MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 598 V.L.S.I. Memo 80-31 September, 1980

THE DESIGN PROCEDURE LANGUAGE MANUAL

by John Batali and Anne Hartheimer

Abstract: This manual describes the Design Procedure Language (DPL) for LSI design. DPL creates and maintains a representation of a design in a hierarchically organized, object-oriented LISP data-base. Designing in DPL involves writing programs (Design Procedures) which construct and manipulate descriptions of a project. The programs use a call-by-keyword syntax and may be entered interactively or written by other programs. DPL is the layout language for the LISP-based Integrated Circuit design system (LISPIC) being developed at the Artificial Intelligence Laboratory at MIT. The LISPIC design environment will combine a large set of design tools that interact through a common data-base.

This manual is for prospective users of the DPL and covers the information necessary to design a project with the language. The philosophy and goals of the LISPIC system as well as some details of the DPL data-base are also discussed.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's V.L.S.I. research is is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-80-C-0622 and in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014-75-C-0643.

CONTENTS

1. INTRODUCTION 6

1.1 The Manual 6
1.2 Designing With DPL 7
1.3 Credits 10

2. INTRODUCTORY EXAMPLES 11

3. THE DATA-BASE 16

3.1 Types 16
3.2 Prototypes 17
3.3 Virtual-Copies 17
3.4 Instances 19
3.5 Storing and Accessing Information 22
3.6 What Happens When Something is Made 22

4. BUILDING THINGS 24

4.1 Creating Types 24
4.2 The Type RECTANGLE 25
4.3 Instantiating Types 26
4.4 The Structure Built by a Type 28
4.5 Naming Things 28
4.6 Accessing Parts and Parameters 29
4.7 The General Access Function 30
4.8 Additional Features of DEFLAYOUT 31

5. PLACING THINGS 33

5.1 Coordinate system 33
5.2 Points 33
5.3 Implicit Parameters 34
5.4 Translation 36
5.5 Unitary Transforms 36
5.6 Placement by Parameter 38
5.7 Invoke 39

5.8 *LIST* 40

6. THE DPL WIRING SYSTEM 41

6.1 Wire System Commands 41
6.2 Wire System Example 43
6.3 External Wire Commands 45
6.4 Connection Points 45

7. CONSTRAINTS 47

7.1 Using Constraints 47
7.2 Defining Constraints 48

8. REPLICATORS 50

8.1 Calling Replicators 50
8.2 Accessing Replications 51
8.3 Defining Replicators 52

9. USING DPL 54

9.1 Interacting with DPL 54
9.2 What Objects Look Like 55
9.3 CIF 56

10. EXAMPLE 57

10.1 DPL design style 57
10.2 The REGCELL 58
10.3 Discussion of the REGCELL program 62

11. LIBRARY 65

11.1 Some Constraints 65
11.2 Some Types 65
11.3 Some Replicators 68

12. GLOSSARY 71

12.1 Types 72
12.2 Naming 73
12.3 Access Functions 74
12.4 Points 75
12.5 Transform Functions 75
12.6 Wiring Commands 77
12.7 Constraints 77
12.8 Replicators 78
12.9 CIF 79
12.10 Implicit-Parameters 79
12.11 *LIST* 79
12.12 Layer Sizing 80
12.13 Variables 80
12.14 Constants 81

ACKNOWLEDGMENTS

We would like to thank Ned Goodhue for drawing the figures. Neil Mayle, Howard Shrobe, Jon Taft, Gerald Sussman, Daniel Weise, and Ron Rivest read and commented on drafts of this manual. Chuck Rich helped us figure out how to use the text-justifier.

1. INTRODUCTION

1.1 The Manual

This manual describes the Design Procedure Language (DPL) for LSI design. DPL creates and maintains a representation of a design in a hierarchically organized, object-oriented LISP data-base. Designing in DPL involves writing programs (Design Procedures) that construct and manipulate descriptions of a project. The programs use a call-by-keyword syntax and may be entered interactively or written by other programs. DPL is the layout language for the LISP-based Integrated Circuit design system (LISPIC) being developed at the Artificial Intelligence Laboratory at MIT. The LISPIC design environment will combine a large set of design tools that interact through a common data-base.

This manual is for prospective users of the DPL and covers the information necessary to design a project with the language. The philosophy and goals of the LISPIC system as well as some details of the DPL data-base are also discussed. The implementation of the language is not discussed here except for those details that are felt to be instructive when attempting to understand the language. The manual is organized as follows:

The introduction describes the key features of the LISPIC system, the data-base and DPL.

Chapter 2 contains some introductory examples of the use of DPL. The examples consists of definitions of several cells and pictures of the cells.

Chapter 3 presents an overview of the DPL data-base. Here we discuss abstract structures used to represent designs.

Chapters 4 through 8 present DPL itself. The functions and forms most useful for using DPL when designing a project are presented and explained. The material in these chapters constitutes all of the information necessary to use the language.

Chapter 9 discusses how DPL "looks" to the user. It presents interaction details as well as functions for translating between DPL and CIF.

Chapter 10 presents a fairly hefty example which exercises many of the features

of the language. The example also demonstrates the design style which DPL supports.

Chapter 11 is a library containing the definitions of objects available in the basic DPL system. This chapter is useful both as a set of examples of the use of DPL and as a reference for designing.

Chapter 12 is a glossary of DPL functions and variables. The syntax and usage of functions are summarized. This is where we explain how all the functions evaluate their arguments. The most useful DPL functions will have been met earlier in the manual. The glossary also contains functions which are less useful or more "low level" then the functions explained in the body of the manual.

1.2 Designing With DPL

VLSI design is complicated. A large IC design may contain several thousand or more pieces of material. Designers think of their designs not in their full complication but rather as collections of of parts which may be further decomposed into other parts. Such a hierarchical viewpoint both expresses the designer's understanding of his design and economizes his thinking about it. Unnecessary detail is suppressed so that the gate, module, subsystem or system of interest may be dealt with.

A set of design tools should be able to represent the design in as much the same way the designer does as possible. Thus the basis of a set of design tools should be a data-base representation of designs that is flexible, extensible and hierarchical. The goal of the LISP-based Integrated Circuit (LISPIC) design project is to produce a design system consisting of a large number of design tools -- simulators, design verifiers, routers etc. -- integrated with one another through a common data representation.

The Design Procedure Language (DPL) is a collection of LISP functions that construct and manipulate a hierarchically organized object-oriented data-base. DPL is intended to be a user language -- it can actually be used by a human to build projects. DPL may also be used by programs such as PLA generators, routers, and node-extractors. Since it is embedded in LISP, DPL inherits the power of a full programming language. LISP programs can be written that call DPL functions and

vice-versa.

The LISPIC system is illustrated in Figure 1. The LISP-based data base is manipulated by the DPL language. Other programs and systems communicate with the data-base through DPL. The other systems may communicate and cooperate with each other through this common representation. (Note: Not all of the systems pictured are available yet.)

Using DPL consists of designing a project by specifying a procedure that will build it. This is the reason for the name "Design Procedure Language". A design procedure constructs a data-structure which holds a description of a design. The descriptions may include procedures for further manipulation of the data-base. At some point the designer will want to actually construct a physical implementation of the project, but for most of the design process that is not necessary. What is necessary is the construction and maintenance of a structure that represents the design. Since much in a design description is procedural, a description can help to build itself. Such procedural descriptions also allow the DPL data-base to be modified by other programs.

We should point out that the need to do this sort of thing is the reason why the system is implemented in LISP. More than any other programming language, LISP easily handles arbitrary structures that may contain procedural parts and may even be able to build themselves. The simple syntax of LISP allows programs to write programs easily, and the interpreter allows a "real-time" interaction between the designer and the language.

Parameters to DPL design procedures use call-by-keyword syntax. The parameters may be assigned default values. Constraints among the parameters may be specified by the designer. Parts and parameters of objects may be named and the named information may be accessed by functions which follow path descriptions and transform objects.

The DPL design process involves the following stages: The designer specifies procedures for constructing pieces of the design. These pieces are then used to build more complicated representations. The procedures may refer to information stored in the structure earlier. Mistakes may be corrected and changes implemented by making use of information in the existing structure. The final design is a complicated

LISPIC ENVIRONMENT

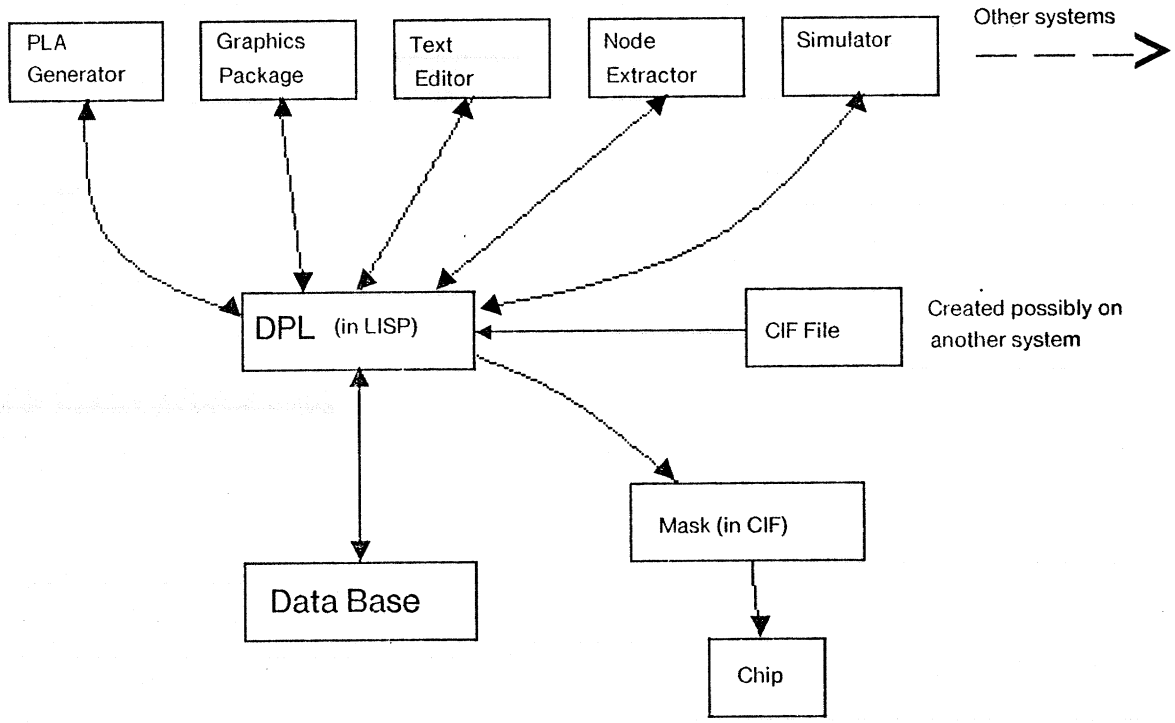


Figure 1

yet organized hierarchical structure which may be used to produce mask specifications.

1.3 Credits

DPL was written by Gerald Jay Sussman, Howard Shrobe, Neil Mayle, and John Batali. The language is based on two earlier IC layout languages, one by Jack Holloway and Sussman, the other by Shrobe. Ron Rivest, Daniel Weise, Anne Hartheimer, Howard Cannon, Tom Knight, Jon Taft and Paul Penfield contributed help, advice and enthusiasm.

2. INTRODUCTORY EXAMPLES

In this chapter we present three illustrations of the use of DPL. The examples are included here to motivate the detailed description of the language in later chapters. We suggest the reader look at the examples before reading on, note the interesting points, and refer back to the examples while reading the rest of the manual.

The three example cells are described in DPL expressions and are accompanied by pictures of their layouts.

The definition of PASS-TRANSISTOR specifies that an object be built from several rectangles called CHANNEL, SOURCE-DIFFUSION, DRAIN-DIFFUSION and POLY-PIECE. The channel region is a rectangle of the layer "channel" whose length and width are determined by the parameters passed when this object is built. All of the parts of the pass-transistor are given names. The positions for the source and drain diffusions are determined by aligning points on them with points on the channel rectangle. Also, a point P is named. An instantiation of the pass-transistor with a particular set of values for its parameters is shown in Figure 2A.

The INVERTER "calls" the pass-transistor. It also calls a "standard-pullup" which is a cell defined elsewhere (see Library). The inverter's parameters are constrained so that the inverter-ratio is equal to the ratio of the pullup-ratio to the pulldown-ratio, etc. Note that the pass-transistor is placed by lining up its top-center with the location of a named point, (DIFFUSION-CONNECTION), the pullup. Also a point, (INPUT-PT), the inverter is named (Figure 2B). The location of INPUT-PT is determined by following a path of named parts: INPUT-PT is located at the CENTER-LEFT point of the part named POLY-PIECE of the part named PULLDOWN of the inverter.

BUFFER is a cell that could be used to refresh a signal. It calls INVERTER twice, with different parameters. It thus makes use of two different versions of PASS-TRANSISTOR. The version used is determined by how the constraint system sets the values of the inverter's parameters when it is called. The second inverter is placed far enough away from the first to allow room for the connection. Variables representing design-rule constants, *POLY-TO-POLY* and *DEFAULT-POLY-SIZE*, are used to specify placement. The connection between the two inverters is made with the DPL wiring system. The wire begins at a point on the boundary of a part of a part of a part of

INPUT-INVERTER. It runs horizontally to a point halfway between the inverters and then "jogs" to the input of the second inverter (Figure 2C).

The figures show the structures built when the three object definitions are called. They also show how the location of the point named P in PASS-TRANSISTOR is transformed as the structures are built.

Note: The definitions of PASS-TRANSISTOR and INVERTER presented here are not the same as the definitions in the library. They are, however, perfectly legal DPL and would "work" as defined. We have made some changes to make the examples simpler. BUFFER is not available in the basic library.

```
(deflayout pass-transistor                          ;;; PASS-TRANSISTOR
   '((primary-parameters
        ((channel-length 2)
         (channel-width 2))))
  (part 'channel rectangle
        (layer 'channel)
        (length (>> channel-length))
        (width (>> channel-width)))
  (part 'source-diffusion rectangle
        (layer 'diff)
        (length *diff-overhang*)
        (width (>> channel-width))
        (top-center (>> bottom-center channel)))
  (part 'drain-diffusion rectangle
        (layer 'diff)
        (length *diff-overhang*)
        (width (>> channel-width))
        (bottom-center (>> top-center channel)))
  (part 'poly-piece rectangle
        (layer 'poly)
        (length (>> channel-length))
        (width (+ (>> channel-width)
                  (* *poly-overhang* 2))))
  (setq-my p (>> center-right poly-piece)))
```

```
(deflayout inverter                                          ;;; INVERTER
   '((primary-parameters
        ((pullup-length 8)
         (pullup-width 2)
         (pulldown-length 2)
         (pulldown-width 2)
         (pullup-ratio nil)
         (pulldown-ratio nil)
         (inverter-ratio nil)))
     (constraints ((c* pullup-length pullup-ratio pullup-width)
                   (c* pulldown-length pulldown-ratio pulldown-width)
                   (c* pullup-ratio inverter-ratio pulldown-ratio))))
  (part 'pull-up standard-pullup
        (channel-length (>> pullup-length))
        (channel-width (>> pullup-width)))
  (part 'pull-down pass-transistor
        (channel-length (>> pulldown-length))
        (channel-width (>> pulldown-width))
        (top-center (>> diffusion-connection pull-up)))
  (setq-my input-pt (>> center-left poly-piece pull-down)))


(deflayout buffer                                            ;;; BUFFER
   '((primary-parameters ((input-ratio 8)
                          (output-ratio 4))))
  (part 'input-inverter inverter
        (pullup-ratio 4)
        (inverter-ratio (>> input-ratio)))
  (part 'output-inverter inverter
        (ratio (>> output-ratio))
        (center-left (pt-to-right (>> center-right input-inverter)
                                  (+ *poly-to-poly*
                                     *default-poly-size*
                                     *poly-to-poly*))))
  (wire 'connection
        (run-layer 'poly)
        (from (pt-above (>> bottom-right gate-poly transistor pull-up input-inverter)
                        (half *default-poly-size*)))
        (to-x (+ (>> x p pull-down input-inverter)
                 *poly-to-poly*
                 (half *default-poly-size*)))
        (jog-y (>> input-pt output-inverter))))
```

