The GENERIC Programming Language Manual[1]

Jon A. Solworth

TR 85-698
September 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

1 This work was supported in part by NSF grant MCS 79-07804, DCR-8503610, DOE grant AC02-76ER03077, and the Cornell Manufacturing and Engineering Productivity Program.

Acknowledgements

Most of the better ideas on programming languages in GENERIC are derived from the programming languages SETL, Lisp, Snobol, and its successor ICON.

I would also like to thank Ralph Grishman for his careful reading (and many suggestions) on an earlier draft of this document, and Hal Perkins for adding significantly to the precision of discussion of programming language features. This work was done in part at New York University as part of a NYU degree under the supervision of Ralph Grishman.

This document represents the language as of September, 1985. Although the language is currently quite stable, this language is not distributed, and is still subject to further improvement.

The GENERIC Programming Language Manual

Jon A. Solworth

Department of Computer Science
405 Upson Hall
Cornell University
Ithaca, N.Y. 14853

(607) 256-4052
(607) 256-7166

Arpanet: solworth@cornell

Abstract

GENERIC is a programming language for the description and manipulation of integrated circuits. GENERIC works on the layout level with the designer in complete control of the layout process. To design an integrated circuit, a program is written which hierarchically describes the chip. The dynamic calling structure of the program determines the integrated circuit's hierarchical cell structure. These cells are created by special procedures called <u>generators</u>. Generators are capable of producing completely custom structures -- they do not consist of predefined layout.

In addition to the specification, GENERIC provides operators for the manipulation of integrated circuit layouts, thus enabling existing geometry to be modified. These modifications can be geometrical, topological or circuit.

GENERIC is a very high level language. The language is general purpose -- the VLSI aspects of the language are layered on top of the basic language as a run-time library. Since the library itself is written in GENERIC, the language is completely extensible.

Chapter 1

Introduction

1.0 Introduction

Integrated circuit design, as traditionally performed, is a costly and time consuming task. In the traditional method, a graphics editor is used to aid in the drawing of the cells that compose the chip. Alternative approaches include the use of silicon compilers or other programming languages which serve as extended declaration languages. In the case of silicon compilers, the designer loses control over the crucial layout process, while the language approach has had limited success because of the weaknesses of the declarative paradigm. In contrast, GENERIC is completely procedural. GENERIC is more than a tool for designing ICs; it is a design methodology for highly automated construction of high quality integrated circuits.

1.1 Abstraction

GENERIC spans three levels of integrated circuit design. The highest level is the electrical circuit, that is, the components and wires which describe the circuit. At this level wires are not assigned to layers, and there is no notion of placement (of wires or components). The electrical circuit level corresponds to switch level network description of a circuit, such as used by RNL or Mossim. These circuits can be simulated (assuming they use Mead/Conway level design methodology), although timing estimates will be only approximate since accurate capacitances and resistances for wires are not known. More sophisticated design techniques such as sense amps, crowbars, and bootstrapping are also handled at this level, but by a different mechanism that we will discuss latter

The second level of design is the topological design. At this level, relative placement is made and wires are assigned to layers and routed. At the electrical circuit level the designer only needed to know what type of circuits are possible (CMOS, nMOS), however at the topological level the designer

must also know what routing layers are available (poly, diff, $metal_1$ $metal_2$), and what the contact structure is between different levels. Switch level simulations with RC timing analysis is possible at this level.

The final level of design is the geometrical level. At the geometric level, absolute distance is determined, and process dependent spacing is handled, as well as fine level manipulations. At this level, Spice (device level) simulations can be performed on the circuit. Furthermore, an integrated circuit mask generation file (CIF) can be generated.

This partitioning of the design process is typical. However, in GENERIC once the circuit design is entered, the design can be refined by operators which which perform correctness preserving transformations. These transformations ensure that the original circuit design is implemented, assuming that reasonable circuit design techniques are used. These transformations are used not only for manipulating the layout but for describing the layout as well.

Since one of the goals of GENERIC is to be able to generate the highest quality layouts, the transformational operators must be both flexible and powerful. In addition to the more standard types of relative placement operators, there are operators for performing topological manipulations of layout. GENERIC is the only language we know of which provides operators for topological manipulation while maintaining design rule correctness. Furthermore, all GENERIC layout operators attempt to maintain connectivity and will ensure that geometric design rules are never violated. We feel that these capabilities are required to automatically generate layouts as dense as those produced by hand.

Three levels of design have been discussed in this section: these levels should not be construed as phases of the design process. Computationally, many problems in VLSI design are intractable (NP-hard). In addition, these NP-hard problems are simplifications of the real problems, and the solution of the simplified problem is not necessarily a viable solution for the real problem. For these reasons we believe that VLSI problems are not amenable to direct algorithmic (graph theoretic) solutions,

although these algorithms must be a part of a complete, and especially a programming based, design solution.

Since a direct, unified solution for general layout is unlikely to be successful due to computational complexity reasons, an alternate synthetic approach is needed. This approach uses general global (graph theoretic) algorithms, special purpose algorithms (eg. recurrence based layouts), local heuristic and/or exhaustive combinatorial search, integration of pieces, and cascading of various methods. To effectively provide this capability it will be necessary to iterate over the layout (not just a graph representation of the layout). For at least parts of the circuit, it will be necessary to go all the way down to geometry, decide to make some changes, and then go back up to the topological level. GENERIC unifies these levels so that, while conceptually these levels are present, they do not limit the range of operations that can be carried out. Hence, all levels may be simultaneously present in a circuit, and the designer can move either up or down in the levels.

1.2 How Generic Works

In this section we will describe the principal objects on each of the three levels of design along with their principal properties. At the electrical circuit level, there are components and connections. Connections define the electrical nets in the circuit; the components are the elements which are responsible for the behavior of the circuit. Circuits which are designed using the Mead/Conway circuit abstraction can be simulated at the electrical circuit level, and the layout developed for the circuit is guaranteed to correctly implement the circuit. More sophisticated circuit design must be dealt with as described below.

The Mead/Conway abstraction provide a powerful basis for designing VLSI circuits; however, the abstraction is based on steady state properties, on voltages which are either at V_{ss} or V_{dd}, and wires which have zero resistance and capacitance (from the point of view of circuit correctness -- not timing). This abstraction covers most of logic design, but does not adequately deal with, for example, sense amps (for which the capacitances of wires are critical to the correct functioning of the circuit),

crowbars (which use as initial states voltages which are midway between V_{cc} and V_{dd}), and bootstrapping (which depends on relative capacitances). However, for the vast majority of logic design the Mead/Conway design abstraction suffices; GENERIC ensures proper generation of layout for these circuits. The non-Mead/Conway portions of the circuit will have to be verified, either through validation before fabrication (static checking), or in the worse case by simulation of the relevant parts of the circuit using transistor models.

Because of their intimate connection, we describe the topological and geometric levels together. A GENERIC layout consists of two types of basic objects: wires and primitives. Primitives are used to describe transistors, contacts, and, in analog circuits, capacitors and are the instantiation (at the topological and geometric levels) for the logical equivalence classes which are denoted by the component names. Primitives represent fixed geometry: a transistor, for instance, is created with a generator, after which its geometry is fixed. If a different transistor is required, the old one must be deleted and a new one created. There are several advantages to viewing primitives in this way. First, having the primitives as fixed entities ensures that the circuit function is not accidentally changed. Second, the generators that produce the circuit can provide accurate parameters, for example of transistor height to width ratios. Third, the design rule checking problem is broken in two: the operators need only be concerned with the spacing between two objects, while the primitive generators are concerned with device construction. Because of the small number and variety of basic primitives they are provided for in built-in primitive generators, implicitly embedded with the device construction design rules. These primitive generators are privileged -- the user may not provide his own. However, since the primitive generators can be fully parameterized, arbitrarily shaped primitives can be created.

Wires are collections of wire segments which are connected together at their endpoints. A wire represents a physical net connecting two or more points and is the physical embodiment of a set of connections. To wire together two points, these points must be connected. However, wiring is an incremental process, and at any given point in the execution of a program two points which are

connected need not be wired. A neccesary condition for fabrication is that all connected points must be wired.

Wires have properties. These properties determine whether a wire is stretchable, shrinkable or slidable. Although the size, shapes and components of a wire are fixed at any given time, these wires may be manipulated (deformed) through the use of operators. These operators enable the user to move objects and collections of objects around in a way that maintains geometric design rule correctness. Operators exist for both topological and geometric manipulations.

Using the two basic objects, integrated circuit layouts can be constructed hierarchically by using cells to group together hierarchically related objects.

In GENERIC, the notions of topological and geometric levels are unified. Hence, the type of objects available at both levels of design are the same, but their style of use and the operators employed differs. Characteristic of topological manipulation are the use of planes (a pseudo 3rd dimension), wiring complexities, and "pick up and move" style operators. Characteristic of geometric manipulations are wires which can almost always be deformed and "sliding"-style operators. These notions should become clearer after reading the chapters on VLSI Concepts and VLSI Operators.

1.3 Implication of Not Using Graphics

The use of graphics is dominant in the design of integrated circuits. Its dominance is partly due to historical reasons, and partly due to some inherent advantages in using graphics. Primary among these advantages is the ability of humans to comprehend quickly information presented in graphical form. Graphics enables the designer to assess the quality of the design (metrics), a means to determine what modifications should be made (iterative improvement), and efficient means of modifying the design (operators). In order for GENERIC to replace graphics, each of these capabilities must be provided in the language.

1.3.1 Compensating for the loss of Graphics

Let us make the analogy of designing integrated circuits with piloting a plane. The use of graphics corresponds to flying a plane by sight. Using programming languages corresponds to flying by instrumentation. Flying by sight has many advantages: it is cheaper, it is easier to learn and it helps in developing intuition about how to fly. On the other hand, it is insufficient for flying a plane in high traffic or bad weather. In the early days of aerospace, there were no instruments and until World War II the instrumentation was not very sophisticated. This is the stage at which we find ourselves now with VLSI design systems.

In order for the designer to accept the possibility of 'flying blind', the instrumentation must be sufficiently powerful, accurate and complete. Our goal is to use the GENERIC programming language as a base for providing this instrumentation, which will replace the three capabilities (metrics, iterative improvement, operators) inherently provided in graphics.

The first issue is measuring the quality of the layout. The notion of quality depends on the design criterion for a particular chip, which is a cost function having as parameters:

- speed
- density
- power consumption
- cost of design

Our goal is to minimize the cost function. The cost is minimized by trading off speed, density, and power consumption, while the cost of the design is measured in time. We shall be primarily interested in indicators of speed, density, and power consumption tradeoffs, which can be used to guide the design process until the improvements possible are not worth the time expended to obtain them. The goal now boils down to determining the metrics for speed, power, and density.

For designing in the large, it is clear that the highest level consideration must be topological. Topological considerations enter in two ways. First, topological planning is needed to develop reasonable initial topology for the design. Second, metrics which evaluate the quality of the design are needed. The scenario for design is then to begin with some automatically determined planar embedding of the IC using standard cells as nodes, and then to evaluate the quality of the embedding. If the quality of the design is not satisfactory, then a more comprehensive exploration of the design space is required. A top-down tool would measure the wiring area and topology and overall density of transistors and wires. The exact topological planning and metrics is a topic for research, and GENERIC will greatly facilitate the effectiveness of this research. Until such metrics are developed, we shall use graphics terminals and plotters to provide the feedback mechanism to evaluate the design. However, feedback differs substantially from the normal use of a graphics interface, in that we are interested in using graphics to *evaluate* the design process, not to *drive* it.

