EPLE:
An Effective Procedural Layout Environment

by

Donald James Gamble
B.Sc., Simon Fraser University, 1985

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY

March 1988

Approval

Name: Donald James Gamble

Degree: Master of Science

Title of thesis: EPLE: An Effective Procedural Layout Environment

Examining Committee:

Chairman: Binay Bhattacharya

Dr. R. F. Hobson B.Sc. (Br. Col.), Ph.D. (Wat.)
Senior Supervisor
Associate Professor

Dr. L. J. Hafer B.S.E.E., M.S., Ph.D. (Carnegie-Mellon)
Assistant Professor

Dr. B. V. Funt B.Sc., M.Sc., Ph.D. (Br. Col.)
External Examiner
Associate Professor
School of Computing Science
Simon Fraser University

Date Approved: April 8, 1988

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

EPLE: An Effective Procedural Layout Environment

Author:

(signature)

Don Gamble

(name)

April 20/88

(date)

Abstract

Procedural language layout is a powerful mechanism for the production of very large scale integrated (VLSI) circuits. By using the algorithmic nature of procedural languages the designer can create generic routines which operate in conjunction with the designer's standard cell library. Once these routines have been created, the designer is relieved of the tedious task of generating repetitive structures by hand; examples of such structures are register arrays, address decoders, and programmable logic arrays (PLAs). The weak point in procedural layout is the creation of non-repetitive structures and the interconnecting of cells. These two tasks remain an awkward and time consuming venture.

EPLE (Effective Procedural Layout Environment) is an attempt to remove many of the awkward or frustrating aspects of procedural layout. It attempts to do so by closing the procedural design cycle so that the user may produce graphical objects from procedural code, as well as producing procedural code through graphical interaction. EPLE does this in several ways. It allows the user to capture the positioning and interconnecting of graphical objects in a graphical representation known as trails; trails may then be automatically translated to procedural code. An interpretive environment is also provided which allows the user to query the state of graphical objects.

The system developed in this project is intended to become part of a tool set used in the production of VLSI circuits. It is difficult to predict its final form, other systems may evolve from EPLE as EPLE has evolved from them.

Acknowledgements

I would like to thank the following people:

- Dr. Richard F. Hobson, my senior supervisor, who provided the opportunity to develop a cad system in a real VLSI development environment.
- John Simmons and Warren Strange, who gave me great insight into the problems with current procedural layout techniques by letting me observe them work over a long period of time.
- Mary Grimmett, for the motivation when things started to come to a halt.
- My parents, who gave me the opportunity to investigate the things which are of interest.

This work has been supported by the Natural Sciences and Engineering Research Council of Canada.

Table of Contents

Chapter 1

Introduction

Before an integrated circuit can be manufactured a precise description of the circuit must be defined by the designer. There are many ways to create this description. The circuit may be described by its characteristics, appearance, or its functionality. In whatever way the designer decides to describe the circuit, it must be a description which can be converted to a form used by silicon foundries. Foundries as a general rule want the description to be in a form which has little or no abstractions and that easily describes the physical patterns which are used in the production of the circuit. CIF[1] and GDSII[2] are both common descriptions accepted by foundries. These descriptions describe the circuit in terms of a series of patterns defined on different logically overlapping layers.

The foundries use the patterns defined on each logical layer to produce the physical image masks or to control the areas of silicon which are exposed during different processing stages. For each logical layer the user works with, the foundry may have to perform several different processing steps to obtain the result specified by the logical layer. Each processing step may require the use of a combination of different logical layers. Processing steps performed are many and varied; they include the vaporized depositing of materials, the hardening of resist through direct beam or mask exposure to light or electron streams, the removal of unexposed areas by etching, and the baking of the wafers to allow for the migration of ion implants.

The user does not generally have to worry about physical mask combinations, ion beams, or materials deposited through vaporization. The only problem the user has to worry about is that the tools he uses produce the correct final specification of the logical layer patterns. The EPLE project to date has only been concerned with circuits created using a Complimentary Metal Oxide Silicon (CMOS) process. The CMOS process has only a few logical layers used for the

construction of N-type and P-type transistors and for the interconnection of these devices. Transistors are created in areas where the 'active' (diffusion) and 'poly' (polysilicon) layers overlap. The type of a transistor, P or N type, is determined by the existence of either a 'p+' area or a 'pwell' and 'n+' area defined to surround the transistor (See Figure 1-1). The

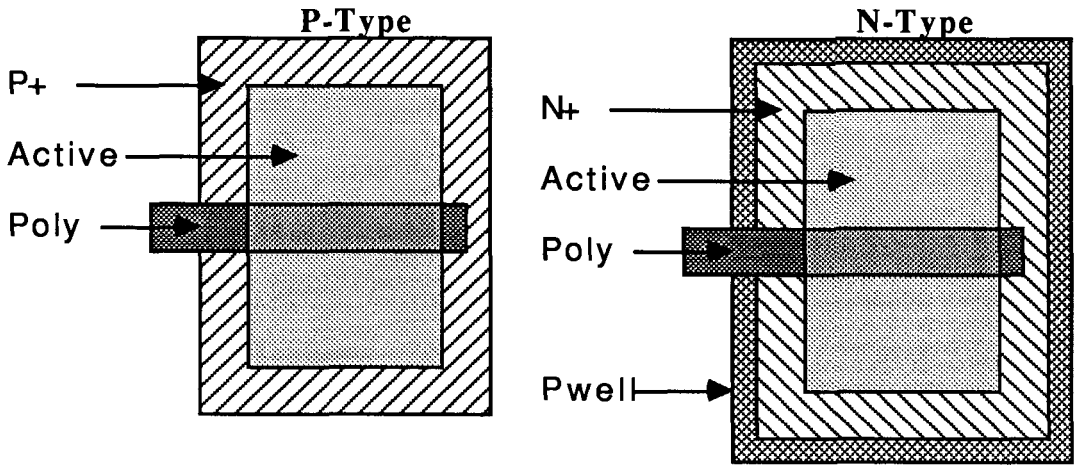


Figure 1-1.

interconnection of the transistors may be done using any of the conducting layers. The normal interconnection layers are 'poly', 'metal1' - sometimes simply called 'metal' - and a possible extra metal layer called 'metal2'. The conducting layers are joined together by two logical layers. The first is the 'cut' layer which joins 'poly' and 'metal1'; the second is 'via' which joins 'metal1' and 'metal2'. The metal layers are better conductors, with 'metal1' being the conductor of choice to run signals because of its low resistance and the ease in which it connects to 'poly'.

The patterns which describe the mask image may be represented in a number of different ways. One representation is to create all patterns out of trapezoids. EPLE represents its patterns

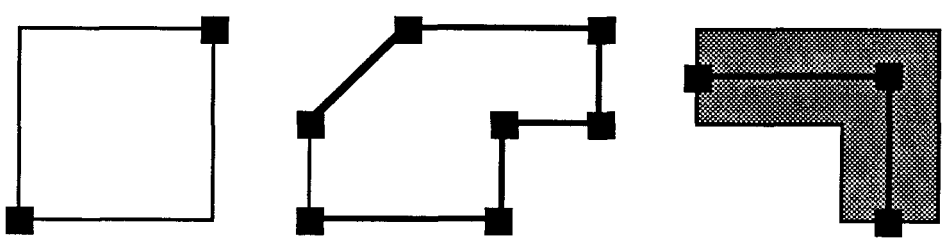


Figure 1-2.

the way a number of other systems do, using boxes, polygons, wires, and cells. A box is defined by two points, its lower-left and upper-right corners. Polygons are defined by a series of N points which describe an N-sided object. Wires are described by a center line and a width. Figure 1-2 shows examples of a box, polygon, and wire. Cells are simply named collections of boxes, polygons, wires, and possibly more cells.

In the following sections the patterns produced in the logical layers will be referred to as the physical layout . The true physical layout or chip produced by the foundry will be forgotten for the moment and we will assume that the chip produced by the foundry is composed of only the mask layers the user manipulates. Changing the way we refer to the layers the user manipulates, allows us the freedom to refer to the higher levels of layer abstractions as logical layers.

1.1. Approaches to Layout

There exist several different methodologies for producing physical layouts, as was stated above. In the past some designers have been limited in their choice of tools, due to lack of money or resources. This has lead to many different and alternative solutions being developed. Some solutions exist because they are cost effective, others have evolved from less sophisticated approaches; still others have been developed through major efforts at private companies.

Layout generation can be divided into several different methodologies. Each methodology supports a different level of data abstraction. Layout generation done with the least sophisticated methods of abstraction tend to be the simplest to implement, but also tend to be the most awkward to use. While less abstraction tends to make the tool more awkward to use, it also offers the designer the greatest degree of control over the final product. High-level layout generation systems, those which present their data in a highly abstracted view, can be the easiest to use. The penalty extracted from the designer is that he has less control over the details of the generated layout.

Geometric languages, for example CIF[1], have the least data abstraction. They represent the geometric layout information by defining objects in terms of their shape and by collecting objects together into common blocks. Typically these languages support no flow control as would be found in regular programming languages. They consist entirely of a sequential list of objects and calls to previously defined blocks of objects. The calls are simply a notation for macro substitution and take as parameters only the placement point and the orientation of the block. The blocks are commonly referred to as cells.

The only abstraction provided by the use of a geometric language is the cell. The cell allows the designer to locate large pieces of a circuit without having to worry about the interrelationships between objects within the cell. The cell, though, does not remove the concerns the designer must have for interactions between cells and between cells and other geometric objects. All other factors, such as co-ordinates which form objects and the layers on which objects are created, are left for the designer to specify.

Geometric languages are textual in form resulting in the designer having to visualize what the geometric layout will look like. An aid would be a program which interprets the language and displays the resulting image so that the designer would not have to try to visualize the image; but even with such assistance the designer is still left with the task of specifying all of the geometric information. The specifications have a textual format, which the designer is forced to develop and modify using a text editor or editor like program. It is because the format of the input does not correspond with the objects being described that this form of layout is considered awkward and tedious.

Interactive mask editors provide a methodology which offers a greater degree of data abstraction for producing circuit layouts. Using this method of circuit description the designer

interacts directly with the geometric objects. The actions which the designer performs while working with this methodology consist of defining the size, shape, layer and relative position of objects. Some systems, like CAESAR[3] and VALE[4], refer to the construction of objects as painting. The designer no longer has to be concerned with coordinate information, because this is implicit in the methodology of interaction which is being used, and may be generated from the underlying layout description when needed. This methodology is one of the most common forms of layout generation because of its 'what you see is what you get' or WYSIWYG approach.

While the interactive mask editor approach is simple to use and comprehend it is restrictive in what operations it will allow you to perform. The approach of painting a canvas with different layers of paint allows little in the way of algorithmic or structured control. For this reason it has many of the same disadvantages as geometric languages. Interactive mask editors provide little if any way to specify common objects such as register arrays, address decoders, or PLA's. Some support is provided in the form of arrays of objects, this may be acceptable for some forms of register arrays; but will not work in any case where the contents of an object are determined by the parameters of its creation, such as address decoders and PLA's. In addition, the adoption of a painting methodology requires that objects be represented in a form which allows for quick screen image updates. This generally means that a maximum horizontal trapezoid algorithm must be used to represent the areas painted[3]. The result is that single objects are represented by many trapezoids. This causes greater delays when other programs try to examine the layout because trapezoids must then be merged to form complete objects.

While having disadvantages similar to geometric languages, an interactive layout editor also has similar advantages. Since the designer completely defines the shape, size, position, and layer of every object he has complete control over the final geometric information which describes the VLSI circuit. This may be critical in situations where the analog characteristics of devices must be controlled. The designer has the extra advantage that he does not have to work with two

different formats which describe the circuit, as was the case with the geometric languages. This is a trade-off, because when everything is of the WYSIWYG form then you can not produce objects which are parameterized.

Another form of interactive layout editor is the constraint based editor, e.g. Electric[5, 6]. This is similar to a paint editor in that objects are created in an interactive fashion. The advantage is that objects - transistors, ports, and cells - are connected by wires which have defined constraints placed upon them. For example, a wire between two objects may be defined as being of variable length. Thus, when the object at one end of the wire is moved, the wire stretches to accommodate the object's new position. When examined, this type of system reveals that it is a graphical representation of a netlist, where the nodes have shape, layer, and positional properties and the edges have layer properties and spacing constraints.

It is interesting to note that Electric has been developed as an extensible system in which new tools and data representations may be added. For example, there is a design rule checker and simulator which have been added, along with a data representation for schematic diagrams. This system provides a very open view of its internal data structure and allows queries to be generated to extract information on all sorts of information contained within its database. This openness and the fact that objects within the database are positioned by constraints has allowed Electric to be interfaced with Prolog. From Prolog a user may evaluate expressions which result in modifications to the internal database of Electric.

There are disadvantages with using the Electric system. Since it represents everything as a net-list and all tools and database information are contained in virtual memory the size of the system needed to run Electric is very large. Some designers report needing a system with a minimum of eight megabytes of memory to obtain the desired performance.

A different type of interactive layout editor is the one which uses an abstraction to hide some of the effects of layer interaction, but still allows the designer to work with the physical layers of a circuit. MAGIC[7] is this type of layout editor. This methodology is very similar to the WYSIWYG editor, but it modifies its user interface to the designer by hiding some of the detail of the geometric layout. This form of editor could be referred to as a 'what you see is almost what you get' editor or WYSIAWYG. The details this kind of editor hides are the picky details of physical circuit generation, like the position and shape of dopings and wells. These details are hidden by moving the designer slightly further away from the physical layout. To form a transistor at the physical level requires that the poly and active layers be positioned with respect to each other so that they overlap as shown in Figure 1-1. Then the doping layers must be defined to enclose the transistor to determine is operational characteristics. By using a WYSIAWYG approach it is possible to define a transistor as a logical layer; this is done in both MAGIC[7] and Electric[5, 6]. This means that to define a transistor only involves placing a single logical geometric object. MAGIC, a very powerful system, also relieves the designer of spacing concerns because all objects are checked for their interrelationship with surrounding objects at the time they are created. For this to be done means that the well and doping characteristics of a transistor must also be modeled by the logical transistor layer.

This form of interactive mask editor has the same disadvantages as the first interactive mask editor described. It still has the problem of having no way to describe a PLA or any form of parameterized cell. Since the designer has moved further away from the physical layers, he has less control over exactly how objects will be generated. This could lead to problems in some rare pathological situations where the designer needs complete control over dopings or other hidden layers. One remaining problem that is apparent is that as the internal data structures of these programs become more complex trade-offs will have to be made. Magic uses a data structure called corner-stitching. It describes all the occupied and empty areas as rectangles which point to their neighbours. One of the trade-offs made by Magic was that all geometries had to be

manhattan. To produce a manhattan object means that all of the object's edges are parallel to either the X or Y axis and that all corners intersect at 90 degrees. Such a trade-off restricts the designer's ability to create extremely space efficient cells. With further work it may be possible to remove the manhattan rule, but only by increasing the complexity of an already complex data structure. Another possible alternative is to allow non-manhattan geometries, but to ignore them during more complex operations like design rule checks (DRCs). This is the trade-off the Electric[5, 6] design system made.

The interactive systems described so far all have a language which describes the objects which are manipulated. The presence of the language has not been directly apparent, but all of the above forms of systems do keep an internal representation which may be considered to some degree to be a language. There is a third type of interactive editor system, which are known as instant plot systems. These attempt to make both the graphical and the textual form of the layout directly accessible to the designer without one having to be created from the other; SAM[8] is such a system.

SAM generates both a textual representation and a graphical image for a cell from a common third representation. The third representation is a parse tree of the textual program. When modifications are made using the text editor it is not the textual information that is modified but it is the parse tree that is changed. Since the textual image displayed is simply another view of the parse tree, the textual image changes to show the current interpretation of the parse tree. The graphical image displayed is also just a different view of the parse tree, and as such it also changes to remain consistent with the underlying representation (See Figure 1-3). Using a third representation for the cell being described allows the designer to make changes to either of the two visible representations, which is really a modification to the underlying representation, while keeping both visible views consistent.

The language that SAM uses is very simple, having only a for loop and if-then-else for flow control and only a box primitive to generate objects. The reason for the simplicity is that when editing changes are made to either the graphical or textual image they have to be mapped onto the parse tree. Complex flow control would make the analysis of graphical changes much more

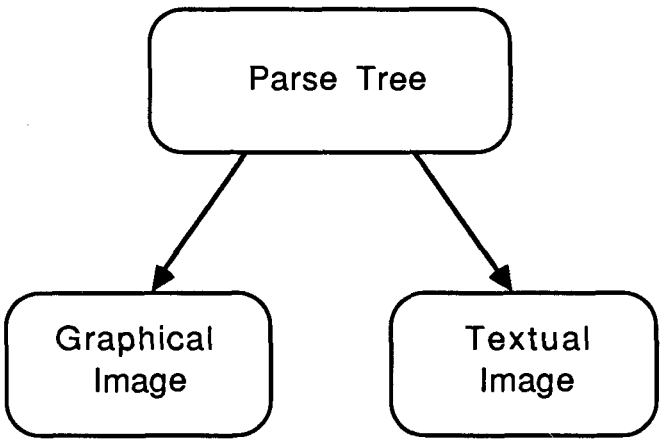


Figure 1-3.

complex. Having a single graphical command, which is self contained, allows Sam to determine exactly where an object was created. This is the opposite of systems which support more complex objects which require them to be created over several lines of code. Being able to identify exactly where an object was created allows SAM to easily update the parse tree when changes are made to the graphical image.

Systems like SAM hold great promise because they allow the designer to use the strengths of both interactive layout and procedural layout without having to deal with the short-comings of either. The problem is that SAM, while only experimental, was very slow and required a virtual environment to operate within. This currently makes using SAM for anything more than a demonstration infeasible.

