can be a vertical line that sweeps from $-\infty$ to $+\infty$ (we are still generating constraints for the x dimension). The scan line contains information of what a viewer on the scan line looking toward the left would see. In Figure 6.7 the viewer on the scan line would see the x_2, x_3 segment of the left box and will see the x_1, x_2 segment as belonging to the insides of the right box. Constraints are placed between what the viewer can see in the scan line and the objects that currently intersect the scan line. More details on this scan line technique and relevant data

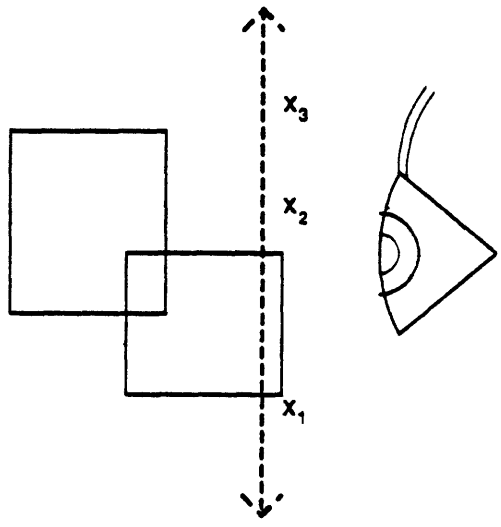


Figure 6.7: Correct scan line method

structures can be found in [11] and [24] [5]. The advantage of this method is that hidden edges are automatically taken care of because they do not show up in the scan line. Hence merging of boxes is implicitly taken care of.

6.4.2 Solving the Constraint System

The Bellman Ford algorithm [17] was used to solve the graph based constraint system. The Bellman Ford assigns to each vertex the lowest possible abscissa subject to the constraints. The algorithm proved to be extremely fast, especially if the edges are traversed in sorted (according to their abscissa) order, i.e. a preliminary sort on the edges according to their abscissa in the initial layout is performed. This is because the initial ordering of the edges is a good estimate for the final ordering. Going through the edges in a suitable order considerably reduces the number of Bellman Ford relaxation

[5] [28] uses this method.

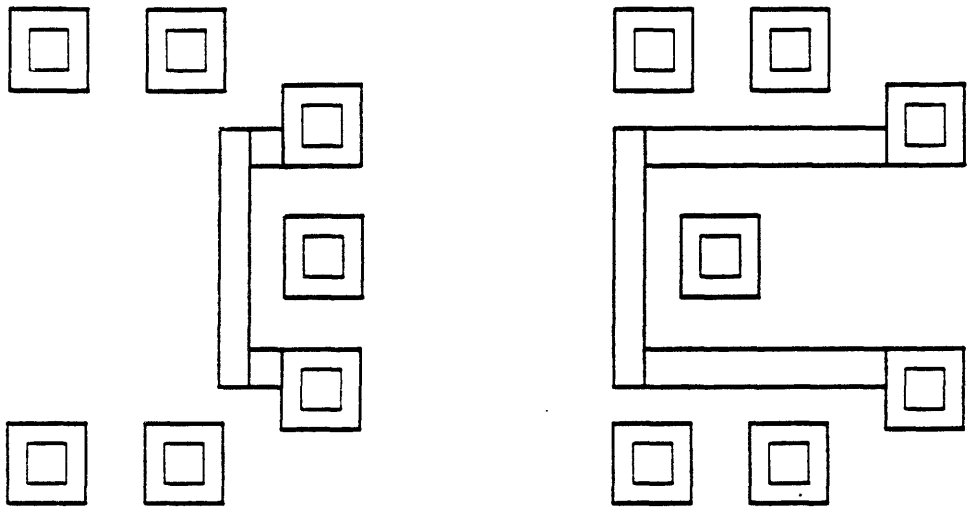


Figure 6.8: Worsening of a layout Jog

steps. In the case where the initial ordering is preserved in the final layout exactly one relaxation step is required instead of the $\mid E \mid$ (where $\mid E \mid$ is the number of vertices in the constraint graph) required in the worst case. Unfortunately while Bellman Ford does a good job of minimizing the total size (bounding box) of the layout it can generate electrically poor layouts. This is because although the algorithm minimizes the longest path it can actually increase the length of other paths (up to the length of the longest path).

The Bellman Ford algorithm consists of pushing all the objects in a layout as much to the left as they can go subject to the constants. When applied to the layout of Figure 6.8(a) the resulting layout of Figure 6.8(b) develops a jog in it. A more appropriate algorithm would be one that tries to bring all objects close together as if they were all connected by rubber bands instead of trying to move them all to one side as if they are being attracted by a large magnet on the left.