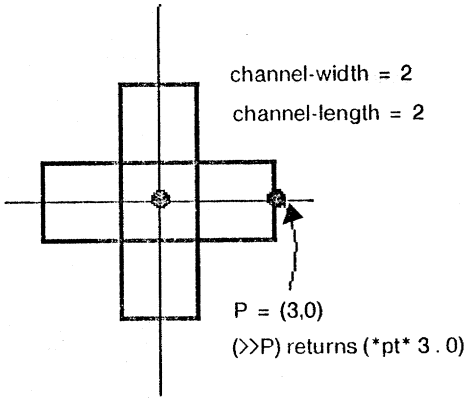


Figure 2A -- pass-transistor

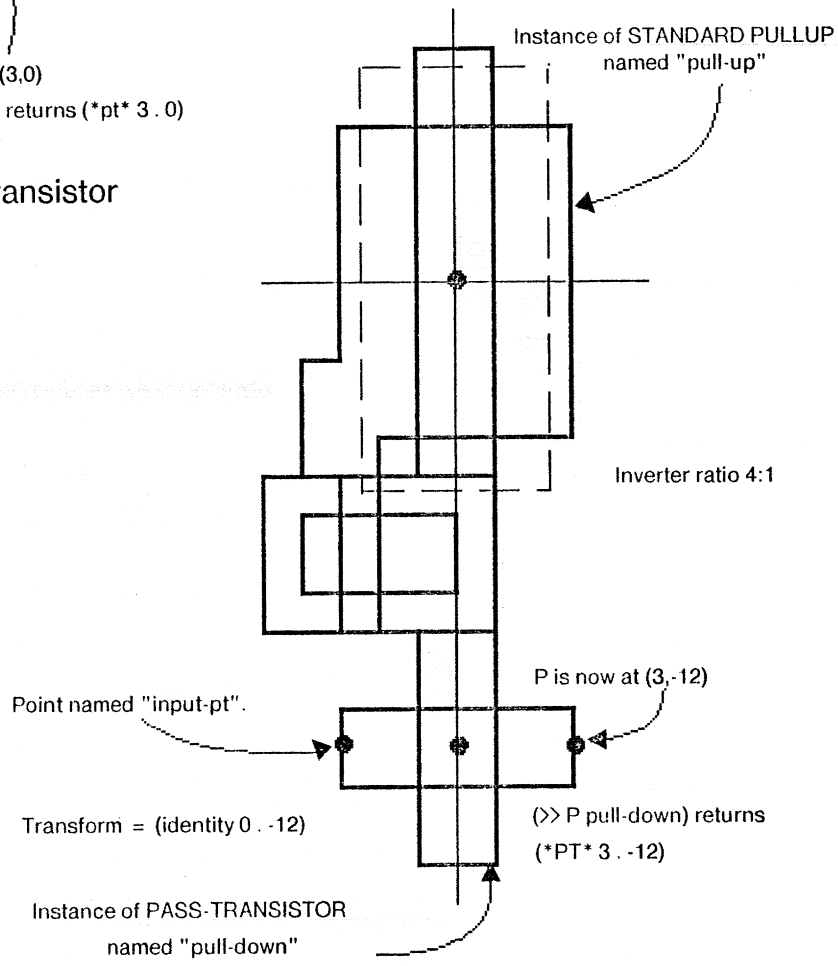


Figure 2B -- Inverter

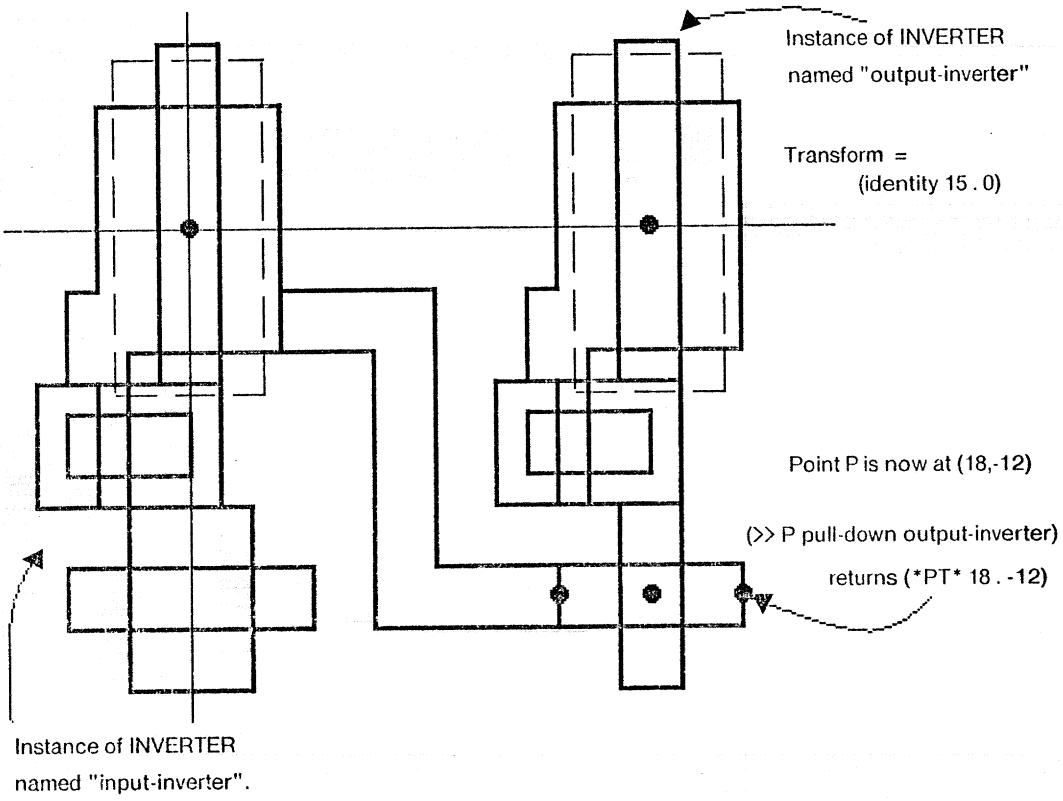


Figure 2C -- Buffer

3. THE DATA-BASE

In this chapter we discuss the data base used by DPL and the vocabulary needed to explain the particulars of the language. We discuss the data structures of DPL, explain what is in them, and show how they are used by the language to build descriptions of designs.

This chapter need not be understood fully to use the language. The implementation details and more complicated ideas are reasonably interesting but the only way to understand DPL is to use it. We recommend that this chapter be skimmed for the basic ideas and the later chapters examined more carefully -- they are more useful for using the language. For the most part, the difficulty of a concept is inversely proportional to its utility.

In the sections that follow, various abstract objects are introduced and discussed. The low-level implementation of these structures (i.e. lists or arrays or whatever) is not important and in most cases is invisible to the user.

3.1 Types

A **type** in DPL holds a procedure that builds data structures. These structures contain descriptions of various kinds. The procedure stored in a type is called the **maker** function of the type. A type also contains information about its maker function such as the parameters it may take, their default values, and constraints among the parameters. A type may also contain information about the structures produced by its maker function.

A type may be thought of as a description of a class of objects that share some common features. These objects are the structures produced when the maker function is run with various values for its parameters. The structures produced by a type thus are related by the way they were created. Usually one creates a type whenever a certain object or module is important enough to be given a name.

In the introductory example, `PASS-TRANSISTOR`, `INVERTER`, and `BUFFER` are all types. (The command `DEFLAYOUT` defines a type.) In all cases we have a certain conceptual entity which can nevertheless take a wide variety of forms. Inverters can be built with

many ratios, NOR-gates can be built with different numbers of inputs.

A type may specify that the structure it builds includes structures built by other types. Types may thus "call" other types. Objects are built by defining simple types which are called by progressively more complicated ones.

3.2 Prototypes

The structure that is built by a type is called a **prototype**. A prototype holds the description of certain parts of a design. The prototype and the description in it are produced by a "call" to the type with a particular set of values for its parameters. The description depends on the values of the parameters in a way that depends on the details of the maker function of the type. The different prototypes produced by a type will resemble each other since they were produced by the same procedure, but they will differ if their parameters differ.

The distinction between types and prototypes is this: Types hold programs that produce prototypes. Prototypes hold descriptions. The user defines types by specifying the details of the maker function. The maker function is used to construct a prototype and thus a description of a piece of the design. The user never directly touches a prototype -- he only tells a type how to build one.

In addition, the user may place any other information on a prototype he desires. It is often useful to name a part of an object or specify the value of a numerical parameter. This, as well as the addition of parts to a prototype, is done by inserting the appropriate commands in the maker function of the type.

3.3 Virtual-Copies

If the maker function of a type "calls" another type, the prototype built by the "called" type will be a part of the prototype built by the "calling" type. A prototype that is a part of another prototype is called a **virtual-copy** (VC) in DPL. A VC always has two pieces of information on it: its **parent**, which is the prototype it is a part of, and its **prototype**, which is the prototype being called.

The virtual copy is so named because the description of the prototype is

PART HIERARCHY

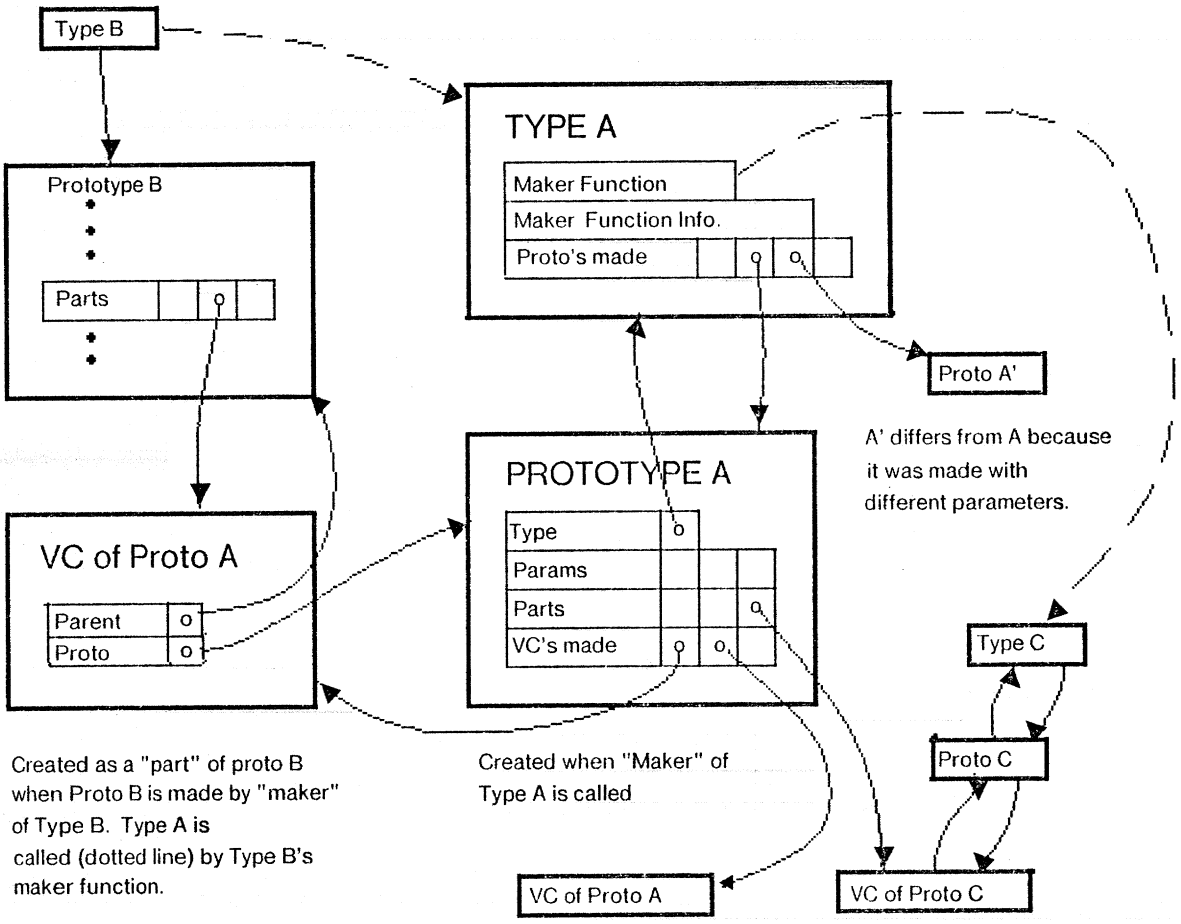


FIGURE 3

Note: VC's, Types, and Prototypes contain other fields not shown

available in the VC. The copy is "virtual" because the information is not on the VC, but on the prototype of the VC. Thus several prototypes can use a prototype as a part, each having a different VC of the that prototype. (See Figure 3.)

Information about a VC may be obtained by accessing the corresponding part or parameter of its prototype. For example, a prototype of INVERTER will have a value for its pulldown ratio. To find the pulldown ratio of a VC of that prototype, an access function finds the prototype and gets the information from there. The VC "looks" just like its prototype. One may pretend that the prototype's structure is really copied into the VC. Prototypes are not copied into VCs because it is more efficient to have only one prototype which is pointed to by its VCs.

It may be necessary to attach other information to a VC besides its parent and prototype. A particular VC may have a certain "reason" (for being a simple inverter, say, as opposed to a superbuffer). In general, information that is shared by more than one VC belongs on a prototype, while information that is specific to a VC may be stored there.

In addition to using some prototypes as parts of others, it is possible to specify that a certain type includes "all" of another type, plus some more information. In this case, the original type is called the **supertype** of the one that specifies the changes. The **subtype** has all the parameters of its supertype, plus any others declared when the subtype is defined. (See Figure 4.)

The difference between calling a type as a part of another and declaring a type a supertype of another, is that the subtype is really a modified version of its supertype, while a part is a different entity from its parent.

3.4 Instances

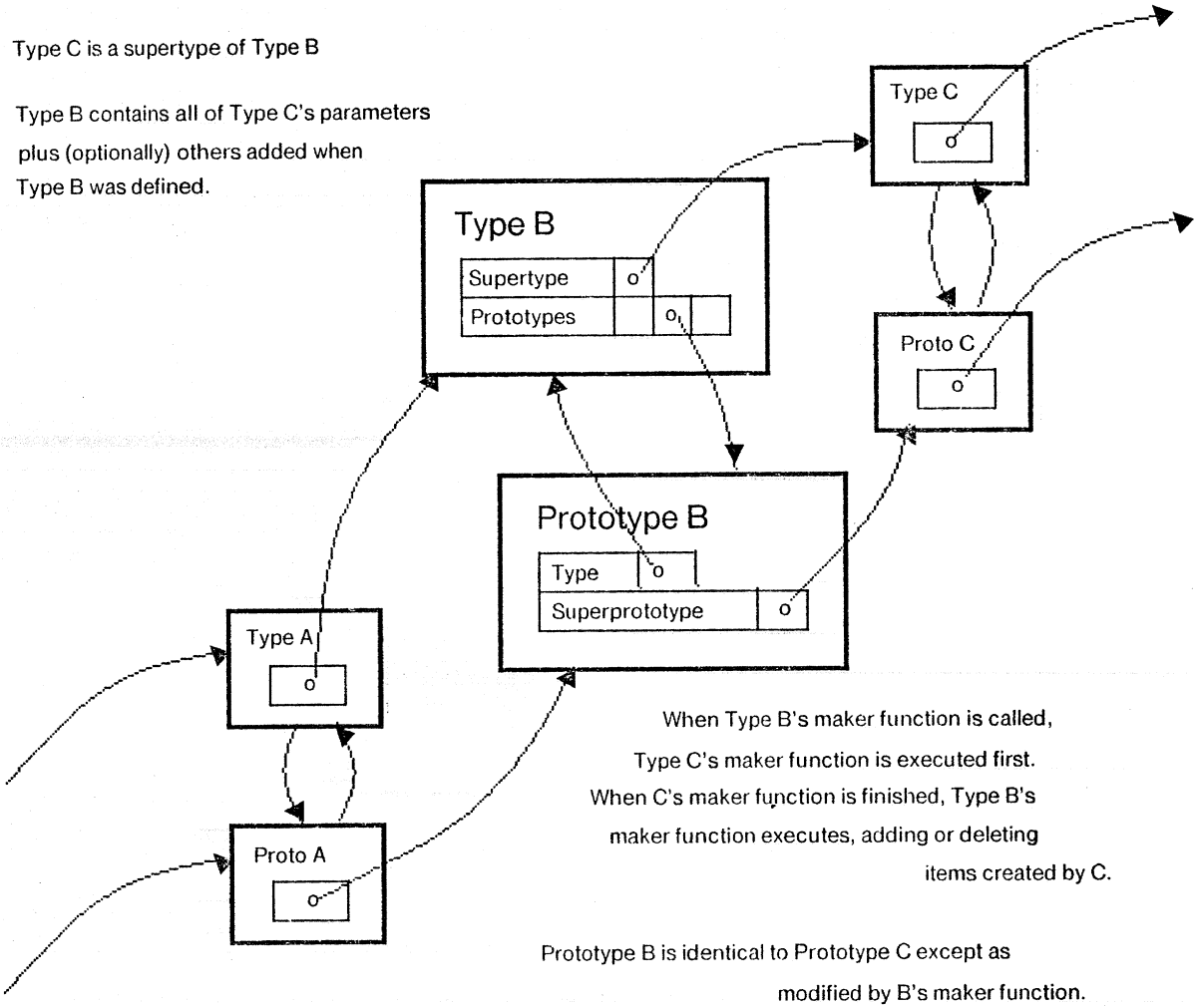
Up to this point have spoken only of information that is "fixed" on objects. Parts, parents, and parameters are all kinds of information that must be specified when describing an object. It is possible for some information about an object to depend on the context in which the object is viewed. An example of this sort of information is the geometric transform a VC undergoes when it is actually placed somewhere in a design.