The second issue is how to determine what modifications are to be made. The metrics discussed above are integral part of the process, since they provide relative figures of merit. The metrics will indicate how good the design is, but will be an imperfect measure. If a part of the circuit need be more highly tuned for layout, a number of strategies will be needed to improve the layout. These strategies fit into two broad classifications: iterative (local) improvement and alternative global analysis. The latter would include, for example the use of a strip methodology instead of random layout.

Given the metrics, which will pinpoint which parts of the circuit we wish to improve, operators must be applied to the circuit. Of most interest is automating local operators. This means deciding which specific layout elements should be changed, and deciding how the operators are to be applied. For geometric manipulations, we intend to use both stylized (for example, H tree layout or linear arrangement) as well as general purpose compaction methodologies. We intend to provide a number of local, stylized operators for performing local topological manipulations.

Finally, the operators are based on a small number of basic operators, which can be implemented relatively inexpensively, and which are general enough to serve as the basis for arbitrarily complex operators.

1.3.2 Inherent Advantages of Programming Languages

The previous sections have discussed some of the problems with a programming language approach to integrated circuit design, as well as the ways in which we intend to address these issues. In this section the advantages of a programming language approach are discussed.

The primary advantage of the programming language approach is that the design process can be abstracted, and stylized methods of attack developed. The *process of layout* (execution of GENERIC programs) is inherently captured by the programming semantics. To reuse the process in a different design, the program is simply re-executed. The graphics approach, in contrast, captures only the final result of the layout process, losing the process that created the layout. This is a principal reason why cell libraries have not been more effective: cells have not been tailorable to their specific environments.

A second issue is that of naming. Naming cells or points on an object are inherently non-graphical operations, and require textual input. Clearly, this operation fits in very well with a programming approach.

A third issue is that a large body of tools which have been developed for programming are available for VLSI design, for example, archiving programs, editors, etc.

A major contribution of GENERIC is the increase in the richness of the design data base, combined with a very natural structure. This structure makes it possible to read in a VLSI design, modify it in some relatively small way, and then write it out. The GENERIC language is a natural medium for writing utilities such as compactors, aspect ratio modifiers, and various types of optimizers. The structure which represents the layout is now a reasonable replacement as a

standard, integrated circuit interchange format, drastically reducing the number of conversion programs needed in the alternative CIF based environment. So utilities could be simpler and more powerful then CIF based tools.

Chapter 2

Syntax

2.1 Variable Names

Variable names contain an arbitrary number of characters. The first character must be chosen from the set {'a'-'z', 'A'-'Z', '$'}. Remaining characters must be chosen from the set {'a'-'z', 'A'-'Z', '$', '_', '0'-'9'}. The variable *a* is different from the variable *A*.

2.2 Comments

Comments begin with a double hyphen ('− −') and are terminated by the end of the line. There are no enclosing comments (such as /* .. */) in GENERIC.

2.3 Integers

An integer is a sequence of digits, optionally preceded by a minus sign (−).

2,4 Float

A floating point number consists of 1 or more digits, followed by a decimal point, followed by one or more digits, followed by an optional scaling factor. The scaling factor is an 'e' followed by an optional sign, followed by one or more digits.

```
{0-9} + .{0-9} + [e[-]{0-9} + ]
```

2.5 Strings

Strings literals begin with a single quote and are terminated with a single quote.

2,6 Keywords

The words used in programs in this document which are emboldened are reserved keywords. When typed on a normal crt, these keywords are typed in with all letters in capitals. A list of reserved words are in an appendix to this document.

2.7 Case

The language is case sensitive. That is 'E' is always different from 'e'.

2.8 Printing Format

In addition to the format for reading in a program, the compiler can produce a typesetting input from the program with keywords converted into special characters (FORALL -> ∀); if there is no appropriate special character, the word will be converted to lowercase and emboldened.

Chapter 3

Data Types and Variable Names

3.0 Introduction

In this chapter, we describe the basic data types. GENERIC has declarations which serve both to specify the scope of a variable (global or local) and to provide a type restriction for that variable. Variables need not be declared, except where default scoping rules are not acceptable. Since variables need not be declared, they need not be typed although there are both performance reasons (we can eliminate type determination code) and correctness reasons (compile and run time checks for correct data types) for specifying the set of types a variable may be bound to. Variables which are not restricted in the type that they may hold can be declared (and hence scoped) by using the restriction **UNSPEC** (UNSPECified).

Variables may be given an initial value. If no initial value is specified, the value will be given as **Ω**, which can be typed as **OM**. OM transcends the type system -- no matter what restriction a variable has, it may have the value OM. OM can be tested for using equal or not equal comparisons; other comparison operators are meaningless on this value. In addition, OM can be assigned to a variable. Attempting to perform any other operations will cause an exception.

3.1 Simple Data Types

GENERIC provides the standard data types commonly found in many language. These are integers, floating point numbers (presumably double) and strings. Because GENERIC is a hardware description language, the integers can have a bit-length, although in the initial implementation the bit-length field is ignored. The user can assume that the default bit size is at least 32 bits. GENERIC provides automatic conversion between float and integers.

Strings contain an arbitrary number of characters, i.e., they need not be specified at compile time, and the string assigned to a variable may be of arbitrary size. Operators for accessing strings are similar to those that access tuples, and will be discussed after tuples.

3.2 Compound Data Types

Besides the simple data types, there are two compound data types which have as component types the compound and simple data types. These two compound data types are **TUPLE**s and **PROTOTYPE**s. Tuples are ordered collections of elements and are used like linked lists or arrays in other languages. Prototypes are collections of related data with named parts.

3.2.1 Tuples

A tuple is a finite ordered sequence of values. These values may be restricted by a declaration for the tuple. One way to specify a tuple is with a tuple former, which enumerates the values in the tuple. Syntactically, a tuple former consists of a left bracket, a sequence of values separated by commas, and a right bracket. For example:

```
[]
```

is a tuple former for the empty sequence (i.e. a sequence containing no values). A tuple former containing three elements is shown below:

```
[1, 'd', 'f']
```

Elements of a tuple are indexed starting at 0. #A is the index of the last element of the tuple A, hence #A is the number of elements in the tuple minus one (-1 if the tuple is empty). Since the indexing of elements in a tuple starts at zero, #A + 1 is the length of the tuple.

Elements of tuples can be extracted. Below are the first and last elements of the tuple assigned to A are extracted.

```
A[0]
A[#A]
```

If the index into a tuple A is greater than #A, then the value OM is returned. Assignment to tuple elements can also be made, for example:

```
A := ['x', 'y', 'z']
A[0] := 1
A[#A] := [1, 2, 3]
A[#A + 3] := 'z'
```

The second line changes A to [1, 'y', 'z']. The third line changes the last element of A; A will then be equal to [1, 'x', [1, 2, 3]]. The last assignment causes the tuple to be extended by three elements, making the new last element of the tuple 'z'; the other two elements (#A + 1, #A + 2) have the default value OM, that is [1, 'x', [1, 2, 3], OM, OM, 'z'].

In addition to extracting elements, we can extract tuples. This is sometimes called extracting a subtuple, although of course subtuples are the same as tuples. To extract a subtuple, the initial and ending positions in the base tuple must be specified. The positions specified with a from-element-index to to-element-index notation. If the from-element-index > to-element-index then the subtuple is the empty tuple. For example:

```
A[0..#A]
A[0..0]
A[#A..#A + 2]
```

the first extracts a subsequence, which is in fact the whole sequence. The second extracts a subsequence which consists of one element which is the first element of A, unless A is the empty sequence, in which case the value is [OM]. The second could be written [A(0)]. Finally the third line extracts a subsequence consisting of the last element of A followed by two OMs.

Tuples can be assigned to in a manner analogous to elements. The sequence (subtuple) specified by the left hand side (lhs) is removed, splicing in its stead the sequence on the right hand side (rhs). In particular, if the size of the subsequence being replaced is different than the size of the rhs sequence, then the size of the resulting tuple will change. For example:

```
A := [1,2,3,4,5];
A[1..3] := [9];
A[3..3] := [8, 9, 10];
A[0..0] := [0, 100];
```

If the above statements are executed in order, then the value of A after the second line is [1, 9, 5] and after the third line is [1,9,5, 8,9,10] and after the fourth line is [0, 100, 9, 5, 8, 9, 10].

Two tuples can be joined together by concatenation. For example,

A + B

creates the tuple containing all the elements of A followed by all the elements of B.

Tuples have a mixture of value and pointer semantics. For example, if we execute:

```
A := [1, 2, 3];
B := A;
```

then A and B both point to the same item. If the assignment:

```
B[1] := 'a';
```

is then made, A = [1, 'a', 3]. (Note that := is assignment and = is a logical operation). However, if at some later point

```
A := B + ['x', 'y', 'z'];
```

Then B and A will be separate tuples, and assignment of values to A[i] will not affect B. The semantics (copy or pointer) for each operation type is given below:

Operation	Semantics
Element Assignment	Pointer
Tuple Assignment	Pointer
Tuple Former	Copy
Tuple Extraction	Copy
Concatentation	Copy
Slice Assignment	Copy

3.2.2 Prototypes

A Prototype is a mechanism for creating new types. It is similar to similar to Pascal records [PASCAL]. The new type is a collection of related values which can be accessed by their named parts.

A prototyped variable is a variable which has a value with a prototyped type. For instance, if a prototype PointT is declared which component parts x and y (both of type **float**) and P is declared type PointT, then P.x and P.y are the two component parts of P. If the whole aggregate component is needed, it is referenced by P. P can be used as a parameter to a procedure or returned as a value from a function.

Prototypes are created in one of two ways. If a global or local variable is declared with a type that is a prototype type, the variable is initialized to the prototype. Otherwise, a prototype can be created by a generator (see procedures).

3.3 String Operations

String operations are similar to tuple operations, and in fact use most of the same notation. A string is a sequence of characters. Operationally, the major difference is that strings use copy semantics while tuples sometimes use copy semantics and sometimes use pointer semantics. This means that changing one string variable never changes the value of a different string variable (no side effects). A few strings are written below:

```
''
'abc'
'|19z'
```

The first line is the empty string, the second line is a string of length three and and the third line is a string of length four. Like tuples, the # operator can be applied to strings and represents the index of the last character in the string

Substrings are accessed using the same notation as for tuples, and, as in tuples, the lowest index is zero. However, unlike tuples the element returned by a string which is indexed is itself a string. Thus, A[i] is short hand for A[i..i]. For example

```
A := 'wxyz';
A[0]
A[3]
A[4]
```

After A is assigned the string 'wxyz', A[0] is the string consisting of the single letter w and A[3] is the string containing only the letter z. The final line returns the empty string.

Substrings can be extracted in a manner similar to tuples. Finally, substring assignment works just like tuple subsequence assignment. Substring element assignment is just like sub-tuple assignment. For example:

```
A:= 'wxyz';
A[1]:= 'q';
A[2]:= 'cde';
A[0..#A]:= 'x';
```

If the above statements are executed, then after the second line is executed, A is 'wqyz'; after the second line is executed A is 'wqcdez'; and after the final line is executed A is 'x'.