6.4.3 Dealing with layer Interaction

Some design rules such as those for contacts or gates are hard if not impossible to express in terms of minimum spacing constraints between the mask layers of a layout. These kind of constraints often occur due to the interaction of several layers at a time. For example the width of poly may be 3λ except over diffusion (gate of a transistor) where it might have to be 5λ. Not knowing beforehand where in the compacted layout poly will end up over diffusion it is hard to determine which regions of poly should have a 5λ width constant on them. This is because constraints are generated based on the initial layout whose topology will change during compaction.

One way of solving this class of problems is to create new layers that do not correspond to actual mask layers in the lithographic process. This method is already used in editors such as Magic [26]. For example Magic has a special layer called *contact* which has design rules similar to those of any other layer. This special layer is comprised of metal, poly and the actual contact cut (or cuts) between them. At mask creation time the contact layer is converted into actual lithographic mask layers which may contain one or several contact cuts depending on the size of the *contact* layer. The appropriate metal and poly overlaps as well as the size and spacing of the contact cuts can be looked up in a table. Figure 6.9 shows an example of what this translation process when applied to a large *contact* layer might look like. The same type of strategy can be used for transistors, buried contacts, etc.. The benefit of this strategy is that often the *new layers* that result from the interaction of several primitive layers can be characterized by simple design rule constraints while as the interaction of the different layers often can not.

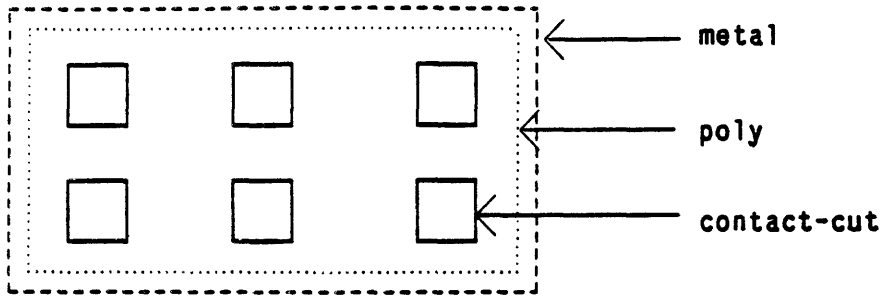


Figure 6.9: Contact layer Expanded

6.5 Summary and new directions

In this chapter some of the benefits and difficulties of *leaf cell* compaction have been explored. A constraint representation for *leaf cell* compaction has also been proposed. Difficulties encountered during the design and implementation of an experimental compactor (a flat layout compactor) have been described and improvements have been suggested. The rest of this section describes a plausible sequence of steps leading to the implementation and evaluation of an efficient *leaf cell* compactor.

Section 6.4.3 relates the problems of dealing with layer interaction. This problem occurs because design rules arising from layer interaction cannot be described in terms of minimum spacing constraints. A successful compactor must be built on top of underlying mechanisms for transforming a set of *physical* mask layers into special layers as prescribed by Section 6.4.3, and transforming these special layers back into physical layers. A flexible constraint generator (for flat layout compaction) implementing the right kind of scanning technique and a carefully constructed set of constraint generation rules must be built. The ultimate goal is to modify the constraint generator

to do *leaf cell* constraint generation. Provisions for interfacing the constraint generator to a device sizing tool such as [22] must be considered. Care must be taken not to underestimate the difficulty inherent in constraint generation, and a carefully charted course must be generated before any actual code is written. Testing the constraint generator for larger than simple test cases cannot be accomplished without building a throw-away test constraint solver (for flat compaction). The constraint solver's purpose will be to facilitate testing of the constraint generator by outputting actual compacted layouts instead of constraint graphs. Once testing is completed the constraint generator must be modified to do *leaf cell* compaction and an appropriate constraint solving algorithm for *leaf cell* compaction must be selected or developed. The effects of different cost functions on the new *leaf cell* compactor must be evaluated and catalogued. Finally an exploration of how the compactor and the RSG can together constitute an efficient layout module in a larger silicon compilation system must be investigated.