TYPE HIERARCHY

Type B is a supertype of Type A

Type C is a supertype of Type B

Type B contains all of Type C's parameters plus (optionally) others added when Type B was defined.



When Type B's maker function is called, Type C's maker function is executed first. When C's maker function is finished, Type B's maker function executes, adding or deleting items created by C.

Prototype B is identical to Prototype C except as modified by B's maker function.

FIGURE 4

When building a prototype, it is necessary to specify where each of its parts is to be placed. We must specify how the coordinate system of the prototype being used as a part is to be transformed in the coordinate system of the parent prototype. In the introductory example, the pullup is placed so that its origin is at the origin of the inverter's coordinate system (by default). When we use this inverter in another prototype we must specify where the inverter is to be placed. The output-inverter of the example is placed with its origin at the point (15, 0) in the buffer's coordinate system.

The specification of a transformation is not a part of a VC because the same VC may be viewed in different ways. The pullup transistor inside the inverter may be viewed in the coordinate system of the inverter as the inverter prototype is being built. We may then wish to view it later, from a coordinate system in which the inverter is a part.

Another reason that a transform must be separate from a VC is that we want a VC to be "the same" (EQ) no matter what coordinate system it is viewed from. The transform, on the other hand, changes with the viewpoint.

A VC is contained in an object called an **instance** which contains a VC and "augmentation" describing the context in which the VC is viewed. In the artwork description of an object, the augmentation is the transformation of the coordinate system of the prototype into the coordinate system of the parent. So in this case: an instance is a VC plus a transform.

Much of this is similar to CIF. Prototypes correspond to CIF symbols and instances correspond to calls to symbols. CIF has no need to distinguish between VCs and instances because objects are never accessed in CIF except to call them.

Plato probably had the idea first, speaking of "ideals" of which real world objects were just crude imitations. The term "virtual copies" comes from Scott Fahlman who used them in much the same way we do.

3.5 Storing and Accessing Information

DPL has functions for storing and accessing information. Some information is necessary to specify a prototype's structure, such as its parts. Other kinds of information may simplify the design process, such as the named points where wires may be attached. A number that is the result of a complicated computation may be computed once and stored. For that matter, totally useless information may be stored if desired.

To store information on a prototype, a "cell" is created on the prototype and given a name. Any LISP object may then be placed in the cell as its "value". The value of an existing cell may be modified by placing a new value into the cell.

If one of the parts of a prototype is named, a cell on the prototype will be created with the part as its value, and the part's name as the name of the cell. This part, an instance, may have as its prototype, another prototype with a named part, and so on. If one asks for the location of a named point or part "deep" inside several levels of parts, the transforms of all the parts must be composed to find the location of the object in the current coordinate system.

DPL provides access functions that are used to extract values from cells. In addition to simply extracting values, the access functions will apply the appropriate transforms to objects that depend on the context in which they are viewed. In Figure 2 we show the transforms that must be applied to point P in successive instances of the type PASS-TRANSISTOR.

DPL also provides functions for locating the corner points of an instance as well as its bounding box and horizontal and vertical dimensions.

3.6 What Happens When Something is Made

When a type is called:

1. The values of the supplied parameters are evaluated.

2. For each parameter defined on the type, a cell is created with that name. At this point all the cells have no value assigned to them, but each contains information

about any constraints that apply to it.

3. The supplied values are then used to fill the cells. If placing a value in a cell allows a constraint to run -- because the added value completes one of the sets of parameters on which a constraint depends -- the procedure for that constraint then runs. If the result is to set a previously unassigned cell, then the process repeats until no more constraints can run.

4. When all supplied parameter values have been placed in cells and all triggered constraints run, any cells that still have no values are given their default values. At this point the system has all the information it will get about how to build and place the prototype.

5. All prototypes built from the same type with the same parameter values will be identical. Therefore, before a new prototype is constructed, all previously constructed prototypes of the type are examined. If one is found with parameters identical to the ones being requested, that prototype is used. Otherwise the maker function of the type is run to construct the new prototype.

6. Once the prototype is made or found, a new instance is constructed from the prototype. The transform given to the instance depends on whether a transform is specified when the type is called. If no transform is specified the instance is given the "identity" transform.

7. The instance may then be named. DPL commands also exist to move it around, rotate it, and extract information from it.

4. BUILDING THINGS

In this chapter we introduce the DPL functions which build objects.

4.1 Creating Types

The most important aspect of the DPL design process is the creation and calling of types. A type is created by a DEFLAYOUT expression, which contains the **maker** function of the type, a procedural description of the structure created when the type is called.

Deflayout takes the form:

```
(DEFLAYOUT <type-name> <param-list>
        <form-1>
        <form-2>
        .
        .
        .
        <form-n>)
```

where <type-name> is the name being given to the new type, and <form-1> through <form-n> constitute the maker function of the type. The forms may include any LISP or DPL expressions. <param-list> is a list in which each element is a pair of the form (<param-name> <value>). <value> may be any LISP object. The only form in this expression that is evaluated is <param-list>.

<param-list> holds information about the parameters that the maker function of the type may take. This information may include parameter names, their default values, and constraints among the parameters. <param-list> may contain all, some, or none of the above information. It may also contain other information about the type besides that used in the type's maker function.

To name parameters and assign them default values, a list of the following form must be included in <param-list>:

```
(PRIMARY-PARAMETERS
        ((<name-1> <val-1>)
         (<name-2> <val-2>)
         .
         .
         .
         (<name-n> <val-n>)))
```

This list is a pair whose first item is PRIMARY-PARAMETERS and whose second item is a list

of parameter names and values. The <name-i> are the names of the parameters that the type's maker function may take. The <val-i> are the parameters' default values.

An example of the use of DEFLAYOUT is:

```
1   (deflayout square-contact '((primary-parameters ((layer 'poly))))
2           (part 'cut rectangle (layer 'cut))
3           (part 'cover rectangle (layer 'metal))
4           (part 'stuff rectangle (layer (>> layer)) (length 4) (width 4)))
```

Here we define a type called SQUARE-CONTACT. It has one parameter: LAYER (the material it will be made of) which is given the default value of POLY. Lines 2-4 are uses of the DPL procedure PART which adds a part to the object being built by calling another type. Making a SQUARE-CONTACT involves creating the parts described in lines 2-4.

4.2 The Type RECTANGLE

The artwork description of an IC design is ultimately decomposable to a collection of rectangles. The DPL type RECTANGLE is the primitive type used to build the artwork descriptions of other types. RECTANGLE specifies the mask layer and dimensions of a rectangular piece of a design.

The primary-parameters of RECTANGLE are LAYER, LENGTH, and WIDTH. LENGTH refers to the Y dimension of the rectangle. WIDTH refers to the rectangle's X dimension. Rectangles may only be built with their sides parallel to the X and Y axes. LAYER refers to the material from which the rectangle will be made.

RECTANGLE is defined with the constraint that if either LENGTH or WIDTH is not specified it will be set to the default size for LAYER. A list of the default layer sizes is included in the glossary.

When using NMOS technology, the available DPL layers are: DIFF, POLY, CUT, METAL, ION and CHANNEL. The CHANNEL layer is used to represent the channel region of transistors. The CHANNEL layer is included in DPL to make it possible to explicitly refer to the active region of a transistor, as well as to separate the source and drain diffusions. In addition, this representation is physically more accurate than simply crossing rectangles of POLY and DIFF, since the channel region of an NMOS transistor actually contains no diffusion. It is possible, however, to design without using the

channel layer.

4.3 Instantiating Types

Once a type has been defined with DEFLAYOUT, instances of that type can be built. PART is the procedure used to instantiate a type. A call to PART creates an instance of a type.

PART takes the form:

```
(PART <name> <type> (<param-1> <val-1>)
                    (<param-2> <val-2>)
                    .
                    .
                    .
                    (<param-n> <val-n>))
```

where <name> is the name given to the instance which PART creates and <type> is the name of the type that will be used to create the instance.

The remainder of the PART procedure consists of the parameters that will be passed to the maker function of <type>. Parameters and values are passed with a "call by keyword" syntax. Each parameter is specified by a pair in which <param-i> is the parameter name and <val-i> is its value. Any primary parameters of the type may be assigned values. Parameters not listed will take their default values. The parameters may be specified in any order. Other information may be placed in the parameter list as well (see Section 5.6).

In the PART command, <type> is not evaluated, <name> is evaluated. For each of the parameter pairs, <param-i> is not evaluated, <val-i> is evaluated.

PART is usually used within the DEFLAYOUT procedure of a type. It is a way of specifying the structure that the type being defined will build. When the type is called, the instance created by the PART command will be included in the structure being built.

The instance built by a PART command is considered to be a "part" of the object in which it is placed. It is the parts of an object which constitute the object's structure. It is the parts of an object which are displayed when the object is viewed. "Partness" is contrasted with other kinds of information stored on an object that do not explicitly specify the object's structure.

An example of a PART procedure is:

```
(part 'trans-1 pass-transistor (channel-length 8) (channel-width 4))
```

This creates an instance of PASS-TRANSISTOR. The instance is named TRANS-1 and built with a channel length of 8 and a channel width of 4 (lambda). It is shown in Figure 5. The type PASS-TRANSISTOR is defined in the introductory example.

Another transistor is created by:

```
(part 'trans-2 pass-transistor (channel-length 8))
```

This produces an instance of PASS-TRANSISTOR named TRANS-2 with a channel length of 8 and a channel width of 2. Since CHANNEL-WIDTH is not listed in this PART procedure, it is set to the default value specified for CHANNEL-WIDTH in the DEFLAYOUT of PASS-TRANSISTOR. See Figure 5.

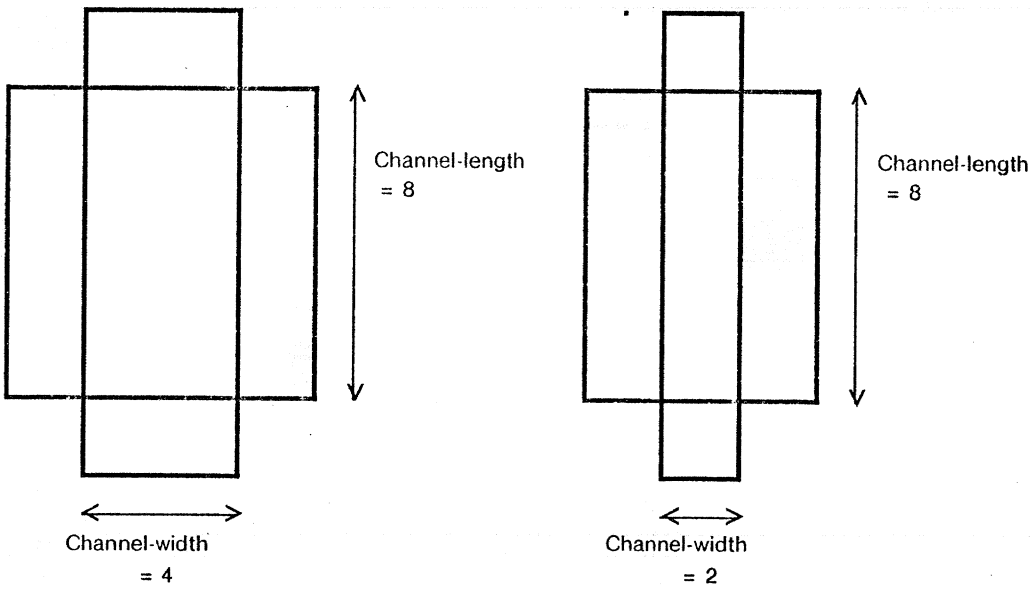


Figure 5 -- Two instances of the type PASS-TRANSISTOR

4.4 The Structure Built by a Type

When a type is called with PART, the LISP variable *ME* (pronounced "star-me-star") is bound to the representation of the structure being built by the type's maker function. In other words, while a structure is built, it is attached to *ME*. This means that when a type is called, *ME* will be bound temporarily to the type's parts as they are built. When the maker function of the type is finished, the structure in *ME* is placed in an instance. This instance may very well be a part of another structure.

Since a DEFLAYOUT is a specification of a type, when writing a DEFLAYOUT, one can think of the structure that will be built when the type is called as *ME*. *ME* acts as the temporary name of the structure while it is being built, so that it is possible for information to be added to and retrieved from the structure. We have already seen one way in which information is added to the structure built by a type -- the PART command makes the instance it produces a "part" of *ME*.

4.5 Naming Things

In the course of the design process, it is often useful to store information on *ME*. Such information may be useful for constructing *ME* or it may be useful later, when the instance is complete, for building other objects.

The function SET-MY stores a piece of information on *ME* and gives it a name.

```
(SET-MY <name> <value>)
```

stores <value> on *ME* and names it <name>. <value> may be any LISP object. SET-MY evaluates both its arguments. The function SETQ-MY is identical to SET-MY except that SETQ-MY does not evaluate <name>. Thus SETQ-MY is often more convenient to use, and we will use it in most examples.

The values stored by SETQ-MY may be accessed by the DPL access functions described below.

For example, within the DEFLAYOUT of a type, the locations of several parts may depend on the height of the VDD bus. (SETQ-MY VDD-HEIGHT 20) will store the value 20 on *ME* and will name it VDD-HEIGHT. It is possible to access this value later in the

DEFLAYOUT to specify the locations or dimensions of parts of *ME*.

4.6 Accessing Parts and Parameters

Within a DEFLAYOUT procedure it is often necessary to refer to parts of *ME* (or parts or parameters of those parts), as well as to information placed on *ME* by SETQ-MY. Accessing such information can be done with the function >> (called "arrow-arrow").

```
(>> <thing-1> <thing-2> . . . <thing-n>)
```

will retrieve the value of <thing-1> which belongs to <thing-2>. . . which belongs to <thing-n> which belongs to *ME*. (None of the elements in an >> form are evaluated unless they are non-atomic.)

For example, suppose we are writing the DEFLAYOUT procedure for a type named REGISTER which includes as a part an instance named SHIFT-CELL. SHIFT-CELL, in turn, contains as a part an instance of PASS-TRANSISTOR named TRANS-3. To find the value of the CHANNEL-LENGTH of TRANS-3 of SHIFT-CELL of REGISTER, we write:

```
(>> CHANNEL-LENGTH TRANS-3 SHIFT-CELL).
```

(This expression is read: "The channel-length of trans-3 of shift-cell of *ME*.")

To access a part or parameter of *ME*, or a something stored on *ME* by SETQ-MY, we write:

```
(>> <thing>)
```

For example, (SETQ-MY VDD-HEIGHT 20) followed by (>> VDD-HEIGHT) will return 20.

>> is used to access the values of the parameters with which the type was called. This is the way in which the parameters direct the construction of an instance. The DEFLAYOUT of PASS-TRANSISTOR has two parameters, CHANNEL-WIDTH and CHANNEL-LENGTH, (see Introductory Examples). The following command is included in the DEFLAYOUT of PASS-TRANSISTOR:

```
(part 'diff-piece rectangle
        (layer 'diff)
        (length (+ (>> channel-length) (* 2 *diff-overhang*)))
        (width (>> channel-width)))
```

This command specifies the values for the parameters LENGTH and WIDTH of the type RECTANGLE by accessing the values that CHANNEL-LENGTH and CHANNEL-WIDTH are assigned when PASS-TRANSISTOR is called. If we call PASS-TRANSISTOR in the following way:

```
(part 'trans-4 pass-transistor
      (channel-length 4)
      (channel-width 100))
```

CHANNEL-LENGTH and CHANNEL-WIDTH are assigned to the values 4 and 100. These values are then used when the type RECTANGLE is called by the maker function of PASS-TRANSISTOR.