A fourth approach to layout is procedural layout. This is where the total power of a programming language is used to create the physical layout. It is hoped that the powerful techniques which are applied to software can be used to help in the creation of a physical layout. The techniques being considered are parameterization, functional evaluation, and hierarchical design.

Icewater[9, 10] is a system which implements a procedural design environment. It provides a functional unit called a constructor. A constructor is a piece of code which describes procedurally how a cell should be built. A constructor is fully parameterized and may change how a cell is built based on the values of its arguments. In the Icewater system there is no concept of a standard cell library, instead constructors are evaluated as required.

A constructor contains declarations of variables, some of which may create instances of cells. Once an instance has been created it may be placed or abutted against other cells. The connections to a cell are made by naming two terminals when calling the connect primitive. Icewater tries to obtain a symbolic appearance by encapsulating the description of a transistor within a constructor. This causes higher level constructors to look very symbolic because they can refer to transistors as p1, n1, n2, etc.; and inter-connections may be stated as connect(p1.drain, n1.drain). While this is a step towards symbolic design, it does so at the expense of the space used to produce a cell. Later versions of Icewater, called IGLOO[11], have moved even further in this direction; they now provide a form of symbolic placement and routing.

There are other procedural languages which operate directly at the physical level. The reason they operate at this level is to permit optimum device and cell sizing and because the necessary macrocells have not been provided to allow them to work at a more symbolic level as was the case with Icewater. It should be noted that the constructors of Icewater allow it to be more gracefully symbolic because constructors are named objects which may be parameterized.

These other procedural languages have the disadvantage that they are more awkward to use because everything must be specified, but this is also their main advantage in that everything may be specified.

CDL[12], C Design Language, is an extendable system which allows for the specification of cells through the use of a library of construction functions. The library contains functions for the creation and placement of cells, nodes - named points in a cell, wires, boxes, and polygons. The library also allows for extensions through the integration of generators. Generators are functions which perform a specific task such as creating a register array, PLA, or address decoder. The idea behind CDL was that as generators were created the environment would become increasingly more powerful. CDL has lately begun also to show an increasing use of relative placement, which may lead to a system similar to Icewater.

There is a third form of procedural language in which the geometric objects are described by specifying the relationships which are true between objects, e.g. ALI[13]. The relationships between objects are specified in the language and then the system generates the placements of the objects. This form of language may better be described as a declarative language. When considering the positions of objects within a cell a binding must occur to the objects outside of the cell when it is placed. The process may be thought of as a form of resolution, where objects are moved around until all relationships are true.

Much of the power of this type of methodology comes from the fact that only relative positioning information is used. Having a method of describing the geometric layout without concern for the absolute sizes and positions of objects allows the designer to create the layout by simply describing the topological relationships. Some concern must still be given to gate size and this may be done by providing the maximum and minimum gate length relationships. The ability to describe just the topological relationships would greatly reduce layout time, but if complete

control of sizing information is given up this would result in unwanted circuit characteristics. The net result is that the extra time saved in placing the objects would be used in correctly specifying the operating features.

When abstraction is allowed to hide the physical shape and position of a device we have reached the layout methodology referred to as Sticks. Two such systems are MULGA[14] and LAVA[15, 16]. Using the Sticks methodology objects no longer have width, only relative position, type, layer, position, and drive strength. Inter-connection between devices is accomplished by using wires. Wires may or may not have a layer assignment. If a wire has no layer assignment one may be assigned by examining the layers associated with the terminals to which the wire is connected. If a wire needs to change from one layer to another, possibly to pass an obstruction, this may be done symbolically by overlapping the two layers, not by defining a cut as done in a less abstract methodology. The placement of objects and the routing of wires is normally done through graphical interaction, but may also be done procedurally as allowed by LAVA. The routing and placement of all objects is relative, no absolute positioning is used. This means that objects will maintain their topological relationships but will be physically moved to adjust for design rules, drive requirements, and the physical instantiation of symbolic objects.

The positioning of objects is done by defining their location on a virtual grid. The grid is, as expected, constructed of horizontal and vertical lines which intersect at 90 degree angles. The reason the grid is referred to as a virtual grid is because lines that do not intersect objects are removed to produce a more compressed design. Also, the spacing between lines is not fixed, even though it is graphically represented as a fixed distance. The true distance between grid lines is computed when the symbolic representation is converted to physical masks. This is done by examining the design rule constraints between objects which appear on neighboring grid lines.

In addition to the graphical interface, the LAVA system provides a procedural language description of the symbolic representation. This is more than a simple sequential listing of the objects and their grid locations. The language provides many of the benefits of procedural layout, flow control and variable assignment. This allows for common or repetitive structures to be coded as compact procedural units.

While stick systems are very powerful they also have significant disadvantages. Since the absolute physical dimensions and shapes of objects are not controllable there is no way to produce technology specific objects such as input and output pads. The solution to this is to add size specific objects, such as pads, to the primitive set of the language. Stick systems also share the disadvantage which is common to all abstracted methodologies - they can not in general be space optimal. If these disadvantages can be tolerated, or if a complementary system - one where space optimal cells can be produced and then later used in the stick system - can be found to augment the sticks system, then a powerful system may exist to allow the designer to work symbolically.

Using schematic diagrams is probably the easiest method for an inexperienced VLSI designer to describe a layout because designers are familiar with schematic diagrams and generally use them as their initial form of circuit description. To produce physical layouts from a schematic requires the following steps. First the components are placed and oriented so that the best interconnecting positions are obtained. Next the high level components are expanded into their transistor equivalents. Finally the last two steps are performed, establishing the wiring paths and assigning material types to the interconnecting wires. These steps were implemented in the DUMBO[17] system. Each step was implemented as a separate program, with the results produced by the last stage being passed to the LAVA sticks compiler.

There are several problems associated with this form of system. At this level of abstraction there is no control over the resulting geometric form of the design. This results in

increased space usage as was found in the DUMBO project. The average area penalty was 120%. Other problems are that the techniques used for constructing circuits at the LSI, MSI, or SSI level do not always extend well to a VLSI solution. There may be a better solution at the VLSI level which is not representable using the LSI, MSI, or SSI building blocks.

As designers select methodologies which use a greater degree of abstraction the job of designing a chip becomes smaller. The methodologies discussed above all use a form of description which describes the structure of the circuit. Even the schematic diagram describes the circuit at a level where the structural characteristics of the circuit are specified. As we move further away from the mask level of design, we no longer use a structural description - the description becomes behavioral in nature. The tools used to interpret a behavioral description are called Silicon Compilers, e.g., MacPitts[18], CMU-DA[19], F.I.R.S.T.[20], CAPRI[21], and the Yorktown Silicon Compiler[22].

The input to a silicon compiler is behavioral in nature because the interest has shifted to how the data moves, when it should move, and how it is transformed. The behavioral aspect of the description also means that the circuit may be simulated at a much higher level. When simulations are performed at this level it is possible to see if the function specified operates as expected, as opposed to seeing if subcell timing is correct. The behavioral input to a silicon compiler is normally a textual description of a data flow graph. The graph represents the movement of data through the architecture. Added to this information is a description of conditional movements of data and the operations which may be performed on the data as it arrives at nodes in the graph. Most descriptions are flexible enough to also allow the description of parallel operations. The silicon compiler then maps the operations and transfers that occur in the behavioral model to a hardware implementation.

By using silicon compilers the designer has no control over the structure of the layout or the form of the architecture produced. The architecture selected for the design is chosen from a small set of hardware implementations that the silicon compiler was designed to implement. The only part of the design the designer still has control over is the behavior of the resulting circuit, but for many designers this is the only factor with which they are initially concerned. This results in very fast development times for chips which fit within the architectures available, but the trade-off is possibly unacceptable space and speed requirements. This is becoming less a problem as silicon compilers improve.

The final level of abstraction is removing the need for a VLSI designer and replacing him with an expert system[23, 24]. These systems allow, with limited success, a non-VLSI designer to produce VLSI circuits. The goal of such systems is to produce the required layout by interacting with the user and using the encoded knowledge of an expert to invoke the correct generators, constructors, or silicon compilers to produce the layout. Some expert systems have tried to replace the other levels of layout production and perform all of the layout tasks through the use of resolution and rules; this has resulted in very slow toy like systems. The role of the expert system in layout design is better as a supervisor not a worker.

1.2. The Problem and Solution

A system was wanted which allowed the designer complete control over the final geometric mask layout, as well as a method of generating repetitive and algorithmic structures quickly. The main reason was to allow the designer the ability to be space efficient, and to allow the operating environment to be extended through the use of generators - so that common blocks could be generated automatically - and object oriented functions - so that the designer could more freely refer to the objects instead of the co-ordinates that compose an object. There were two options when beginning. One option was to use an interactive graphical editor and the other was to use procedural code. While a graphical editor allows the rapid construction of cells, it is a poor

substitute for cell generators when common structures are being created. Procedural languages allow for the construction of generators, which once built may be used to generate parameterized blocks in the future. Their problem is that when non-regular structures are created they are awkward to use. What really was needed was an integration of both types of system, but that was too large an initial step. So for historic reasons[12, 25] the choice was made to start by building a procedural layout system which could handle graphical sketches of segments of procedural code.

At the time the EPLE project began, a procedural system called CDL[12] was in use; CDL had replaced an older procedural system called SDL[25]. CDL is an extension to C[26] which allows the user to create geometric layout information in a hierarchical fashion. This system is

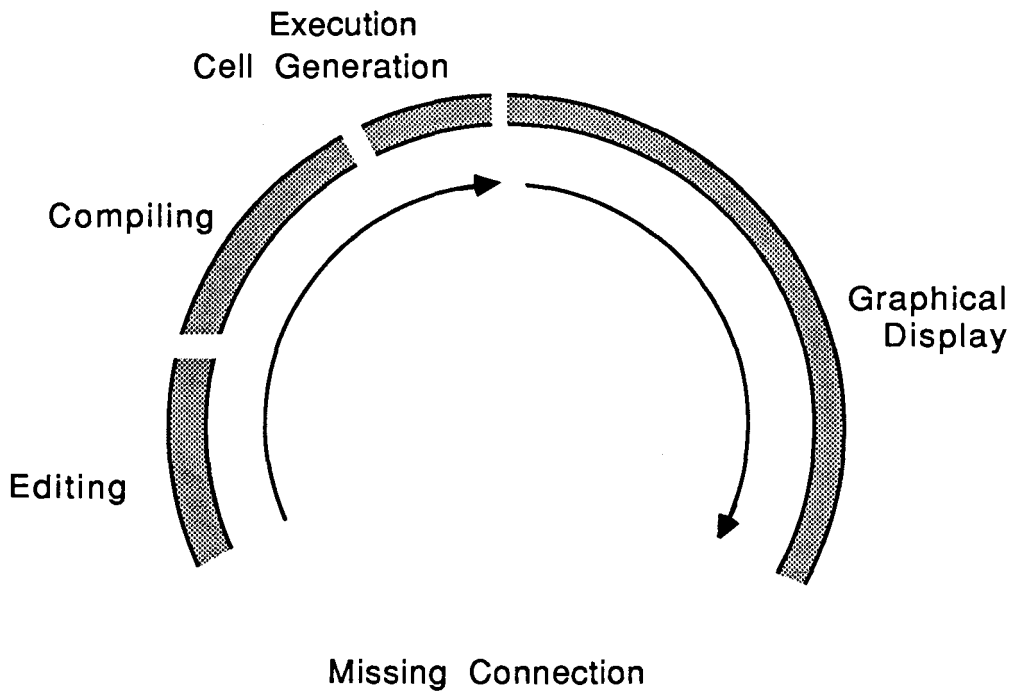


Figure 1-4.

used in conjunction with a viewing program called VLSIplot[27] which allows the designer to view graphical images which have been generated. There are several problems with CDL. The first problem is the design cycle. Procedural code is produced with a text editor; the text of the code is then compiled by a C compiler to produce a runtime module, the module is then executed

to generate a cell and then the cell is viewed by using VLSIplot (See Figure 1-4). It is felt that the design loop could be shortened if the compilation/execution phases could be replaced by a single interpretation phase. More importantly it is expected that the awkward time consuming side of procedural layout may be reducible if a link is constructed between the viewing and editing steps.

EPLE implements a link between the viewing and the editor environment by using a new graphical notation called trails. To allow the power of trails to be recognized the language that EPLE uses is no longer C, but is a vector based language, currently APL[28, 29]. Through the use of trails and object oriented enhancements (made possible because of the vector based

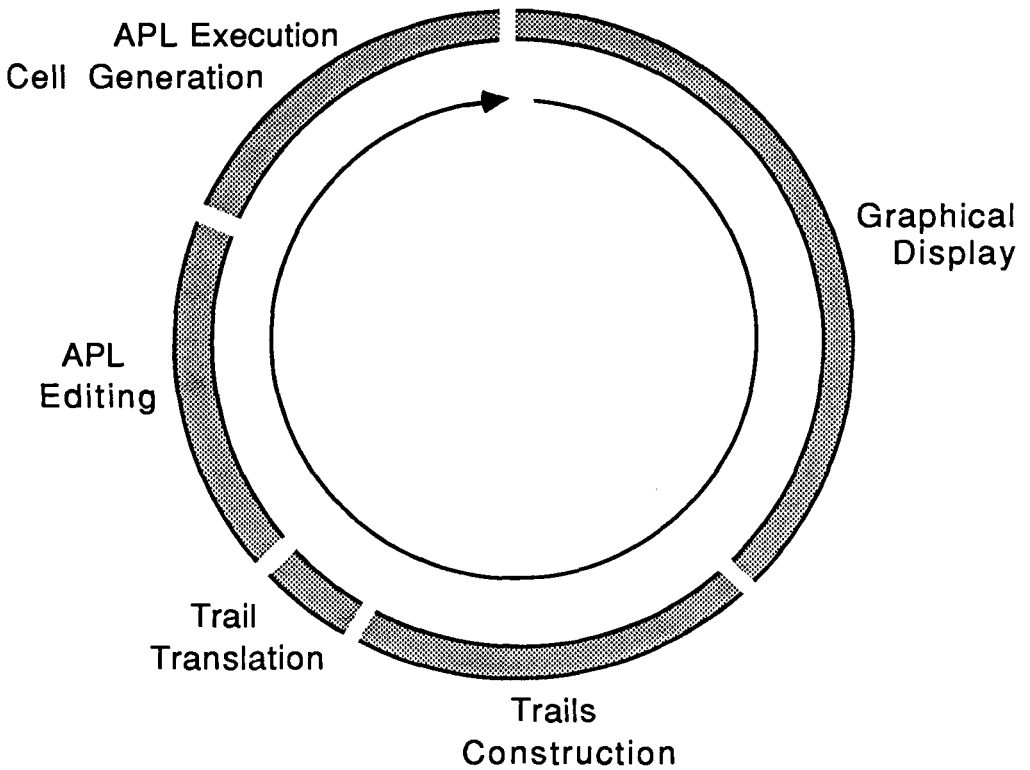


Figure 1-5.

language) a clean interface for the graphical construction of procedural code has been developed. The resulting design cycle is interactive and continuous in flow (See Figure 1-5). This allows objects to be initially created in a textual format and then allows for further modifications to be performed through both textual and graphical specification. In this way the design cycle is

continuous, textual changes may be made to effect the graphical image, while graphical operations may be used to create or replace text in the textual description.

The following chapters introduce each part of the system. Chapter two discusses the graphical display. Chapter three describes the naming of objects in the EPLE environment. Chapter four covers trails, what they are and how they are used. Chapter five discusses what the interactive language is and why an interactive vector based language was chosen. The final chapter is a collection of conclusions reached at this point in the development of EPLE. Following chapter six there is an annotated appendix which shows the EPLE system in operation.

Chapter 2

Graphical Display

The graphical display section, known as the graphical composer, forms one of the largest parts of EPLE. It is the most apparent part of the system because it shows color images of the cells under construction and also allows for the construction of trails (See Chapter 4). The graphical composer is organized as a link in the design cycle. Its job involves more than plotting cells, though that is one of its major functions. It is also responsible for integrating with the phases of the design cycle before and after it. The graphical composer displays cells which already exist, compiles information about a cell under construction, generates answers to queries from the interactive programming environment (See Chapter 5), and allows for the construction of trails and their translation into code. All these activities must be performed while maintaining an interactive viewing environment which allows the user to extract needed information rapidly.

2.1. User Interface

The graphical composer consists of several different windows. There is a display window which shows the graphical images, a control panel window, a layer selection window, a color table selection window, and a command window. The latter four allow the user to control the amount, type, or way the information associated with a cell is displayed. By examining our earlier system, CDL and VLSIplot, it was concluded that being able to remove unnecessary items from the screen would be beneficial. Removing unneeded items allows the display window to occupy a greater proportion of the screen area; this in turn allows the user to view more detail or cell area. EPLE allows the user to remove windows by displaying them as icons. This allows for the rapid selection of the window when it is needed, but keeps the window in an unobtrusive form when not needed.

The user interacts with the display composer through the use of a three button mouse and the keyboard, with a majority of the interactions being performed with mouse actions and/or a single keystroke. The three mouse buttons are organized as follows. The right mouse button, when depressed in conjunction with the shift key, displays a three item pop-up menu. This menu allows the user to select the types of operations to be performed. The first entry in the operations menu is *bounding box*. If this operation is selected then the left and middle mouse buttons are used to define the lower-left and the upper-right corners of a selection box. The second item in the operations menu is *trails*. When trails is selected the right mouse button, unshifted, produces a menu of valid trail constructions; while the left and middle mouse buttons define locations in trails and abort their construction respectively. The last operation menu item is *locations*. When locations has been selected from the operations menu, the left mouse button selects spots in the window to be converted to world co-ordinates and displayed for the user to examine.