Chapter 7

Conclusion

The push to design larger and more complex VLSI chips has spurred the creation of more sophisticated design tools. By restricting the target architecture to designs that are regular and can be algorithmically described, efficient and flexible layout generators that function well in a realistic VLSI setting can be built. Regularity, however, does not exclude complexity in the personalization of these structures. This thesis has demonstrated the importance of the appropriate abstraction mechanisms — *macrocells*, *interfaces*, and *interface inheritance* — in generating layouts for realistic regular structures. The RSG is an operational tool that supports true macro abstraction and inheritance. Due to the flexible target architecture, greater generality than specialized module compilers can be achieved without the loss of efficiency incurred in silicon compilers with a fixed target architecture. The RSG presents a convenient interface to the user by separating the graphical and procedural description of a circuit along a natural boundary, making it an extremely easy tool to utilize, extend, and upgrade. Information is efficiently partitioned into a *design file* which describes the global layout

connectivity and a *sample file* which specifies the local placement constraints and the specifics of the primitive cells. Tangible proof of the efficiency and applicability of the RSG method to intricate regular structures that arise in meaningful applications was demonstrated by the design of a (class of) pipelined multiplier. The RSG's power can be further enhanced by a special kind of compactor which will make the RSG technology transportable and allow it to perform device and bus sizing. The simple mechanisms used in the RSG can be easily embedded in a complete VLSI design system. Such a design system would include placement and routing and also compilation from a functional specification. The RSG could then be an efficient link in the design chain from functional specification to silicon.

Appendix A
BNF Grammar

⟨procedure definition⟩	:=	⟨function definition⟩
	:=	⟨macro definition⟩
⟨function definition⟩	:=	(defun ⟨function name⟩ ⟨formals⟩ ⟨locals⟩ ⟨body⟩)
⟨macro definition⟩	:=	(macro ⟨macro name⟩ ⟨formals⟩ ⟨locals⟩ ⟨body⟩)
⟨formals⟩	:=	(⟨variable list⟩)
⟨locals⟩	:=	(local ⟨variable list⟩)
⟨variable list⟩	:=	⟨variable⟩ ⟨variable list⟩
	:=	⟨empty⟩
⟨variable⟩	:=	⟨simple variable⟩
	:=	⟨indexed variable⟩
	:=	⟨2indexed variable⟩
⟨body⟩	:=	⟨statements⟩
⟨statements⟩	:=	⟨statement⟩ ⟨statements⟩
	:=	⟨empty⟩
⟨statement⟩	:=	⟨conditional⟩
	:=	⟨do loop⟩

:= ⟨assignment⟩

:= ⟨function call⟩

:= ⟨macrocall⟩

:= ⟨primitive function call⟩

:= ⟨print statement⟩

:= ⟨read statement⟩

:= ⟨prog statement⟩

:= ⟨variable⟩

:= ⟨connect statement⟩

:= ⟨make instance⟩

:= ⟨subcell⟩

:= ⟨make cell⟩

:= ⟨declare interface⟩

⟨conditional⟩ := (cond (⟨cond exprs⟩))

⟨cond exprs⟩ := ⟨cond expr⟩ ⟨cond exprs⟩

:= ⟨empty⟩

⟨cond expr⟩ := ⟨if part⟩ ⟨then part⟩

⟨if part⟩ := ⟨statement⟩

⟨then part⟩ := ⟨statement⟩

(do ((⟨simple variable⟩ ⟨initial value⟩

⟨do loop⟩ := ⟨next value⟩⟨exit conditional⟩)

(body))

⟨initial value⟩ := ⟨statement⟩

⟨next value⟩ := ⟨statement⟩

⟨exit statement⟩ := ⟨statement⟩

⟨assignment⟩ := (assign ⟨variable⟩ ⟨statement⟩)

⟨function call⟩ := (⟨function name⟩ ⟨variable list⟩)

⟨macro call⟩ := (⟨macro name⟩ ⟨variable list⟩)

⟨primitive function call⟩ := (⟨primitive function name⟩ ⟨variable⟩ ⟨variable⟩)

⟨print statement⟩ := (print ⟨statement⟩)

⟨read statement⟩ := (read)

⟨prog statement⟩ := (prog ⟨statements⟩)

⟨connect statement⟩ := (connect ⟨variable⟩ ⟨variable⟩ ⟨statement⟩)

⟨make instance⟩ := (mk_instance ⟨variable⟩ ⟨statement⟩)

⟨subcell⟩ := (subcell ⟨variable⟩ ⟨statement⟩)

⟨make cell⟩ := (mk_cell ⟨simple variable⟩ ⟨statement⟩)

:= (declare_interface ⟨statement⟩ ⟨ statement ⟩
⟨statement⟩ ⟨statement⟩
⟨statement⟩ ⟨statement⟩)