4.7 The General Access Function

The general DPL access function is THE.

```
(THE <name> <thing>)
```

finds the information named <name> on <thing>. (Both arguments are evaluated.) THE knows about all DPL structures and may be used to access parts and parameters of all of them. For example, one may use the following to find the PRIMARY-PARAMETERS of a type named A-TYPE:

```
(the 'primary-parameters 'a-type)
```

THE may also be used to access named information from *ME*. For example, the following two expressions will return the same value:

```
(the 'channel-length *me*)
(>> channel-length)
```

In fact, >> is defined in terms of THE. The expression

```
(>> mumble fumble grumble)
```

expands to:

```
(the 'mumble (the 'fumble (the 'grumble *me*)))
```

>> is the most useful way to get information from *ME* and is usually used in DEFLAYOUT expressions. THE is used to get information from objects other than *ME* and thus is most useful when interacting with LISP while debugging designs.

When accessing information from parts of objects THE and >> transform the objects correctly so that the object is always viewed in the current coordinate system.

4.8 Additional Features of DEFLAYOUT

There exist additional capabilities of DEFLAYOUT which are not used as often as those described above. This section will introduce these capabilities.

4.8.1 Supertypes

DEFLAYOUT allows one to make new types by adding to old ones. <type-name> may be written as a list of two elements, the first name referring to the name of the new type, the second name referring to the name of the old type. A new type is made which is identical to the old type, but with the addition of whatever information is included in the new DEFLAYOUT procedure. The previously defined type is called a **supertype** of the new one. The new type has all the parts and parameters of the supertype, yet it may be given additional parameters as well as additional parts.

An example of the use of a supertype is a depletion-mode transistor. It is identical to a normal transistor with the addition of a rectangle of ion implantation. The definition of a depletion-mode transistor is:

```
(deflayout (rect-d-fet rect-fet) '()
           (part 'implant rectangle
                 (layer 'ion)
                 (length (+ (>> channel-length) (* 2 *ion-overhang*)))
                 (width (+ (>> channel-width) (* 2 *ion-overhang*)))
                 (center (>> center channel))))
```

In this example, RECT-D-FET is the name of the new type. RECT-FET is the name of the type that constructs "normal" transistors -- it will be the supertype of RECT-D-FET. RECT-D-FET includes all the parts and parameters of RECT-FET with the addition of the part created in the example above, a rectangle of ion implant. Note that our new type, RECT-D-FET, includes no parameters in its parameter list. However it actually does have parameters -- the parameters of RECT-FET (which happen to be a channel-length of 2 and a channel-width of 2). RECT-D-FET could have been given additional parameters, but here we have limited its parameters to those of RECT-FET.

A supertype is used if a type being defined differs only slightly from a pre-existing type, and it is desired that the new type have the same or very similar parts and parameters as the supertype. The advantage to using the supertype construction rather than calling the supertype as a part is that only the differences between the new type and the supertype need be specified.

4.8.2 Additional Parameters to DEFLAYOUT

The values of a type's primary parameters determine the structure of its instances. DEFLAYOUT may also be given parameters which specify information about the type other than that which directly determines the structure of its instances.

The **auxiliary-parameters** of a type is a list of some of the names used to store information on the instances of the type. For example, if a point named CONNECTION-PT is named with a SETQ-MY command inside a DEFLAYOUT, the name CONNECTION-PT may be included in the auxiliary-parameters of the type. Auxiliary-parameters are usually used by programs which manipulate types, such as the constraint system described in Chapter 7. Auxiliary-parameters are included in a type by placing a list of the following form in the <param-list> of a DEFLAYOUT:

```
(AUXILIARY-PARAMETERS (<name1> <name2> . . .))
```

where the <namei> are names that will be assigned to things in the body of the DEFLAYOUT.

Other information may be specified in a DEFLAYOUT parameter-list. Any pair of the form:

```
(<name> <value>)
```

in the <param-list> of a DEFLAYOUT will cause the information in <value> to be stored on the type and named. Documentation, version numbers, device parameters are all kinds of information one might want to store on a type.

5. PLACING THINGS

When defining a type it is necessary to specify both the structure of its parts and their location. This chapter will explain the DPL functions, data structures, and variables used to specify placement.

5.1 Coordinate system

Every structure specified by a DEFLAYOUT has its own coordinate system. Each PART procedure in the DEFLAYOUT will place the instance it creates at a certain position in the coordinate system of *ME*. Unless the PART command is given explicit placement information, the part is placed with its origin at the origin of *ME*.

Since parts are themselves calls to types, each part is constructed with its own coordinate system in which its parts are placed. However, within a DEFLAYOUT procedure all parts of parts are transformed when accessed by THE or >> to their positions in the coordinate system of *ME*.

5.2 Points

Points may be created by the function PT.

```
(PT <x-coord> <y-coord>)
```

creates a point with the given coordinates in *ME*. For example, `(pt 4 3)` creates a point with the coordinates (4,3). PT is often used with the DPL placement functions which will be explained in this chapter.

The X and Y coordinates of a point may be accessed by the functions >> or THE. If the expression

```
(setq-my connection-point (PT 5 6))
```

is used in a DEFLAYOUT, the function

```
(>> x connection-point)
```

will return 5.

DPL provides a number of functions which deal with points. They include functions which construct new points from existing ones, a function which tests

values for "pointness", and functions dealing with the placement of points. These are explained in the glossary.

5.3 Implicit Parameters

Every instance possesses pre-named information useful for placing the instance. This information is called an instance's **implicit parameters** because it is never explicitly placed on the object, yet it may be accessed.

Probably the most useful of the implicit parameters are **corner-parameters** and **apparent-corner-parameters**, pre-named points on every object which contain the locations of an object's corners, center-side-points, and center. The corner-parameters are:

```
top-left        top-center      top-right
center-left     center          center-right
bottom-left     bottom-center   bottom-right
```

The corner-parameters refer to the locations held by the appropriate points in the coordinate system of the object before the object had been transformed. If the instance has been rotated or mirrored the corner-parameters will be transformed as well. Thus, for example, the TOP-CENTER of an instance that has been rotated 90 degrees counterclockwise will be on the left side of the instance.

It is often more useful to refer to the points on an instance which indeed appear to be the top-right, bottom-center, and so on. The apparent-corner-parameters are provided for this purpose. The names of the apparent-corner-parameters are obtained by concatenating "apparent-" with the names of the corner-parameters.

Figure 6 shows how the corner-parameters and apparent-corner-parameters of an instance transform when the instance is transformed.

The values of corner-parameters and apparent-corner-parameters may be accessed by >>. For example,

```
(>> apparent-bottom-right gate-poly pulldown inverter-1)
```

will access the APPARENT-BOTTOM-RIGHT of the GATE-POLY of the PULLDOWN of INVERTER-1 of *ME*.

Corner-parameters and apparent-corner-parameters are useful for the

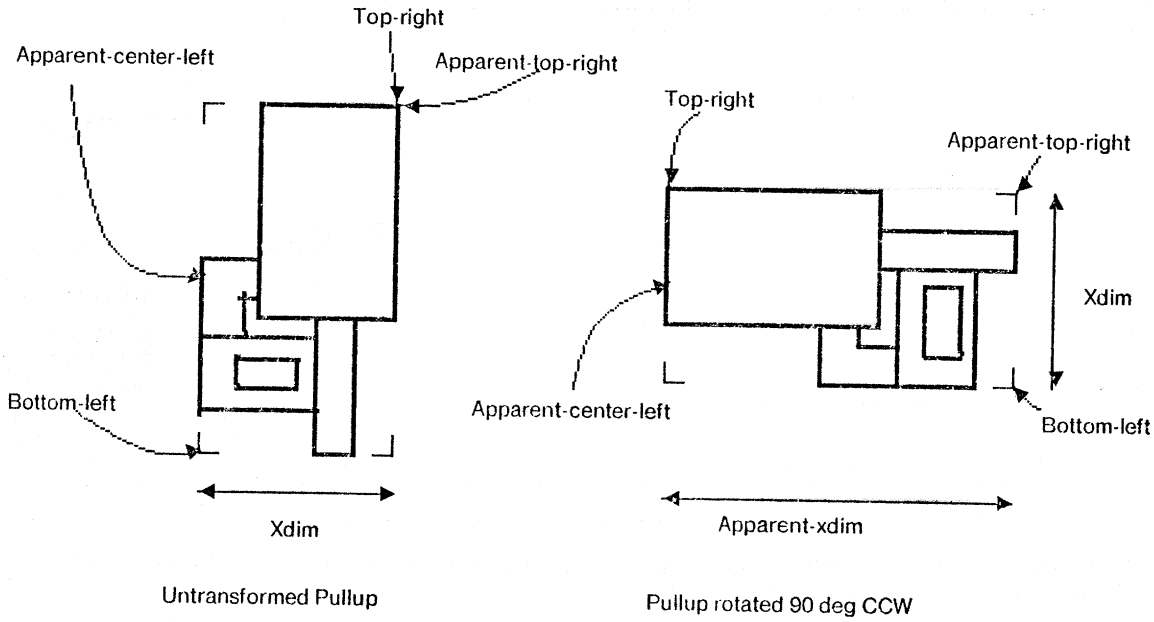


Figure 6 -- Some implicit parameters of an Instance

placement of objects. They allow objects to be placed by reference to other objects, rather than by their numerical coordinates. Corner parameters contribute to the flexibility of the design because relative placement allows one to change or move a part without changing the specification of the objects near it. Corner parameters are also useful in that they reveal the reason for the particular positioning of a part, far more than a numerical parameter.

Other implicit parameters of every instance are BOUNDING-BOX, ORIGIN, XDIM, and YDIM. The BOUNDING-BOX of an instance consists of a representation of two diagonally opposite points on an imaginary rectangle surrounding the instance. It is computed by taking the extreme values of the coordinates of the bounding-box of the parts of the instance, or the corner points if the instance is a RECTANGLE.

The ORIGIN of an instance is the point (0, 0) in the coordinate system of the instance. Accessing the ORIGIN of a part of *ME* or of a part of a part will give the point in the coordinate system of *ME* that the origin of the part occupies.

The XDIM of an instance is the distance between the CENTER-LEFT and CENTER-RIGHT of the instance. The APPARENT-XDIM is the distance between the APPARENT-CENTER-LEFT and APPARENT-CENTER-RIGHT of the instance. YDIM and APPARENT-YDIM give the corresponding distances between the TOP-CENTER and BOTTOM-CENTER.

5.4 Translation

One way to place parts within a DEFLAYOUT procedure is to make the part with a PART procedure, and then move the part to the desired location. When a part is created it is placed at the origin of *ME*. It may then be translated to the desired location by the DPL function ALIGN.

ALIGN moves an object to another location, maintaining its orientation (without rotating or mirroring it).

ALIGN takes the form:

```
(ALIGN <object> <ref-point> <target-point>)
```

where <object> is the thing to be moved, <ref-point> is a point, usually on the object, and <target-point> is the point to which <ref-point> is to be relocated. <object> will be moved so that <ref-point> is at <target-point>. All three arguments to ALIGN are evaluated.

For example,

```
(align (>> contact-1)
       (>> top-center contact-1)
       (>> bottom-center source-diffusion trans-1))
```

will move CONTACT-1 so that its TOP-CENTER is at the BOTTOM-CENTER of the SOURCE-DIFFUSION of TRANS-1. This example is illustrated in Figure 7.

5.5 Unitary Transforms

Instances may be rotated or mirrored by the unitary transform functions. Each unitary transform function corresponds to an element of the group of symmetries of the square. Each unitary transform function takes an instance as an argument and transforms the instance as described below. Rotation and mirroring is performed

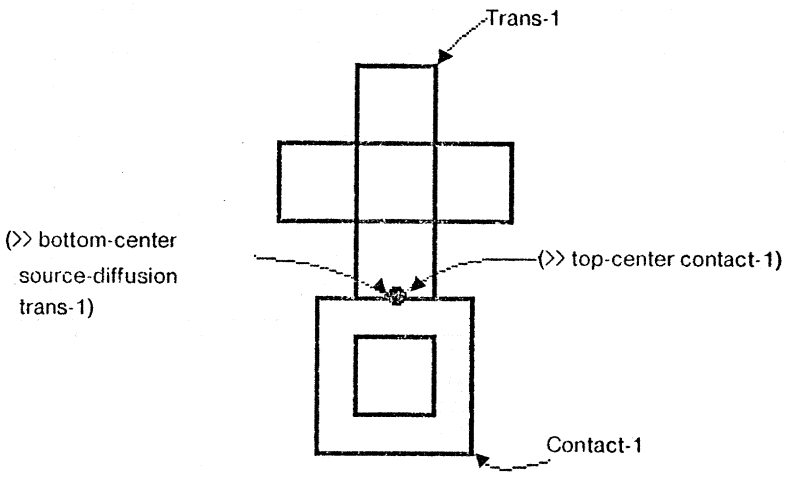


Figure 7 -- Placement by alignment

about and across the origin of *ME*.

IDENTITY Performs no rotation or mirroring of the instance.

ROT90 Rotates the instance 90° counterclockwise.

ROT180 Rotates the instance 180°.

ROT270 Rotates the instance 270° counterclockwise.

NEGX Negates the X coordinates of the instance. (Mirrors the instance across the Y axis.)

NEGY Negates the Y coordinates of the instance. (Mirrors the instance across the X axis.)

INT-POS The composition of ROT90 followed by NEGX. (The values of the coordinates are interchanged.)

INT-NEG The composition of ROT90 followed by NEGY. (The values of the coordinates are interchanged and negated.)

For example,

```
(negx (>> trans-7))
```

will mirror the instance named TRANS-7 across the Y axis.

Note that each application of a unitary transform function to an instance

composes the new transform with the previous transform of the instance.

If both ALIGN and unitary transform functions are to be applied to an instance, it is usually more convenient to apply the unitary transform functions before translating. This is because translation is usually used to place the object in its final location. If one translates before rotating or mirroring, it is very difficult to predict where the instance will be ejected.

5.6 Placement by Parameter

We have seen that objects may be created and then moved around. It is also possible to place objects by supplying parameters to the PART procedure.

The translation of an object may be specified by including as a parameter to PART the name of a reference point on the object and its target location. The points most commonly used as translation parameters are corner-parameters and apparent-corner-parameters. For example,

```
(part 'contact-1 square-contact
      (layer 'diff)
      (top-center (>> bottom-center source-diffusion trans-1)))
```

will do the same thing as

```
(part 'contact-1 square-contact (layer 'diff))
(align (>> contact-1)
       (>> top-center contact-1)
       (>> bottom-center source-diffusion trans-1))
```

Note that when passing the name of a point to PART, as a reference point on the object, >> is not necessary. (Like the point TOP-CENTER in the example above.)

Points other than corner-parameters and apparent-corner-parameters may also be used as reference points. For example, if CONNECTION-POINT was a named point on an inverter,

```
(part 'inverter-1 inverter
      (connection-point (>> bottom-left poly-rect)))
```

would place INVERTER-1 so that CONNECTION-POINT was at the BOTTOM-LEFT of POLY-RECT.

Rotation and mirroring can also be performed by supplying parameters to PART. PART may be called with a parameter whose name is XFRM and whose value is the name of one of the unitary transform functions, in which case the unitary transform function is applied to the instance.

If both a translation parameter and a unitary transform are passed as parameters to a PART procedure, they may be listed in any order. The unitary transform will be applied to the part before it is translated.

For example,

```
(part 'bc butting-contact
      (xfrm 'rot90)
      (bottom-center (>> center-left gate-poly trans-1)))
```

will rotate butting contact BC 90°, and then place it so its BOTTOM-CENTER is at the CENTER-LEFT of the GATE-POLY of TRANS-1 of *ME*.

The two methods of placing objects -- placement by function and placement by parameter -- each have advantages in certain situations. Placement by parameter is less wordy than placement by function. However placement by function is often easier to read, especially when either the reference point or the target point is a complex formulation, or when one is passing many parameters to the PART procedure.

For example,

```
(part 'iv2 inline-inverter
     (enhancement-width 10)
     (enhancement-length 3))
(int-neg (>> iv2))
(align (>> iv2)
       (>> apparent-top-right iv2)
       (pt (- (>> x top-left poly-contact) 2)
           (- (>> y bottom-right VDD-contact1) 3)))
```

is equivalent to

```
(part 'iv2 inline-inverter
     (enhancement-width 10)
     (enhancement-length 3)
     (xfrm 'int-neg)
     (apparent-top-right  (pt (- (>> x top-left poly-contact) 2)
                              (- (>> y bottom-right VDD-contact) 3))))
```

5.7 Invoke

Many of the objects created during the course of a design are meant to "line up" or "fit" other objects. The driver for a PLA column, for example, must be the same width as the column, and its outputs must match up with the inputs of the column. The design of such types is made easier if an instance of the other object is available to match up with the one being made.