When viewing a cell it is not enough to be able to simply see it plotted. Due to the complex nature of the information and the limited resolution of the display it is necessary to be able to zoom and/or pan around the cell. Most systems allow panning and zooming but many of these systems do not allow these operations to occur while creating objects[3, 4, 5]. To disallow the movement of the viewing window during the specification of objects was found to make the creation of many large objects awkward. To create an object at an exact location requires that the cell being viewed be magnified so that the grid is visible, but this generally means that the location of the other end of the object to be created is not visible. To overcome this problem the EPLE project has attempted to create a viewing and construction environment which allows for movement of the visible part of a cell while construction of objects is occurring. For this to occur EPLE tries to implement a none mode environment for display movement. This means that the area displayed in the graphical window should be movable no matter what stage of a command the user is currently in. This was done by using a large amount of state information which tracks the

state of an operation. By using the state information it is possible to allow most operations to happen in conjunction with panning and zooming.

In addition to the single keystroke panning and zooming operations there is also a set of additional commands which allow for the movement of the viewing window. A home key is provided which allows the user to return to the entire view of the cell. Pressing home with the shift key depressed causes the system to display the area selected by the bounding box. This operation was found to be necessary to allow for rapid movement around a cell. There are also long commands, commands that begin with a ':', which cause the current view to be pushed onto a stack, or popped off the stack, or swapped with the top view on the stack. The long commands appear in the command window so that the user can verify what he is typing, but since the mouse is generally located over the display composer window all input typed to this window is redirected to the command window. Redirecting typed input as part of EPLE's user interface was found to greatly reduce excess mouse movement, and helped prevent commands from being typed into oblivion because of incorrect mouse positioning. Another important keyboard operation is the stop key. This key allows the user to stop the plotting process. This is very important because plotting large cells can take a long time and if the view displayed is incorrect then the users time is wasted. In many cases the user may not want to plot all of a given area. The user may wish to only see enough detail to determine a selection area to zoom in on.

2.2. Cell Data

Part of the job of the display composer is to manage the storage of cell data. This is the logical place for the data to be administered because the display composer makes the most frequent use of it. The data is traversed when a cell is plotted, when answers to queries are generated, or when the viewing mode of cells, or cell instances, is changed.

Each cell has associated with it a cell structure header and a cell body (See Figure 2-1). The body of a cell does not have to be present in memory unless the information it contains is needed. A cell header contains some basic information about a cell. It is possible for just the header of a cell to be in memory if the body information has not been required by any action. If the user references or performs an action which requires the body of the cell then the cell body will be loaded into memory. If a cell is needed which has never been referenced then neither the header or body will be in memory. This causes the header to be loaded, under the assumption that the information in the header will satisfy the needs of the action which caused the reference. If further data is then needed then the body is loaded. It is hoped that in the future the routines that manage the loading of cells may be extended to allow for a working set of cells to be kept in memory. This would allow for larger cells to be handled without increasing the size of the virtual workspace provided by the workstation.

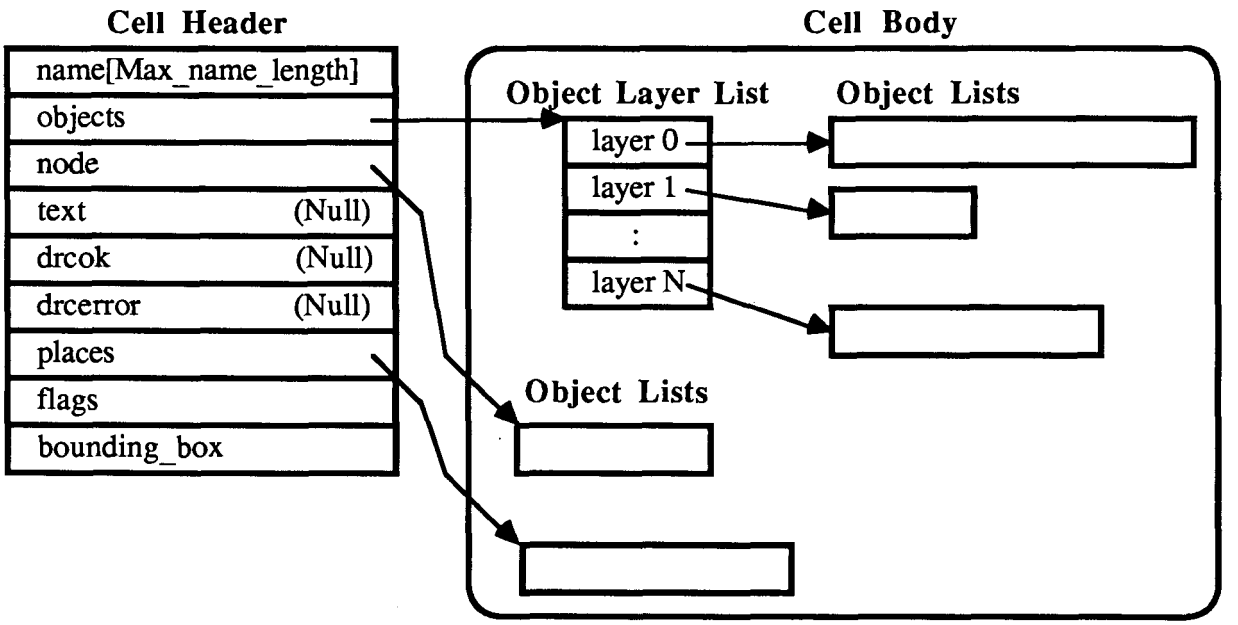


Figure 2-1.

The cell header contains pointers into the cell body but in addition to these it also contains the name of the cell, flags, and the bounding box of the cell. The name of the cell is simply a character array of fixed size into which the cell name is copied. The flags field contains

information which is used when cells are displayed. It indicates if all instances of this cell should be displayed, displayed as bounding boxes, or not displayed at all. These flags can be changed by the user. The bounding box field contains four values, the lower-left and the upper-right coordinates of the cell in the coordinate space of the cell.

A cell body is present in memory when the objects field in the cell header is not null. The objects field points at a dynamic array of object pointers called the Object Layer List (OLL). Each entry in the OLL is associated with a different logical layer. For example the first entry may be associated with pwell and the second with n+. There is one entry in the OLL for every logical layer and the entry is in the same position in the OLL as the layer is found in the layer mapping table (See Section 2.4). If the objects field in the cell header is null then so will be the node, text, drcok, drcerror, and the places fields. These fields point to Object Lists (OLs) in the cell body and are only valid when the objects field is valid. Cell data is separated into these different categories to make searching for objects on certain layers, or for only places, or for drcerrors or nodes more efficient.

The Object Lists (OLs) contain the actual data which define a cell. The data is stored in a sequential list to avoid the pointer overhead which was experienced in EPLE's display composers predecessor. Each object in the earlier system, which was VLSIplot[27], was represented as an object in a linked list. This lead to a significant percentage of space being used for pointers. It is suspected that because of the allocation/deallocation scheme and the small size of object structures that adjacent elements in the linked lists were not kept within the same page of memory. This caused page references to be over a large working set of pages, which resulted in varying degrees of system thrashing depending on the size of the cell being plotted. To avoid these problems in EPLE it was decided to keep OLs compacted in a sequential array of memory. The overhead of pointers was reduced by not maintaining all variable length data objects as linked lists, instead they are stored as sequential values.

An OL can be thought of as an array of variable length arrays. Each object in the OL has a type, an offset to the next object in the OL, a name length, a name, and a data area (See Figure 2-2). There are nine different types of objects which may appear in an OL. The first three are wires, boxes, and polygons. The fourth type is a cell placement, followed by nodes, text, drcok, drcerror, and finally end. The type end is used as a sentinel to indicate when the traverse of an OL has reach the end. Since the objects which form a cell are divided into groups by the organization of the cell header - ie: placements, text, layers, etc. are kept separate from each other - then some of the OLs may contain only a single type. This was done so that one common access routine could be used to traverse all OLs.

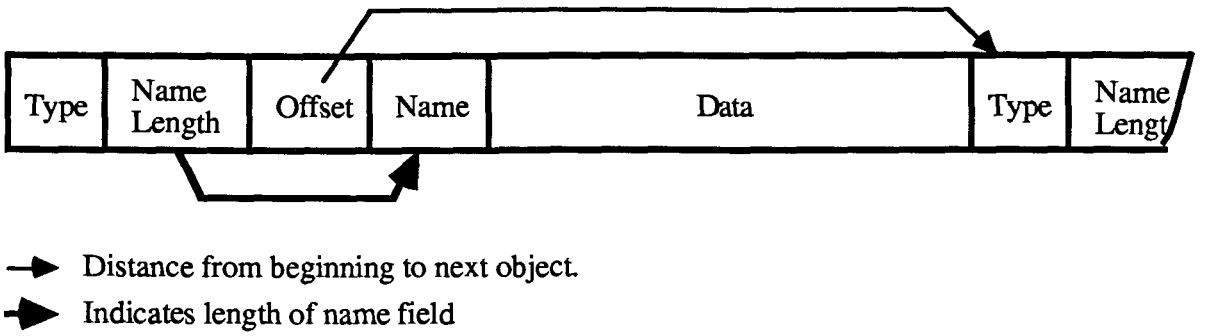


Figure 2-2.

Different information is contained in the data segment for each different object type. An object of type wire has a data segment which contains a width and N pairs of (x,y) values which represent the center path of a wire. The data segment for a box contains just two pairs of values, the lower-left and the upper-right co-ordinates of the box. A polygon object's data segment is similar to the data segment of a wire, but the polygon has no width and the N pairs represent the shape of the polygon. The node and text objects have a data segment which contains only the x and y co-ordinates of the node or text. A placement object is more complicated because its data segment contains a second data segment (See Figure 2-3). The data segment of a placement contains the following fields, cell name length, cell name - the name of the cell being placed, and

then a second data field. To find the second data field requires using the name length as an offset. The second data area contains the x and y co-ordinates of the placement, and a flag field. The flag field contains the orientation of the placed cell and display information - should this instance be plotted, not plotted, or plotted as a bounding box. The last two objects, drcok and drcerror, share the same format for their data area. The data area for these objects contains an (x,y) pair that is the location to place the name of this object. Following the (x,y) pair is a list of N (x,y) pairs that represents a polygon that surrounds the cell area in error. The types wire, polygon, drcok, and drcerror all contain lists of length 2N, but the value 2N is not explicitly stated in the data area. The routines which access these objects calculate the value of N by using the address of the next object in the list. This is possible because these lists always appear at the end of the data field. The sentinel at the end of the OL has only a type field, when its object type of end is found the traverse ends.

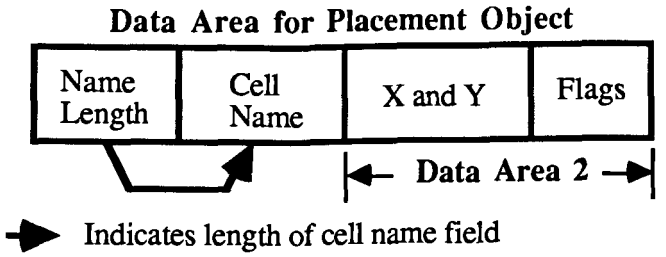


Figure 2-3.

Since cells are accessed frequently by their name during plotting and during the generation of responses to queries it is necessary to have a method of access which is reasonably efficient. EPLE stores the cells in a binary tree which is organized by cell name. When a request is made for a cell the tree is searched and if the cell is present in the tree then a pointer to the cell header is returned. If the wanted cell is not found in the tree then the header of the cell is loaded into memory by searching for the cell in the directories indicated by the search path. To allow for the smooth transition to EPLE from CDL and VLSIplot, EPLE will automatically search for the cell in the CDL file format if it does not find an EPLE formatted cell. If no cell could be loaded into memory then the user is told that the requested cell was not accessible.

2.3. Cell Data Under Construction

When a cell is under construction, this is from the time a start_cell call is done until an end_cell call is performed, the data representing its structure is kept in a separate database which is independent from the other cells which the display composer controls. This separation is for several reasons. The foremost reason is that the objects within the active cell have a much higher degree of chance of being referenced by name than do the objects in other cells. Since EPLE's procedural language builds objects which make their references relative to objects already created within the current cell, then it follows that the objects in the current cell will be referenced the most. To support the high number of references to current cell objects efficiently, requires that when an object is accessed a sequential search through the object lists not be required. Objects in the active cell are organized in a binary tree. This means that finding named objects in the current cell can be done in O(log n) time, where n is the number of objects which exist at the time of the query. Another reason for the separate database is that the cell structure discussed in the previous section required that objects be separated into layers from the time of their creation. When considering the functionality of the language it was considered beneficial not to have to define the layer an object was to be created on a priori (See Chapter 5).

When an object is created a header is built for it which contains its name, type, default layer, and a pointer field to its data. The contents of the data area varies depending on the type of the object, but it is very similar to the contents of the data area in a object on the object list. Once the header and data fields have been initialized they are linked into the binary tree (See Figure 2-4). During construction of a cell there will be many updates and queries from the language environment. Most of the queries will be for information from the data segment of a named object; while the updates will change wire widths or layer assignments of named objects. When the language environment indicates that the active cell is completed, by giving an endcell command, the cell is written out into the current directory. To force the new cell to be viewed if it is a replacement for an existing cell, the cell with the same name in the cell database is deleted.

This will result in a demand load occurring when the just written named cell is requested. An additional feature of this is that all cells are written to disk at the time of their creation so that they are available to others immediately without the user having to explicitly give a write command.

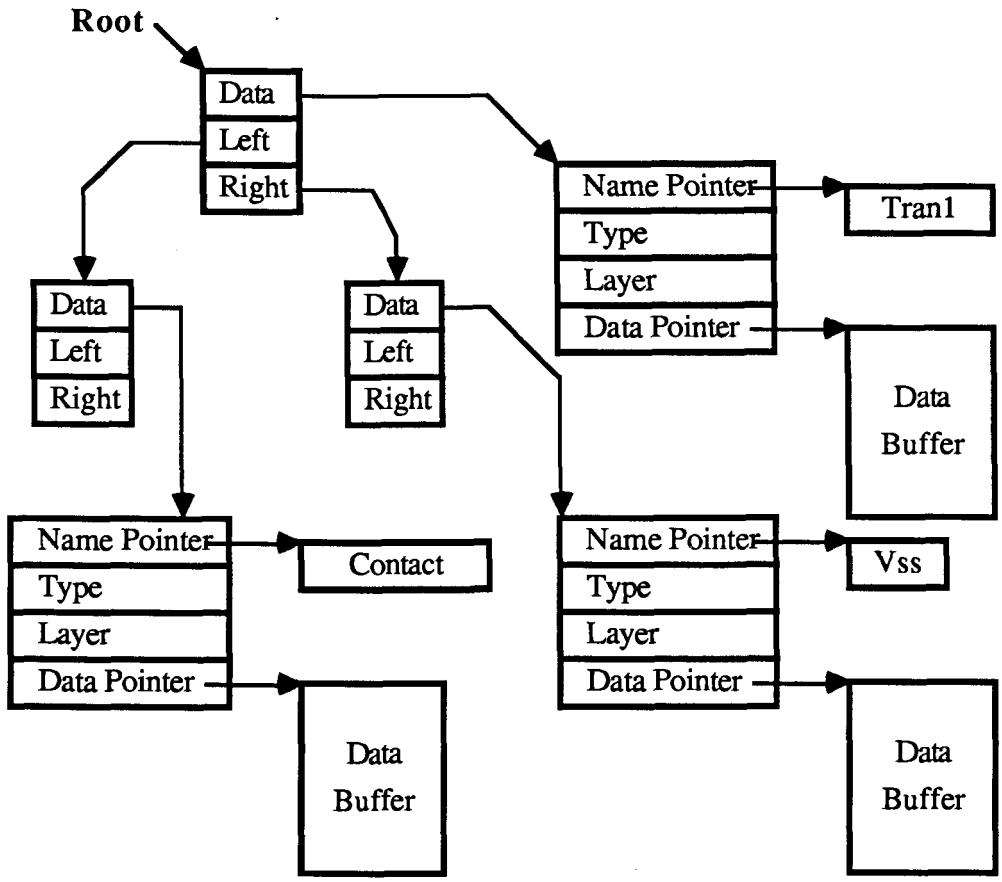


Figure 2-4.

2.4. Plotting, Layer Map, and Color Map

The image displayed in the viewing window of the graphical composer is produced by traversing the object lists associated with a cell, and by recursively traversing all of the cells placed within the cell. The user is provided with interactive mechanisms which allow him to control the selection, way, amount, and form of the information displayed. The user may additionally configure control files, before startup, which specify the plotting sequence of layers and the initial color map values.

A request for the plotting algorithm to begin may occur in two different ways. The first type of request is for a cell to be plotted so that it is completely visible. The second type is for the current cell to be replotted so that the view port into the cell's co-ordinate space is not changed. The second type of request will be transformed into the first if there is no current view port. Two reasons for replotting the current view port of the active cell are to correct for erased or missing parts of the displayed image due to window interaction, and to display changes that have occurred because of the replacement of subcells. Erased or missing pieces of the display window are not automatically updated because of the long time associated with the plotting of some large cells. It was thought that the user could better deem when it was necessary to correct the displayed image. The cell to be plotted is determined by examining the contents of the control panel window which provides a field for the specification of the current cell. How the cell is plotted is extracted from both the control panel window and the layer control window.