Strings can be concatenated with the + operator.

3.4 Constructed names

In this chapter names data values have been obtained either through simple names (see the syntax for variable names), through tuple accesses or prototype component access. There is another mechanism for obtaining a value; this is the constructed name.

Anywhere a sequence of characters can be used to specify a name (including in prototype components), a part of the name can be constructed using the percent operator. The syntax is given below: The expression between percent signs is evaluated, and the print value of the expression is

```
[[sequence_of_characters]*[%expr%]*]
```

concatenated with the constant sequnce_of_characters to obtain the name.

Chapter 4

Values and Declarations

4.1 Values

Most of the constructs in GENERIC are expressions. Thus the constructs such as loops and if-then-else can be used as part of an expression. The result of evaluating an expression is a pair <**value, cond**>, where the value has as its type one of the built in data types of GENERIC and cond is either **true** or **false**.

In this manual, we will use the term value to mean a member of one of the types, expression to be a value-condition pair, and cond to be one of two values **true** or **false**. There is no difference between the use of expressions for computational purposes and for conditional purposes. Once the expression is calculated, the conditional part will be extracted for use in, for example, an if clause, or the whole expression used in a computational clause. This means that operations are computed on expressions and not values. The use of these value-cond pairs is similar to that in ICON [ICON].

This approach has several advantages, among them the ability to write expressions such as:

a > b > c

This expression yields the expression <c, **true**> if a>b and b>c. Otherwise it returns <b, **false**> if a < = b, and < c, **false**> if a>b and b< =c. In the case in which <b, **false**> is returned, c is not evaluated. This has important consequences if c is a function call. This cascading of logical operators is called **chaining**

4.2 Declarations

The declaration of global variables is optional, but all local variables and parameters must be declared. Global variables are those which could be seen from any part of the program. Local variables are local to a procedure.

The simplest type of declaration, which gives no type information, is to declare a variable of type **unspec**. An example of such a declaration is given below:

```
unspec a;
unspec x,y,z;
```

In addition to declaring and typing a variable, a local or global variable can be given an initial value. For a global variable, that means the value is initialized once, at load time. A local variable which has an initial value is given that value every time the procedure is entered. Ex.

```
unspec a := 97;
unspec b := 'this is a string';
unspec x := 2, y:= 4;
```

4.2.1 Simple Declarations

The user may want to specify types. The simplest types are integers, floating point numbers and strings. The syntax for each simple type is given below:

```
int var [<<msb#:lsb#>>] [, var [<<msb#:lsb#>>]]*;
float var [, var]*;
string var [, var]*;
```

Example declarations for each are given below:

```
int x, y;
int x<<0:32>>;
```

```
int x<<5:1>>;
float z;
string str1, str2, str3;
```

The second declaration is for an integer of arbitrary size. This integer is a 33 bit twos complemt number, with the high order bit is x<<0>> and the low order bit is x<<32>>. Similarly, the second declaration is for a 5 bit number with the bits numbered in the reverse direction, ie. the high order bit is x<<5>> and the low order bit is x<<1>>. In the initial implementation, bit specifications are ignored. Each of these variables may be given initial values ion the declaration, as with **unspec**.

4.2.2 Complex Declarations

Using the simple structures, we can build up two types of complex objects: tuples and prototypes.

A tuple consist of a sequence of values called elements, and each element is restricted by the type of the tuple. Unless tuples have a declared size, they have an unbounded number of elements. Declarations exist for fixed length tuples, but are not taken advantage of in the current implementation. All other tuples (the only one currently supported) are variable length, as described in the data type sections. The syntax of tuple declarations is as follows where the bold square brackets are not metasymbols, but are actual square brackets to be typed:

```
type [] + var [, var];
```

The number of square brackets indicate the degree of the list. For example, two sets of square brackets indicates a list of lists; three indicates a list of lists of lists.

Some examples are:

```
unspec[] x;
unspec[] y,z;
int[][] a, b;
string[] v;
```

The first declaration creates a one dimensional tuple; each element is of unspecified type. The second declaration declares two one-dimensional arrays. Tuples may have arbitrary dimension. In addition, variables of type **unspec** may be assigned tuples (or tuples of tuples, etc.). The third declaration defines two variables having as a type a two dimensional array of integers. The fourth declaration defines v to be a tuple, each element of which is a string.

The second type of complex declarations is a prototype. Prototypes have named and typed subcomponents. Subcomponents can optionally be assigned initial values. Prototyped variable definitions are created in two steps. The first step is to create a prototype form. The second step is to declare a variable as of that type. For example:

```
prototype PointT :=
    [[float, x := 1.0],
     [float,  y := 9.7],
    [unspec, pt__number]];

PointT p1, p2;
```

The prototype declaration declares a type called PointT. PointT specifies a prototype form with three components named x, y, and pt__number. Both x and y are given initial values in this type, pt__number is given the default from its declared subtype (in this case OM). PointT can then be used to declare two variables p1 and p2.

4.3 Typedef Declarations

As an information hiding technique, types may be declared which are built up of other types. The form for this declaration is the keyword **typedef**, followed by the new name, and finally the type. Below are some examples:

```
typedef BOOLEAN int<<0>>;
typedef t2 unspec[1];
typedef triple__of__ints int[2];
```

The first line declares the type BOOLEAN to be an integer having one bit. The second declaration is for a pair of values of arbitrary type. Finally, the third declaration for triple__of__ints

is for an tuple having exactly three integers. After the typedefs are given the following declarations can be used:

```
t2 a,b;
BOOLEAN x;
triple__of__ints c;
```

The result of these declarations is that a and b have type **unspec**[2], x has type **int**< <0> >, and c has type **int**[3].

4.4 Predef Declarations

All operators must be declared before they are used. Since operator syntax is different from procedure syntax, operator syntax must be defined before the operator is used. The syntax can be defined in one of two ways; either the whole operator can be defined or the operator can be declared. This enables the language to support separate compilation. The way of declaring this is with a **predef** (for pre-defined) declaration. The form for this declaration is:

```
predef binary__op opname;
predef unary__op opname;
```

for example:

```
predef binary__op mod;
predef unary__op sign;
```

Chapter 5

Expressions, Statements and Procedures

5.1 Scopes & Procedures

The language contains both procedures and operators. The only difference between operators and procedures is the way that they are called. Operator are infix, and are either binary or unary. All procedures are declared at the top level (there are no nested procedures). Procedures have a type restriction which specifies the type of the value returned. This type can be any of the standard types or the type **void** which means that the procedure does not return a value. A procedure has a fixed number of parameters, and each parameter has a type restriction.

Procedures can be called either by positional parameters or by keyword parameters. It is possible to call a procedure with fewer actual parameters then formal parameters. In this case, there is a default mechanism which provides values for missing parameters. The default mechanism works as follows: in the procedure definition, a **default** can be listed for each (or any) parameter. Furthermore, the defaults need not be listed in the same order as parameters in the procedure parameter list. When the procedure is called, the defaults for each unspecified parameter are evaluated in the order that the defaults are listed (not by the order of the parameters). If an unspecified parameter has no default it is given a value of OM. It is undefined in GENERIC whether variables which have the value of **OM** will be treated as unspecified parameters.

Below are the declarations for procedures

```
type
procname(type1 param1, type2 param2, . . . , typen paramn )
    { default parami : = expri; }
    decls
    stmts
end;

type is one of the data types or void
```

Operators are declared similarly, the only difference is that there is an extra token specifying what type of operator (binary or unary). The operands to the operator are not declared explicitly. However, additional parameters may be declared. See the declaration below:

```
type
opr__decl
procname(type1 param1, type2 param2, . . . , typen paramn)
    { default parami : = expri; }
    decls
    stmts
end;

where opr__decl is one of BINARY__OP, UNARY__OP
type is one of the data types or void
```

In what follows, everything said about procedures applies equally to operators.

Declarations inside procedures are the same as declarations outside of procedures, with the addition that declarations can be prefixed with the keyword **local** or **global** (with the obvious meanings).

5.1.1 Non-Prototyped Procedures

If the type returned by the procedure is not a prototype (or **typedef**ed to a prototype), then the procedure semantics are as follows. Parameters are bound as in assignment, either by value for simple expressions, or by pointer for complex expressions. Parameters which are unbound are given values by the default setting for the variable, if present. It is undefined whether defaults are applied when the parameter has the value OM

Variables appearing in the procedure can be either local, global, or parameter names. Variable names are by local if they are declared within the scope of the procedure (or are parameter names); otherwise they are global. Values returned by a procedure must be coercible into the result type of the procedure.

5.1.2 Prototyped Procedures (Generators)

If the result type of a procedure is a **prototype** (or **typedef**ed to a prototype), the procedure has special semantics and is called a generator. Generators use the same parameter passing mechanisms as procedures.

Generators differ substantially from non-prototyped procedures in other ways. First, all of the component names of the declared prototype are available in the generator as if they were local variables. In addition, variable names that are evaluated during the course of an invocation of the generator, but which are not declared in the generator are added as components to the prototyped value being built. Thus, the resulting prototype value, which is returned as the value of the generator, contains as components all those names declared in the prototype declaration plus those variables which are encountered during the invocation of the generator.

Return statements for generators do not need to specify the returned value. The return will provide the prototyped value it has implicitly built up. The name for this prototype value is ____self____.

In addition to ____self____, there is another predefined variable name called ____genv____, which is available in all procedures. At any given time in the execution of the program, there is a series of procedures which are active; these processes have their variables on the stack. ____genv____ is the structure which is in the process of being constructed by the last active generator, or is the value __top__level__ if there is no previous generator. To get the generator two steps away, the value ____genv____ . ____genv____ is used. **NB**: if ____genv____ is used in a generator it refers not to ____self____ but to the structure of the generator before ____self____.

In order for variable names to not become part of the prototyped value, they must be declared as a local variable, a global variable (within the generator) or a parameter.

5.2 Expressions

An expression yields a pair <**value, cond**> when evaluated, where value is a member of one of the data types and cond is either **true** or **false**. In this section all of the expressions in GENERIC are described.

5.2.1 Variable Expressions

Each variable has a value which is one of the data types that can be defined in the language. When a variable with value **val** is used, it evaluates to <val, **true**>.

5.2.2 Logical Operators

The logical operators in GENERIC are **and** (&), **or** (|) and **not** (!).

and_expression	::=	$expr_1$ & $expr_2$
or_expression	::=	$expr_1$ | $expr_2$
not_expression	::=	! $expr_1$

The **not** operator takes expr which is a pair <val, cond> and returns a pair <val, cond'> where cond' is **false** if cond is **true** and cond' is **true** if cond is **false**.