⟨2indexed variable⟩ := ⟨simple variable⟩.⟨statement⟩.⟨statement⟩

⟨indexed variable⟩ := ⟨simple variable⟩.⟨statement⟩

⟨simple variable⟩ := ⟨string of chars⟩

⟨function name⟩ := $\overline{m}$⟨string of chars⟩

⟨macro name⟩ := m⟨string of chars⟩

⟨string of chars⟩ := a string of charecters

⟨empty⟩ :=

Appendix B

Multiplier Design File

```
(macro mcell (xsize ysize xloc yloc)
       (locals c foo)
       (mk_instance c corecell)
       (cond ((= xsize xloc)
      (cond ((= ysize yloc)(connect c (mk_instance foo type1) t1inum))
    (true (connect c (mk_instance foo type2) t2inum))))
     (true (cond ((= ysize yloc)
  (connect c (mk_instance foo type2) t2inum))
 (true  (connect c (mk_instance foo type1) t1inum)))))
       (cond ((= (mod xloc 2) 0)
      (connect c (mk_instance foo clock1) clk1inum))
     (true (connect c (mk_instance foo clock2) clk2inum)))
       (cond ((= yloc ysize) (connect c (mk_instance foo top2) top2inum))
     (true (connect c (mk_instance foo top1) top1inum))))

(macro mline (xsize ysize currentline)
       (locals 1. ref)
       (assign 1.1 (mcell xsize ysize 1 currentline))
       (setq ref (subcell 1.1 c))
       (do (i 2 (+ 1 i) (> i xsize))
   (assign 1.i (mcell xsize ysize i currentline))
   (connect (subcell 1.(- i 1) c) (subcell 1.i c) hinum)))

(macro m2darray (xsize ysize)
       (locals c1. topright  bottomright)
       (assign c1.1 (mline xsize ysize 1))
       (setq topright (subcell c1.1 ref))
       (do (i 2 (+ 1 i) (> i ysize))
   (assign c1.i (mline xsize ysize i))
   (connect (subcell c1.(- i 1) ref) (subcell c1.i ref) vinum))
       (setq bottomright (subcell c1.ysize ref))
       (mk_cell mularrayname bottomright))

(macro mtopregs (size)
       (locals 1. ref)
       (assign 1.1 (array topreg 1 topregvinum))
       (setq ref (subcell 1.1 c.1))
       (do (i 2 (+ 1 i) (> i size))
   (assign 1.i (array topreg i topregvinum))
   (connect (subcell 1.(- i 1) c.1) (subcell 1.i c.1) topreghinum))
```

```
       (mk_cell topregisters  ref))

(macro mbottomregs (size)
       (locals l. ref)
       (assign l.1 (array bottomreg size bottomregvinum))
       (setq ref (subcell l.1 c.size ))
       (do (i 2 (+ 1 i) (> i size ))
   (assign l.i (array bottomreg (- (+ 1 size) i) bottomregvinum))
   (connect (subcell l.(- i 1) c.(- (+ size 1) (- i 1)))
    (subcell l.i c.(- (+ 1 size) i)) bottomreghinum))
       (mk_cell bottomregisters  ref))


(macro mrightregs (size)
       (locals l. ref length regnum)
       (setq regnum (+ 1 (* 3 size)))
       (setq length (// regnum 2))
       (cond ((= (mod regnum 2) 1) (setq length (+ 1 length))))
       (assign l.1 (array rightreg length rightreghinum))
       (assdirection l.1 1 length regnum)
       (setq ref (subcell l.1 c.1 ))
       (do (i 2 (+ 1 i) (> i size ))
   (assign l.i (array rightreg length rightreghinum))
   (assdirection l.i i length regnum)
   (connect (subcell l.(- i 1) c.1)
    (subcell l.i c.1) rightregvinum))
       (mk_cell rightregisters  ref))

(defun assdirection (rarray index length regnum)
       (locals ins outs bi foo doublereg)
       (setq ins (* index 2))
       (setq outs (- regnum ins))
       (setq bi (fmin ins outs))
       (cond ((> ins outs) (prog (setq doublereg inward)
 (setq singlereg sinward)))
     (true (prog (setq doublereg outward)
 (setq singlereg soutward))))
       (do (i 1 (+ 1 i) (> i bi))
   (connect (mk_instance foo bidirectional)
    (subcell rarray c.i) rtoregsinum))
       (connect (mk_instance foo singlereg)
(subcell rarray c.(+ bi 1)) rtoregsinum)
       (do (i (+ bi 2) (+ i 1) (> i length))
   (connect (mk_instance foo doublereg) (subcell rarray c.i) rtoregsinum)))
```

```
(macro mall (xsize ysize)
       (locals arrayfoo tregs bregs rregs tri arrayi bri rri)
       (setq rregs (mrightregs ysize))
       (setq bregs (mbottomregs xsize))
       (setq arrayfoo  (m2darray xsize ysize))
       (setq tregs  (mtopregs xsize))
       (declare_interface topregistername arrayname 1 (subcell tregs ref)
  (subcell arrayfoo topright) cell_to_topreginum)
       (connect (mk_instance tri topregistername)
(mk_instance arrayi arrayname) 1)
       (declare_interface  arrayname bottomregistername 1
   (subcell arrayfoo bottomright)
   (subcell bregs ref) cell_to_bottomreginum)
       (connect (mk_instance bri bottomregistername) arrayi 1)
       (declare_interface  arrayname rightregistername 1
   (subcell arrayfoo topright)
   (subcell rregs ref) cell_to_rightreginum)
       (connect (mk_instance rri rightregistername) arrayi 1)
       (mk_cell "all" arrayi))

(defun fmin (x y)
       (locals)
       (cond ((> x y) y)
     (true x)))

(mall xsize ysize)
```