The control panel's fields control how a cell is plotted, with the exception of how the different physical layers are represented. It contains a field for the name of the cell to plot, the initial transformation, the zoom and pan factors, the plot depth, and the granularity factor. The initial transformation determines how the cell will appear in the display window. The cell may be viewed in a combination of mirrored and/or rotated multiples of 90 degrees. This allows the user to view greater detail by adjusting the cell so that the aspect ratio of the cell is similar to the aspect ratio of the display window, the result is that there is less unused viewing area which means that displayed objects are larger. The zoom and pan factors do not effect the way the cell is plotted, but effect the degree of change that occurs when either a zoom of pan operation is performed. The plot depth is an effective tool for limiting the amount of information displayed. By setting the plot depth to a value less then the cell hierarchy results in cells at one level deeper being shown only as bounding boxes, and cells deeper than one more level not being shown at all. The granularity value is used to hide information which would be to small to be of use. It is used to determine if text, node names, and other textual objects should be plotted. These objects are not plotted if the

scaling value to transform them from cell co-ordinates to screen co-ordinates is less than the granularity factor.

The layer control window contains controls associated with layers and some additional controls which are associated with pseudo layers. The top of the layer control window contains a vertical list of the physical layers and how each layer is currently displayed. The way a layer is displayed may be changed by clicking the left mouse button over the current display state. This results in the selected layer being displayed using the next display type. The display types are organized in a cycle so that if the user continued to click on a layer he will return to where he began. The display type may also be selected by pressing the right mouse button over the current display setting. This results in a menu of choices being displayed. The choices are: solid, outlined, hatched, Xed, and off. If solid is selected then objects displayed on the associated layer are displayed filled in. Outlined causes objects to appear with just an outline of their image. Hatched is not currently implemented, but should display objects with a fill pattern. Xed is used mostly for vias. If the object to be displayed is a box, then the outline of the object is drawn and a X is formed by drawing two diagonal lines between opposite corners of the box. If the object is not a box then it is displayed using the outline mode. The last display option is off. By selecting off as the display method for a layer causes all objects associated with that layer not to be displayed. When a layer is set to off it is also not included in some searches invoked through user actions. This means that if the user wants to know the name of a poly wire, he may set all layers off except for the poly layer, so that the search will consider only poly objects. Below the layer display list are two buttons which are shorthand methods of setting the display values. One button sets all layers to off, so that selected layers may then be turned on, and the other button sets all layers to their default values. The default values may be set for each layer before starting EPLE by modifying the layer mapping file.

At the bottom of the layer control window are many check boxes which indicate if extra features or pseudo layers should be displayed. The nodes button controls if node marks and node names should be displayed. The node text button controls only if the names associated with nodes should be displayed. The text button allows the user to control if text objects are displayed. The bbox (bounding box) button, when on, causes all placements of cells to be displayed with their bounding box, instance name, and cell name. The cell name button controls if the instance name and cell name should be displayed when the bbox button is on. The trails button indicates if trails constructed by the user should be displayed on the screen. Drc errors and oks are also controllable by using the Drc ok and Drc error buttons. The last three buttons control if the path points that make an object are displayed. There is one button for each type of object; Vrx Box shows the points that form boxes, Vrx Wire shows the points and the center path of wires, and Vrx Polygon shows the points which form polygons. Once these buttons are set, their values will be used the next time a plot is generated.

The default values for how layers are displayed and if they are transparent or opaque are controlled by values in the layer mapping file. The layer mapping file contains a line for each physical layer. On the line is the layer's name, its colormap index number, the way it should be plotted (solid, outlined, etc), the pen number to use for hardcopy output, and the pen force and speed. After the physical layers list there is a system layers list. This list contains EPLE system layer names and the color map index to be used to display them. The order that the lines appear in the physical layer list is extremely important. This order controls the order in which layers will be plotted. The first layer defined will be the first layer plotted, the last defined will be the last layer plotted.

The color map index number selects the entry to use from the color map. The color map is numbered 0 through 255, with entry 0 always being the background color. When displaying VLSI images it is important to be able to see the effects of layer interactions. The ideal solution

would be to assign each physical layer to its own bit plane. This would allow for all possible layer interactions to be represented. Since the SUN workstation only has eight bit planes it would only be possible to display eight layers if one layer was mapped to each bit plane. To limit the designer to only eight physical layers is totally unacceptable. The alternative is to only show some layer interactions and to hide less important interactions. For example, the interactions which occur between doping, active, and poly should be represented so that it is possible to see where transistors are formed, but the interactions that occur between the two different dopants may be ignored because they do not form meaningful objects. To accomplish the goal of selectively hiding some interactions we treat some layers as if they are opaque and others as transparent. When an opaque layer is displayed it hides all layers which have already been displayed in the same location, but when a transparent layer is displayed it interacts with the layers that are already present.

A constraint which had to be considered in conjunction with the above problem was how to represent the formation of trails†. It was thought that trails should exist in a very interactive format. Such a format would allow for the creation, movement, modification, and the destruction of trails without affecting the underlying physical plot. The only way to accomplish this was to draw trails with a method which would allow them to be erased without damage to the physical layers image. This meant that trails would have to be drawn by exclusive-oring their color map index with the images, because they could then be erased by a second exclusive-oring operation. The ramification of this was that if trails were to be displayed in one uniform color, instead of a rainbow of color complements for each color they intersected, then a bit plane had to be reserved exclusively for trails. This meant that there were only seven remaining bit planes to express the physical layer interactions.

---

† Trails are graphical descriptions of procedural code fragments (See Chapter 4).

The interactions of the physical layers were handled by using the opaque, transparent model discussed above. Currently EPLE is coded to have three transparent layers, but this could be made a parameter in the layer mapping file. Each transparent layer requires one bit plane, this leaves four of the seven undefined bit planes free. Three of the bit planes are used to encode the opaque colors. This gives us seven opaque colors, the eight opaque color must be used to indicate that there is no opaque color present. The remaining single bit plane in the graphics buffer is used to select super-opaque layers. Super opaques are opaques which must appear after all transparents because they do not map to legal colors when transparents are drawn on top of them. See Figure 2-5 for examples of how color map indexes are formed. Figure 2-5a shows the typical format for an index in the frame buffer which would represent the combination of some transparent layers and one opaque layer. The second example, figure 2-5b, shows an entry from the frame buffer which is displaying a super opaque image. Figure 2-5c shows what an entry in the frame buffer would look like when a trail is displayed in the selected location.

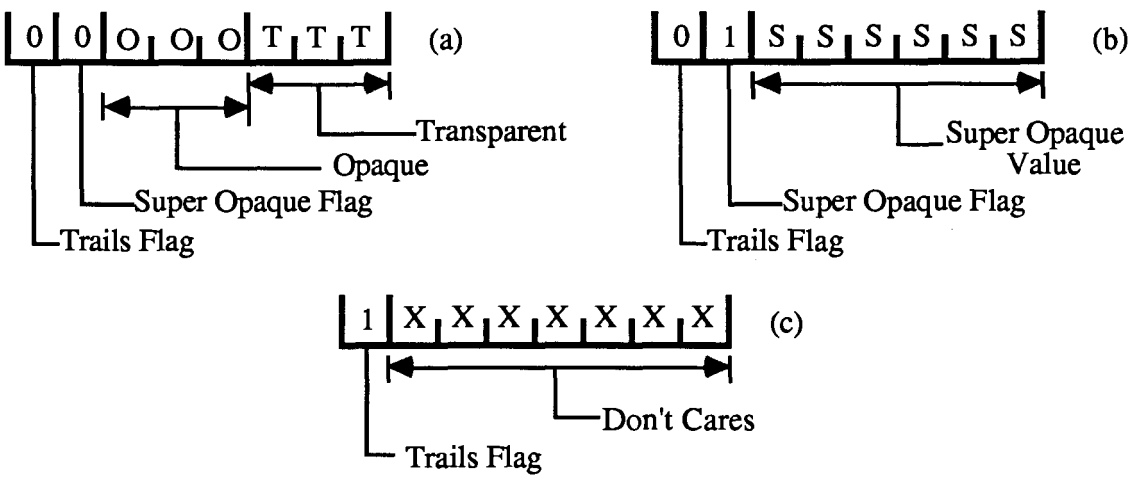


Figure 2-5.

From figure 2-5a we can see that the legal transparent values are 1, 2, and 4; while the legal opaque values are 8, 16, 24, . . ., 56 or 8k {k = 1 to 7}. Super opaque values range from 64 to 127. This means that when the layer mapping table is constructed the user may define the order the layers are drawn by the order they are listed; if they are transparent they may be assigned so by using a map value of 1, 2, or 4, and if they are opaque a value of 8k {k = 1 to 7} may be

used. The user may freely order the transparent and opaque layers as long as it is understood that any opaque layer drawn after other layers will completely hide the layers it intersects. Once all the transparent layers have been listed the user is then free to use the super opaque values.

To completely understand how a given color is generated requires an examination of the color map. When opaque and super opaque indexes are written into the frame buffer their eight bit value replaces the contents of the pixel which is being written to. When a transparent layer is written to the frame buffer its index is ored with the index of the affected pixel. This results in the color map containing values which define the color of an opaque layer and then the next 8 entries define the opaque layer with all possible combinations of the transparent layers. This pattern is repeated once for each opaque layer, including opaque layer zero which is the background. At entry 64 in the color map the super opaque entries begin. The super opaque entries are never ored with a transparent layer so they appear sequentially until entry 128. Entries 128 through 255 should have the same color values, because all these entries are used to represent trails when the top bit of a pixel is complemented by the exclusive-or operation in the process of drawing a trail. Entry 255 in the color map is also the foreground color of all the windows, as such the user may want to specify it to be different from the color of the trails. This may be done by reserving super opaque entry 127. By never using entry 127 there will be no pixel that could possibly have its top bit complimented to produce a value of 255.

The user may edit the color map entries for the physical layers interactively by using the color map selection window. The appropriate index for the color map will be computed from the layers selected in the window. The user may then modify the selected index by adjusting the red, green, and blue sliders. Once the desired color combinations have been selected the new color map can be saved. To change the color associated with trails or to change any of the system layer colors requires that the color mapping file be edited with a text editor.

The plotting algorithm used to draw the graphical image has essentially been discussed in the preceding comments but a few details have been omitted, they are discussed here. Step 1 is to scale the co-ordinates of the cells world bounding box to have the same aspect ratio as the display window. Step 2 is to examine each layer in the order they are listed in the layer mapping file, and if they are not set to off then scan_objects is asked to plot all objects on that layer. Scan objects first recursively calls the plot routine to do all the subcells. The plot routine, after transforming the co-ordinate space, recursively calls scan_objects to process all of the objects in the subcell. This continues until the lowest visible subcell is reached. Once the lowest level has been reached scan_objects begins to consider the geometric objects in the subcell, calling the plot routine to display each. The recursion backtracks until all of the objects on the current layer have been plotted for all subcells and for the parent cell. If the recursion reaches a depth greater than the viewing limit - measured by the number of subcells deep - or if a cell is determined to be drawn as a bounding box - either because the cell or cell instance is set to bounding box - then the recursion backtracks without drawing the subcell. Step 3 is to repeat step 2 until all of the layers have been considered in order. Step 4 is to traverse the plot tree one more time. This time any bounding boxes, text, nodes, and vertex points are drawn. It is important that these are drawn after the physical layers because these layers are drawn with a super opaque index. Steps 5, 6, and 7 are to draw the grid if it is selected, then to display the selection box, and finally to display any trails that exist.

Chapter 3

Naming of Objects

3.1. Previous Naming Practices

Graphical editors tend to name cells, cell instances, nodes, and ports. These names commonly exist to convey functional information to the designer or they act as reference points to which other programs may refer. The functional information communicated by nodes and ports may be used for the specification of inputs, outputs, and control. For example, four ports of a cell may be named *inputa*, *inputb*, *output*, and *select*. Nodes or ports act as reference points by allowing locations within the physical layout to be matched with areas in alternate representations. The comparison of the physical net-list and a net-list generated from a different representation is much simpler if the comparison of the net-list can begin at locations which have common names.

When procedural language systems began to evolve they used a combination of ports, nodes, cell names, and cell instance names[9, 10, 12, 13, 25, 30], the same as the graphical editors. Procedural language systems which are declarative and constraint based generally declare instances by listing the cell name as a type and the instances of a cell as simple variables or arrays. The declared instances are then positioned either explicitly or by constraining their position with respect to other objects. Below is an example of the declaration of several objects:

Declare INV: inva, invb, inv_out;

or

Declare PTYPE: t1, t2, phi0, carry[5];

Non-declarative procedural language systems generally place objects by performing procedure calls without previously needing to declare that a given cell type is going to be used. The calls return a handle to the cell instance. The handle may be used to refer to the cell instance in the future. Procedural language systems which use calls to create cell instances are more flexible than declarative systems. This is because when parameterization of a functional generator is used to

indicate the number of cells to be placed, the language which uses calls may place each instance using loops; while the declarative systems have to define the number and names of each cell type used a priori. One of the advantages of the declarative system is that the code which produces the cell instances appears together at the beginning of the cell description. This makes understanding the creation of a cell easier because the reader knows all the objects manipulated within the cell right from the beginning.

Procedural systems which return a number as the handle associated with a cell instance have a distinct disadvantage when compared to systems which use names to refer to cell instances. Once the definition of a cell is complete the number used to reference a cell instance conveys little if any useful information. It is awkward to rely on the referencing of a cell instance within a cell by a constant number, because changes to the definition of the cell might result in the reference number of the subcell being increased or decreased. If a modification occurs to the cell's code which causes an extra cell placement, or the removal of a cell placement, then constant references which are made to cells placed after the change will be incorrect . The current solution to this problem requires that the designer assign the result from a cell placement to a variable. The variable is then used to refer to the placed cell indirectly. In this way the subcell is referred to by name for the length of time the code segment is active. When trying to construct a reference to a cell which was placed within a cell, that was placed in the current cell - ie: a reference to a subcell placed two levels deep - it is impossible to assign the instance number of the deepest subcell to a variable because the instance number of that subcell became unavailable at the conclusion of its parent's execution. For this reason the reference to the instance has to be made using a constant value, but doing so does not guarantee that the cell instance referenced will be to the correct cell instance if changes are made to the definition of the subcell. These problems could be avoided if alpha-numeric names were associated with objects at the time of their creation.

Nodes, named locations in cells, exist in most procedural layout languages. The position and layers associated with a node may be retrieved by referring to the node by name. Nodes are generally used to define cell instance alignment points, input and output connections, and other points which are considered to be of interest. For example, Figure 3-1 shows a cell which has been designed so that its pwell may abutt with the next cell's pwell, but there is also a control signal which passes along the edge of the cell. This control signal causes the bounding box of the cell to expand past the edge of the pwell. This means that the bounding box of the cell is not useful in determining where the edge of the pwell is when trying to abut the two cells. As a result the designer may need to define a location which may be referenced in order to determine the edge of the pwell. The designer may define such a point by placing a node at the top-right hand corner of the pwell (See Figure 3-1). This node may then be referenced later when placing the second cell.

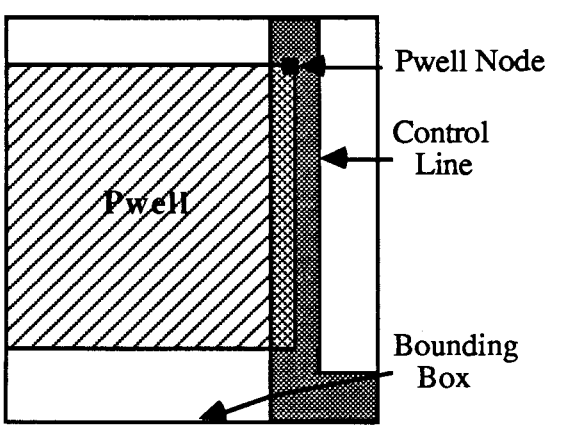


Figure 3-1.

In general, what role do the nodes defined play in the construction of a VLSI circuit? By examining the SJ16[31] and SJMC[32] chips, which were implemented using CDL, it is possible to notice some patterns of node usage. Nodes, when used, are generally placed at locations which are easy to extract from the surrounding known objects. This is felt to mean that the designers are not willing to compute yet another co-ordinate within a cell, but instead prefer to locate nodes at locations which have already been computed. The reasons associated with the designer's

unwillingness to use more nodes or to compute the extra needed co-ordinates for new nodes is felt to be due to the awkward way the language treats objects once they have been created. For example, if a designer wants to place a node near the corner of an 'active' area, he is forced to determine the co-ordinate values of the corner or alternatively to save the co-ordinate of the corner at the time the 'active' area was created. The first alternative has the problem that the designer must examine the code already written to find the values. The second alternative requires that the user modify the code which produced the 'active' area so that the corner's values are saved in variables. Neither alternative allows the user to produce a query to find the co-ordinate of an object after it has been created unless the user explicitly saved the information himself. This makes it very awkward to compute the new values.

Some cells, when constructed in a hurry, have few or no nodes at all. This results in the placement of objects without any references to other objects. Procedural code of this type is filled with magic numbers, numbers which seem to have been pulled from a hat. This type of code can make a designer cringe and flee in disgust when faced with the job of modifying a cell. The code appears to be simply a series of objects placed independently of each other. This type of code can make even a simple modification, the movement of one wire, a monumental task. It can be awkward, if not almost impossible, to identify the line of code that produced the wire since all objects are essentially created using absolute co-ordinates with no references to related objects. Blaming the designer for code of this form can only explain some of the reasons for why such code exists. The problem is similar to the early versions of the Basic language. With Basic if you used great care, a consistent style, and the problem was not overly complex, it was possible to write manageable but unstructured code; but if any of the above concerns were relaxed then the resulting program exhibited a strong resemblance to spaghetti, parts going here and there with no common strand to unite the code.