The DPL function INVOKE is identical to the PART function but the instance that is created is not made a "part" of the new object. The invoked instance may be moved

around and the information inside it may be accessed, but the structure of the new type will not include the structure of the invoked type.

Example:

```
(deflayout pla-driver ()
        (invoke 'column pla-column-cell)
        (part 'vdd-bus rectangle
                (layer 'metal)
                (center-left (>> center-left column))
                (center-right (>> center-right column)))
        <and more forms of the maker function>)
```

This shows how a type is invoked and then used to specify the dimensions of another cell. Note that the PLA-COLUMN-CELL will not be a part of the PLA-DRIVER, but the information in the column cell can be used as the driver is being built.

5.8 *LIST*

Many DPL objects keep lists of other kinds of objects. Some of the lists contain objects that are "transformable", like instances or points. In this case it is useful to have them transformed into the current coordinate system. The DPL structure that allows this to be done is called a *LIST*. If a *LIST* is stored in a cell on an object, each element of the *LIST* will be transformed as it is "brought out" of the structure with THE or >>.

A *LIST* is made out of a LISP list by the function MAKE-LIST-OF, the list inside a *LIST* is obtained with the function THE-LIST-OF. For example:

```
(deflayout a-useless-type ()
        (part 'moe rectangle (layer 'poly))
        (part 'larry rectangle (layer 'diff))
        (part 'curley rectangle (layer 'metal))
        (setq-my stooges (make-list-of
                          (list (>> moe)
                                (>> larry)
                                (>> curley)))))
```

creates three rectangles of default sizes on top of one another and a *LIST* containing them is made. If the following code is in the maker function of another type:

```
(part 'meanies a-useless-type (bottom-left (pt 100 100)))
```

asking for `(the-list-of (>> stooges meanies))` gives the list of the three instances, each transformed correctly.

6. THE DPL WIRING SYSTEM

Most of the rectangles in a large design serve to connect objects. This chapter introduces the DPL wiring system which consists of special procedures for creating and manipulating such rectangles.

Wires are specified by indicating the layer and width to be used, and the path the wire is to follow. Wires may change layers, in which case the wiring system will automatically insert the correct contact, or make any number of side branches.

Wires are made by placing rectangles. In the DPL wiring system, wires are placed by aligning new rectangles of a specified width and layer so that they join previously placed rectangles. The length of each rectangle is determined by the path the wire is to follow.

To join rectangles, wires make use of a special kind of point called a **connection-point** (CP). A CP is a data structure which keeps track of the layers of the wires connecting to it as well as its coordinates. The **current-CP** of a wire is a CP containing the point where the next rectangle is to be attached. The current-CP may be thought of as the current position of the wire.

Wires are considered to be parts of *ME*. During the construction of a wire, the variable *THE-WIRE* will be bound to the wire being constructed. Wires are instances of the special type WIRE. Like all instances, they have implicit parameters which may be useful when placing other parts of *ME* near wires. The bounding-box of a wire is computed by finding the extreme values of the coordinates of its component rectangles.

6.1 Wire System Commands

The most common way to use the wiring system is with the procedure WIRE. WIRE is usually used within a DEFLAYOUT and takes the following form:

```
(WIRE <name> ..<list-of-wire-commands>..)
```

WIRE names the wire it builds <name>. <name> is evaluated. If <name> is NIL the wire is given no name. The first and last CP's of the wire are named START and END. The

construction of the wire is controlled by the following commands which may occur inside a WIRE procedure. (Each wire command evaluates its argument.)

(RUN-LAYER <layer>) sets the layer of the wire. If the wire had already placed rectangles of a previous layer, the next time a rectangle is placed the appropriate contact is made at the wire's current-CP. The RUN-LAYER command sets the width of the wire to the default width for that layer (see Glossary). A RUN-LAYER command must come before any of the commands that actually place rectangles.

(RUN-WIDTH <width>) sets the width of the wire. RUN-WIDTH must come after the RUN-LAYER command if it is to affect the width of that layer.

(FROM <place>) sets the current-CP to <place>. <place> may be a point, a CP or another wire. If <place> is a wire, the last CP of the wire is used. A FROM command must precede any of the commands which actually place rectangles. The FROM command may be used at other places inside a WIRE command in which case it moves the current-CP from its previous location to <place> without placing any rectangles.

(TO-X <place>)

(TO-Y <place>)

These commands place rectangles. <place> may be a number, in which case it is interpreted as a coordinate of the destination of the wire in the specified direction. If <place> is a point or a CP, the appropriate coordinate will be extracted and used. Like all commands that place rectangles, this may cause a contact to be dropped if the layer has been changed since the last rectangle was placed. The current-CP is then updated to the new point.

(TO-PT <place>) places a rectangle which extends from the current-CP to <place>. <place> may be either a point or a connection-point. One of its coordinates must be equal to a coordinate of the current-CP.

(+X <number>)

(+Y <number>)

(-X <number>)

(-Y <number>)

Each of these commands extends a rectangle from the current-CP the specified distance in the specified direction.

(JOG-X <place>)

(JOG-Y <place>)

These commands extend the wire to the point or CP specified by <place> by running first in the specified direction and then in the other. For example, if the current-CP is at point (0,0) and the command (JOG-Y (PT 10 10)) is given, the result is a rectangle from point (0, 0) to point (0, 10), and then a rectangle from point (0, 10) to point (10, 10).

(SAVE-CP <name>) names the current-CP of the wire <name>, and stores it so it may be accessed later.

(RESTORE-CP <name>) moves the current-CP of the wire to the specified connection-point.

(DROP-CONTACT) places a square contact of the current layer at the current point.

In addition, any LISP or DPL forms may be placed in a WIRE form. It is often useful to use LISP conditionals to direct the construction of a wire. Any DPL commands that create or name structure (such as PART and SETQ-MY), except the above wiring commands, will affect the object the wire is part of, not the wire itself.

6.2 Wire System Example

The following is an example of the use of the DPL wire system. In the example we assume that *ME* has been given the parts TRANS-1 and TRANS-2 which are pass-transistors, and CONT-1 which is a poly-to-metal contact. The example is illustrated in Figure 8.

```
(wire 'wire-ex
      (run-layer 'diff)
      (from (>> bottom-center source-diffusion trans-1))
      (-y 4)
      (to-x (>> center cont-1))
      (save-cp 'fork)
      (jog-x (>> bottom-center source-diffusion trans-2))
      (restore-cp 'fork)
      (run-layer 'poly)
      (to-y (>> top-center cont-1)))
```

Our wire, WIRE-EX, begins as a wire of diffusion at the bottom-center of the source diffusion of TRANS-1. After running 4 lambda down, it runs to the X coordinate of the contact CONT-1. The current-CP is then saved and named FORK because we will

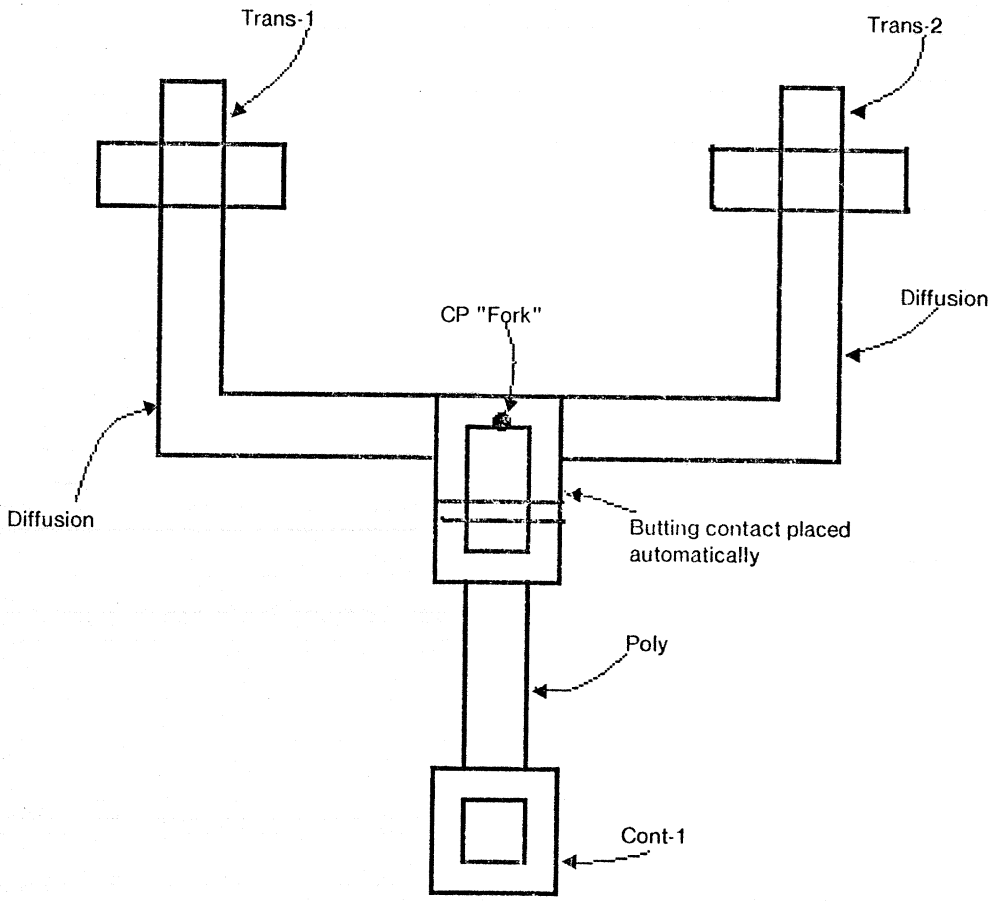


Figure 8 -- Wire System Example

want to resume wiring from there later. The wire is then run with a JOG-X command to the bottom-center of the source diffusion of TRANS-2. The saved CP FORK is then restored, and the layer is changed to poly. Finally, we run the wire from the restored CP to the top of CONT-1. Note that the wiring system automatically places a butting contact where the layer changes.

6.3 External Wire Commands

Wires may be given additional rectangles after they have been made. The commands which do this correspond to the above commands except that the names have "wire-" concatenated to the front. Each external wiring command takes two arguments: the first is the wire to be manipulated, the second is the same argument as that passed to the corresponding internal wiring command. For example:

```
(WIRE-JOG-X <wire> <place>)
```

It is possible to build a wire entirely from the "outside". An empty wire may be created by using WIRE with a name and no other wire commands. The wire can then be given rectangles with the external wiring commands.

6.4 Connection Points

CPs named during the construction of a wire may be accessed during the construction of the wire. The special symbol *CURRENT-WIRE*, when used as the last symbol in an >> command, allows one to access a named CP of the current wire. For example:

```
(>> fork *current-wire*)
```

After a wire is complete, a named CP may be accessed as if it were named information on the wire. In the wire system example above,

```
(>> fork wire-ex)
```

will access the CP FORK.

The point which is the location of a CP is obtained with the function PT-CP. For example,

```
(pt-to-left (pt-cp (>> fork wire-ex)) 5)
```

will return the point which is 5 to the left of the FORK of WIRE-EX.

One may access the X and Y coordinates of a CP as if it were a point:

```
(>> x fork wire-ex)
```

It should be noted that when the wiring system automatically places a butting contact at a CP, the butting-contact is placed with its center at the CP. This means that other poly or diffusion wires connecting to the butting contact may not end directly at the CP, but at a point to one side of it. In the example in figure 8, if we were to attempt to run a horizontal poly wire from the CP named FORK, the poly actually must connect 1 lambda below the CP -- or else it would partially cover some diffusion and make an illegal butting contact. The wiring system automatically adjusts the endpoints of the wires to make legal butting contacts, but in so doing may cause the wires to begin and end on points other than the specified CP. One should use the JOG commands near butting contacts to allow the system to bend wires when such situations develop.

7. CONSTRAINTS

In many cases the parameters of a type are related. Often some parameters may be derived from others. The DPL language uses **constraints** to allow the user to specify only those parameters necessary to "constrain" the rest.

For example, a standard transistor's resistance is determined by the ratio of its channel length to its channel width. If the ratio is `Z`, the channel length `L`, and the channel width `W`, then `Z = L/W`. But also `ZW = L` and `W = L/Z`. The complete specification of the transistor may be accomplished by setting only two of the three parameters.

Constraints do more than allow one to specify fewer parameters. Much of the computation necessary for determining the layout of an object can be done with constraints. For example, it may be necessary to fix a transistor's ratio and width, allowing its length to vary. In another case one may want to specify the length and the width of a transistor and later ask for its ratio. Or, one could specify all three and let the program complain if they were set inconsistently.

In some cases it is useful to specify constraints among parameters which are not used to build the instance. For example, the width of a cell may be determined by the distance between two control lines. We may write a constraint between the positions of the control lines and the width of the cell. If the points where the control lines enter the cell are named as auxiliary parameters of the type, it is possible to express the width of the cell in terms of these positions.

7.1 Using Constraints

To specify constraints between parameters of a type, the following list must be included in the `<param-list>` of the type's `DEFLAYOUT`:

```
(CONSTRAINTS ((<c-1> <param-1> <param-2>)
              (<c-2> <param-3> <param-2> <param-4>)))
```

The `<c-i>` are names of constraints. Several constraints have already been defined (see library); others may be defined by the user. The `<param-i>` are the names of primary or auxiliary parameters of the type among which the constraint is to be applied. A single parameter may be mentioned in several constraints.

A useful constraint is C* which constrains three arguments so that the first is the product of the second and third. C* may be used to specify the constraints of a transistor:

```
(deflayout 'transistor-1 transistor
        '((primary-parameters ((channel-width 2)
                               (channel-length 2)
                               (ratio 1)))
          (constraints ((c* channel-length ratio channel-width))))
        ...forms of the maker function...)
```

This example uses only one constraint and applies it to three primary-parameters.

If it is desired that one of the arguments be a constant, that value (usually a number or a point) may be given to the constraint directly, instead of passing the name of a parameter. For example, one may wish to maintain a ratio of 4 between an inverter's pullup-ratio and its pulldown ratio. This could be accomplished by:

```
(c* pullup-ratio 4 pulldown-ratio)
```

If an attempt is made to specify an inconsistent set of parameter values to a type defined with constraints, an error will be signaled.

7.2 Defining Constraints

Constraints are defined by the command DEFCONSTRAINT. A constraint must be defined before it can be used in a DEFLAYOUT.

DEFCONSTRAINT takes the form:

```
(DEFCONSTRAINT <name> <arglist>
          (<arg-1> (<arg-2> <arg-3>) <procedure>)
          (<arg-2> (<arg-1> <arg-3>) <procedure>))
```

where <name> is the name used to call the constraint. <arglist> is a list of the variables that will be bound to the values specified when the constraint is called. The forms following <arglist> each begin with the name of one of the variables. This is followed by a list of the variables that can be used to compute the first variable if they all have been given values. The last item is the actual procedure that can be run, using those variables, to compute the value of the first variable. The best way to clarify DEFCONSTRAINT is to present the definition of a simple constraint:

```
(defconstraint c* (prod m1 m2)
  (prod (m1 m2) (* m1 m2))
  (m1 (prod m2)
      (if (= m2 0)
          'bail-out
          (// (float prod) m2)))
  (m2 (prod m1)
      (if (= m1 0)
          'bail-out
          (// (float prod) m1))))
```

This is the C* constraint discussed above. It takes three arguments, and constrains the first to be the product of the other two. If the procedure evaluates to the string BAIL-OUT, the constraint will not attempt to set a value.

Constraints may be used between primary-parameters, auxiliary-parameters and constants. It is also possible to specify corner and apparent-corner parameters as arguments to a constraint. In fact, the type RECTANGLE is defined this way. One may specify the bottom-left and upper-right of a rectangle, for example, and the constraint will determine the proper length and width to make the rectangle.

8. REPLICATORS

Many designs contain situations in which a small number of objects are replicated in highly regular arrangements. A row of identical register cells may be used to store the result from a column of bit-slice adders. A PLA or a shifter is typically made of a two-dimensional grid of identical parts.

Replicators in DPL allow the user to create such structures. They also make use of the regular nature of the structures to represent them efficiently. All that is stored is the list of instances which appear in the replicator and a function that computes the transform of an instance for a given set of coordinates. The functions examining the structure of the replication put this information together and produce the parts of the replication. When examined with THE, a replication "looks like" it has many parts, while actually only a few instances are stored.

The most useful replicators build rows, columns or 2-dimensional grids of instances. These replicators are defined in the library. It is also possible to define replicators for special purposes.

A "replicator" is similar to a type. It holds the procedure to construct a replicated set of instances. A "replication" is what a replicator produces -- an instance that looks like it has many regularly placed parts.