Appendix C

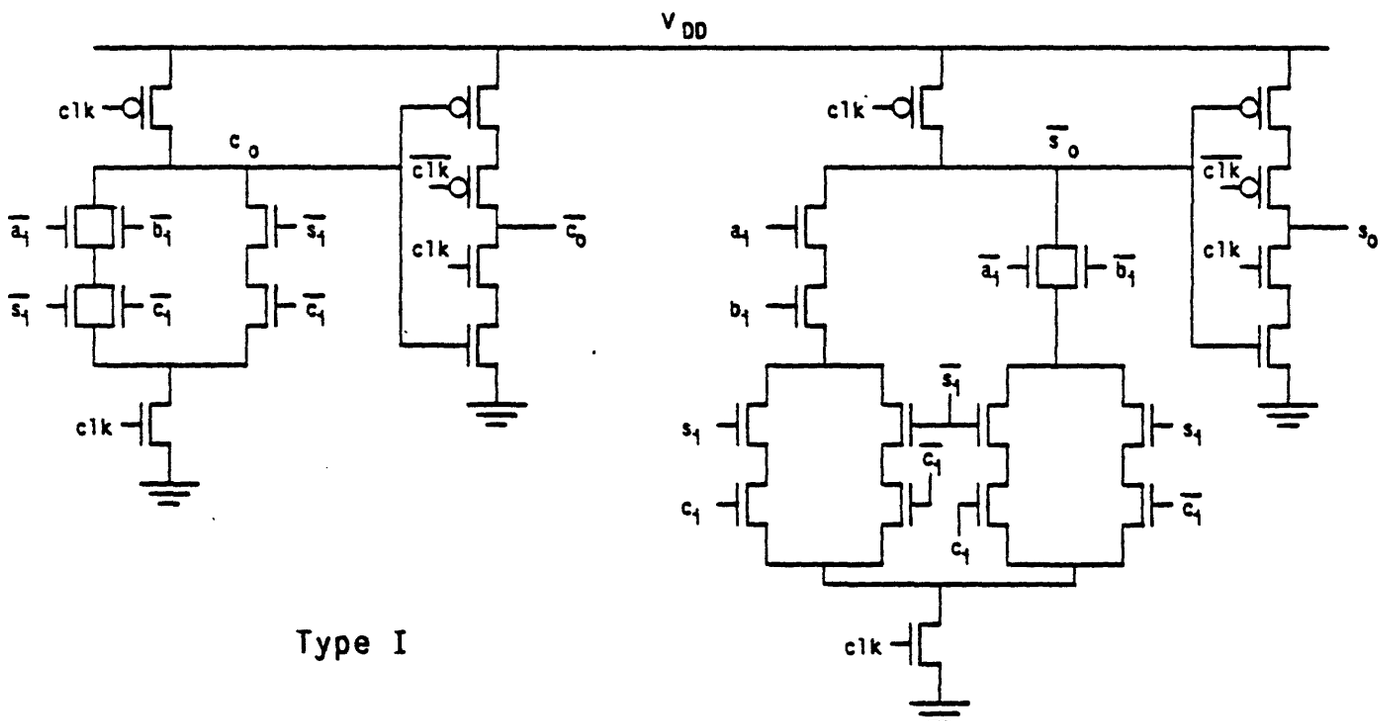
Multiplier Parameter File

```
.example_file:/u/bamji/demo/mult.def
.concept_file:/u/bamji/demo/mult.con
.output_file:/u/bamji/demo/multout.def

vinum=2
hinum=1
t1inum=1
t2inum=1

mularrayname="array"
arrayname=array
corecell=cell
type1=t1
type2=t2
clk2inum=1
clk1inum=1
clock1=clk1
clock2=clk2
top1=top1cel
top2=top2cel
top1inum=1
top2inum=1

topregvinum = 2
topreghinum = 1
topreg = tr
topregisters = "topregs"
topregistername = topregs

bottomregvinum = 2
bottomreghinum = 1
bottomreg = br
bottomregisters = "bottomregs"
bottomregistername = bottomregs

rightregvinum = 2
rightreghinum = 1
rightreg = rr
rightregisters = "rightregs"
rightregistername = rightregs
```