Nodes in the two chips, SJ16 and SJMC, were used to define the locations of interconnections between cells and for the placement of one cell with respect to another. When care was shown and a large degree of consideration for the future use of the cell was exhibited, then the placement and interconnection of cells was a relatively painless process. The problem is that the designer in many cases was simply content to complete a cell; designers generally did not continue the construction of a cell to the point of defining nodes at all of the external interconnection points. This again is due to the awkward nature of computing node locations. It appears a designer, in most cases, would prefer to define only a few nodes. The locations the designer selects for these nodes are selected to exhibit some useful characteristics. For example, a node may be placed so that it is aligned with the first output of a decoder. The remaining outputs of the decoder are then connected by using a magic number as an offset from the first output (See Figure 3-2). Cells defined this way cause problems when future cells use the defined cell as

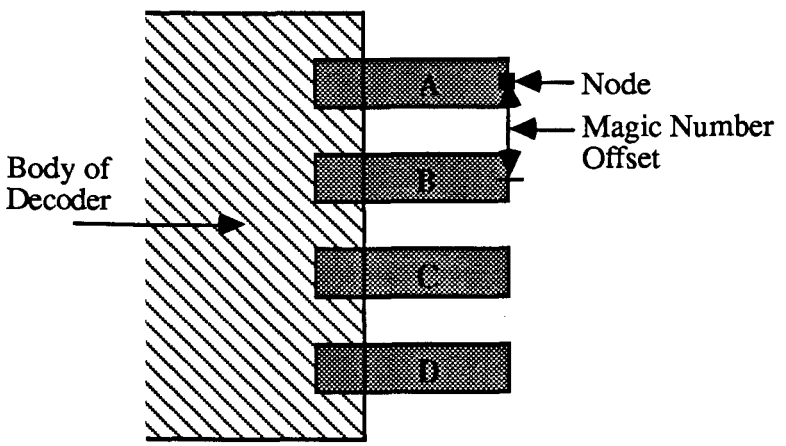


Figure 3-2.

a subcell. When interconnecting the new occurrence of the subcell in some future cell the displacement between outputs will generally not be known. This means that the designer of the new cell is forced to search through the code associated with the subcell to determined the output spacing. This is frustrating for the designer and could have been totally avoided if each output had been given a named node. The problem can be even more complex when the spacing doesn't

exhibit a constant step size, but instead uses some formula or algorithm which is dependent on the output or input under consideration.

3.2. A Better Alternative

Nodes, and handles to cell instances are good at what they do. They allow the designer to specify points of interest which may be referred to later and allow for the referencing of placed cells. The questions to be asked are 'What are nodes and instance handles really doing?', 'Can the specification of points of interest within a cell be defined in some better way?', and 'Can many nodes be eliminated by using an alternative methodology for referencing points of interest?'.

By examination of the SJ16 and the SJMC chips it can be seen that nodes allow the designer to reference a location within a subcell. The referenced locations generally reside very near or co-exist with co-ordinate points which define objects within a cell; or the node is present in the cell because it was imported from a node which was contained in a subcell inside of the cell under construction. Allowing direct references to be made to objects eliminates the need for many nodes which currently exist. EPLE allows direct references by requiring the user to provide a name for each object as it is created. Once an object has been given a name it may be referred to by using its name in a query.

In EPLE the naming of objects is not restricted to just cells, cell placements and nodes. The naming of objects is also extended to the primitive geometric objects, boxes, polygons, and wires, as well as text, drc errors, and drc oks. The use of nodes has not been eliminated, nodes still exist to cover situations where geometric objects are not conveniently placed. The name given to an object when it is created in EPLE must be unique within the cell it is defined. If an attempt is made to create an object which does not have a unique name then the object is not created and the error is signalled to the user.

Naming cell placements as well as all other objects has allowed EPLE to extend the concept of referencing objects by name. EPLE does this by allowing for the construction of hierarchical name references. The result of this is that an object may be referenced which is not in the current cell, but that is in a subcell of the current cell, or a subcell of a subcell of the current cell, etc. This is accomplished by specifying the name of the cell instances recursively followed by the name of the object for which the query is generated. The textual form of a query in EPLE is formed by joining together subcell instance names followed at the end by the name of the object desired, where each name is separated by a period. The types of objects which may be queried are all geometric objects, cell instances, nodes, drc errors, drc oks, and text. The following are examples of valid query names. The first may be the query name for the gate of the p-type transistor within an instance of an inverter. The second example may be asking for information about the output of the second driver instance within a decoder instance named decoder.

'inv1.ptran.gate'
or
'decoder.driver2.out'

The concept expressed above is very similar to structure referencing in a structured programming language. The difference here is that while most programming languages are scalar in nature, the responses generated to queries in EPLE are generally vector results. This is due to the form of the objects being handled. When a query is made of a wire, polygon, or box the query normally returns a vector of values which defines the shape of the object. Similar vector results are returned when queries are made of instance names, drc errors, drc oks, text, and nodes. The unix program 'pic' allows objects to be named in a similar way but does not allow the names to have arbitrary depth.

EPLE has adopted one approach for the naming and referencing of objects which is consistent across all object types. Within the EPLE environment every object has a path. A path is the point or points which define the shape or position of an object in space. Allowing every object to be named allows for the construction of cells which exhibit a high degree of relative

referencing. This results in objects being defined and positioned with respect to the the objects that naturally surround them, as opposed to being defined with respect to some arbitrary point in space.

Consider the difference between positioning the gate of a transistor some arbitrary distance away from a predefined point and placing the gate of a transistor a fixed distance below the bottom of the transistor's VSS contact. The first form of placement is relative, but not to the other objects in the layout. The second form of placement is described in terms of the other objects which make up the layout. This results in the layout being less sensitive to change. For example if the VSS connection in the above discussion had to be moved down, the first form of layout description would probably result in an error, but the second form would simply move the gate to remain a fixed distance below the contact. By carefully constructing a cell it is possible to create cells which may exhibit many different characteristics by changing only a few parameters. For example the channel pitch of an inverter may be affected by simply modifying the relative spacing distance between the two transistors which form the inverter. Consider how easily the problem of Figure 3-1 could have been solved if the pwell was named 'pwell' and EPLE's naming methodology was used. The position of the pwell within the placed instance of the cell could have been determined without using a node by simply generating a query using the form shown below.

'instance_name.pwell'

There would be no confusion which pwell is being requested because all object names within a cell must be unique. Thus, if there were two instances of the same subcell, each pwell would have a unique name because of different instance names. Figure 3-2 could have been solved in a similar way. Since the outputs are named, A, B, C, and D, respectively, then the connection point of each output could have been determined by four queries which used the following names.

'decoder_name.A'	'decoder_name.B'
'decoder_name.C'	'decoder_name.D'

Chapter 4

Trails

4.1. What are Trails?

A trail is a graphical description of a shape formed interactively in a way which allows the shape to be translated into procedural code, while the code retains the relative placement information described in the construction of the trail. Later evaluation of the code produces the

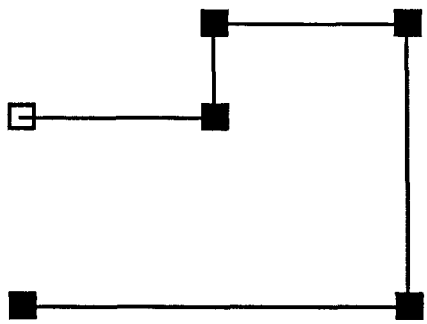


Figure 4-1.

non-relative absolute co-ordinates which describe the object's physical layout. Trails can be thought of as a short hand method for the writing of fragments of procedural code which describe

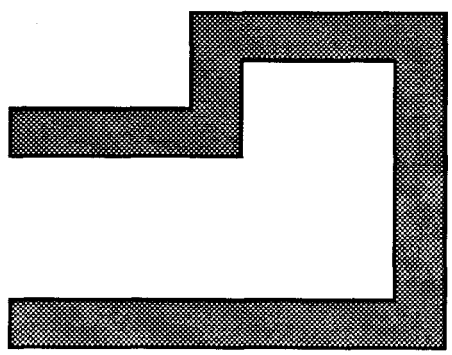


Figure 4-2.

the relative positioning and shape of objects being created. Trails are not objects, but are object's backbones, describing their shape and position of the objects. For example, the trail in Figure 4-1 may describe the shape of a wire or a polygon or neither. To determine what type of object is

formed by a trail depends on how the trail is used in the context of other functions in the procedural language. All a trail does is define the shape and position of an object, maybe more than one object. If the trail in Figure 4-1 was used to describe a wire then the result would look like Figure 4-2, but if the same trail was used to describe the shape of a polygon then the object created would look like Figure 4-3.

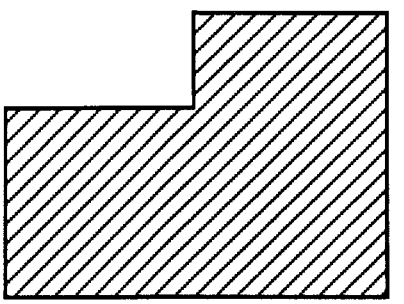


Figure 4-3.

Trails are created interactively, by using a mouse to define their shape and position. The user sees a trail as a series of marked points which are connected by line segments to form a multiple segment line. Each marked point in a trail represents a spot which will be resolved to an absolute co-ordinate in the physical layout when the code representing that trail is evaluated. The marked points in a trail - where two segments are jointed - are referred to hereafter as steps. For example in Figure 4-1 there are six steps.

Steps in a trail may be displayed differently depending on the type of step they represent. Steps may be divided into two groups, reference steps and relative steps. A reference step is a step which may be evaluated to obtain its absolute co-ordinate without referencing other steps which are within the same trail. A relative step is a step which is defined in terms of an offset from its predecessor.

EPLE has several different types of reference steps. The simplest reference step is an absolute reference step. An absolute reference step specifies a point in the physical layout by defining its x,y co-ordinate value. The user indicates where such a step should be positioned by

selecting the desired location with the mouse, there is no need for the user to know the co-ordinate values at which the step is placed.

Another form of reference step is a node reference step. This type of step is defined by using an already defined node to indicate the position of the new step. For example, there might be a node called 'input' marking the position of an input connection. By using the mouse the user may define a node reference step to occur at the node 'input' by selecting the node while in the create node reference step mode. During the evaluation of the code which represents this new step a query would be generated to determine the physical co-ordinates of the node 'input'. In this way the step is defined as a reference to the node 'input'. The step is not simply placed at the same location as the node, but has its position defined to be the same as the node. This means that objects created from a trail which includes this node reference step will move if and when the node 'input' moves.

A similar type of reference step is the path point reference step. Since every object is defined by a set of steps and each object has a name, then it is possible to produce a query of an object to obtain a desired step's co-ordinate. For example, in Figure 4-3 a query could be

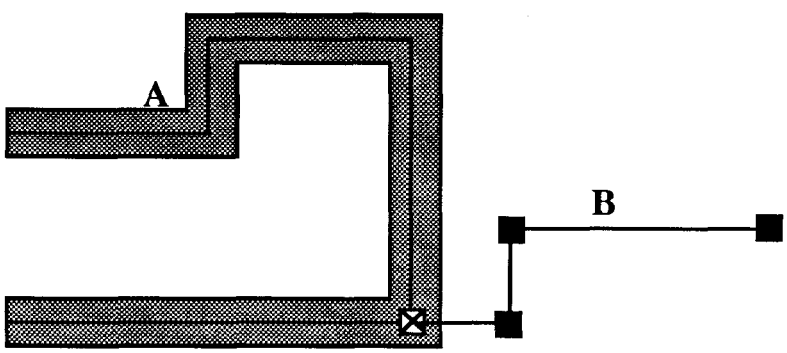


Figure 4-4.

generated to return the fourth step which defines the shape of the polygon. The fourth step in the polygon may be determined by examining Figure 4-1, because that was the trail responsible for defining the shape of the polygon in Figure 4-3. It is through the use of queries of this type that it

is possible to create reference steps which are defined in terms of path points. The user defines a path point reference step interactively by using the mouse to select a step in a trail of an object which is to serve as the reference object. Once a step has been selected as a new path point reference step then any trail which contains the new step will move as the point in the object to which the new path point step has been defined moves. For example consider Figure 4-4. If object A moves then the trail for object B will move because the start of the trail for object B is defined to be the fifth path point of object A. In this way the object B has been positioned in space relative to the fifth path point in the object A, without the need of introducing unneeded or unwanted nodes or mathematical expressions.

There are two remaining types of reference step, both extensions to the two previously described reference steps. The first is a node reference step with offset and the second in a path point reference step with offset. The difference between these reference steps and their earlier

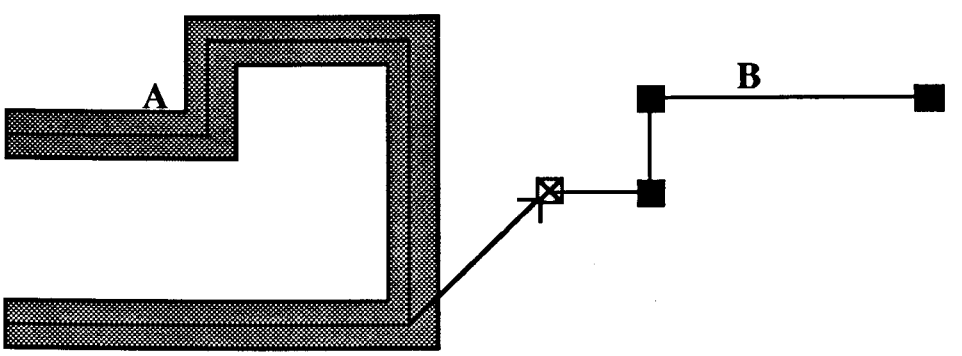


Figure 4-5.

counterparts is that their position in space is not defined to be at the node or path point that they reference. Instead they define a position in space which is a fixed constant distance away from the node or path point they reference. This allows for the description of objects which are placed relative to, but not connected to, other objects. Figure 4-5 shows a modified example of the objects from Figure 4-4. In Figure 4-5 the object to be created by the trail B is defined to be a fixed distance away form the fifth path point in object A.

In addition to reference steps there are also relative steps. A relative step is a step which is defined in terms of a connected offset distance from its predecessor. Its predecessor may also be a relative step, but eventually a relative step must have a reference step as its rooting ancestor. Figure 4-1 is a simple example of an absolute reference step which is followed by five relative steps. Each relative step is defined as an x,y offset from the previous step. In Figure 4-1 the first relative step has an offset of 100,0; the second 0,50; the third 100,0; the fourth and fifth 0,-150 and -200,0 respectively. Relative steps are used to build rigid structures. These structures are described as rigid because if the reference step to which the relative steps are defined moves then the object created by the relative steps also moves without changing its shape. In addition, if a relative step in a sequence of relative steps is moved, then all of its descendents - the steps lying further away from the reference step - will also move to maintain the same offsets they were initially defined to have. This rigidity allows the user to describe the shape of an object independently of its position in space.

From each reference step there may be none, one or two relative step sequences leading away from the reference step (See Figure 4-6). The single reference step and the two or fewer relative displacement sequences leading away from it are collectively called a path. A path is used

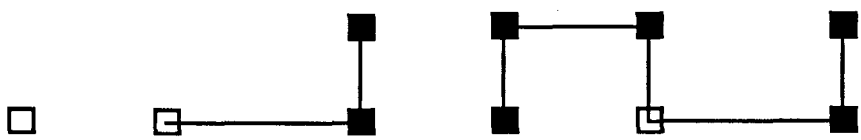


Figure 4-6.

to build basic geometric shapes which have their shape determined independently from their position in space. For example, the gate of an n-type transistor may generally be defined to have a given shape which is independent of how far it is away from the corresponding p-type transistor in an inverter. Thus, the gate should be defined by a rigid structure so that the transistor may be moved without affecting the shape of the gate.

If paths - as just described above - were the only method of describing objects then the designer of layouts would not receive many new benefits not currently available using the present methods of layout description. Paths as rigid structures have the above stated advantage of maintaining their shape when moved, but this is also their greatest weakness. Since paths are rigid they are incapable of describing a connection between two objects which may move closer or further apart in the future. This is also one of the biggest downfalls of other layout systems, except for ELECTRIC[5, 6] which allows the user to interactively change the ridgity of a wire; if a group of objects should need to be moved in the future then all of the interconnections normally have to be redone. This problem can be expected if the object being moved is moved a long distance, but the same problem existed in many cases when an object is shifted only a short distance within a channel.

To alleviate this problem it is recognized that a method is needed to describe connections which should stretch to accommodate slight movements. As it happens, the only place where stretching connections may occur is between paths, because paths are rigid structures. Let the stretching connections between paths be called bridges, for lack of a better name. This finally gives us the set of structures which form trails. We have two basic types of steps, reference and relative; a reference step and up to two relative step offset sequences form a path; paths may in turn be connected by stretchable bridges, and finally the whole collection of structures are know as trails.

The last class of objects introduced above - bridges - may be divided into two groups. The first group has only one member which is a bridge which goes directly from point A to point B via a straight line path. The second group, referred to as L-bridges, has two members which take their shape from their name. Figure 4-7 provides examples of the three types of bridges, direct, L-lower and L-upper. A direct bridge is useful when connecting objects which lie on the same horizontal or vertical axis. Examples of where this is common are the interconnection of signals which are present between objects and cells within a channel. Connecting the objects and cells in

a channel with bridges allows for the shifting of items without the need to reconnect or shorten signal paths. Not having connections which stretched was found to make the modification process for the SJ16 chip long and tedious. The designer could see the desired modification very easily, but its implementation was not easily obtainable because it required a great deal of modification to interconnecting signals. A simple thought like, 'It will work if we slide that cell over a bit in the channel,' generally required a significant amount of work because the signals which connected it to the remainder of the circuit did not accommodate this simple slide. If the code describing the connection between the two objects being moved had been created with a trail containing a bridge, then the connection would stretch when the position of the objects changed. It must be recognized that bridges do not remove the problem of unwanted signal intersections when a part is moved, it simply allows for the movement of some objects without the designer having to concern himself with rewiring every signal path.