8.1 Calling Replicators

If a replicator has been defined, it is called with the command REPLICATE:

```
(REPLICATE <name> <replicator-type> <dimensions> <list-of-instances>
                        ...<other-parameters>...)
```

This command is similar to the PART command except for the <dimensions> and <list-of-instances> arguments, which are both evaluated. It calls the replicator named <replicator-type> and gives it the name <name> in *ME*. The <dimensions> argument is a list of the values of the dimensions the desired replicator. The <list-of-instances> is a list of instances that will be used in the replication. These instances may have been INVOKED or actually made as parts of *ME*. In either case, after the REPLICATE command, they will be removed from wherever they were before and treated as parts of the replication.

The replicator ROW takes a single instance and places a row, spacing the parts using the sum of the parameters PITCH and SPACING. If the parameter PITCH is not specified in the call to REPLICATE, the instance is checked to see if it has an H-PITCH parameter. If so, that value is used. Otherwise, the APPARENT-XDIM is used. (Needless to say, this is done with constraints.) The SPACING parameter defaults to zero. The following commands

```
(invoke 'p1 rectangle (layer 'poly) (length 10) (width 10))
(replicate 'r1 row (10) (list (>> p1))
                (spacing 1))
```

give a replication of ten poly squares, spaced one apart.

8.2 Accessing Replications

The parts of a replication may be accessed by the form

```
(*REP <coord-1> <coord-2>)
```

within a call to THE or >>, where the <coord-i> are the coordinates desired. The arguments are evaluated. The coordinates start with (0 0). If the replication is one-dimensional, the second coordinate may be unspecified, or else must be 0. Thus

```
(>> (*rep 3) r1)
```

gives the fourth element in the ROW produced above. The symbol *REP-FIRST is used to access the (0 0) element in a replication, and *REP-LAST accesses the element in the replication with the largest coordinates. These are useful when the size of a replication is a parameter and one wants to refer to the end or beginning of it. For the above replication then, the following forms are equivalent:

```
(>> *rep-first r1)
(>> (*rep 0) r1)
```

and

```
(>> *rep-last r1)
(>> (*rep 9) r1)
```

Besides their parameters, replications contain additional information. The INSTANCE-LIST of a replication contains the list of instances used when the REPLICATE command was called. The form (*REP-INSTANCE <n>) is used to get the nth element of the INSTANCE-LIST of a replication. The cell MAX-DIMS of a replicator contains a list of the maximum-dimensions of the replication. MAX-DIMS is the same as the argument <dimensions> to REPLICATE.

8.3 Defining Replicators

Replicators are a special kind of type. They may be defined with constraints and parameters. A replicator is defined as follows:

```
(DEFREPLICATOR <name> <coordinates>
                     <param-list>
               <body>)
```

The <name> and <param-list> are as in DEFLAYOUT: <name> will be the name of the replicator and <param-list> its parameter-list. The <coordinates> argument is a list of one or two symbols. The length of <coordinates> will be the dimension of the replication. <body> is a collection of LISP forms that are called when the replication is accessed. The functions in <body> determine the instance and transform of the "part" of the replication determined by the coordinates. As in DEFLAYOUT, only <param-list> is evaluated.

The call (*REP <i> <j>) binds the variables in the <coordinates> list to <i> and <j>, and binds *ME* to the replication. Then <body> is evaluated.

<body> must call the functions:

```
(REPLICATOR-INSTANCE <instance>)
(REPLICATOR-TRANSFORM <unitary-part> <x> <y>)
```

The argument to REPLICATOR-INSTANCE is the instance at the position specified by the value of the coordinates. It is obtained by a call to *REP-INSTANCE and is determined by the values of the coordinates and perhaps some of the parameters.

REPLICATOR-TRANSFORM takes as arguments the three parts of the transform to be composed with the transform of the argument to REPLICATOR-INSTANCE. This yields the transform of the part of the replication. THE and >> use the arguments to these functions to construct and return the correct instance.

The following is the definition of the ROW replicator:

```
(defreplicator row (i)
        '((primary-parameters
               (pitch
               (spacing 0)))
          (constraints
               (h-pitch instance-list pitch)))
        (replicator-instance (>> (*rep-instance 0)))
        (replicator-transform 'identity
                              (* i (+ (>> pitch) (>> spacing)))
                              0))
```

The H-PITCH constraint acts on the INSTANCE-LIST of a replication to find the horizontal-pitch as described above. ROW takes only one instance and all its parts have the same unitary-transform and Y position. The X position is determined by the product of the coordinate and the sum of the parameters PITCH and SPACING. Thus for the ROW replication created above, the call `(>> (*rep 4) r1)` returns an instance of a 10 by 10 poly rectangle with a transform of 44 in the X direction.

9. USING DPL

In this chapter we discuss how the DPL language can be used to construct a project. We explain what the various kinds of DPL objects look like when they are displayed on a terminal and how the designer may use DPL at a terminal to probe the structure of his design. We also discuss the functions that translate between DPL and CIF.

9.1 Interacting with DPL

When the DPL system is loaded into a LISP environment, a special object is created. This object, called a **top**, is very similar to a prototype. *ME* is bound to a top so that commands which would otherwise appear in the maker function of a type may be executed at top-level LISP, the results affecting the top.

For example, typing

```
(part 'iv1 inverter)
```

at top-level will make an inverter and name it IV1 in *ME* -- which is bound to the top. One may then say

```
(>> iv1)
```

and get the instance of the inverter. Tops are represented by the string "layout" followed by a number: LAYOUT-1. In most cases this is the only top that will be seen.

Calling types interactively (by calling PART at top-level) is the way to debug designs. One defines types while working with a text-editor and then loads the text files into a DPL environment where he may then examine the structures of the types. On some systems where DPL may be used, graphics programs can display pictures of instances. Just running the maker function of a type, by using PART, is a good way to see if the code is syntactically correct and contains no LISP errors.

The function EXAMINE is useful for inspecting the structure of DPL objects. The command

```
(EXAMINE <object>)
```

will place the user in an "examiner-loop" for examining the structure of <object>. He may type commands to identify the components of the structure he wishes to view. The EXAMINE function is system-dependent and will not be described in detail here. It is self-documenting -- typing "?" or "help" will print out the available options on the

system being used.

9.2 What Objects Look Like

When interacting with DPL it is necessary to know what the DPL structures look like when they are printed. The printed representations of DPL structures are for the user's convenience and have little to do with the way structures are stored in the computer.

The printed appearance of types and prototypes will vary greatly among different implementations of DPL. In general, the EXAMINE function should be used to look at them.

The function

```
(TYPE-FROM-TYPE-NAME <type-name>)
```

returns the type with the name <type-name>.

Components of the structure of a type or prototype may be inspected with THE:

```
(THE 'TYPE <prototype>)
(THE 'PROTOTYPES <type>)
```

For the remainder of this chapter, we represent types by their names, and prototypes by the name of their type concatenated with a number.

An instance contains a VC and a prototype:

```
(INSTANCE (VC <prototype> <parent>) <augmentation>)
```

The <augmentation> of an instance is a transform:

```
(<unitary-part> <x>  <y>)
```

(or it may be NIL which is the same as the "identity-transform," (IDENTITY 0 0)). So a whole instance may look like this:

```
(INSTANCE (VC INVERTER-5 REGISTER-12) (ROT90 34  10))
```

Since there are so many of them, prototypes of the type RECTANGLE are treated differently from other prototypes. The only time this makes a difference is when they are printed out. In this case they look like:

```
(RECTANGLE <layer> <length> <width>)
```

<length> is the Y dimension of the prototype and <width> is its X dimension. Rectangles are used in place of normal prototypes, so an instance of a rectangle could look like this:

```
(INSTANCE (VC (RECTANGLE POLY 2 4) (IDENTITY 4 5)))
```

9.3 CIF

Users of DPL must interact with CIF for two reasons: Interesting cells from other designers are available in CIF, and CIF must be given to the fabrication companies to actually produce the chips.

CIF descriptions of cells may be translated into DPL with the CIFTRAN function. The form

```
(CIFTRAN <cif-file> <load?> <output-file> <lambda>)
```

translates the CIF in <cif-file> into DPL in <output-file>. The argument <lambda> is the size of lambda in microns used to produce the CIF. If <load?> is not NIL, the resultant DPL definitions are immediately loaded into the LISP environment and the types produced are available to call. All the arguments to CIFTRAN are evaluated.

The DPL produced from CIF is extremely "bare". Nothing is named, for example, and thus none of the path-following features of DPL may be used. It is possible to edit the definitions of the types in the output from CIFTRAN and name some of the parts.

To obtain a CIF representation of a design, the function CIFOUT is used. The form is:

```
(CIFOUT <instance> <file-name>)
```

CIFOUT takes an instance and recursively travels through the data-base, translating into CIF all the types on which <instance> depends. The final CIF command is a call to the symbol that corresponds to the prototype of <instance>. "User extension 9" is used for the names of the prototypes as they are translated. The name of each type is included in a comment after the DS command. All the arguments to CIFOUT are evaluated.

10. EXAMPLE

In this chapter we explain in detail a DEFLAYOUT of a register cell. The program uses most of the features of DPL, so serves as a good illustration of the language in action. We have two reasons for presenting this example. First, the example should help the user learn how to use DPL. Second, we want to demonstrate some of the ways to use DPL to its best advantage.

10.1 DPL design style

The program below is one of many possible programs that could be written to lay out this cell. There certainly is no one place to start or one way to proceed in such a program. However, there are a number of guidelines to be kept in mind.

Note that very few numbers are used in the program. Almost every new part is placed by reference to the locations of previously placed parts. Most of the spacings and offsets between parts are determined by the design rules of the processing technology. DPL provides variables to specify such values.

It is helpful to begin at some point in the cell and move in a particular direction, creating new parts when their location may be described in terms of the existing structure. Specifying placement by reference to previously placed parts is encouraged because it makes apparent the justifications for the positioning of parts. Specifying placement numerically incorporates no information about why an object is placed where it is. On the other hand, a "symbolic" specification of position enables one to see, for example, that the end of a wire is to connect to the input of an inverter.

Symbolic specification of a design contributes to easier debugging. Since all the parts depend in precise ways upon one another, moving an object will cause the parts specified in terms of it to maintain their distance. This ability is totally absent in a cell specified numerically, where each affected part would have to be moved.

10.2 The REGCELL

The object we are designing is meant to store a bit of data; it can be written from a bus and can write its value onto the bus. The circuit, shown in figure 9, consists of two inverters with a feedback path to refresh the data and connections to the data bus. During phi2 the value is refreshed. The cell is given a new value by placing the bit on the bus and taking the phi1&load signal high. The value of the cell may be read onto the bus by taking phi1&read high.

Figure 10 is a drawing of the cell's layout in NMOS technology. We have chosen the most straightforward design possible, to allow us to concentrate on using the language -- as opposed to a fancy layout. This layout is certainly not the smallest for this cell. We leave it as an exercise for the reader to come up with a "minimal" layout for this cell.

On the next pages we present the definition of REGCELL.

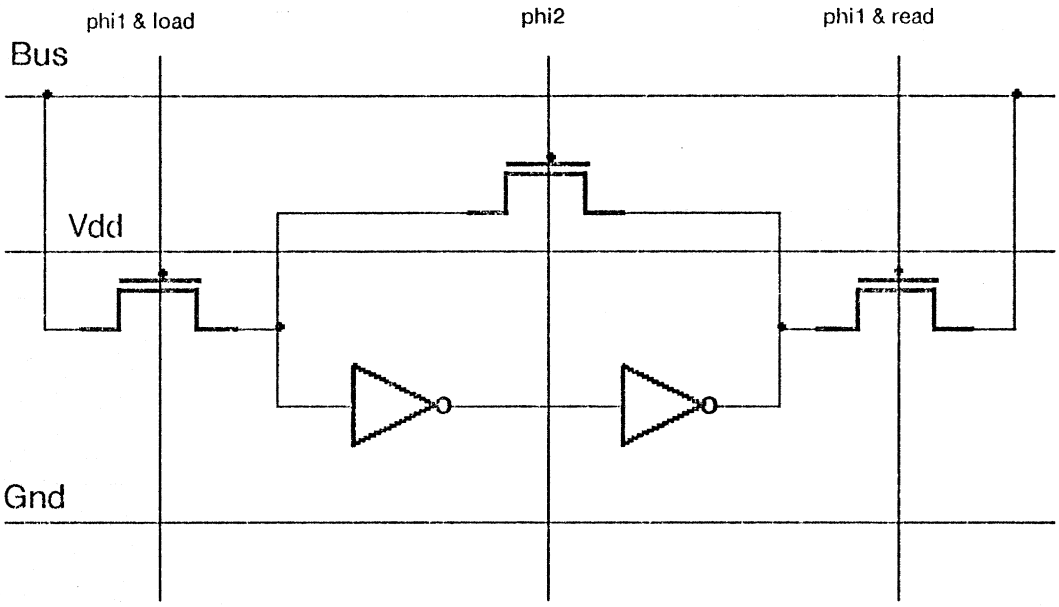
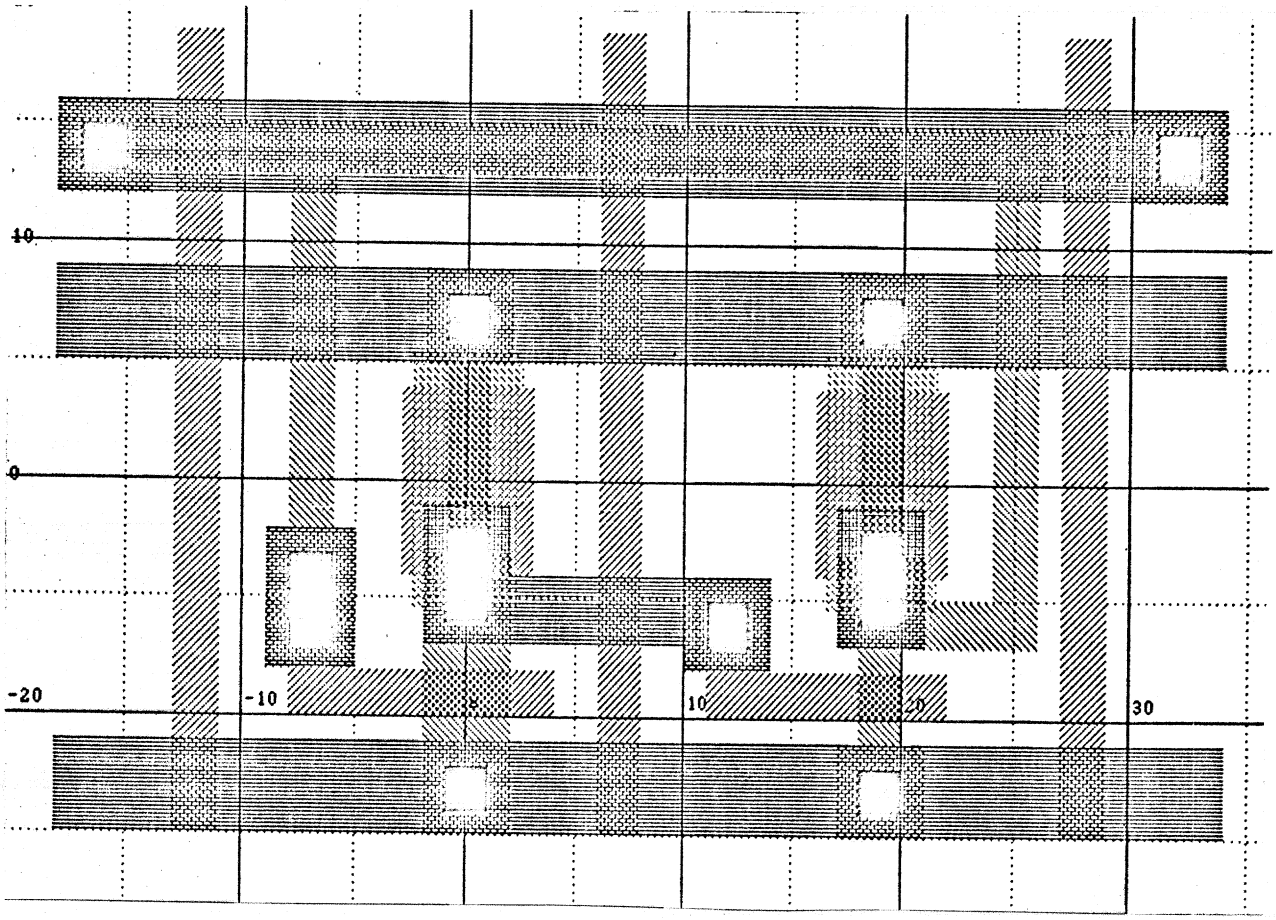