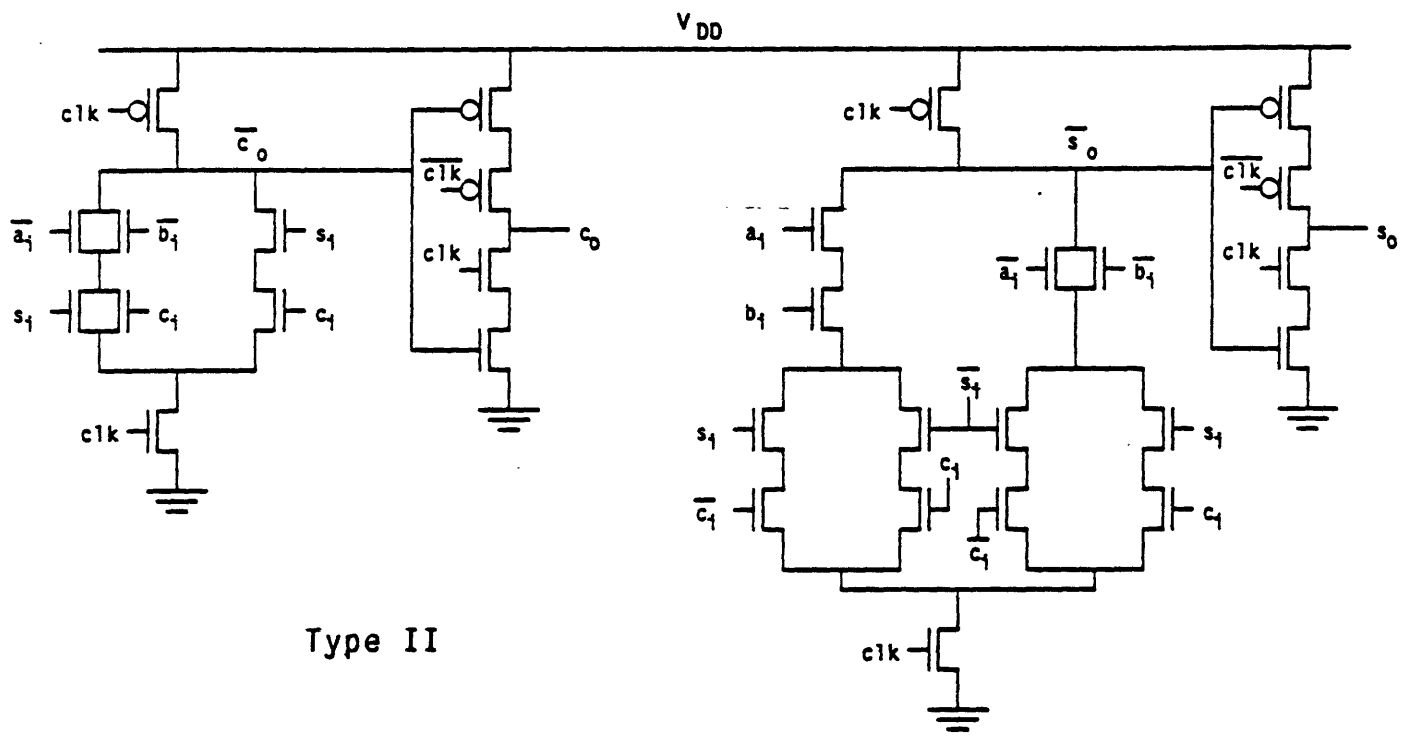
```
bidirectional= goboth
inward=goleft
outward=goright
sinward=gosleft
soutward=gosright

rtoregsinum=1

xsize=asize
ysize=asize

cell_to_topreginum=1
cell_to_bottomreginum=1
cell_to_rightreginum=1

asize=16
```