The **and** of two expressions ($expr_1$ and $expr_2$) evaluates as shown below. Let $expr_i$ be rewritten to <val_i, $cond_i$> pairs. If $cond_1$ is **false** then $expr_2$ is never evaluated. This has important consequences if $expr_2$ has side effects.

and evaluates to:	
<val_1, **false**>	if $cond_1$ is **false**
<val_2, $cond_2$>	if $cond_1$ is **true**

The **or** operator evaluation is shown in the box below. If val_1 is **true**, then $expr_2$ is not evaluated.

or evaluates to:	
<val_1, **true**>	if $cond_1$ is **true**
<val_2,$cond_2$>	if $cond_1$ is **false**

5.2.3 Arithmetic Expressions

GENERIC supports the standard operations of addition(+), subtraction(-), division(/) and multiplication(*) on integer and floating point numbers. The evaluation is performed as follows: where the op is the corresponding operator applied to the values. If one of the operand's val is a floating point number, and both cond are **true**, the resulting value will be a floating point value. In

```
if cond1 is false then
     the resulting expression is exp1
elseif cond2 is false then
    the resulting expression is expr2
else
    the resulting expresion <val1 op val2, true>
```

general, coercions are applied when required by the operator or by the type restrictions for a variable. If $cond_1$ is **false**, then $expr_2$ is not evaluated.

5.2.4 Assignment Operator

Assignment (: =) is an operator in GENERIC. Assignment binds the lhs name to the value part of the evaluated expression. The result of the assignment is the expression which the rhs evaluates to. Additionally, C style computed assignments can be used with the arithmetic operators (+ : =, − : =, *: =, /: =). The effect of a computed assignment is that the lhs is evaluated, the operation is

```
assignment-operator              ::=    lhs      := rhs
assignment-compute-operator      ::=    lhs op := rhs

    where op is one of +, -, *, /
```

applied to the evaluated lhs and the rhs, and the result is bound to the left hand side. The left hand side is evaluated only once, which means that in the expression

A[i + : = 1] *: = 7,

the value of i is incremented only once. The expression returned by a completed assignment is the expression that is the result of performing the operation.

5.2.5 Procedure Calls

Procedure can be called either by positional parameters or by keyword parameters. A procedure call can also have fewer actual than formal parameters, even when positional notation is used. Furthermore, positional and keyword parameters can be used in the same call; in this case, the positional parameters must come before the keyword parameters. The syntax for a procedure call is:

procedure-call ::= proc(param-list)

param-list ::= [formal__param$_1$:=] expr$_1$, [formal__param$_2$:=] expr$_2$, . . . [formal__param$_n$:=] expr$_n$

where n is less than or equal to the number of formal variables in the procedure definition. Furthermore, if positional formal__parameter$_i$ is present then for all $j<i$ positional formal__parameter$_j$ must be present. The result of a procedure call is the expression returned by that procedure.

5.2.6 Operator Calls

Operators are a form of procedure which takes in addition to possible arguments one or two operands. The use of a binary and unary operators are given below.

binary-operator-call ::= operand$_1$ bop|{param-list}| operand$_2$
unary-operator-call ::= uop|{param-list}| operand$_1$

Operator calls can be combined. The syntax is as follows:

bin-operator-conjunction ::= operand$_1$ op$_a$&op$_b$&op$_c$...&op$_z$ operand$_2$
bin-operator-disjunction ::= operand$_1$ op$_a$|op$_b$|op$_c$...|op$_z$ operand$_2$
un-operator-conjunction ::= op$_a$&op$_b$&op$_c$...&op$_z$ operand$_1$
un-operator-disjunction ::= op$_a$|op$_b$|op$_c$...|op$_z$ operand$_1$

Operator conjunction conjunction (&) is equivalent to the below expansion, where the & is now the logical **and** which has been previously defined.

bound-expr-t1 ← operand1
bound-expr-t2 ← operand$_2$
(bound-expr-t1 op$_a$ bound-expr-t2) &
(bound-expr-t1 op$_b$ bound-expr-t2) &
. . .
(bound-expr-t1 op$_z$ bound-expr-t2)

Where ← means that <expr, val> pair is bound to the lhs. **NB:** ← is not an operator of the language, it is just a notation used in this manual.

The disjunction is the same as the conjunction, however, the &s are replaced by |s

5.2.7 Conditionals

The operations >, <, >=, <=, =, != are defined on integers, floats and strings. These data types all have value semantics so that the comparison is on the values. All the standard operations are possible, in addition to the chaining of operations.

In addition, the operations = and != are defined on tuples and prototyped values. Two complex objects are equal if and only if

1. They are both tuples or both derived from the same prototype
2. they have the same length and
3. for each corresponding pair of elements, they have the same value if they are a simple type, or are identical (point to the same thing) if they are a complex type.

5.2.8 If Statement

The **if** statement is a selector operator. If the $cond_i$ part of $expr_i$ is equal to **true** and $cond_j$ is **false** for all $j < i$, then $stmt\text{-}list_i$ is executed. If all $expr_i$ evaluate to **false** then $stmt\text{-}list_{n+1}$ is executed. If $stmt\text{-}list_i$ is executed, then the value of the **if** is the value of $stmt\text{-}list_i$. Finally, if all $expr_i$ evaluate to **false** and there is no else then the value of the **if** is <OM, **true**>.

```
if expr_1 then
    stmt-list_1
[elseif expr_2 then
    stmt-list_2
 . . .
[elseif expr_n then
    stmt-list_n]]
[ else
    stmt-list_n+1]
end
```

5.2.9 Existentials

Existentials are implicit iterators which sequence through a tuple to find the first element that satisfies (in the case of notexists, does not satisfy) the predicate *st-expr*. For **exists**, the expression evaluates to <elmt, **true**> where elmt is the first element in list for which *st-expr* has a **true** cond when evaluated, and <OM, **false**> otherwise. For **notexisits**, <elmt, **false**> is the result of the evaluation of the expr for the first element in list which satisfies the conditional, and <OM,**true**> otherwise.

> **exists** *var* **in** *list* **st** *st-expr*
>
> **notexists** *var* **in** *list* **st** *st-expr*

5.2.10 Loops

There are three different loop forms in GENERIC. The **for** loop is similar to for loops in Pascal, and is used for iterating through a numeric sequence. **Forall** is a loop for iterating through each element of a tuple. Each of these loops allows an optional "such that" clause, which when present causes the body of the loop to be executed only when the *st-expr* evaluates with a cond of **true**. The third form is a general, all purpose loop structures, which have as special subcases **while** and **repeat** loops.

5.2.10.1 For loops

The **for** loop names a variable which must be local to the procedure enclosing the loop. If the variable is not declared local, it will implicitly be declared local; however, in the case of implicit declaration, var must not be mentioned before it is used the first time as a loop variable. The expressions *from* and *to* delimit the beginning and end of the iteration. These two expressions, in addition to the optional *next* are evaluated once, on entrance to the loop.

Var is initialized to *from*. The increment is calculated based on whether *next* is specified. If *next* is present, then the increment is (*next* – *from*); this number may, of course, be negative. If *next* is omitted, then increment is 1 if $to \geq from$, and -1 otherwise. The comparison for loop termination is based on the increment. That is, if the increment is positive, then the terminating condition is $var > to$, otherwise, it is $var < to$. Note that it is possible to write infinite loops using the **for** construct.

The body of the loop is executed for each value of *var* for which the *st-expr* evaluates with a cond of **true**.

The result of evaluating the for loop expression is the last statement evaluated in the loops statement list. If the statement list is never executed, the value returned is <OM, **true**>.

```
for (var := from [, next] ..to [st st-expr])
    stmt-list
end
```

5.2.10.2 Forall loops

The **forall** loop sequences through all the elements of a list, binding *var* sequentially to the first through last element. The body of the loop is executed once for each element which satisfies the *st-expr*. As in the **for** loop, the list is evaluated once at the start of the loop, the value of the loop is the same as in the case of the **for** loop, and the termination value of *var* is the last member of the list (unless terminated early by **break** or **return**).

```
forall (var := list [st st-expr])
    stmt-list
end
```

5.2.10.3 General loops

The general loop construct, which begins with the keyword **loop**, is a generalization on standard loop constructs, and is similar to the loop structure used in SETL [SETL]. The **init** keyword specifies those actions to execute once at the start of the loop. The next keyword, **doing**, is used to specify statements to be executed at the beginning of every loop iteration. **While** is the conditional test at the start of the loop. The loop will terminate if while evaluates to a cond which is **false** The **do** part are the statements forming the body of the loop. If the **while** condition evaluates to **true**, the body of the loop is executed next. At the end of each iteration of the loop , the **step** statements are executed. Finally, if the **until** condition is present and evaluates to **true**, the loop terminates. Otherwise execution resumes at the **doing** statement (or if not present the first of **while** or **do** which is present).

5.2.11 Type

```
loop
    [init expr]
    [doing stmts]
    [while expr]
    [step stmts]
    [until expr]
    do
        stmts
    end
```

The datatype of any expression can be tested. The built-in function **type** evaluates to a string which is the datatype of its argument. The the resulting expression's value is either 'int', 'float', 'str', 'tuple' or a prototype name.

```
type(expr)
```

5.2.12 Names

The names of each member of a prototype can be determined at runtime. The built-in function **names** returns a tuple of strings, each string is the print value of one of the member names. Using the percent operator, these strings can be used to access the prototype. The **names** call is shown below:

```
names(expr)
```

For example if a structure P was created with the the fields x and y, then names(p) would be ['x', 'y'].

5.2.13 Printing

There are two print statements in GENERIC, **print** which produces an unformated print, but can print recursive structures, and **printf** which is a formatted print which is of the same form as in the standard library function in C [C].

5.2.13.1 Printf

Printf produces a formatted print as in the standard library function of C. The arguments must be of the type implied by the format or an exception occurs. The form of printf is given below:

printf(format, $expr_1$, ... $expr_n$)

5.2.13.2 Print

There is a print statement in GENERIC which accepts an arbitrary number of arguments and prints them using a method similar to that of SPITBOL. Print takes expression which can contain arbitrary data structures (including recursive ones) and prints their values.

print($expr_1$, ... $expr_n$)

The algorithm for printing out the print list consists of printing on one line, if possible, the print value for each expression, separated by spaces. The print value for an integer is just the digits needed to print the integer preceded by an optional sign. The print value for a string is the value of the string without surrounding quotes. The print value for a floating point number is at least one digit, a decimal point, and then five significant digits. If this floating point number would have more than x digits in front of the decimal point or if the integer part is zero, and there are at least y digits of zero immediately following the decimal point, then the number is printed in scientific notation.

Tuples and prototyped variables are recursive structures and hence cannot be directly printed (lest we run the risk of infinite print loops). Each tuple or prototype is printed in the print line, or pointed to by some element of the print line is printed exactly once. To do this, the **print** command keeps track of all complex values encountered and prints out each element exactly once in the order it was encountered using a breadth first search. Hence, two values list the same indirect print index (see below) iff they point to the same element.

Tuples print as #n, where the nth structure printed is the tuple. Prototyped variables print as prototype-name#n, where prototype-name is the type of the variable.