Figure 9 -- Circuit of Regcell

Figure 10 -- The REGCELL



```
(deflayout regcell '()
  (part 'iv1 inline-inverter (ew 4))
  (part 'GND-contact1 square-contact (layer 'diff)
      (top-center (>> center source-diffusion pulldown iv1)))
  (part 'VDD-contact1 square-contact (layer 'diff)
      (bottom-center (>> center drain-diffusion transistor pullup iv1)))
  (part 'bc butting-contact (xfrm 'rot180))
  (align (>> bc)
  (>> apparent-bottom-right bc)
       (pt (- (>> x bottom-left contact pullup iv1) *metal-to-metal*)
           (+ (>> y top-center GND-contact1) *metal-to-metal*)))
   (wire ()
      (run-layer 'poly)
      (from (>> apparent-bottom-center bc))
      (jog-y (>> center-left gate-poly pulldown iv1)))
   (wire 'phi2
      (run-layer 'poly)
      (from (pt (+ (>> x bottom-right gate-poly pulldown iv1) *poly-to-poly* 1)
                (>> y bottom-right GND-contact1)))
      (to-y (+ (>> y top-right VDD-contact1)
               *metal-to-metal*
               *default-metal-size*
               *metal-to-metal*)))
   (part 'poly-contact square-contact (layer 'poly))
   (align (>> poly-contact)
       (>> bottom-left poly-contact)
       (pt (+ (>> x center phi2) *poly-to-poly* 1)
           (+ (>> y top-right GND-contact1) *metal-to-metal*)))
   (wire ()
      (run-layer 'metal)
      (run-width *min-metal-size*)
      (from (pt-above (>> bottom-right cover contact pullup iv1)
                      (// *min-metal-size* 2.0)))
      (to-x (>> center poly-contact)))
   (part 'iv2 inline-inverter)
   (align (>> iv2)
       (>> top-left gate-poly transistor pullup iv2)
       (pt (+ (>> x top-right poly-contact) *poly-to-poly*)
           (>> y top-left gate-poly transistor pullup iv1)))
   (part 'GND-contact2 square-contact (layer 'diff)
      (top-center (>> center source-diffusion pulldown iv2)))
   (part 'VDD-contact2 square-contact (layer 'diff)
      (bottom-center (>> center drain-diffusion transistor pullup iv2)))
   (wire ()
      (run-layer 'poly)
      (from (>> bottom-center poly-contact))
      (jog-y (>> center-left gate-poly pulldown iv2)))
   (wire 'feedback
      (run-layer 'diff)
      (from (pt-above (>> bottom-right diff contact pullup iv2) 1))
      (to-x (pt-to-right (>> center-right vdd-contact2)
                         (+ *diff-to-diff* 1)))
      (to-y (pt-above (>> top-center vdd-contact2)
                      (+ *metal-to-metal*
                         (// *default-metal-size* 2.0))))
      (save-cp 'read-turn)
      (to-x (>> apparent-top-center diff bc))
      (save-cp 'load-turn)
      (to-y (>> apparent-top-center diff bc)))
   (wire 'phi1&load
      (run-layer 'poly)
      (from (pt (- (>> x apparent-bottom-left poly bc)
                   (+ *poly-to-poly* 1))
                (>> y start phi2)))
      (to-y (>> y end phi2)))
```

```
66    (wire 'phi1&read
67       (run-layer 'poly)
68       (from (pt (+ (>> x center-right feedback)
69                    (+ *poly-to-diff* 1))
70                 (>> y start phi2)))
71       (to-y (>> y end phi2)))
72    (part 'load-trans pass-transistor (xfrm 'rot90))
73    (align (>> load-trans)
74        (>> center load-trans)
75        (pt (>> x center phi1&load)
76            (>> y load-turn feedback)))
77    (part 'load-contact square-contact
78       (layer 'diff)
79       (center-right (pt (- (>> x center phi1&load)
80                            (+ *poly-to-diff* 1))
81                         (>> y load-turn feedback))))
82    (wire ()
83       (run-layer 'diff)
84       (from (>> load-turn feedback))
85       (to-x (>> apparent-center-right load-trans)))
86    (part 'read-trans pass-transistor (xfrm 'rot90))
87    (align (>> read-trans)
88        (>> center read-trans)
89        (pt (>> x center phi1&read)
90            (>> y read-turn feedback)))
91    (part 'read-contact square-contact
92       (layer 'diff)
93       (center-left (pt (+ (>> x center phi1&read)
94                           (+ *poly-to-diff* 1))
95                        (>> y read-turn feedback))))
96    (wire ()
97       (run-layer 'diff)
98       (from (>> read-turn feedback))
99       (to-x (>> apparent-center-left read-trans)))
100   (wire 'data-bus
101      (run-layer 'metal)
102      (from (>> center-left load-contact))
103      (to-x (>> center-right read-contact)))
104   (wire 'vdd-bus
105      (run-layer 'metal)
106      (from (pt (>> x start data-bus)
107                (>> y center vdd-contact1)))
108      (to-x (>> end data-bus)))
109   (wire 'gnd-bus
110      (run-layer 'metal)
111      (from (pt (>> x start data-bus)
112                (>> y center gnd-contact1)))
113      (to-x (>> end data-bus)))
114   (setq-my h-pitch (- (>> x center read-contact)
115                       (>> x center load-contact))))
```

10.3 Discussion of the REGCELL program

We have laid out REGCELL by beginning with the leftmost inverter, creating the parts surrounding it, and moving around the cell building each part as its position is determined by its neighbors. We built the central pieces of the cell first and built the busses extending the entire width of the cell last. The busses were built last so there would be objects by which to specify their dimensions.

Let us look at the program in more detail. We begin with the leftmost inverter, which we name IV1, because it is a prominent piece of the cell and seems like a good place to start. IV1 is created by using the PART function which makes an instance of the prototype INLINE-INVERTER (see Library) with an enhancement channel width of 4. This will give the inverter a ratio of 8-1 which is necessary because it will be driven through a pass-transistor. The other parameters of INLINE-INVERTER will default as follows: The enhancement channel length will be 2, the depletion channel length 8, and the depletion channel width 2. (line 2)

We may now create things whose locations depend on that of IV1. We create GND-CONTACT1 and VDD-CONTACT1 placing them below and above IV1. They will be used later to connect the inverter to the power and ground busses. (lines 3 - 6)

After making the two square contacts, we create the butting contact on the left of IV1, and call it BC. We rotate BC 180° because the untransformed butting contact has its diffusion on the bottom and we need it on the top. We then align BC the minimum metal-to-metal distance from the metal to its right and the same distance above the ground bus. We use GND-CONTACT1 to find out the height of the bus. Note that it was necessary for us to create the square contacts before making BC in order to specify the Y coordinate of BC. Of course we could have aligned BC with a part of IV1, say making the APPARENT-BOTTOM-RIGHT of the CUT of BC 3 lambda to the left of the BOTTOM-LEFT of the CONTACT of the PULLUP of IV1. While this would work fine, it doesn't really represent the reason that BC is placed where it is. The location of BC is determined to the right by the metal in IV1 and below by the eventual location of the ground bus. The placement we use indicates these dependencies. (lines 7 - 11)

We can finish up this segment of the cell by connecting a poly wire from the poly of butting contact BC to the GATE-POLY of the PULLDOWN of IV1. The wire starts from the APPARENT-BOTTOM-CENTER of BC and does a JOG-Y to the CENTER-LEFT of the GATE-POLY of

the PULLDOWN of IV1. The JOG-Y command here will actually create a rectangle of 1 lambda in the Y direction and then a rectangle extending in the X direction. We use a jog here to allow us the flexibility of being able to raise BC at some point in the future without having to change the description of the poly wire. Since there is nothing requiring BC to be as close to the ground bus as we have made it, it is possible to move it higher if necessary. (lines 12 - 15)

We now move to the right and create the clock line PHI2, the next object whose position is determined by those we have already built. Its position in the X direction is determined by the minimum poly-to-poly spacing from the PULLUP of IV1. We want the wire to run from the bottom of the cell to the top, plus one metal-to-metal distance at the top. The extra is so that several of these cells may be stacked to allow several bits to be stored at once. Since there will be a data-bus at the top of the cell, in addition to the power bus, we must allow room for one metal line of the default size, plus two metal-to-metal distances. (lines 16 - 23)

Next we form the poly contact to the right of PHI2. It can be no closer to ground than the minimum metal-to-metal distance and no closer to PHI2 than the minimum poly-to-poly distance. We place it as close as it can get to these wires with an ALIGN procedure using these specifications. Next we run a metal wire from the BOTTOM-RIGHT of the CONTACT of the PULLUP of IV1 to POLY-CONTACT. We make it 3 lambda (minimum metal width) wide. We must begin the wire 1.5 lambda above the bottom of the contact so that the connection will be 3 lambda wide. The TO-X command then runs the wire to POLY- CONTACT. (lines 24 - 34)

We are now ready to make our second inverter which we name IV2. We give it no parameters as we want its ratio to be 4-1 which is the default. The 4-1 ratio is justified because this inverter is driven directly from IV1. Its placement is determined on the left side by the minimum poly-to-poly distance from POLY-CONTACT. The vertical placement is to be the same as that for IV1. The power and ground contacts are then placed for IV2 in the same manner as for IV1. A poly wire from POLY-CONTACT to the GATE-POLY of PULLDOWN of IV2 completes the connection of the output of IV1 to the input of IV2. We again place the wire with a jog to allow us future flexibility. (lines 35 - 47)

We will now construct the feedback path that refreshes IV1 through PHI2 and makes the connections to the data-bus through the control signals. The wire FEEDBACK runs from the diffusion in the CONTACT of the PULLUP of IV2, to the right far enough to put

it one diff-to-diff distance away from VDD-CONTACT2, then up to where the center of the data-bus will be. Two CPs of the wire, READ-TURN and LOAD-TURN, are saved for later placement of transistors. FEEDBACK then jogs to the diffusion in BC, completing the feedback path. Note that a transistor is created when the diffusion of FEEDBACK crosses the PHI2 wire. (lines 48 - 59)

The control lines, PHI1&LOAD and PHI1&READ, may now be placed. Both are specified horizontally by minimum distances from existing structure and vertically by requiring that they begin and end at the same Y values as the START and END of PHI2. (lines 60 - 71)

The connections from the feedback wire to the data-bus may now be made. The same procedure is used for the connections on the left and right sides of the cell. First a pass-transistor is placed at the appropriate position. Then a contact to the data-bus is created as close as possible to the control line. A wire is then run connecting the pass-transistor to one of the named CPs on FEEDBACK. (lines 72 - 99)

All that remains is the busses. They extend from the left edge of LOAD-CONTACT to the right edge of READ-CONTACT. The Y coordinate of each bus has already been determined. (lines 100 - 113)

The last command names the H-PITCH of the cell. Note that the cell may be replicated so that the center of LOAD-CONTACT on one instance of REGCELL could line up with the center of READ-CONTACT on the instance to its left. We specify this here so that the constraints in the ROW replicator will place REGCELL correctly. We have designed REGCELL so that its vertical dimension is the correct one to use for replication so we need not specify it explicitly. (lines 114 - 115)

11. LIBRARY

The definitions here are automatically loaded into the DPL environment. There will probably be other cells available.

11.1 Some Constraints

```
(defconstraint c= (v1 v2)
  (v1 (v2) v2) (v2 (v1) v1))
```

This causes the values of two parameters to be equal.

```
(defconstraint c+ (v1 v2 v3)
  (v1 (v2 v3) (+ v2 v3))
  (v2 (v1 v3) (- v1 v3))
  (v3 (v1 v2) (- v1 v2)))
```

The first value is to be the sum of the other two.

```
(defconstraint c* (prod m1 m2)
  (prod (m1 m2) (* m1 m2))
  (m1 (prod m2)
      (if (= m2 0)
          'bail-out
          (// (float prod) m2)))
  (m2 (prod m1)
      (if (= m1 0)
          'bail-out
          (// (float prod) m1))))
```

The first value is to be the product of the other two.

```
(defconstraint offset (v1 v2 v3)
  (v1 (v2 v3) (pt-sum v2 v3))
  (v2 (v1 v3) (pt-difference v1 v3))
  (v3 (v1 v2) (pt-difference v1 v2)))
```

All the values are points. The first point is to be the vector sum of the other two.

11.2 Some Types

```
(deflayout rectangle
    '((primary-parameters
        ((layer nil)
         (length nil)
         (width nil)))
      (constraints
        ((default-size-for-layer layer length)
         (default-size-for-layer layer width)))))
```

The structure of a rectangle is determined entirely by the values of its parameters.

```
(deflayout square-contact
    '((primary-parameters ((layer 'poly))))
  (part 'cut rectangle (layer 'cut))
  (part 'cover rectangle (layer 'metal))
  (part 'stuff rectangle (layer (>> layer))
        (length 4) (width 4)))


(deflayout horizontal-contact
    '((primary-parameters ((layer 'poly))))
  (part 'cut rectangle (layer 'cut)
        (length 2) (width 4))
  (part 'cover rectangle (layer 'metal)
        (length 4) (width 6))
  (part 'stuff rectangle (layer (>> layer))
        (length 4) (width 6)))


(deflayout butting-contact '()
   (part 'cut rectangle   (layer 'cut)
         (length 4))
   (part 'cover rectangle (layer 'metal)
         (length 6))
   (part 'poly rectangle  (layer 'poly)
         (length 3) (width 4) (center (pt 0 1.5)))
   (part 'diff  rectangle (layer 'diff)
         (length 4) (width 4) (center (pt 0 -1))))
```

```
(deflayout rect-fet
    '((primary-parameters
        ((channel-length 2)
         (channel-width 2))))
   (part 'gate-poly rectangle  (layer 'poly)
         (length (>> channel-length))
         (width (+ (>> channel-width)
                   (* 2 *poly-overhang*))))
   (part 'channel rectangle (layer 'channel)
         (length (>> channel-length))
         (width (>> channel-width)))
   (part 'source-diffusion rectangle (layer 'diff)
         (length *diff-overhang*)
         (width (>> channel-width))
         (top-center (>> bottom-center channel)))
   (part 'drain-diffusion rectangle (layer 'diff)
         (length *diff-overhang*)
         (width (>> channel-width))
         (bottom-center (>> top-center channel))))
```

This is the standard transistor -- a rectangular FET.

```
(deflayout (rect-e-fet rect-fet) '())
```

A "rectangular enhancement FET" is the same as a "rect-fet".

```
(deflayout (rect-d-fet rect-fet)
           '()
  (part 'implant rectangle
        (layer 'ion)
        (length (+ (>> channel-length)
                   (* 2 *ion-overhang*)))
        (width (+ (>> channel-width)
                  (* 2 *ion-overhang*)))
        (center (>> center channel))))
```

A "rectangular depletion FET" has ion implant.

```
(deflayout (pulldown rect-e-fet) '())

(deflayout (pass-transistor rect-e-fet) '())
```

Both of these are alternate names for rect-e-fet.

```
(deflayout standard-pullup
    '((primary-parameters
       ((channel-length 8)
        (channel-width 2))))
  (part 'transistor rect-d-fet
        (channel-length (>> channel-length))
        (channel-width (>> channel-width)))
  (part 'contact butting-contact)
  (rot90 (>> contact))
  (align (>> contact)
         (>> bottom-right cut contact)
         (>> bottom-center
             source-diffusion transistor))
  (part 'poly rectangle
        (layer 'poly) (length 3) (width 2)
        (bottom-right
           (>> bottom-right poly contact)))
  (setq-my diffusion-connection
       (pt 0 (>> y
                 bottom-left diff contact))))
```

A "standard pullup" has a contact on its left side and a connection from the contact to the gate of the transistor.

```
(deflayout (inline-pullup pullup)
    '((primary-parameters
       ((channel-length 8)
        (channel-width 2))))
  (part 'transistor rect-d-fet
        (channel-length
           (1+ (>> channel-length)))
        (channel-width (>> channel-width)))
  (part 'contact butting-contact)
  (align (>> contact)
         (>> bottom-center poly contact)
         (>> bottom-center gate-poly transistor))
  (setq-my diffusion-connection
           (>> bottom-center contact)))
```

An "inline pullup" has the butting contact directly below the transistor. The channel must be made one lambda longer than necessary for the correct ratio, because the contact will cover part of it.