Appendix D

Adder Cell Schematic



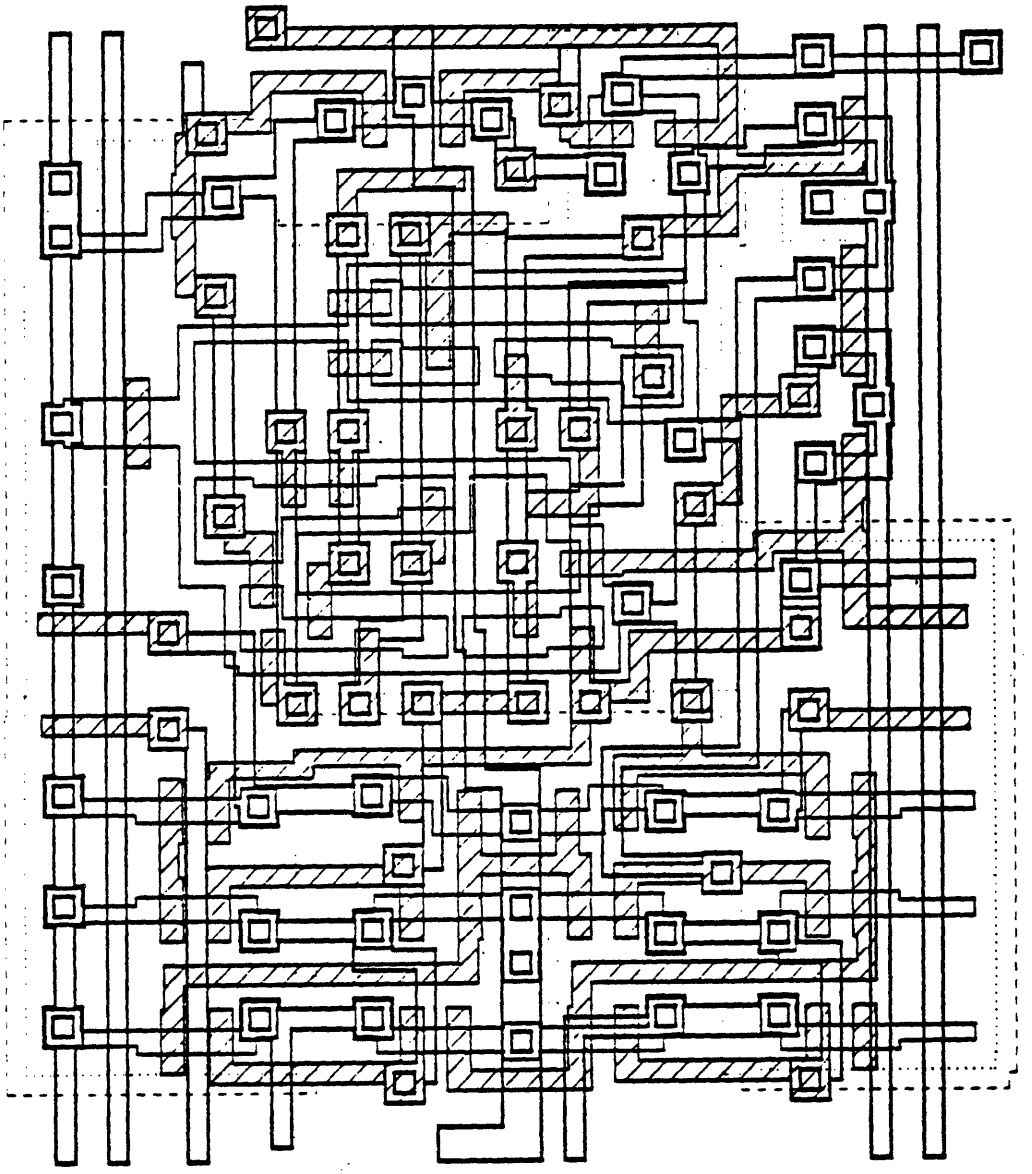
Type I


V_{DD}
clk
$\overline{c}_0$
$\overline{clk}$
$\overline{a}_1$
$\overline{b}_1$
s_1
c_0
clk
clk
$\overline{s}_0$
$\overline{clk}$
a_1
b_1
$\overline{a}_1$
$\overline{b}_1$
clk
s_0
$\overline{s}_1$
s_1
c_1
$\overline{c}_1$
s_1
c_1
$\overline{c}_1$
clk

Type II

Appendix E

Adder Cell Layout



Bibliography

[1] A. Aho, J. Ullman and J. Hopcroft, *The Design and Analysis of Computer Algorithms,* Computer Science Press, pp. 111-113.

[2] R. Armstrong, "HPEDIT Reference Manual", MIT Research Laboratory of Electronics, MIT, 1982.

[3] R. Armstrong, "HPDRAW Reference Manual", MIT Research Laboratory of Electronics, MIT, 1982.