Printing the nth structure when it is a tuple, is shown below depending on the size of the tuple. If the tuple will fit on a line, the first version will be printed, otherwise the second version will be printed.

```
#n := [print-value1, print-value2, . . . , print-valuen]

#n := [print-value1,
        print-value2,
        . . .
        print-valuen]
```

When prototyped structures are printed, the prototype field name is printed before the value for ease of reading.

```
prototype-name#n :=  [field1 -> print-value1, . . . . , fieldn -> print-valuen]

prototype-name#n := [field1 -> print-value1,
        field2 -> print-value2,
        . . .
        fieldn -> print-valuen]
```

5.2.14 Operator Precedence

<table>
<tr><th>Type name</th><th>Precedence</th><th>Associativity</th></tr>
<tr><td>EXISTS, NOTEXISTS</td><td>1</td><td>left</td></tr>
<tr><td>: = , op: =</td><td>2</td><td>right</td></tr>
<tr><td>ATTEMPT</td><td>3</td><td>left</td></tr>
<tr><td>binop</td><td>4</td><td>right</td></tr>
<tr><td>unop</td><td>5</td><td>right</td></tr>
<tr><td>|</td><td>6</td><td>left</td></tr>
<tr><td>&</td><td>7</td><td>left</td></tr>
<tr><td>!</td><td>8</td><td>right</td></tr>
<tr><td><, >, =, ! =, < =, > =</td><td>9</td><td>right</td></tr>
<tr><td>+, -</td><td>10</td><td>left</td></tr>
<tr><td>*, /</td><td>11</td><td>left</td></tr>
<tr><td>#, unary -</td><td>12</td><td>right</td></tr>
</table>

5.3.0 Statements

Statements in GENERIC are terminated by semicolons. Wherever a statement can be used, a sequence of statements can be used. Hence, the terms statements and statement lists are used for the most part interchangeably. There are two types of statements: expressions terminated by semicolons and control flow statements. The control flow statements are the **return, freturn, break and continue.**

5.3.1 Expression Statement

A statement which is formed from an expression has as its result the value of the expression. If this has a cond of **false**, then a procedure return is invoked. The value returned is the value of the expression, which means that it is, in effect, a failure return (see failure returns). In this case the statement has no value (a control transfer has occurred). The rational behind this mechanism is that in performing complex operations, the failure of a low level operation should "bubble up" to the appropriate level to handle it.

5.3.2 Control flow statements

Control flow statements do not evaluate to a value since they form unconditional control transfers. Control flow statements differ from expression statements in that they have an optional number of parameters and/or are context dependent. **Return**, **freturn** and **break** all have optional parameters, while **break** and **continue** can only be used in a loop.

5.3.2.1 Continue and Break

A **continue** or **break** may be placed in the body of any loop (**for**, **forall** or **loop** construct). A **break** causes the innermost loop enclosing the **break** to end, and execution continues after the **end** statement of that loop construct. The value returned for the loop is the value of the **break**, or <OM, **true**> if the **break** had no value.

Continue causes the current iteration of the loop to end, and causes control to transfer to the code which computes the next loop iteration. In the case of the loop construct, control transfers to the evaluation of the step part. For the **for** and **forall** loops, the next value is computed and execution begins at the top of the loop.

```
continue;
break [expr];
```

5.3.2.2 Return and Freturn

Return and **freturn** are statements for exiting form a procedure before the last statement of the procedure (or generator) is reached. (after the last statement of the procedure is evaluated, an implicit return is provided).

```
return [expr];
freturn [expr];
```

If the **return** is in a procedure, the value returned is the expr, or if none is given, the value <OM, **true**>, If it is an **freturn**, then the value returned is the <val, **false**>, or if none is given, the value <OM, **false**>,

If the **return** is in a generator and no value is specified, the expression <____self____, **true**> is returned. If it is an **freturn,** and no value is specified, the expression <____self____, **false**> is returned.

Chapter 6

VLSI Concepts

6.1 Introduction

The basic VLSI concepts which underlie the GENERIC library are described here. Together with the introduction and operators chapters, these concepts form the heart of the VLSI design methodology and mechanisms of GENERIC.

6.2 Geometric Design Rules

GENERIC layouts are guaranteed to satisfy the geometric design rules. The method for guaranteeing these rules consists of several parts. Firstly, the devices used in GENERIC are generated by systems supplied generators (of type PrimitiveT). The user is not allowed to provide his own primitive generators. Hence, the correctness of devices (also called primitives) can be guaranteed by hand verification of the rather small primitive generator code. This is the only way to create devices, for example it is impossible in GENERIC for the user to create a transistor by crossing polysilicon and diffusion wires.

The second part consists of providing spacing rules to ensure that two wires, a wire and a primitive, or two primitives are not too close together. Since devices and wires are created in a way that ensures that spacing rules are not violated, we need only ensure that transformations do not introduce spacing errors. For each technology, a table is given which defines minimum spacing between layers. This table is used by the basic layout operators in GENERIC to ensure that transformations made do not introduce design rule errors. Since all operators are written in terms of basic operators, and since the basic operators are part of the runtime library of GENERIC, it is sufficient to ensure that the basic operators make proper use of the spacing rules.

The only system defined, technology-dependent parts of the system are the primitive generators and spacing operators, which together total less than a thousand lines of GENERIC code!

Technological changes from nMOS to HMOS or lambda changes require linking against the revised library and re-executing the code. In addition to system supplied primitives, there is a collection of cells which the user can supply, which presumably would need to be duplicated in the radically different technology as, for example, CMOS vs. nMOS. The rest of the system is technology independent.

As we shall see later, our extended design rules are quite easy to implement and conceptualize. The spacing rules use a table which includes the layer, potential, and spacing for each combination. Thus, if two polysilicon wires (for example) have the same net, they can have arbitrary intersections without causing design rule violations. Equipotential regions described later in this chapter provide this information.

The third part consists of loosening up the design rules enough to allow manipulation of layouts. In this chapter we shall explain how the design rules are extended to provide the ability to perform arbitrary manipulations on circuits.

6.3 Equipotential Regions (Equinet)

Since a GENERIC layout is derived from a circuit specification, the information about what electrical signal is represented by a piece of geometry is always available. This information is needed by the transformation operators for design rule computations

The availability of information on equipotential regions is a powerful addition to intermediate stages in the design. At any stage in the layout, it is possible to determine what needs to be connected, or the total length of wire which is yet to be connected in a net — even if the net is currently in one or more disjoint sections. Another important effect is that it is easier to look at critical paths.

The system which we shall build using GENERIC will rely on this mechanism both for the creation of metrics and as an estimator and planning mechanism. Development of metrics based on the average or maximum wire length are possible.

6.4 Pins

The basic layout primitives of wire segments and primitives must be amalgamated together in order to achieve a complete layout. Since GENERIC allows operators to arbitrarily change a given layout, we must define the relation (connectedness) of wire segments and primitives. Pins are the mechanism which implements this relationship.

Two or more objects can be pinned together. This means that moving or manipulating one object of the pinned set (possibly) involves moving every other item in the pinned set. Objects outside the pinned set are not connected to objects inside the pinned set. Objects in pinned sets are either endpoint of wires (PointT) or geometry in primitives (GeomT).

Wires consists of wire segments that are pinned together.

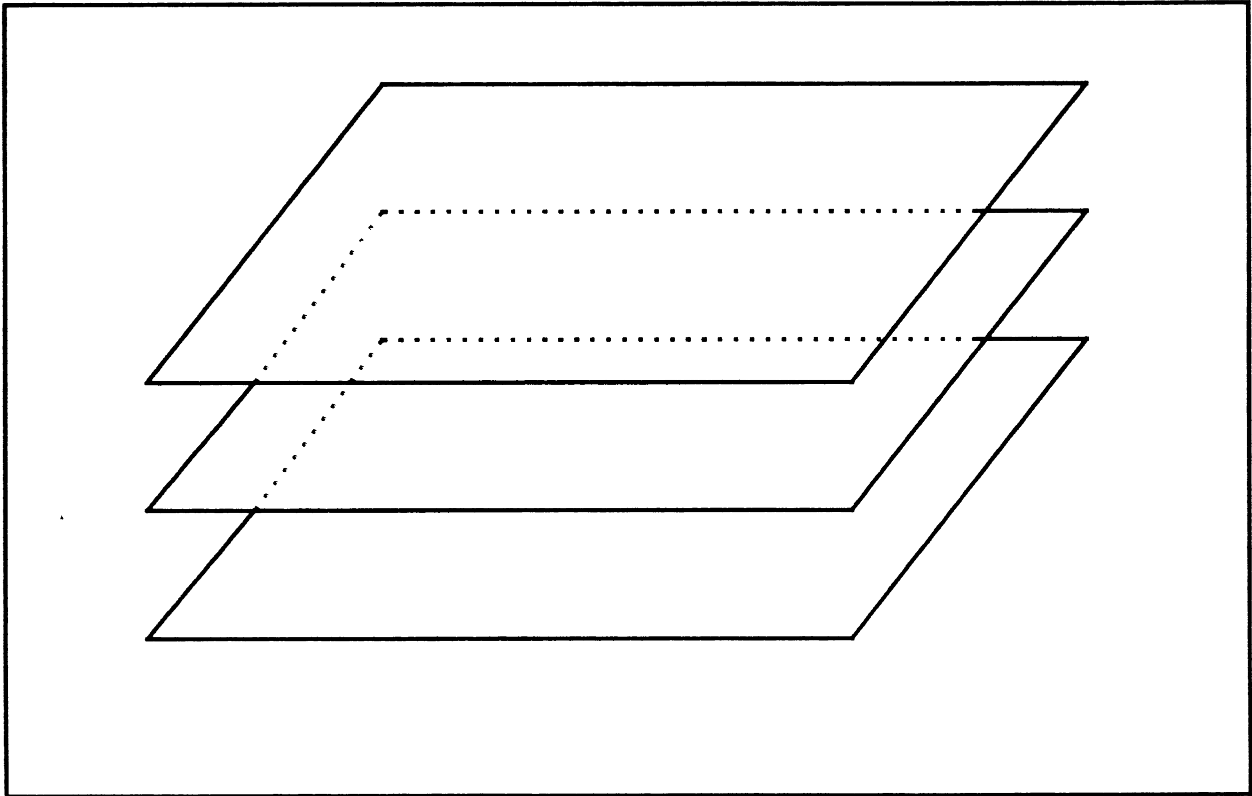
In each pin set, there is at most one element of type GeomT. Hence, two different primitives cannot be pinned together. Instead, a primitive may be pinned to a wire segment which is pinned to another primitive. If the primitives should abut, the wire segment may be of zero length.

6.5 Planes

Planes are part of GENERIC's extended geometric design rules. In the physical integrated circuit, all layers have the potential for interacting with each other. In the case of nMOS, an intersection of polysilicon and diffusion layers causes a transistor to be formed. Other layers interact with each other through contact cuts or ion implants. When restricted to the physical plane, many design-rule constraints inhibit experimentation with different layout variations.

To alleviate these problems, the concept of planes is introduced. Physical integrated circuits are implemented on a single plane on which multiple layers are defined. Planes add a third dimension to layout, allowing components to share x,y coordinates without causing design rule errors. Thus, polysilicon on plane 1 does not create a transistor with diffusion on plane 2

Arbitrary topological manipulation can now be performed by moving geometry to a second plane, moving it relative to the objects on the first plane, and then, if room permits, dropping it back to the first plane.



This mechanism also has an efficiency advantage of reducing the amount of geometry that must be checked for design rule correctness to that geometry which sits in the same plane as the object being moved.

Chapter 7

VLSI Structures

7.0 Introduction

In this chapter, the major VLSI data structures are discussed. These prototypes for these structures are defined in the file 'struct.h' in the run-time library directory. This chapter may be skipped on the first reading.

7.1 Basic VLSI Components

The basic VLSI components are primitives and wire segments. Cells and wires can be constructed from these components. Points and geometry are used to construct primitives and wire segments, . The data structures in GENERIC only support cells constructed using manhattan geometry.