This inverter uses the standard pullup. Note the constraints between the parameter values and ratios.

```
(deflayout inverter
    '((primary-parameters
       ((dl 8.0) (el 2.0)
        (ew 2.0) (dw 2.0)
        puz pdz z))
      (constraints
       ((c* dl puz dw)
        (c* el pdz ew)
        (c* puz z pdz))))
  (part 'pullup standard-pullup
        (channel-length (>> dl))
        (channel-width (>> dw)))
  (part 'pulldown rect-e-fet
        (channel-length (>> el))
        (channel-width (>> ew)))
  (align (>> pulldown)
         (>> top-center
             drain-diffusion pulldown)
         (>> diffusion-connection pullup)))
```

This inverter uses the inline-pullup.

```
(deflayout inline-inverter
    '((primary-parameters
       ((dl 8.0) (el 2.0)
        (ew 2.0) (dw 2.0)
        puz pdz z))
      (constraints
       ((c* dl puz dw)
        (c* el pdz ew)
        (c* puz z pdz))))
  (part 'pullup inline-pullup
        (channel-length (>> dl))
        (channel-width (>> dw)))
  (part 'pulldown rect-e-fet
        (channel-length (>> el))
        (channel-width (>> ew)))
  (align (>> pulldown)
         (>> top-center drain-diffusion pulldown)
         (pt-above (>> diffusion-connection pullup) 1)))
```

11.3 Some Replicators

These replications make use of the H-PITCH and V-PITCH constraints between their instance lists and their "pitch" parameters. The "pitch" of an instance is the minimum distance between points where successive replicated versions of the instance may be placed. In most cases this is the size of the instance in the appropriate dimension "H" (horizontal) or "V" (vertical). However, if a cell is explicitly given a parameter with one of these names, the value in that cell is used.

```
(defreplicator row (i)
  '((primary-parameters
        ((pitch) (spacing 0)))
    (constraints
        ((h-pitch instance-list pitch))))
  (replicator-instance
      (>> (*rep-instance 0)))
  (replicator-transform
        'identity
        (* i (+ (>> pitch) (>> spacing)))
        0))
```

This places a row of objects. The SPACING parameter may be used to insert extra space between the elements.

```
(defreplicator column (i)
  '((primary-parameters
        ((pitch) (spacing 0)))
    (constraints
        ((v-pitch instance-list pitch))))
  (replicator-instance
        (>> (*rep-instance 0)))
  (replicator-transform
        'identity
        0
        (* i (+ (>> pitch) (>> spacing)))))
```

Makes a column.

```
(defreplicator array (i j)
  '((primary-parameters
     ((v-pitch) (h-pitch)
      (v-spacing 0) (h-spacing 0)))
    (constraints
     ((h-pitch instance-list h-pitch)
      (v-pitch instance-list v-pitch))))
  (replicator-instance (>> (*rep-instance 0)))
  (replicator-transform
        'identity
       (* i (+ (>> h-pitch) (>> h-spacing)))
       (* j (+ (>> v-pitch) (>> v-spacing)))))
```

This takes an instance and makes an n x m array of it. The pitches default as for row and column.

```
(defreplicator flipping-array (i j)
  '((primary-parameters
      (v-pitch h-pitch xref yref
      (x-overlap 0) (y-overlap 0)
      (x-space 0) (y-space 0)))
    (constraints
      ((flipping-pitches
        instance-list h-pitch v-pitch xref yref
        x-overlap y-overlap x-space y-space))))
  (let ((x ( i 2)) (y ( j 2)) which unitary)
    (setq which (+ x (* 2 y)))
    (setq unitary (cond ((equal which 0) 'identity)
                        ((equal which 1) 'negx)
                        ((equal which 2) 'negy)
                        ((equal which 3) 'rot180)))
    (replicator-instance (>> (*rep-instance 0)))
    (replicator-transform unitary
                          (if (= x 0)
                              (* (// i 2) (>> h-pitch))
                              (+ (>> xref)
                                 (* (// i 2)
                                    (>> h-pitch))
                                 (- 0 (>> x-overlap))))
                          (if (= y 0)
                              (* (// j 2) (>> v-pitch))
                              (+ (>> yref)
                                 (* (// j 2)
                                    (>> v-pitch))
                                 (- 0 (>> y-overlap)))))))
```

This makes an array in which alternate elements are flipped. It uses a FLIPPING-PITCHES constraint which is similar to the pitch constraints. One can make a row in which alternate elements are flipped by calling FLIPPING-ARRAY with a second argument of 1, or a column of alternately flipped elements by using a first argument of 1.

12. GLOSSARY

Presented here are most of the DPL functions and variables available for designing projects. Functions which are used only for the implementation of the language are not included.

Each function is presented with information about the arguments it takes. For example

```
(A-FUNCTION <arg-1> '<arg-2> . <form-1> <form-2> . . .)
```

introduces the function A-FUNCTION. Arguments are enclosed in angle brackets (<>) and are given reasonably mnemonic names. Arguments not evaluated are shown with a quote before them. (like <arg-2> above). A dot (.) in the form indicates that the terms after the dot are optional and thus may be omitted. Three dots (. . .) at the end of some optional arguments indicates that there may be any number of terms in the argument list at that point.

Every term in the body of a form will be described. In some cases the terms themselves must be structured in certain ways.

The functions will be grouped according to the kinds of DPL objects they operate on -- transform functions are grouped together, as are functions that manipulate points.

12.1 Types

```
(DEFLAYOUT '<type-name>
           <param-list>
           .  '<body>)
```

Defines a type. If <type-name> is an atom, the new type will have that name. If the name is a list of two atoms, the CAR will be the name of the new type, the CADR will be used as the "supertype" of the type. <param-list> is a list of keyword-value pairs. The value of each pair is stored in a cell on the type. <body> is the maker function of the type and may consist of any LISP and DPL forms.

```
(PRIMARY-PARAMETERS
   ((<param-name-1> <default-value-1>)
    (<param-name-2> <default-value-2>)
          . . .))
```

This form in the <param-list> of a DEFLAYOUT specifies the names and default values for the parameters used to build instances of the type.

```
(CONSTRAINTS
  ((<constraint-name <param> <param> . . .)
   (<constraint-name <param> <param> . . .)
     . . .))
```

This form in the <param-list> of a DEFLAYOUT specifies that the constraints named are to be applied to the parameters named.

```
(AUXILIARY-PARAMETERS
       (<param-name>. . .))
```

This form in the <param-list> of a DEFLAYOUT specifies some of the names used by the maker function to store information.

```
(TYPE-FROM-TYPE-NAME <type-name>)
```

Returns the type whose name is <type-name>.

```
(TYPE? <object>)
```

Tests to see if <object> is a type.

```
(PART <name> '<type>
       . <param-1> <param-2> . . .)
```

Creates an instance of <type> and makes it a part of *ME*. If <name> is non-NIL, it is used as the name of the part on *ME*. Each of the parameters has the following form:
('<param-name> <value>)
The (<param-name>) may be a defined parameter of the type, an "implicit-parameter", or parameters used to place the instance.

(DELETE-INSTANCE <instance>)	Removes <instance> from the parts of *ME*.
(INVOKE <name> '<type> . <param-1> <param-2> . . .)	Identical to PART except that the instance produced is not considered to be a "part" of *ME*.
(REMTYPE <type-name>)	Removes all prototypes and instances of the type and all prototypes and instances that use them. The type definition is still available.
(DESTROY-TYPE <type-name>)	Does a REMTYPE of the type and then removes the type from the list of defined types. The type may then no longer be called.

12.2 Naming

(ASSIGN-TO-THE <name> <object> <value>)	Creates a cell named <name> on <object> and places <value> in it.
(SET-MY <name> <value>)	Creates a cell named <name> on *ME* and places <value> in it.
(SETQ-MY '<name> <value>)	The same as SET-MY but <name> is not evaluated.
(BOUND-ON? <name> <object>)	Tests to see if a cell named <name> exists on <object>.

12.3 Access Functions

(THE <name> <object>)

Finds the value of <name> in <object>. If the value is a transformable object, the appropriate composition of transforms is performed to assure that the result is viewed in the correct coordinate system. <name> may be the name of information placed by DEFLAYOUT, SETQ-MY, an "implicit-parameter", or one of the components of a DPL structure (for example, the prototype of an instance).

(>> . <form-1> <form-2> . . .<form-n>)

Expands into a series of calls to THE. The <object> of the last call to THE is *ME*. Accesses information from *ME* as well as information nested in parts of *ME*. If the forms are atoms they are not evaluated. Otherwise they are evaluated and the results used as the names.

(E>> <atom>)

Returns <atom>. It is used if one wants to evaluate an atom in a >> form.

(MY <name>)

The same as: (THE <name> *ME*).

(EXAMINE <thing>)

Used to interactively examine the components of the structure of <thing>. Allows the user to indicate the parts he wishes to examine. The command QUIT exits the program, ? prints documentation.

12.4 Points

(PT <x> <y>)

Makes a point with the given coordinates.

(PT? <object>)

Tests the object to see if it is a point.

(PT-SUM <pt1> <pt2>)
(PT-DIFFERENCE <pt1> <pt2>)

These functions create a point that is the vector sum or difference of the two arguments.

(PT-ABOVE <pt> <offset>)
(PT-BELOW <pt> <offset>)
(PT-TO-LEFT <pt> <offset>)
(PT-TO-RIGHT <pt> <offset>)

These functions create a point offset from the given point the specified amount in the specified direction.

12.5 Transform Functions

(<unitary-transform> <instance>)

Applies the unitary transform function to the instance. This gives the instance a new transform which is the composition of the unitary-transform with the previous transform of the instance.

IDENTITY **ROT90**
ROT180 **ROT270**
NEGX **NEGY**
INT-POS **INT-NEG**

These are the unitary transform functions. Their names may be used as the XFRM parameter in a PART command. The names may also appear as the unitary-part of a transform.

(ALIGN <instance>
<pt-on-instance>
<target-pt>)

Translates the instance so that <pt-on-instance> is at <target-pt>.

(SET-TRANSFORM <instance> <transform>)

Gives the instance a transform of <transform>.

(TRANSFORM-PT <transform> <pt>)

Returns the point that is the result of transforming <pt> by the <transform>.

(TRANSFORM-PT-BY-UNITARY <unitary-transform> <pt>)	The same as TRANSFORM-PT but the argument is the name of a unitary-transform function.
(CREATE-TRANSFORM <unitary-part> <pt>)	Creates and returns a transform with the given unitary-part and a translation part determined from <pt>.
(COMPOSE-TRANSFORMS <trans-1> <trans-2>)	The transform that is returned is the result of first applying <trans-2>, then <trans-1>.
(COMPOSE-UNITARY-TRANSFORMS <ut-1> <ut-2>)	Gives the unitary-transform that results from applying unitary transform <ut-2> followed by unitary transform <ut-1>.
(INVERSE-TRANSFORM <transform>)	Returns the transform that, when composed with <transform>, would give the identity transform.
(INVERSE-UNITARY-TRANSFORM <unitary-transform>)	The same as INVERSE-TRANSFORM except that this takes and returns a unitary transform.

12.6 Wiring Commands

`(WIRE <name> . <body>)`

Makes a wire and names it `<name>` if the name is non-`NIL`. `<body>` is then executed with the wire bound to the variable `*THE-WIRE*`. `<body>` may contain and LISP forms or special wire procedures.

```
(RUN-LAYER <layer>)
(RUN-WIDTH <value>)
(FROM <location>)
(TO-X <location>)
(TO-Y <location>)
(TO-PT <location>)
(JOG-X <pt>)
(JOG-Y <pt>)
(+X <value>)
(-X <value>)
(+Y <value>)
(-Y <value>)
(SAVE-CP <name>)
(RESTORE-CP <name>)
(DROP-CONTACT)
```

These commands are allowed inside a `WIRE` form. All of these have corresponding functions for use from "outside" the wire. Their names are `WIRE-<name>` where the `<name>` is one of the above.

`(PT-CP <cp>)`

Gets the point from the connection-point.

12.7 Constraints

```
(DEFCONSTRAINT '<name>
        '<var-list>
         .  '<body>)
```

Defines a constraint named `<name>`. When called, the constraint will bind whatever variables in `<var-list>` are specified. Then the `<body>` will be executed. The forms in the body look like:

`(<result-var> <depends-on> <code>)`

(None of these are evaluated when the `DEFLAYOUT` is evaluated.) The `<result-var>` is the variable that may be computed if all the variables in the `<depends-on>` list are specified. The procedure to find the value of that variable is specified by `<code>`.

```
(DEFDEFAULT '<name>
        '<var-list>
         .  '<body>)
```

Defines a "default" constraint. The constraint will be effective only if no other attempt is made to set the value of the parameters it constrains, either by explicit passing of values or other "normal" constraints. Form and use are identical to DEFCONSTRAINT.

12.8 Replicators

```
(DEFREPLICATOR '<name>
        '<coordinate-variables>
         <param-list>
          .  <body>)
```

Defines a replicator. The <name> and <param-list> work the same as for DEFLAYOUT. <coordinate-variables> is a list of names that are used in <body> to compute the instance at the corresponding coordinates when the replicator is referenced.

```
(*REP <i> <j>)
```

Used in a THE or >> command, gets the (<i>,<j>)-th element of a replication.

```
(*REP-INSTANCE <n>)
```

Gets the nth instance in the INSTANCE-LIST of the replication.

```
(REPLICATOR-INSTANCE <instance>)
```

Must appear in the body of a replicator. Indicates the instance that is to be returned when the replicator is accessed. Usually <instance> is a call to *REP-INSTANCE that depends on the <coordinate-variables> of the replicator.

```
(REPLICATOR-TRANSFORM
        <unitary-part> <x> <y>)
```

Must appear in the body of a replicator. Indicates the transform that is to be composed with the transform of the instance given to REPLICATOR-INSTANCE.

```
(REPLICATE <name> '<replication>
        <coordinate-sizes>
        . <param-list>)
```

Calls the replicator <replication>. The list <coordinate-sizes> gives the values of the dimensions of the coordinates. The rest of the form is identical with the PART form.

12.9 CIF

```
(CIFOUT <instance> <filename>)
```

Outputs CIF translations of enough of the data-base to build the instance.

```
(CIFTRAN <file>
        <load?> <file-out> <lambda-size>)
```

Translates the CIF in the <file> into DPL and places the result in the file <file-out>. The <lambda-size> is the size of lambda in microns with which the original CIF was produced. If <load?> is non-NIL, the DPL forms are loaded into the current environment.

12.10 Implicit-Parameters

```
BOUNDING-BOX
XDIM            YDIM
TOP-LEFT        TOP-RIGHT
CENTER-LEFT     CENTER-RIGHT
BOTTOM-LEFT     BOTTOM-RIGHT
TOP-CENTER      BOTTOM-CENTER
CENTER
ORIGIN
```

The implicit parameters of an instance. All (except BOUNDING-BOX) have corresponding APPARENT-versions.

12.11 *LIST*

```
(MAKE-LIST-OF <list>)
```

Makes a *LIST* from the list. Used when the list will contain transformable objects.

(THE-LIST-OF <list>) Extracts the list from a *LIST*.

12.12 Layer Sizing

(LAYER-DEFAULT-SIZE <layer>) Finds the default size for the layer. Uses the "default-size" constants below.

(LAYER-MINIMUM-SIZE <layer>) Finds the minimum size for the layer. Uses the "minimum-size" constants below.

12.13 Symbols

ME When prototype is being constructed by the maker function of a type, *ME* is bound to that prototype. Otherwise, *ME* is bound to the "top".

THE-WIRE Bound to the wire being constructed in a WIRE form.

CURRENT-WIRE Used to access CPs of the wire being constructed.

(*UNNASIGNED*) This list is placed in cells between the time they were created and the time they get values. If it is ever seen, it means that an error has occurred -- somehow a cell has been accessed that has no value.

TYPE-LIST Contains a list of the names of all defined types.

12.14 Constants

All these numbers are in lambda. They depend on the design rules of the process being used.

MIN-POLY-SIZE	2	The minimum-size constants.
MIN-CHANNEL-SIZE	2	
MIN-DIFF-SIZE	2	
MIN-METAL-SIZE	3	
MIN-CUT-SIZE	2	
MIN-ION-SIZE	5	
MIN-NOGLASS-SIZE	2	
DEFAULT-POLY-SIZE	2	The default-size constants.
DEFAULT-CHANNEL-SIZE	2	
DEFAULT-DIFF-SIZE	2	
DEFAULT-METAL-SIZE	4	
DEFAULT-CUT-SIZE	2	
DEFAULT-ION-SIZE	5	
DEFAULT-NOGLASS-SIZE	2	
POLY-OVERHANG	2	Other useful numbers.
DIFF-OVERHANG	2	
ION-OVERHANG	1.5	
METAL-TO-METAL	3	
POLY-TO-POLY	2	
DIFF-TO-DIFF	3	
POLY-TO-DIFF	1	
ION-TO-TRANSISTOR	1.5	