[4] R. Armstrong, "Procedural Design of a high speech Floating Point Arithmetic Unit", S.M. thesis, MIT, Febuary 1985.

[5] D. Baltus, "Design of an Assembler of NMOS Fast Parallel Fractional Multipliers", S.B. thesis, MIT, May 1983.

[6] C. Bamji, "Design by example PLA generator", S.B. thesis, MIT, February 1984.

[7] C. Bamji, C. Hauck and J. Allen, "A Design-by-Example Regular Structure Generator", *ACM IEEE* 22^{nd} *Design Automation Conference*, Las Vegas, Nevada, 1985.

[8] J. Bentley and T. Ottmann, "The complexity of manipulating hierarchically defined sets of rectangles", Carnegie-Mellon University, Computer Science Department, Technical Report CMU-CS-81-109, April 1981.

[9] A. Dunlop, "SLIP: symbolic layout of integrated circuits with compaction", *Computer Aided Design*, Vol.10, No.6, November 1978, pp. 387-391.

[10] F. Ficken, *The Simplex method of linear programming*, Holt, Rinehart and Winston, New-York, 1961.

[11] L. Guibas and J. Saxe, Problem 80-15, *Journal of Algorithms*, Vol.4, 1983, pp. 177-181.

[12] C. Hauck, C. Bamji, J. Allen, "The Systematic Exploration of Pipelined Array Multiplier Design", *ICASSP*, 1985.

[13] K. Hwang, *Computer Arithmetic*, John Wiley and Sons, New York, 1979.

[14] D. Johannsen, "Bristle Blocks: A Silicon Compiler", *ACM IEEE* 10^{th} *Design Automation Conference*, June 1979, pp. 310-313.

[15] G. Kedem and H. Watanabe, "Optimization techniques for IC layout and compaction", Technical Report 117, Computer Science Department, University of Rochester, September 1982.

[16] S. Kirkpatrick, C. Gelatt, Jr. and M. Vecchi, "Optimization by Simulated Annealing", *Science*, V. 220, number 4598, May 1983, pp. 671-680.

[17] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976 , pp. 74.

[18] C. Leiserson, F. Rose and J. Saxe, "Optimizing Synchronous Circuitry by Retiming", *Third Caltech Conference on VLSI*, Pasadena, California, March 1983.

[19] T. Lengauer, "Efficient Algorithms for the Constraint Generation for Integrated Circuit Layout Compaction", *Proceedings of the* 9^{th} *Workshop on Graphtheoretic Concept in Computer Science*, June 1983.

[20] T. Lengauer, "The complexity of compacting hierarchically specified layouts of integrated circuits", *Proceedings of the* 23^{rd} *Annual Symposium on Foundations of Computer Science*, November 1982, pp. 358-368.

[21] B. Liskov, et al., *CLU Reference Manual*, Springer-Verlag, New York, 1981.

[22] M. Matson, "Macromodeling and Optimization of Digital MOS VLSI Circuits", PhD. thesis, MIT, January 1985.

[23] S. McCormick, "EXCL: A Circuit Extractor for IC Designs", *ACM IEEE* 21^{st} *Design Automation Conference*, Albuquerque, New Mexico, 1984, pp. 616-623.

[24] E. McCreight, "Priority Search Trees", Xerox Corporation Palo Alto Research Centers Technical Report, CSL-81-5, January 1982.

[25] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Menlo Park, California, 1980.

[26] J. Ousterhout, G. Hamarchi, R. Mayo, W. Scott and G. Taylor, "Magic: A VLSI Layout System", *ACM IEEE* 21^{st} *Design Automation Conference*, pp. 152-159.

[27] K. Pitman, "The Revised Maclisp Manual", Report TR-295, Laborarory for Computer Science, MIT, June 1983.

[28] S. Sur, "Resizing in Automated VLSI Layout Design" S.M. thesis, MIT, February. 1985.

[29] J. Suskind, J. Southhard and K. Crouch, "Generating Custom High Performance VLSI Designs from Succint Algorithmic Descriptions" *Proceedings Conference on Advanced research in VLSI*, January 1982.

[30] A. Vladimirescu, and S. Liu, "The Simulation of MOS Integrated Circuits Using SPICE2", Electronics Research Lab, University of California Berkely, ERL Memo No. M80-7, February 1980.

[31] W. Wolf, "Two-Dimentional Compaction Strategies", PhD. thesis, Stanford University, 1984.

[32] R. Zippel, "An Expert System for VLSI Design", *IEEE International Symposium on Circuits and Systems*, 1983.