7.1.1 PrimitiveT

Primitives contain the components in a VLSI layout. For example, in nMOS there are primitives for enhancement mode transistor, depletion mode transistors, contact over transistor (for pull-ups), poly-to-diffusion buried contact, etc. All of the geometry that will be part of the integrated circuit fabrication masks is defined in either the primitives or the wire segments.

The primitive prototype definition is shown in figure 1.

A primitive contains the geometry ($geom) needed for a component to be realized in the integrated circuit fabrication process. GENERIC supports only orthogonal rectangles. Hence, the geometry consists of a tuple of rectangles each with an associated layer. During the design process, linear transformations are applied to the geometry indirectly through the operators; these transformations can translate, rotate or mirror, but must maintain the size and shape of the primitive.

$plane is the number of the plane on which the geometry of the primitive is located.

```
prototype PrimitiveT : =
    [
        [str,           $design__name],
        [str,           $logical__equiv__class],
        [BOOLEAN,       $marked : = 0],
        [int,           $flags : = 0],
        [int,           $plane],
        [CellT,         $parent],
        [GeomT[],       $geom : = []],
        [ConnT[],       $conn__region : = []],
        [GdrT[],        $bumpers : = []],
        [unspec[],      $except__rules : = []],
        [int,           $pin__cnt],
        [PointT[],      $points]];
```

Each primitive is part of some cell. To easily traverse the design tree, each primitive points to its parent cell ($parent) and each cell points to its component primitives and cells.

A circuit consists of wires and primitives connected together. Primitives cannot be directly connected; instead, two primitives must be logically connected together using a wire, even if that wire has zero length. The connection region ($conn__region) describes where wires can be attached to the geometry.

Given the points at which wires can be connected, the limitations on routing wires in the vicinity of the primitive must be specified. The wiring constraints within the region of the primitive are described in terms of the $bumpers.

Since a primitive describes some rigid geometry, every time the primitive is moved, the connecting circuitry must be moved (wires attached to the primitive). The wire segment end-points attached to the primitive and the primitive are said to be pinned together, and have the same pin number ($pin__cnt). Given the pinned number, the other objects connected to it can be looked up in the pin table.

There are a number of named locations on the primitive called $points. Members of $points must be transformed when the primitive is moved and form useful names for physical locations as well as node names.

There are some miscellaneous fields in the prototype: $design__name describes a name representative of the particular design, while the logical equivalence class ($logical__equiv__class) corresponds to the logical function performed by the circuit, e. x., XOR-3 (3 input exclusive or). $marked and $flags are internally used flags for intermediate computations, and are not relevant to the user.

7.1.2 WireSegT

Wires are composed of wire segments. A wire consist of a number of wire segments, with the wire segments being contiguous. A wire W consists of the transitive closure (in terms of wire segments pinned to it) of any wire segment in W. The wire segments in W need not form a straight line; they can form any connected graph. All segments in W have the same electrical net ($equinet), some segments not in W will also have the same electrical net; these will have to be wired together before the chip can be fabricated.

```
prototype WireSegT : =
    [
        [int,           $plane],
        [str,           $layer],
        [PointT,        $p1],
        [PointT,        $p2],
        [unspec,        $equinet],
        [float,         $width],
        [int,           $flags : = 0],
        [BOOLEAN,       $manip__flags : = -1],
        [BOOLEAN,       $marked : = 0],
        [wire__USER,    $user]];
```

$plane is the current plane that the wire segment is on. $layer is the IC process layer or derived process layer on which the wire is to be fabricated. An example of a derived process layer is the channel of an nMOS transistor, which is derived from poly crossing over diffusion. The layer must be a valid wire layer in the technology or one of the extended wire names. $p1 and $p2 are the end-points which define the centerline of the wire segment. $equinet is the list of the geometry with the

same net number. $width is the width of the wire and must be greater than the minimum wire width for that layer.

Wires are deformed in a number of ways by the operators. To limit the types of operations that can be performed, the user may set (and unset) various bits in the manipulation flags. The list of flags, using their symbolic name is given below.

```
STRETCH_MANIP
SHRINK_MANIP
BREAK_MANIP
VIRTUAL_MANIP
SLIDE_MANIP
STAIR_MANIP
KINK_MANIP
```

7.2 Higher Level VLSI Structures

There are two higher level structures that are constructible: cells and wires. Cells are a hierarchical structuring mechanism for the components of a circuit (i.e. primitives). Cells contain both primitives and other cells.

Wires are a graph structure which which are superimposed on top of the elements of the cell tree. The properties of wires are described in the section on wire segments. There is no explicit structure which defines a wire. instead. a wire is defined indirectly through its pins.

7.2.1 CellT

Cells are the hierarchical structuring element for the layout. Cells contain as components other cells and primitives. Wires form a graph which is external to the cell hierarchy. However, for some purposes, we shall say that wires which are incident only on a given cell (including its descendents and any primitives it contains) are part of that cell. This is true of the operators which we will consider later.

The cell prototype definition is shown in figure 4.

```
prototype CellT : =
    [
        [str,                 $design__name],
        [ioT[],               $io : = []],
        [str,                 $logical__equiv__class],
        [BOOLEAN,             $marked : = 0],
        [int,                 $flags],
        ---------------------------------------------------------------
        --  circuit specification
        ---------------------------------------------------------------
        [CellT,               $parent],
        [connection__map,$connections : = []],
        [CellT[],             $sub__cells : = []],
        [PrimitiveT[],        $prims : = []],
        [PointCellT[],        $points : = []],
        [CELL__USER__TYPE,  cell__USER]];
```

A cell is part of a cell hierarchy. Each cell points to its parent cell ($parent) and to its component cells ($sub__cells). In addition, cells may contain primitives ($prims). There is no requirement in GENERIC that primitives only be defined in leaf cells.

$io is the names of the I/O points for this cell.

Points can be defined as part of a cell ($points). These points are logical points which point to physical points, incident either on primitives or wires. Since points are named, these points propagate names , allowing point names to reflect the level they are used on, and allowing the points to refer to larger objects (cells) which are related to the physical point

Like primitives, cells have fields for specifying the catalog entry of this design ($design__name), the logical equivalence class to which this cell is a member of ($logical__equiv__class) and flags which are used by the runtime library ($flags and $marked).

7.3 Subcomponents

In this section we describe the main ingredients for the basic components. These ingredients are two types of points (PointT and PointCellT) and the geometry used to form the primitives (GeomT).

7.3.1 GeomT

The geometry that composes the primitives are specified by elements of type GeomT. See figure 5.

```
prototype GeomT : =
    [
        [str,           $layer],
        [int,           $equi__raw],
        [unspec,        $equinet],
        [PrimitiveT,    $object],
        [float,         $x1],
        [float,         $y1],
        [float,         $x2],
        [float,         $y2],
        [int,           $pin__cnt],
        [PointT,        $points : = []]];
```

GeomT typed variables specifies a rectangle. The physical location of the rectangle is specified by two *x* locations ($x1 and $x2) and two *y* locations ($y1 and $y2). $layer specifies the layer that the geometry is on. Each piece of geometry is part of some Primitive ($object).

The electrical properties of the primitive are specified to the extent of indicating which geometry has inherently the same equipotential ($equi__raw). For example, in a polysilicon to diffusion contact, both the poly silicon and the diffusion have the same raw potential. When primitives are wired together, the potential of two connected pieces of geometry will have the same potential ($equinet). Both pieces of information are kept around so that circuits can be connected and disconnected, without having to rebuild the whole circuit from scratch.

$pin__cnt is used to indicate the pin number, so that connected wires can be [illegible]. Objects with the same pin count must have the same electrical potential.

7.3.2 PointT

The type PointT defines a point which is anchored to a GeomT or a WireSegT. At any given time, the point will have an x-y coordinate. However this coordinate will change as the object to which it is attached is moved or deformed. The fields of a PointT prototype are given below:

The current coordinates of the point are specified by $x and $y. $pin__cnt specifies the number of the set to which the point is pinned. $pin__cnt can be looked up in the pin table to find the set of

```
prototype PointT : =
    [
        [float,    $x],
        [float,    $y],
        [int,      $pin_cnt],
        [unspec,   $object]];
```

things pinned together. $object specifies the object to which the point is on. This can be either part of a primitive (type GeomT) or a wire segment (type WireSegT), hence a point is incident to at most one equipotential value. If $object is of type GeomT, this does not mean that the point is physically incident on that geometry, only logically incident. The point is incident on a GeomT in the same primitive and the same layer and potential which is physically contiguous to the named GeomT.

7.3.3 PointCellT

The type PointCellT is an indirect specification of a point. A pointcell points to a location, but the location is considered to be on the cell.

```
prototype PointCellT : =
  [
    [PointT,  $point],
    [unspec,  $object]];
```

Since pointcell is an indirect pointer to an object, the first component is the point to which it actually point. $object is the cell to which the point is attached. In the design tree $object should be a 'parent' to $point if $point is on a primitive, or if $object is a wire segment than that wire segment is incident to some descendent of the cell.

Chapter 8

Wire Deformations

8.0 Introduction

In this chapter, we shall describe how wires are manipulated in GENERIC. Wires are not manipulated directly; rather, they are manipulated as the result of applying a VLSI operator to some part.of the circuit. For example, if a cell is to be moved relative to the rest of the circuit, then the wires which attach that cell to the external circuitry must be deformed in order to maintain GENERIC's goal of simultaneously maintaining connectivity while guaranteeing satisfaction of geometric design rules. This chapter describes how GENERIC chooses the wires to deform, and which deformations are to be applied to the chosen wires.

8.1 Types of Wire Deformations

Wire deformations can be seen as consisting of two separate components: direct deformations, which are the shrink, slide and stretch operations and indirect transformations which mostly prepare the wires for the direct operation stretch. Both the direct and indirect deformations are operator specific. Furthermore, these deformations can be inhibited by the setting of *manip_flags* in the individual wire segments and also by the global flag, *GlobalWireManip*.

8.1.1 Direct Wire Deformations

GENERIC supports several different types of direct wire deformations. If the direction that the object is to be moved is parallel to the wire, then the simplest type of deformation is to stretch the wire (if the two points joined by the wire are moving further apart) or shrink the wire (if the two parts are

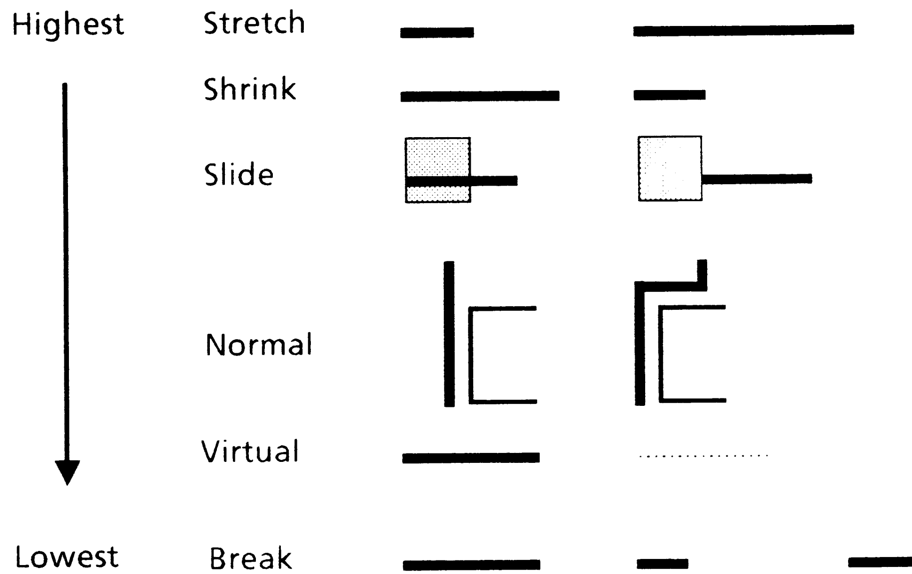


Figure 1 -- Wire deformations and their priorities. See text for full description.

moving closer together). In figure 1, changed wire sizes are shown as the result of applying these deformations.