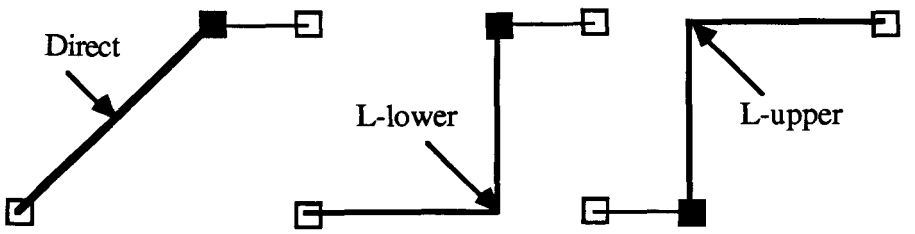


Figure 4-7.

The L-bridges, lower and upper, are used to connect objects which do not lie on the same horizontal or vertical axis, but may have interconnections which need to pass through a diagonal axis (See Figure 4-8). L-bridges are formed by the user defining the two paths which are to be connected. The point at which the 90 degree bend occurs is calculated for the user by combining the co-ordinates of the two end points. If two paths were connected with an L-bridge, where each path is made up of a single absolute reference step, x_1,y_1 and x_2,y_2 respectively, then the resulting bend point would occur at x_2,y_1. If the first point x_1,y_1 is in Cell1 and the second point is in Cell2 of Figure 4-8 then the L-bridge formed would be an upper L-bridge, but if x_2,y_2 is in Cell1 and the first point is in Cell2 then a lower L-bridge would be the result. Thus, the order the user connects the paths when defining an L-bridge defines if the bridge is an upper or lower

L-bridge. L-bridges alleviate the problem of computing a point which is well defined by its end points. It also allows for the bend point to be computed in terms of the end points, not defined as an absolute point or as a fixed offset from one of the end-points. This was observed to be a common problem in the SJ16 and SJMC projects when modifications were being made. The added flexibility of the bend being defined in terms of the end-points again allows for easier movement of the objects associated with the bridge without the need for the redefinition of the interconnecting signals.

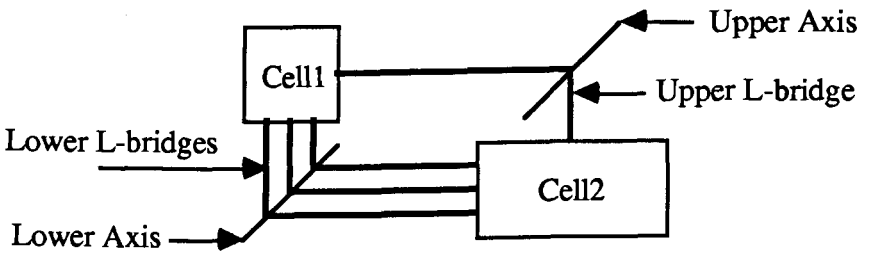


Figure 4-8.

4.2. User Interface to Trails

An EPLE user may enter into the trails mode of operation by depressing the right mouse button while the shift key is depressed. This will produce a menu of operation modes. By selecting the item labeled TRAILS from the menu the user will enter trails mode. While in trails mode depressing the right mouse button causes a menu of valid trail constructions and operations to be displayed. The menu contains the following trail constructions: Node, Offset from Node, Path Point, Offset from Path Point, Absolute, Displacement, L-Bridge, and Direct Bridge. The lower part of the menu includes operations which may be performed on the trails: Move Step, Delete Step, Delete All, and Translate Trail.

Trails to a very high degree are intended to be used in conjunction with the procedural language environment (See Chapter 5). The objects produced by the procedural language environment are displayed in the display composer window and trails are interactively constructed

on top of the displayed objects. Once a suitable trail has been constructed it is expected that the trail will be translated to code and the textual representation of the trail inserted into the current definition of the cell, the same procedural definition which produced the image on which the trail was drawn. This step may be repeated several times until a suitable point is reached to re-evaluate the procedural code definition of a cell. Once the procedural code is re-evaluated the new image of the cell may be re-displayed. When the new image is displaying all of the paths present on the screen are maintained as long as the named reference steps which form the anchoring points of the paths exist within the new image. If a reference point has been removed from the layout produced by the procedural code, for example a node or path point has been deleted from the code which was used in a path, then the path which uses that reference step will be removed from the screen since the path no longer has a valid reference step. Also any bridge which is connected to the removed path is also deleted because a bridge must span the gap between two paths and there now exists only one path associated with the bridge.

Construction of a path begins by first creating a reference step. The five different types of reference steps which may be created are listed when the right mouse button is depressed while in trails mode. The five types are: node references, offset from node references, path point references, offset from path point references, and absolute references. The meaning of these five reference types was discused in the preceding section. How they are used by the user will be discussed here. When a user wants to create a reference step he selects the type of reference step from the menu. Once a reference step type has been selected the user may continue to create steps of that type until a different type is selected from the menu. For all of the reference steps other than the absolute reference step the user depresses the left mouse button over the node or path point to which the reference is going to be made. This will produce a pop-up menu which lists the names of objects within a predefined distance from the point at which the user clicked the mouse. The pop-up menu was provided to allow the user a more error free selection mechanism.

An earlier attempt to use the closest object to the point of selection resulted in unsatisfactory results. This occurred for two reasons. Many times the user believed there was no object closer to the mouse than the desired object but was wrong. This resulted in the incorrect object being selected. The second reason for the change is a result of the first. For the user to make sure he was selecting the correct object required that he zoom-in so that a greater amount of detail was visible. This caused a delay while the image was redisplayed. The alternative of displaying a pop-up menu with a list of possible objects to be select had an added unexpected advantage. It is now possible to easily find the names of nodes and other objects even when the text for these objects is not displayed because the text scaling limit has been reached. The user simply selects the trail mode which best meet his needs; then he selects a spot near the node or object for which he wants the name. This results in all of the nodes or object names in the vicinity of the mouse click to be listed in a pop-up menu. After the user has determined the information he wanted by examining the listed names, he may abort the reference step creation operation by simply not selecting an item from the pop-up menu.

Once an item has been selected from the pop-up menu's list of nearby objects a step mark is placed at the selected reference location. In the case of an offset reference step a rubber-band begins to stretch from the selected reference point to the position of the mouse. The rubber-band continues to follow the mouse until the desired offset from the reference is indicated by the user clicking the mouse. After this second mouse click the step mark for the offset reference step is drawn and an arrow-head is placed on the end of the rubber-band to indicate that this step is an offset reference step. Now the user is free to continue to define other reference steps of the same type or he may select a different type of step or trail operation by depressing the right mouse button.

The simplest of the reference steps is the absolute reference step. When selected from the right mouse button menu an absolute reference step may be created by simply clicking the mouse

button at the desired location. Each new click of the left mouse will generate a new absolute reference step.

Bridges are defined by selecting the desired type from the right mouse button menu. Once a bridge type has been selected the user may then select two path end-points which are to be joined by the bridge. If the bridge type is an L-bridge then the user must take care in determining which path end-point is selected first, because the selection determines if the bridge is an upper or lower L-bridge (See the discussion in the Section 4.1.). If the user wishes to abort the creation of a bridge after the first path end-point has been selected, this may be done by clicking the middle mouse button.

To create relative steps the user selects the displacement item from the right mouse button menu. In this mode when a user clicks on a reference step that has fewer than two relative step sequences connected to it then a rubber-band is formed between the just selected reference step and the mouse's position. The rubber-band stretches as the mouse is moved. If the user should need to pan the display screen or zoom in or out this may be done during any operation. This allows the formation of trails which cross large areas without forcing the user to have the entire area affected visible at once. When the area has been reached where the relative step is to be located the user clicks the left mouse button over the desired location. At this time the position where the mouse click occurred will be aligned with the snap grid - which happens when any object is positioned - and a step mark will be drawn to indicate that a relative step exists at that location. Again a rubber-band will appear. This time from the newly created relative step. The user may continue to define relative steps, each time having a new rubber-band appear. This will continue until the user has finished creating the desired path, at which point the current rubber-band may be terminated by clicking the middle mouse button. The user will still be in displacement creation mode, ie. the relative offset mode, and may begin to define a new relative path starting at a new reference step.

To help position both reference steps and the relative steps EPLE provides a grid which may be turned on and off. The spacing of the grid is measured in multiples of the snap grid's spacing. The snap grid is an unseen grid to which all objects are aligned. The width of the snap grid is set in design units by the user. The multiplication factor chosen for the grid multiple is generally selected by the user to give him a clear reference grid without obscuring too much detail or being too large to approximate where the snap grid intersection points are located.

In addition to the grid EPLE also provides a method to display the path points which define objects already created. The path points for objects may be selected for display in the three basic geometric groups, boxes, polygons, and wires. This is done to avoid cluttering the displayed image with an excess of dots and lines as was the case when just one switch was used to select this feature. When the box path_point_flag is selected, all boxes are shown with dots at the two corners which define the shape of the box. The polygon flag causes dots to appear at the vertices of the polygons. The wire flag causes the center path of the wire to be drawn as a line, with the points along the line which define the shape of the wire to be indicated by dots. Using one or all of these switches allows the user to quickly identify the definition points of objects already created. This makes defining reference points simpler.

To indicate the different types of steps to the user, EPLE displays different types of steps using different notations. An absolute reference step is indicated by an empty box; a path point reference step has a box with an X in it; while the node reference step has a small diamond drawn in the center of its box; the offset reference steps use the same symbols as their non-offset cousins but they have an arrow pointing to the box from the step's true referenced location. The relative steps use a box the same size as the other steps, but the box is solid to indicate that it is a relative step. The lines which connect the steps within a path are formed using a single pixel-wide stroke. While the lines used to indicate both direct and L-bridges are formed using double strokes, where the two lines which form the double stroke are separated by a small amount. Using different

symbols for each type of step in a trail allows a user to identify which operations are associated with which steps. The different step markers may be seen in Figure 4-9.

Given that the user is not perfect, he may select operations from the right mouse button menu which allow him to modify trails. He may select to move a step, delete a step, or to delete all steps. To move a step the user simply clicks over the step to move and then clicks again over the steps new location. The move may be aborted by clicking the middle mouse button. If a reference step is moved then the path associated with it is drawn again in its new location and any bridges that exist are stretched to accommodate the move. In the case where the step moved is a relative step then all of the relative steps after it are drawn again to maintain their correct offsets.

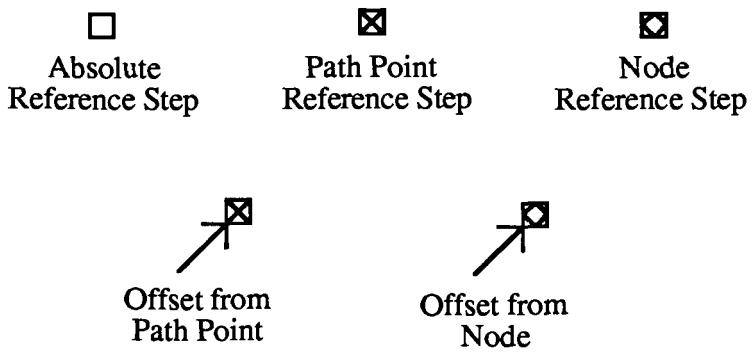


Figure 4-9.

If there is a bridge present at the end of an affected path then its position is updated to handle the change in the path's position. The deletion of a step is done by clicking over the desired step. The user is then prompted to determine if the deletion should really occur. When a step is deleted all of the components of the trail which contain that step are also deleted if they depend on the deleted step for their definition. For example, if a reference step is deleted then all relative steps defined from it are deleted because they are not anchored. A second example, if a relative step is deleted then all of the relative steps further away from the reference step are also deleted. A final example, a path is removed by one of the previous actions and a bridge is left unconnected at one end. This results in the bridge being removed because it is dependent on having two paths to interconnect. It is important to remember that trails are just graphical short-hand representations

for code segments. Once a trail has been translated to code and inserted into a procedural definition the graphical representation of the trail may be deleted. The final operation which may be selected is delete all. This causes the removal of all the trails when you what to start fresh.

When the user is interacting with trails it should be in an attempt to sketch the shape and position of new objects with respect to the objects that already exist. When the sketch of a new object is complete it may be translated by the last item in the right mouse button menu, translate trail. When in this mode the user selects a trail by pointing to one end of it and clicking the mouse. The end selected determines which end of the trail the translation begins from. The result is that the same co-ordinate values are produced when the translated code is evaluated, but the procedural expression to produce them may be different depending on the direction in which the trail was translated. After a trail has been translated into textual code, the code is placed into a buffer which may be inserted into the procedural definition of the cell at any location. It is the context in which a translated trail is used which determines what type of object is created; a trail itself is just a series of shape and position specifications.

4.3. Underlying Representation

Trails are stored as a series of interconnected steps. Each step is stored as an element in an array of structures. The length of this array is currently predefined. When all of the space within the array has been used, the user has no alternative but to delete some old trails to reclaim their space. This may be modified in the future if the current environment limit is found to be to restrictive. An example of an entry from the array which represents a step or bridge is shown in Figure 4-10. Each array entry contains six fields, which occupy a total of 28 bytes excluding the space used to store a copy of the node or path point name being referenced. The six fields are Next, Prev, Type, Win_x, Win_y, and Data. The Data field is a union of structures. The value of the Type field determines how the Data field is interpreted. The Type field may be one of the following types: Trail_Node, Trail_Path_Point, Trail_Absolute, Trail_Displacement, Trail_L_Bridge, and Trail_Direct. The Next and Prev field are short integers and are used to

connect this trail step entry with the step after and before it in a trail. The Win_x and Win_y are integers that contain the x, y screen co-ordinates of where the step is located in the display composer window.

If the Type field value is Trail_Node then the Data field should be interpreted as a Node structure. The node structure has a pointer to a string which contains the node name which defines the location of this reference step. The fields Off_x and Off_y are world co-ordinate offsets which are added to the position derived from the node name string. If both Off_x and Off_y are set to zero then the step is a node reference step. If either of them is non-zero then the step type is an offset from node reference step.

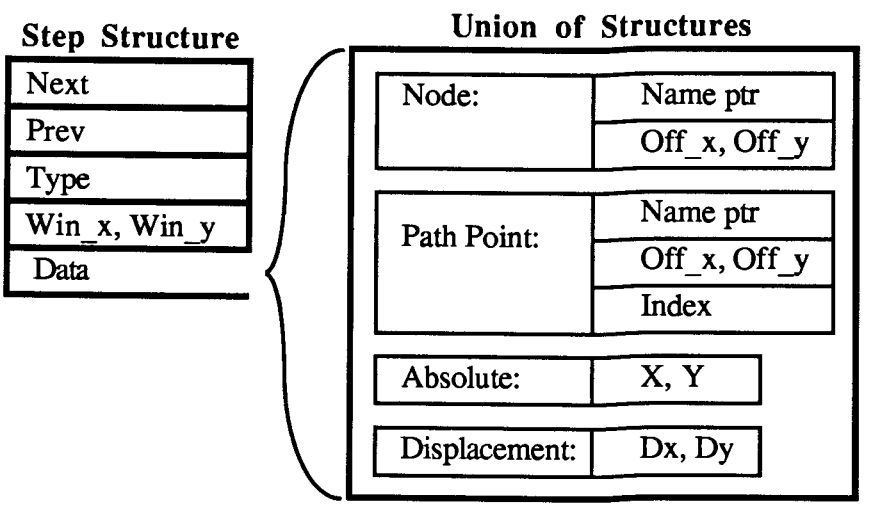


Figure 4-10.

When the Type field indicates that Data field should be interpreted as a Path_Point then the Type field must contain the value Trail_Path_Point. The structure contained in the Data field in this case has four fields: Name ptr, Off_x, Off_y, and Index. The first three fields of this structure are the same as the structure used for a Trail_Node, except the string now contains the name of a referenced object, not a node name. The new fourth field labeled Index in the structure indicates which of the path points in the referenced object should be used as a reference step. For example, if the Name ptr field points at a string containing the name "inv1.output" then the object

referenced is called "output" and it is found in a cell instance named "inv1". Which one of the steps in the object "output" will become the reference step is determined by the value of the structure field called Index. If the value of the Index field is two then the second step in the object "output" will be the reference step represented by this structure. The Off_x and Off_y fields serve the same purpose in this structure as they did in the last except they indicate the offset from a Path Point reference step.

The third type of structure which may occupy the Data field is a structure of only two fields, X and Y. This structure is used to represent an Absolute reference step. The value of the pair X and Y is a world co-ordinate value of the position of an absolute reference step.

The last type of structure which may occupy the data field is a displacement structure. This structure is present when the type field contains the value Trail_Displacement. This structure also only has two fields, Dx and Dy. This structure is present in the data field when the step being represented in a relative step. Dx and Dy are world co-ordinate displacements from the step which appeared before this step. The step which appears before this step is always indicated by the value of the Prev field.

There are two more types which may be found in the Type field, even though they do not have valid Data field structures. The first is Trail_L_Bridge and the second is Trail_Direct. When the type is Trail_L_Bridge the Prev and Next fields indicates the last step of the previous path and the first step of the next path, respectfully which a bridge interconnects. The order of assignment to the fields Prev and Next is important because it determines if the L-Bridge is an upper or lower L-Bridge. When the type is Trail_Direct the order of assignment to the Next and Prev fields doesn't matter, the two fields simply indicate the two steps to be joined together by a bridge.

When any type of reference step is created the first thing that occurs is the allocation of a free structure from the array of trail steps. If there are no free steps then the user is informed that

the limit has been reached. If the step type is Trail_Node or Trail_Path_Point then the name of the referenced object, which has just been determined at the start of the step creation, is copied to a buffer and the Name ptr field is assigned to point at that buffer. In addition when the type is Trail_Path_Point the index field is also set. The system then evaluates the reference step's node name or object name and index to determine its current world co-ordinate position. Once the world co-ordinate or the reference has been established it is converted to a screen co-ordinate and stored in Win_x and Win_y. If the reference step being created is really an offset from a Trail_Node or Trail_Path_Point then instead of stopping at this point the system enters the rubber-band mode. A rubber-band is drawn between the screen co-ordinate held in Win_x, Win_y and the current mouse position. When the next right mouse button click occurs the system exits rubber-band mode and returns to the second phase of our previous mode. Then the world co-ordinates of the last mouse click are computed and the resulting point is snapped to a grid multiple. Next the difference between this and the referenced point is computed and stored in the Off_x and Off_y fields. The final step is to draw the step marker on the screen at the location indicated by the structure just created. An absolute reference step is much simpler to create but it follows similar lines. First the mouse position is changed to world co-ordinates, then snapped to the grid, then saved in the fields marked X and Y, and finally converted back to screen co-ordinates and placed in the fields Win_x and Win_y. Again the step marker for this entry is drawn by examining the structure just created.

The creation of a relative step, a step of type Trail_Displacement, begins by first selecting a step which already exists. When the step that defines the displacement is chosen the rubber-band mode is entered. A rubber-band is displayed between the location Win_x, Win_y of the selected step and the current mouse position. When the next right mouse button click occurs the system returns to the next part of the relative step creation mode code. The relative distance between the referenced step and the mouse is computed - after the mouse's position has been snapped to a grid position. Then the relative distance just computed is stored in the Dx and Dy fields of the Trail_Displacement data structure. To indicate the step from which the displacement step is

defined, the initially selected step's index is placed in either the Prev or Next field - which ever is free - of the currently selected step. One of either Prev or Next will be free because this operation would not have begun if the selected step already had two connections made to it. Before finishing, the code responsible for a relative step computes the screen co-ordinate of this relative step and places it in Win_x and Win_y. Then the step marker and connection to the previous step is drawn. When finished the code returns to the stage where a rubber-band is created between the mouse and the current step, except this time the current step is defined as the step just created. To finally exit from the relative step creation mode the user depresses the middle mouse button. This causes the last step to be deselected as the current step and the rubber-band to be erased.

Bridges are constructed in a straightforward manner. A step is allocated from the array, even though a bridge is not considered to be a step it still occupies a step's location. Then two steps are selected with the mouse, such that the steps selected are at the end of their respective trails. The first is pointed to by the Prev field and the second is pointed to by the Next field in the step representing the bridge. The steps pointed to by this bridge step are in turn made to point back to it. This is done by setting either the Prev or Next fields, which ever is free.

4.4. Plotting Trails

Since trails are made up of a collection of relative step sequences defined in terms of a few reference steps and connected by bridges, it is not possible to plot trails in an arbitrary order. To try to plot a relative step without having already plotted the step to which it is relative does not make sense. The positioning of the step can not be determined without the evaluation of the step it is relative to. The same is true when discussing bridges. To plot a bridge without having plotted the two paths associated with it would require computing both path positions recursively. Avoiding the extra complexity introduced by a recursive algorithm required only a slight modification to the order in which steps were plotted. Steps are now visited such that all reference steps are plotted first, then relative steps and then finally bridges. The orders of the two methods are the same so there is no loss in plotting speed.

Dividing the plotting of trails into three phases allows for the Win_x and Win_y values of any step which needs to be referenced to be known in advance of the reference. This is done in the following way.

1) Plot all reference steps and update their Win_x and Win_y values.
2) For each reference step.
    2a) temp = the step indicated by the field Next.
    2b) While temp indexes a relative step.
        $2b_1$) Compute Win_x, Win_y for this step relative to the step before it.
        $2b_2$) Draw a relative step marker and a line connecting this step to the one before.
        $2b_3$) temp = the step indicated by the field Next.
    2c) temp = the step indicated by the field Prev.
    2d) While temp indexes a relative step.
        $2d_1$) Compute Win_x, Win_y for this step relative to the step before it.
        $2d_2$) Draw a relative step marker and a line connecting this step to the one before.
        $2d_3$) temp = the step indicated by the field Next.
3) For each bridge step.
    3a) Use the Prev and Next fields to find the steps at each end of the bridge. Save the Win_x, Win_y values from each step.
    3b) If the bridge is an L-bridge then draw a L-Bridge between the saved Win_x, Win_y values. Set Win_x, Win_y of the bridge to the point at the bend,
    else
        draw a Direct bridge between the saved Win_x, Win_y values. Set Win_x, Win_y of the bridge to the midpoint of the bridge.

Step one of the above algorithm draws all the reference steps. This requires identifying the type of the reference step and if the type indicates it, the translation of the name pointed to by the structure into a world co-ordinate. The translation of the name can be quite complex. For

example, each subcell instance nested in the name requires that a transformation and translation transformation be applied to the co-ordinates obtained from the final object. Once the world co-ordinates of the named reference step are obtained then any offset from the reference step must be added and the resulting value must be transformed to screen co-ordinates. These values are then saved in Win_x and Win_y and are used to draw the reference step's image on the screen. Step two relies on the fact that all the reference steps now have a valid Win_x and Win_y value. During step two the relative step sequences, all of which start at reference steps, are followed. There may be a step sequence defined to begin from both the Next and Prev fields of a reference step, steps (2b) and (2d) handle these cases respectively. While following a relative step sequence the world co-ordinates of a relative step are computed by remembering the world co-ordinates of the last step in the sequence and adding the relative offset of the current relative step. These world co-ordinates are then translated to screen co-ordinates and the image of a relative step is drawn with a line drawn to connect this step to its predecessor. The next step in the relative step sequence is found by using the Next field in the current relative step structure. The third and final step in the plotting of trails is to search the array of steps for all bridges. When a bridge is found the two end-points are determined by examining Win_x and Win_y for each of the steps connected to it by the fields Next and Prev. The code then simply draws the correct form of bridge, as selected by the Type field, between these two screen co-ordinates.

4.5. Translation of Trails to Code

The translation of a trail to an APL code fragment is done by the user identifying one end of the trail as the starting point. Since a trail consists of paths connected by bridges the translation of the trail may be done in pieces. As each path is encountered during the translation, starting from the beginning of the trail as defined by the user, the path may be translated as an independent structure and then joined to the path which follows it by an expression that represents the type of bridge separating it from the next path. This can be done because a path is a rigid structure which is only affected by its reference step and the sequences of relative steps which are associated with

it . Below is the algorithm used to translate trails into code fragments. Following the algorithm is an example.

1) start = the step indicated as the beginning by the user.
2) current = start; buffer = null string; previous = -1
3) prev_start = previous.
4) Traverse the trail, advancing current as the index, looking for the reference step. When the reference step is found then ref = current. Previous always lags behind current by one step.
5) Continuing from ref, again using current as an index, looking for the end of the path. Previous lags behind by one step from current. The path ends if we walk off the end of the trail or if we find a bridge. If we find a bridge then bridge = current.
6) end = previous. This is the step before walking off the end of the trail or the step before the bridge in step (5).
7) Call path_converter(prev_start, start, ref, end) put the resulting string in path_string.
8) If we found a bridge in step (5) then
    8a) buffer = buffer || "(" || path_string || ") ".
    8b) If the type of step[bridge] is an L-Bridge then
        $8b_1$) If the Prev field in step[bridge] = end then
                buffer = buffer || "lxy ",
            else
                buffer = buffer || "lyx ".
        else the bridge is direct, buffer = buffer || ",".
    8c) start = the step the other side of the bridge, ie. the side not traversed; current = start; previous = bridge.
    8d) Go to step (3)
9) We didn't find a bridge so we must be at the end of the trail. Add the last path to the buffer. buffer = buffer || path_string.
10) Copy buffer to a text buffer which may be inserted using the editor in the procedural environment.

Step (7) in the above algorithm has the call path_converter(prev_start, start, ref, end). The arguments to path_converter are as follows: the index of the step before the start of the path, the

index of the step which is the beginning of the path to be translated, the index of the reference step in this path, and the index of the step at the end of this path. Path_converter uses these arguments to indicate which path in the step array will be translated to code. Below is the algorithm used by path_converter().

Path_converter(prev_start, start, ref, end)

1) Is there no relative step sequence before the reference step.
   If start != ref then
   1a) Traverse the steps from start to ref counting the number of them, call this value B, also let prev_ref be the index of the step before ref.
   1b) Allocate space to hold 2*B values, call it 'before'.
   1c) Copy the displacement values from the relative steps that are before the reference step into the array 'before' such that steps which are closer to the reference step have their displacements placed first in the array.
2) Is there no relative step sequence after the reference step.
   If ref != end then
   2a) Traverse the steps from ref to end counting the number of them, call this value A.
   2b) Allocate space to hold 2*A values, call it 'after'.
   2c) Copy the displacement values from the relative steps that are after the reference step into the array 'after' such that steps which are closer to the reference step have their displacements placed first in the array.
3) Find the type of function exhibited by the displacements just copied. If there are no relative steps in either the 'before' or 'after' arrays then don't make the call for that array. The function displacement_function may modify the contents of the array and adjust the second argument to indicate the new length of the array.
   before_function = displacement_function(before, B).
   after_function = displacement_function(after, A).
4) after_string = null string, before_string = 'rxy '.
5) If A != 0 then convert the 'after' array to text by the following,
   5a) For each value in the array 'after', let i be the index

5a1) after_string = after_string || textof(after[i]).

5b) If the after_function is 0, 1, or 2 then
after_string = after_string || "dxy ",
after_string = after_string || "dxty", or
after_string = after_string || "dytx" respectively.

6) If B != 0 then convert the 'before' array to text by the following,

6a) For each value in the array 'before', let i be the index

6a1) before_string = before_string || textof(before[i]).

6b) If the before_function is 0, 1, or 2 then
before_string = before_string || "dxy ",
before_string = before_string || "dxty", or
before_string = before_string || "dytx" respectively.

7) Examine the type of the reference step indexed by ref and generate code for a query which will get the co-ordinates of this step. Place the text of this query in the string ref_string.

8) string = after_string || before_string || ref_string

9) Free(after) if allocated, free(before) if allocated.

10) Return string.

The above algorithm works on basically the same principle as the earlier algorithm, it divides the problem into subpieces. Here a path is taken and divided so that the relative step sequences may be handled separately and then the reference step is processed. The relative step sequences are each examined in turn to determine what types of characteristics are exhibited by their displacements. If a characteristic pattern can be identified in the displacement vector then extra zero values are removed from the vector and a function for the application of the displacements to the reference point is selected. For example, if we consider the vector of displacements to be labeled v[0], v[1], ..., v[2*N], then a vector which exhibits the following characteristic

SUM(v[1+4i] + v[2+4i]) = 0 for all i producing valid indexes,

produces a manhattan geometry and may be represented as a vector of N values and a function which applies the values alternately as displacements in x and then in y. This function is known as "dxty" or Displacement in X Then Y. An alternative characteristic which may be found in the vector is

v[0] + SUM(v[3+4i] + v[4+4i]) = 0 for all i producing valid indexes.

When this is found it also indicates that the displacements produce a manhattan geometry except that this characteristic requires the function "dytx" or Displacement in Y Then X to be applied to the N remaining displacements after the unnecessary zero values have been removed. The final characteristic is the vector of values which fits neither of the above characteristics. In this case the function that is used to combine the displacements with the reference point is dxy or Displacement in X and Y.

When both of the possible relative step sequences have been converted to text, then the algorithm continues and generates the textual query needed to extract the co-ordinates of the reference step, see step (7) above. There are the five different type of reference steps, only four of these involve the production of real procedural queries. The fifth type of reference step, an absolute reference, is simply converted to code as two literal numeric values. The four remaining reference steps may be divided into two groups, simple references and references with offset. To understand the following queries completely it may be useful to refer to chapter 5 and remember that the language being used is evaluated from right to left, not left to right as normal. Considering the simple reference steps first there are two types, node references and path point references. Both of these queries request the path associated with a named object, for example if a node named 'output' was present in an instance called 'inv1' then the query for the path of the node would be as follows.

```
path 'inv1.output'
```

The word 'path' in the above example is a monadic function. It takes as its argument the name of an object and returns as its result a vector containing the co-ordinates which describe the queried object. Since a node and a path point are both objects, then part of the query generated for both would be of the form shown above. A node has a path which is simply an x,y co-ordinate pair; while an object for which a path point is requested may have N x,y co-ordinate pairs returned as a 2N element vector. This means that the query for a node is just as shown above. A query for a

path point is more complex because the path point must be selected from the vector result returned by the path function. The construction of the more complicated query is accomplished using a selection function to select the x,y pair from the vector. For example, if the same above named object was a wire having three points instead of a node and the selected reference point was the last point in the wire then the query would look as follows.

3 sxy path 'inv1.output'

Here the diadic function sxy uses its left argument to select the third pair of values in its right argument. As a result the above example is the format for a path point reference step query.

If a reference step is also defined to have an offset, as is the case with offset from node and offset from path point reference steps, then the above format of queries is still valid except that an additional offset is added. For example, if the last example reference step was defined as an offset from path point reference step with an offset of 100 in x and 200 in y, then the query would be the following.

100 200 + 3 sxy path 'inv1.output'

Once the translator has generated the after_string, before_string and the ref_string then they are joined together, step (8) of the above algorithm, to form the textual query for a path. Consider the path shown in Figure 4-11. Assume that the translation begins at the left most end

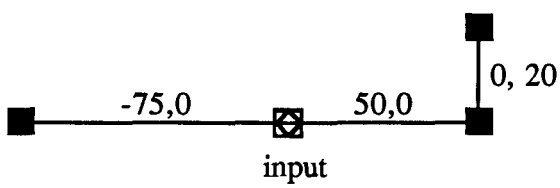


Figure 4-11.

of the path. The textual code produced when Figure 4-11 is translated is shown below.

-75 dxty rxy 50 20 dxty path 'input'

Reading from right to left we first have the query for the path of the object named 'input'. Assume the node input is located at 100, 200. The query, path 'input', generates the vector

100 200. This vector then becomes the right argument to the function dxty which has a left argument of 50 20. The result of the application of the dxty function is the vector

100 200 150 200 150 220.

The next function, rxy, is placed after any relative step sequence which appears before the reference point. The purpose of this monadic function is to reverse the order of the pairs in its right argument. This must be done to allow the next displacement function to continue the path correctly. After the function rxy has been evaluated the resulting vector is

150 220 150 200 100 200.

The final piece of code to be evaluated, -75 dxty, completes the path by using the above vector as its right argument. This produces the final vector of co-ordinate pairs, which describe the paths physical placement. The final vector is shown below.

150 220 150 200 100 200 25 200.

Remember, that this is just the translation of a path. When a trail is translated the same thing occurs for each path in the trail and then all of the path translations are joined together with expressions which represent the interconnecting bridges. For example, if two paths are joined by a direct bridge then the resulting code fragment would have the following format in which the two paths are joined by concatenation.

(path expression one) , path expression two

If the two paths were joined by an L-bridge then the comma in the format example above would be replaced with the function LXYor LYX whichever was required to produce an upper or lower L-bridge. Note in the format example that path one is bracketed. This must be done so that it will be completely evaluated before it is joined to path two.

Chapter 5

The Procedural Language and Environment

Part of the desired EPLE system is a procedural environment where the user can compose and edit procedural routines with graphical assistance, evaluate immediate expressions and functions, debug procedural routines in a step wise fashion, cause modifications to the cell database, allow user enhancements to the environment through procedural definition, and cause both graphical and non-graphical results to be displayed. These operations are to occur without the need for a compilation step which would have extended the design cycle. In addition, the language used should have the ability to express the designer's wants in a way that is similar to the way the objects were envisioned; further, it should capture the contents of an object within a variable and be able to express operations upon these variables.

The current implementation of EPLE has some of the features listed above, the remainder will have to be added in the future. The missing language features can mostly be attributed to the fact that the prototype language environment for EPLE was not constructed from scratch, but was molded out of an existing APL[28] environment. This presented limitations which had to be worked within. In the following sections of this chapter the current APL/EPLE procedural language and environment will be described as well as envisioned extensions to EPLE which will lead to the desired system.

5.1. The APL/EPLE Procedural Language

The procedural language that was to be selected or designed for EPLE had to exhibit several essential characteristics. First and foremost, the language had to allow for the construction of geometric objects dynamically and allow for these objects to be manipulated by expressions and stored in variables. Secondly, the language had to avoid a long compilation phase, and finally the third requirement was that the language had to support a clean interface to the cell database. In the end, because of the limited implementation time, the STSC APL[29] language/environment was

selected, extended and connected to the remainder of the EPLE system. By using APL, the concepts of procedural design within EPLE could be demonstrated, while only having to partially sacrifice the third requirement - a clean interface to a cell database.