If a wire is incident on a primitive then another type of operation can be applied called a slide. Primitives define the geometry necessary for a well formed device; however, it is often possible to overlap connected wires or other primitives on part of the geometry. This is shown below for an inverter (see figure 2 where the black section is the overlap). A slide allows a wire (and any connected geometry) to overlap the geometry used by a primitive.

8.1.2 Indirect Wire Deformations

Direct wire deformations change the visible apperance of the circuit: a wire is longer (shorter) than it was before, or a wire is moved relative to a piece of geometry. Indirect deformations change the underlying data structures but do not change the locations of wires in the circuit. Thus, the

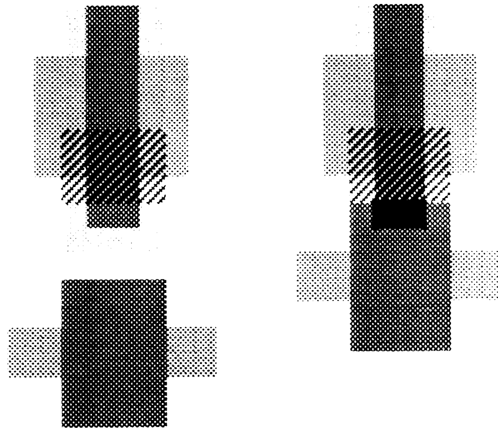


Figure 2 -- Slide. Dark black area on right is the result of overlapping geometry which in this case is part of a primitive, but could also be a wire overlap with primitive.

indirect wire deformations must be used in conjunction with the direct wire deformations to actually change the layout. The changes made to the data structures by the indirect wire deformations include splitting a wire segment into multiple wire segments, changing a wire to be on a virtual layer, or breaking the wire into two disjoint parts. The effect of applying these operators is shown in figure 1. For the change to virtual, the result is shown as a change in the "color" of the wire, for the normals, the result shown is the result of a normal indirect deformation followed by a stretch, and finally, the break is the result of applying a break indirect deformation followed by a move operator. We shall now describe each of these deformations in greater detail.

If the direction of movement is orthogonal to the orientation of the wire, then the wire can be broken into several segments -- a orthogonal component and a zero length wire segment parallel to the direction of movement. The zero length section can then be stretched. This operation is called taking a normal. Normals come in several varieties, and while they are all described loosely by that

term, they are individualy inhibited by the manipulation flags, and not all of them apply to all operators. There are conceptually three normal types which are described below.

The first normal we shall consider is the normal at a point. This normal at a point does not break any line segment, but adds zero length line segments at endpoints). We show an example below (figure 3) of a normal being added to the object being moved, which consists of the box, and part of the wire.

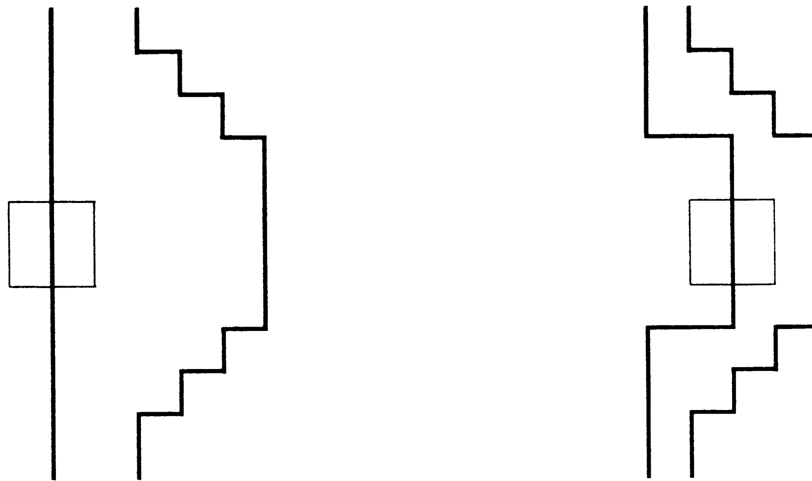


Figure 3 -- Normal at a point. The rectangle and some of the wire constitute the object to be moved.

The second type of normal we shall consider are the staircasing normal, which changes the shape of the object which is moving to form fit against the (unchanged) shape of the object that it is moving against. An example is shown in figure 4.

The implementation of the staircasing normal is done in GENERIC as follows. As much of the object is moved as possible. If an obstruction is encountered, then the wires that are moving are

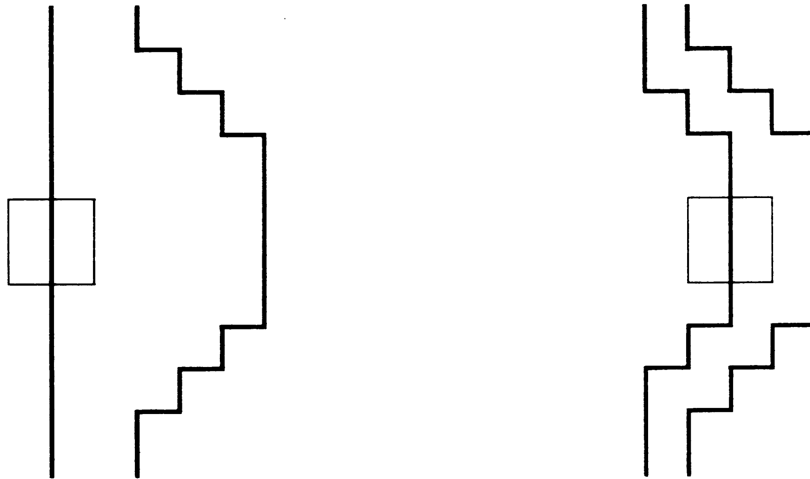


Figure 4 -- Staircase moving object against stationary one. The rectangle and all of the wire constitute the object to be moved.

broken into moving and non-moving sections. The process is then iterated. The structure of the GENERIC runtime library places this iteration in the operator. This deformation is used only in the **move** operator

The final type of normal is the **kink** environment. The object being moved does not change, but when an obstruction is encountered, the obstruction is fragmented into moving and stationary parts as shown in figure 5.

Another type of operation that can be performed is to make a wire into a virtual wire. These wires do not interact with any other object (including virtual wires) but can be routed. If one is used to the Mead/Conway coloring scheme, then making a wire virtual corresponds to making the wire clear -- it can (barely) be seen, but does not interact with other wires. Since GENERIC keeps track of the electrical potential of wires, there is never any confusion.

Figure 5 -- Kink environment. The object which causes an obstruction to the moving object is kinked.

Finally, the last deformation is to disconnect a wire. This deformation breaks the wire into two separate, physically unconnected wires.

8.2 Which wires to deform

The object to be moved is specified by an operator. The wires connecting that object to the external circuitry will be specified starting with those wire segments incident on the object. For each wire segment, the deformations will be chosen in the order of precedence. First, the lowest level deformation is applied. When this deformation no longer can be applied, the lowest level deformation that can be applied for that segment is used. When there are no deformations that can be applied to that segment, the wires attached to that segment are deformed (moving away from the object).

8.3 Precedence of wire deformation

The precedence of wire deformations is in the order specified in figure 1.

8.4 Control over wire deformation

If the user wishes to prevent a particular wire segment from being deformed in a given way, there are manip__flags which specify the allowable deformations. Typically, break is disallowed. For more information see the data structures chapter in this document.

Chapter 9

VLSI Operators

9.1 Operators

GENERIC has both binary and unary infix operators. In this chapter we describe those operators defined in the run-time library for manipulating layout. These operators are design rule safe and are intended to be complete. By completeness, we mean that all possible layout manipulations can be expressed using the base level operators. Furthermore, these operators should be "natural", that is, the base operators should provide direct and intuitively appealing support as building blocks for constructing larger operators. The final constraint is that these operators should be efficient to implement.

9.1.1 Operator Use

Most of the VLSI operators in GENERIC are for the purposes of manipulating the layout. Operators include such functions as placing some geometry relative to other geometry, orienting geometry to some angle orthogonal to the x-y axis, changing the plane on which the geometry lies. While performing these operations on its operands, the operators attempt to preserve connectivity by stretching or otherwise deforming wires. Furthermore, they ensure that the results of an operation does not introduce design rule errors.

The requirement that the geometric design rules not be violated means that occasionally it will be impossible to perform a requested operation. In that case, the operator fails, and the circuit is left unchanged (as it was before the operation was applied). Higher level operations built up of lower level ones should have fall back strategies to achieve their goals, in case the base level strategies fail.

9.1.2 Operator Operands

The VLSI layout operators share some general structural properties, especially in terms of their operands. In this section we will describe the scheme by which these operators are used. Exceptions to the scheme outlined here are specified in the description of that operator.

For both binary and unary operators, op_1 specifies the object ($object_1$) to be manipulated. For example, $object_1$ might be an inverter which is to be moved. The operator (**move** in this case), could cause changes to the environment of the inverter; the environment would include the wires connecting the inverter to the remainder of the circuit. The **move** operator both translates the inverter and deforms the wires connected to the inverter.

Op_2 is the reference operand. It specifies what is to be done to $object_1$. In the case of **move** it describes both the distance and the direction to move the inverter.

Op_1 specifies two things; first the object which is to be manipulated and second, possibly a point on the object which can be used for exact computation of the desired manipulation. If a point is not given, then the operator computes some reasonable point based on the object specified by op_1.

Op_2 specifies, depending on the operator, either an object (possibly with a point), or just some information which is sufficient to compute what the operator should do. This is specific for each operator. For the move operator, op_2 either specifies a distance to be moved to or a point to be moved to in either the x or y direction.

The objects specified by op_1 are shown in Table 1. These operators can either be specified directly

PrimitiveT
CellT
WireSegT
TUPLE containing only the above three types

Table 1

or through one of the point types in Table 2.

9.2 Placement Operators

PointT PointCellT
Table 2

The placement operators are the first class of operators we will define . These operators take the object to be moved (called the **transformation set [ts]**), and attempt to place it at the point specified by op_2. The transformations to be performed on **ts** are a subset of the linear transformations. On a given call, these operators usually cause translation in either the X or the Y direction. The direction in which $object_1$ is to be moved is either an explicit, required parameter or is calculated by the operator.

Given the direction, an interval can be calculated which is the range over which the object will be moved. The best (closest to the desired distance within the interval) valid move which fits in this range will be performed; if there is no valid move which fits in this range the operator will fail, and no change will be made to the layout. The legal interval depends on two factors, the side op_1 is on versus the side (direction) that the operator specifies to move it to, and the distance which would be the ideal move. The distance is highly operator dependent, and will be described under each operator.

Intervals have two forms. If the object is on the same side as it is to be moved, then the object is just moved closer (1c, 1d). The legal interval is [0, distance]. If $object_1$ is on the other side then the side it is to be moved to, then the legal interval is [distance, ∞] (1a, 1b).

9.2.0.1 Translation Operators

In this section, a general scheme is given for the translation operators. The objects referred to in this section are either wire segments or primitives; cells are broken down into their component parts for the purposes of these operators.

The scheme for these operators is as follows. A set is computed which is the transformation set (**ts**). The **wire frame set (ws)** is defined as the set of wires connecting the transformation set to objects outside the translation set. The wire segments in **ws** are exactly those wire segments which are currently (at some point in the life of the operator) being deformed. For each wire segment in **ws**,

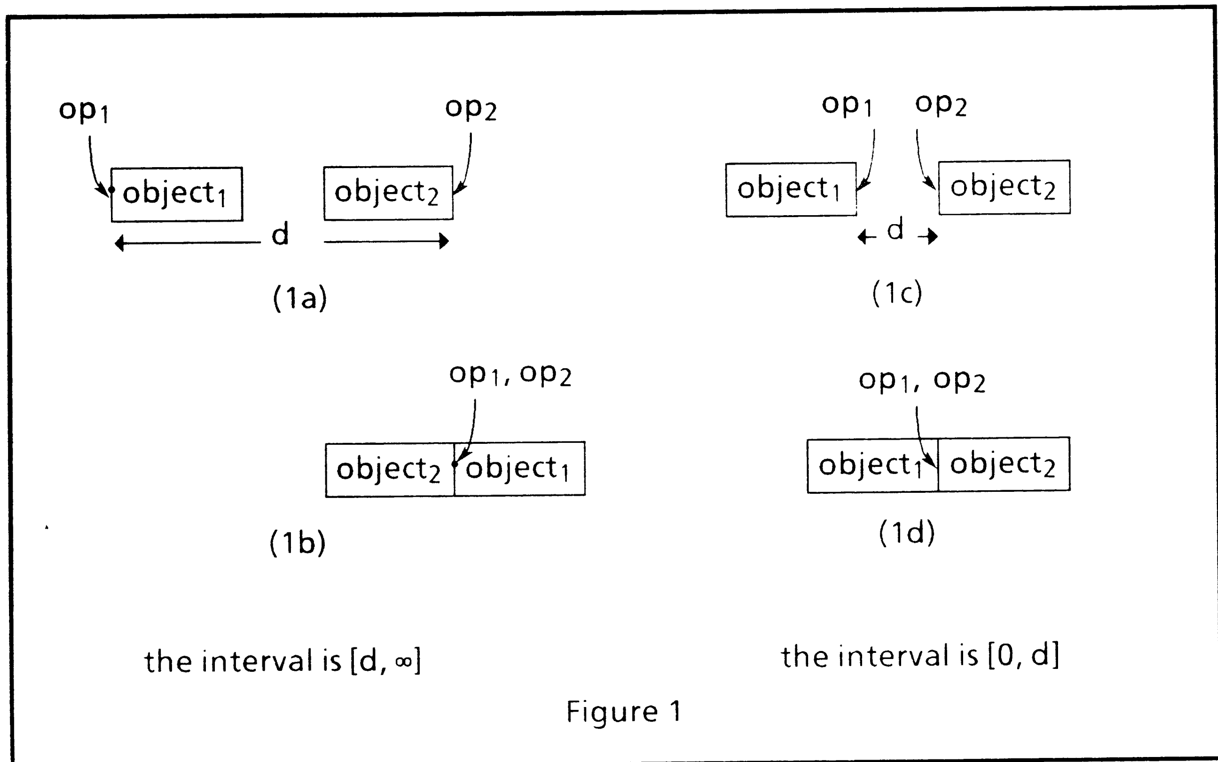


Figure 1

there is defined a current deformation on that wire segment, and the maximum 'play' or distance that the wire can be deformed given the direction that the **ts** is moving. A third set, which is a subset of **ws**, is the **constraint wire set** (**cw**) which is defined by the property that for each member in **cw**, the amount of deformation possible for that wire segment, using the current deformation for that wire segment, is a minimum over **ws**. If an object which is not a wire segment is connected to an object in **ts**, but that object is not an element of **ts**, then no movement is possible, and the translation is halted.

It will, in general, take a number of iterations for the operator to move **ts** to its desired location. At each iteration, the maximum distance that **ts** can be moved is calculated. This maximum distance is the minimum of the **cw** distance and the minimum distance before an obstruction is encountered (this is operator dependent). Another constraint is that, if the desired interval is [0, d], then the objects would be moved no further than d, and if the constraint is [d, ∞], the object would be moved the minimum distance beyond d for which it is possible to satisfy the design rules.

The **ts** set is moved the calculated distance. If this is less than the desired distance, another iteration of the loop is required. If the **cw** distance was the constraint, then each wire segment in **cw** is checked to see if a different deformation type can be applied which provides a non-zero constraint. If for a particular wire segment in **cw**, this is not possible, that wire segment will be added to the translation set, and the sets **ws** and **cw** will be incrementally recomputed.

If the limitation on the translation is an obstruction and normals are allowed, then a normal is created. Otherwise, if the obstruction is a wire segment, then the obstruction is added to **ts**. If the obstruction is not a wire segment (i.e. is a primitive), then translation will be halted.

ts - transformation set: the set of wires and primitives which are being moved without deformation
ws - wire frame set: the set of all wire segments connecting **ts** to everything outside of **ts**, and is the current set of wires being deformed.
cw - constraint wire set: the subset of **ws** wires which have the maximum deformation distance possible in **ws**, given the deformation type currently being used for that wire segment.

The new (augmented) **ts** now becomes the basis for the next stage in the iteration. **Ws** and **cw** are recomputed, and the operation is continued until either it is impossible to move any further (a failure return), or the object falls within the interval (a success return)

9.2.1 Alignment Operators

op_1	**over**	op_2
op_1	**align__horizontal**	op_2
op_1	**align__vertical**	op_2

Over is a composite operator which performs both an **align__vertical** and an **align__horizontal**. For **over** to succeed, the exact distance specified must be moved. The points to be aligned are either

the points specified by op_1 and op_2, or if either op_1 or op_2 does not specify a point, then the point to be used in place of op is the center of the bounding box for the object specified by that op.

Align_horizontal only changes the x coordinates of the objects being translated. The distance of the two points in the x direction is calculated. The operator will succeed if it is possible to place **ts** the exact distance specified by op_1, deforming the wires incrementally as specified. The deformed wires , of course, may not introduce design rule errors, and the order and types of deformations have already been discussed. Ts, on the other hand, need only satisfy the design rules in its final position -- not in the intermediary positions of the iterations. The conceptual effect is as if **ts** was on a new plane, moved to the desired position and dropped onto the old plane(s). The objects move the distance calculated, and if it is impossible to move that distance, the operator fails.

Align_vertical is the analog of **align_horizontal**, but in the vertical direction.

9.2.2 To Operators

op_1	**to{direction}**	op_2
op_1	**to_left**	op_2
op_1	**to_right**	op_2
op_1	**to_up**	op_2
op_1	**to_down**	op_2

To is similar in operation to **align** but differs from to in that it:

1) does not require that the exact distance be moved, but that the best move within the computed interval be made, and

2) ensures that $object_1$ does not interfere with $object_2$ (even if these aobjects are on different planes).

The first constraint means that **to** much more closely follows the model of the general translation operator presented in section 1.0. The second constraint is present for another reason: **to** should be usable in conjunction with the operator **drop**, which is described latter in this chapter. Thus, $object_1$ is design rule checked not only with objects on the same plane(s), but with $object_2$ but acts

additionally as if $object_2$ was on the same plane(s) as $object_1$. Of course, there may be other objects on the plane(s) of $object_2$ which prevent $object_1$ from being dropped.

To_left is equivalent to **to**{LEFT}.

9.2.3 Sliding operators

op_1	**move**{direction}	op_2
op_1	**move_left**	op_2
op_1	**move_right**	op_2
op_1	**move_up**	op_2
op_1	**move_down**	op_2
op_1	**move_away**	op_2
op_1	**move_closer**	op_2

The sliding operators are similar to the placement operators, except that all positions which $object_1$ are moved through must be legal. This means that objects between $object_1$ and $object_2$ become compressed as the $object_1$ frontier moves towards $object_2$. This is similar to the plow operator in MAGIC [MAGIC].

9.3 Orientation operators

orient{angle, point}	op_1
orient_left{point}	op_1
orient_right{point}	op_1
orient_up{point}	op_1
orient_down{point}	op_1
op_1 **face**{point1, point2, constraint}	op_2
mirror{x,y}	op_1

Orient takes $object_1$ and orients it by an angle which is an integer multiple of 90 degrees. The default value for angle is 90 degrees. The optional parameter point specifies the center of rotation. if point is not present, the center will be chosen as the center of the bounding box enclosing $object_1$.

Rather then deform the wires attached to op1, **orient** detaches all wire segments connected to op_1, performs the rotation, and then reconnects the wire segments using a simple router. The user may, by defining the parameter no__reroute, do their own routing. **Orient** returns as its value the list of endpoints which were not rerouted.

The operators **orient__left, orient__right, orient__up, orient__down** require op_1 to be of type PointT or PointCellT. The bounding box is calculated, and the side of the bounding box nearest to op_1 will determine the angle that the object is to be rotated. For example, if the operator was **orient__up**, then the side of the bounding box would be rotated the minimum absolute angle possible so that the computed bounding box side is pointing up. The default point to be used as the center of the rotation is op_1.

Face performs a double rotation, rotating both $object_1$ and $object_2$. The faces of $object_1$ and $object_2$ are calculated as above for **orient**. In order to make these sides face each other, all that needs to be specified is what direction is side1 is relative to side2.

Mirror does not perform a rotation at all, but transforms by a mirror image about either the x (horizontal mirror) or y point (vertical mirror).

9.4 Plane Operators

op_1 **drop** op_2

Drop changes the plane(s) that $object_1$ is on to the plane specified by op_2. Op_2 can either be the number of the plane, or some objects, all of which lie on the same plane. If there is not room to put $object_1$ on the specified plane, then the operator fails.

9.5 Other Operators

Operators not covered here are listed below:

```
trim
trim__left
trim__right
trim__top
trim__bottom
abut__left
abut__right
abut__top
abut__bottom
```

References

[PASCAL] K. Jensen and N. Wirth, Pascal user manual and report, Springer-Verlag, 1978.

[ICON] R. Griswold, and M. T. Griswold, *The ICON programming language*, second edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983

[Magic] J. K. Ousterhout, Hamachi, G. T., Mayo, R. N., Scott, W. S., and Tayor, G. S., "Magic: a VLSI layout system", Proceedings of 21st Design Automation Conf., pp 152-159, Albuquerque, NM, June 1983.

[C] B. Kernighan and D. Ritchie, The C Programming Language

[SETL] R. Dewar, E. Schonberg, and J. T. Schwartz, "Higher level programming: Introduction to the use of the set-theoretic programming language SETL", CIMS, New York University, 1981.