The reason APL was chosen instead of a more conventional language was because it has two major advantages over most other languages. APL is interpreted and handles vector arguments and vector results very easily. Being able to handle vector arguments easily allows for a completely different style for expressing the formation and the queries of VLSI objects. Systems not having vector arguments, vector variables, and vector results generally create objects through a series of procedural calls which extend an object already created. For example, Figure 5-1 shows a CDL code fragment which creates a polygon on the active layer and then a wire on the poly layer. Note how each of the objects is completed by a second or third line of

```
layer(active);
polygon(0,  0);
     dxty(100,  50,  50,  END);
layer(poly);
width(30);
wire(100,   100);
     xy(200,  200,  200,  400,  END);
     dxty(100,  -100,  END);
```

Figure 5-1.

code. The extra lines of code are needed to complete an object because it is not easy or not possible in many scalar languages to express a series of co-ordinates as an expression. Using APL, with a few extra functions to interface with the remainder of EPLE, we can accomplish the same thing while also naming the objects. An Example is shown in Figure 5-2. Remember that APL expressions are evaluated from right to left, not left to right as are most other language's expressions. The first line of Figure 5-2 creates a polygon on the 'active' layer and names the polygon 'obj1'. The second line of code computes the co-ordinates of the wire. The third line

then creates the wire 'obj2' defined by the values in the variable A; then the wire's width is set to 30 and the wire's physical layer set to the 'poly' layer.

```
'active' LYR 'obj1' PLYGN 100 50 50 DXTY 0 0
A←100 -100 DXTY 100 100 200 200 200 400
'poly' LYR 30 WIDE 'obj2' WR A
```

Figure 5-2.

Vector based languages, like APL, have an advantage over scalar languages when returning the results from a query. The queries generated by EPLE users, in code or as immediately evaluated expressions, will generally be returning information about the shape of an object, which in EPLE is a series of co-ordinate values. The most suitable form of object to represent this type of result is a vector. It is better than an array because it has no static length and it doesn't have to be predeclared. It is also better than a data structure, like a linked list, because there are no extra fields which are awkward and confusing to the result being conveyed. A third advantage of APL is that the vector result may be returned and used in an expression. This allows for a very powerful way of expressing object creations and modifications.

Even though APL is a very powerful computation language, it was not meant to handle objects other then numeric and character vectors of arbitrary rank. Since EPLE needs to deal with descriptions of objects like nodes, wires, polygon, and boxes it was necessary to introduce our own set of functions to APL. These functions control the construction and the manipulation of the objects within the EPLE database. In function calls that effect EPLE's database the objects are referenced by using their names as handles. The additional functions added to the APL workspace to support EPLE have taken several forms to date. There are functions which create named objects in the database from a vector of co-ordinate values and a character string, functions which assign or change properties of existing objects in the database, functions which query named objects for values, and the simplest of the functions, functions which operate on numeric vectors to produce new numeric vectors.

There are five different functions for producing primitive EPLE objects. They all take a vector of co-ordinate values as their right argument - a path describing the objects shape, a character vector as their left argument - the object's name, and return the left argument character vector as the result. Figure 5-3 contains a list of the current functions added to the APL/EPLE workspace to allow it to create primitive geometric objects in the EPLE database. The general format of an APL statement in which these five operations are found is shown in the following example.

<functions modifying object> 'obj_name' function <expression producing co-ordinates>

In the example above the expression producing the co-ordinate values may be a simple vector of literal co-ordinate values or more generally will be an expression consisting of queries to the EPLE database and continuation functions. If the expression to produce the co-ordinate values is extremely complex, which generally means the expression is too long to fit into the same line as the construction function, then the expression may be placed on a preceding line with its results

- PLYGN - creates a polygon.
- WR - creates a wire of default width.
- BX - creates a box.
- NODE - creates a node.
- TEXT - creates a textual label.

Figure 5-3.

assigned to a temporary variable. This process may be repeated several times, if needed, so that arbitrarily long or complex co-ordinate expressions may exist. The example below shows how this may be done.

```
TEMP←<first part of the expression producing co-ordinates>
TEMP←<continuation of the expression producing co-ordinates> TEMP
<object modifying functions> 'obj_name' function TEMP
```

The most common query into EPLE's database is the PATH query. The PATH function is a monadic function which takes as an argument the name of the object for which the co-ordinates

of its shape are wanted. The queried object may be contained within the current cell or several layers deep within subcells. The naming conventions introduced in Chapter 3 are used to indicate which of several objects the query selects. The resulting vector of co-ordinate values is transformed so that the co-ordinate values are with respect to the co-ordinate space of the current cell. This query is generally used to find the location of an object so that another object may be created with respect to it. Graphical trails, described in the preceding chapter, make heavy use of this function in implementing reference steps.

Since the PATH query returns the entire co-ordinate vector describing an object's shape it is necessary to have some additional functions to make working with co-ordinate vectors more convenient. For example, when a path point - a single co-ordinate describing part of an objects shape - is used as a reference step we only need the co-ordinate of a single step within the path of

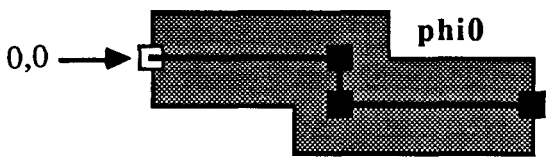


Figure 5-4.

the object. This means that we need an operation which will allow us to select a step within the vector returned, i.e. a single x,y co-ordinate pair. Figure 5-4 shows an object named 'phi0', which is defined to begin at 0,0. If another object was going to be defined from the third step in 'phi0' then a PATH query would be made to determine the co-ordinates of the wire, but this would result in the vector of values shown below.

0 0 200 0 200 -50 400 -50

Only the third pair 200,-50 is needed, so it must be extracted from the vector returned by PATH. This could be done with an APL expression but since it is a common EPLE operation the operating environment was extended by the addition of the function SXY. SXY takes a vector of

co-ordinates as its right argument and an index i as its left argument. The result returned by SXY is the i'th x,y pair in the right argument. Thus, the expression needed to determine the third step in the object 'phi0' is:

```
3 SXY PATH 'phi0'
```

Continuation functions are another type of function found in expressions which produce co-ordinate vectors. There are three different types of continuation function, but they all take a vector of co-ordinates as their right argument, a vector of displacements as their left argument, and produce a result which consists of the right vector extended by the displacements of the left vector. The currently defined set of continuation functions are shown below.

- L DXY R - extends the vector R by sequentially adding displacement value pairs from L to the last value of R.
- L DXTY R - extends the vector R by alternately adding displacement values, first x then y, from L to the last value of R.
- L DYTX R - extends the vector R by alternately adding displacement values, first y then x, from L to the last value of R.

The general way continuation functions are used is to extend a path by fixed offsets. This is the way the continuation functions are used when graphical trails are translated to code. The continuation functions represent the series of relative offset steps which form a path from a reference step. Since the result of a continuation function is simply a vector of co-ordinates then it is possible to apply a continuation function to the result of a previous continuation function. This may be done when it is convenient to describe a path as a series of displacements of one type followed by a series of displacements of another type. For example, a path may consist of several manhattan edges followed by a line segment at 45 degrees. In this case the expression for the path's description would be much shorter if the manhattan part of the path was described using either of the functions DXTY or DYTX; then the latter part of the path could be described with the continuation function DXY. The reason the description would be shorter is because all of the zero displacements in the manhattan part of the path would not be present since DXTY and DYTX only produce manhattan geometries.

Another function which exists because it is easier to use than an APL expression is RXY. This monadic function reverses the order of the pairs in a co-ordinate vector. It places the first x,y pair in last pair's location, the second pair in the second from last position, and so on, and returns the reversed vector as a result. This function is needed to reverse the order of a co-ordinate vector when the user wants to continue a co-ordinate vector from its head instead of from the tail using a continuation function. The reason the vector must be reversed before the continuation function can be applied is that continuation functions only extend the tail of co-ordinate vectors.

Bridge functions may sometimes be confused with continuation functions. Instead of continuing a path with relative displacements, bridge functions join two co-ordinate vectors together to form a vector which represents a trail containing a bridge. The bridge functions below represent the three different types of bridges found when graphical trails are created. The functions are used to represent upper, lower, and direct bridges when graphical trails are translated to code.

- L LXY R - concatenate R following L with an extra co-ordinate point in the middle which is formed using the last x value of L and the first y value of R.
- L LYX R - concatenate R following L with an extra co-ordinate point in the middle which is formed using the last y value of L and the first x value of R.
- L , R - concatenate R following L with no extra point added.

Currently the only time objects may be created is when a cell is under construction. To begin construction of a cell a STARTCELL call must be performed. STARTCELL is a monadic function whose argument is the name of the new cell. If a cell is already under construction at the time a STARTCELL is performed then the current cell is considered finished. To explicitly end the construction of a cell a ENDCELL call may be made. ENDCELL is a niladic function. In the future it is hoped that the interactive nature of EPLE may be increased so that objects may be created without being defined as part of a cell. These objects would be considered transient, used

only for the purpose of displaying immediate graphical execution results, and would be erased when a new cell is started.

To place a cell which has been previously constructed the PLC (place) function may be called. The PLC function is diadic, the right argument is the origin of the cell to be placed and the left argument is a two element vector, where each element of the vector is a character string. The first character string is the name of the cell to be placed and the second element is the instance name to be given to the placed cell. If the user wishes to assign a default name - to this or any other object - the default name may be generated by the Δ function. Δ is a niladic function which generates a new numerically increasing character name each time it is called. Below are two examples of an inverter being placed. The first call places the inverter and gives it the instance name 'inv1', the second call places another instance of the inverter, but this time the name is created by the default naming function. To allow the designer to latter reference the cell placed with its default name, the default name is captured in the variable NAME. It should be noted that

```
('inv' 'inv1') PLC 100 500
('inv' NAME←Δ) PLC 300 500
```

use of default names is not recommended because if at a later time it is found that a reference needs to be made to that object, possibly from a higher level cell, then the object doesn't have a name which is fixed in time. Without a name fixed in time a reference to an object may not remain constant across code modifications. Modification of the code resulting in more or fewer calls to Δ before the call to Δ in question will cause a different name to be produced. The inclusion of the Δ function is to allow for a short hand method of naming objects which have a high probability of never being referenced.

There are a few other functions which exist in the current version of EPLE which have not been discussed yet. These functions are used to change properties of objects and to set global defaults. For example, the functions LYR, TRANS, and WIDE take as their right argument the

name of an object to modify and as their left argument the new value associated with the object's layer, cell instance's placement transformation, or wire's width respectively. In addition, the functions SETLAYER and SETTRANS establish a new default physical layer on which objects are initially created and a default initial transformation applied to cell placements, respectively.

When creating layouts using the above set of functions it is often desirable to locate an object with respect to a point in another object. The problem is that EPLE's database does not store every vertex of an object. This has the unfortunate problem that even though a vertex may appear to graphically define the shape of an object, the vertex in question may not define the shape of the object within EPLE's database. As an example, a box is defined by its lower-left and upper right co-ordinates. The lower-right and upper-left corners also visually appear to be valid points, but it is not possible to create a query to determine these two co-ordinate values since these two corners do not exist as values stored in the database. To overcome this problem secondary functions have been defined to query the named object and compute the unrepresented corners from the lower-left and upper-right corners. The box is currently the only graphical object which has this problem, so there has only been a need to add two new functions, LR and UL (lower-right and upper-left). These two functions, both monadic, can be used to generate their respective co-ordinate of an unrepresented corner when their right argument is the name of a box.

5.2. Extending the Current Procedural Language

The functionality described by the previous section simply allows for the construction of VLSI objects. EPLE's long term goal though is to provide more than a static construction environment. What is desired is an extendable system in which new commands and system parts may be constructed as part of the procedural environment. Where the flavor of commands will more mimic the thoughts of the designer, causing descriptions to be object oriented, instead of being sequences of raw co-ordinate specifications. Before this can be accomplished more primitive operations, possibly involving replacement of the current APL environment, will have to be added to allow the user greater access to the complete EPLE environment. New operations will

have to be added in two areas. The first area will be operations which allow greater access to the structure of objects in the database; the second will be operations which allow the user more control over interactive input and output. In addition to the added commands the underlying database will have to be enhanced, through the addition of auxiliary data structures, to support the added information requirement needed in an extendable environment.

In attempting to increase access to the database a set of operations will be needed that can inspect and modify objects in the database. These operations will have to work on whatever form the objects in the database exhibit. Since the desired system is extendable it is impossible to predict the exact nature of the information which will need to be recorded. To solve this problem the objects in the database will be organized as properties. Where a property has a name and a value - either scalar, vector, or a vector of sub-properties. Properties should be dynamic objects, added and deleted as needed. The only required property of every object will be the property 'type'. Certain types of objects may have other required subproperties, for example all wires will have to have a width property to be plotted correctly. With this change made to the database structure, queries could be generalized from function specific calls like PATH to property selection operations. For example, path, layer, and type would be standard properties of geometric objects. To extract this information the user would simply use a generalized property query. Figure 5-5 shows three property queries. The first gets the object's type, the second the

```
'type' # 'object_name'
'path' # 'object_name'
'layer' # 'object_name'
```

Figure 5-5.

path, and third the layer. Assume that a # is the property query operator. Along with property queries the user needs the ability to identify what property names are contained within an object, associate new properties with an object, and assign values to properties.

Some of the desired extensions would be the addition of operations which mimic the thoughts of the designer, some of these object oriented operations are also possible using the current EPLE system and should be added. These operations would attempt to allow the designer

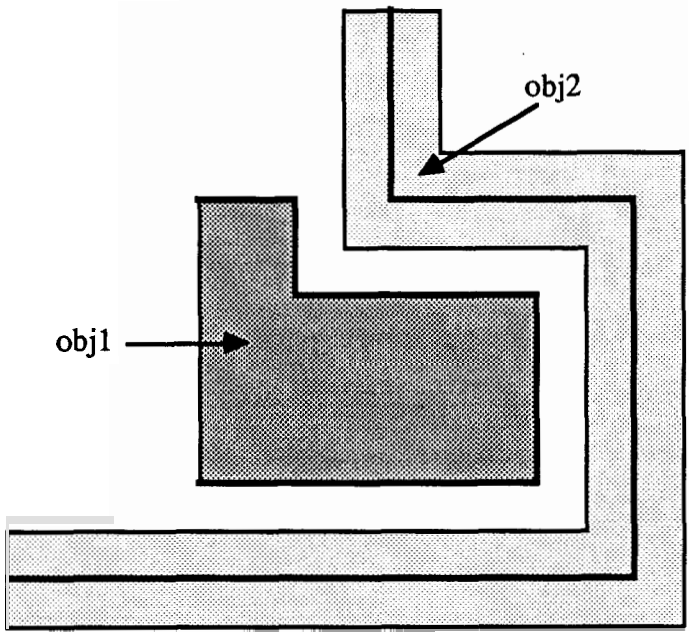


Figure 5-6.

to express the layout in the way it was conceived. For example, many times a designer will want to create an object which is 'just like that one, but a little different'. The current solution is to copy the code which produced the object and then modify the co-ordinates to place it at its new location. A better solution would be to allow the designer to generate a copy of the named object and then use procedural editing operations to modify the new objects exact appearance. For example, Figure 5-6, shows a polygon with a wire following part of the side of the polygon. This is what the designer wants to produce in the end but currently he only has the polygon. The designer thinks to himself, 'I want the wire to come down and then bend to follow the shape of that polygon', but what he is currently forced to do is compute the new co-ordinate for each point beside the polygon. What the designer is forced to code and what the designer thinks are radically different. This could be overcome by the addition of object oriented operations. Note that the bold lines in Figures 5-6 and 5-8 are the paths which define the shapes of the two objects and that

in Figure 5-8 'obj2' is represented only by its path. To produce Figure 5-6 the user might enter the code seen in Figure 5-7. The first line of the code makes a copy of 'obj1' and names it 'obj2'.

```
50 SWELL 'obj2' COPY 'obj1'
50 WIDE 'metal' LAYER 'wire' CHTYPE 'obj2'
temp←1 DROPXY 'path' # 'obj2'
('obj2' 'path') SET -1 50 0 STRETCH temp
```

Figure 5-7.

It then swells 'obj2' by 50 units. The swelling operation is an example of object oriented design, a thought is applied to the objects instead of numeric operations. The designer doesn't have to worry about if the offset from a given co-ordinate is positive or negative, he just states that the object is to be made 50 units larger. The objects resulting after the first line of code has executed are shown in Figure 5-8. Note, that only the path defining 'obj2' is shown and that the closing

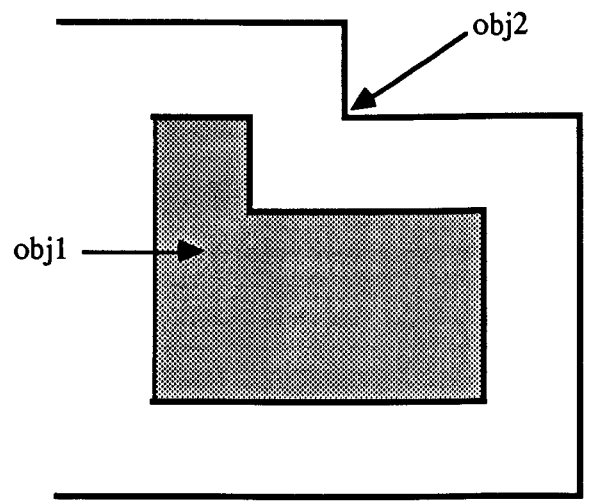


Figure 5-8.

edge of 'obj2' to form a polygon is not shown. The second line of code from Figure 5-7 changes the type of 'obj2' to a wire, sets its layer to metal and its width to 50. The third line gets the path associated with 'obj2', using the property method described above, and then discards the first co-ordinate in the path. The resulting path is saved in the variable temp to be used in line four. Line four stretches the last point, indicated by the index of -1 as the first value in the left argument, by -50 units in x. After stretching the last point in the path, the new path is assigned to the path

property of 'obj2' using the SET operation. This results in the desired image shown in Figure 5-6.

In addition to the database changes and the object oriented changes the language also needs to have access to the basic graphical routines used for interactive description of objects. These would include the ability to transform world co-ordinates to screen co-ordinates and vice-versa, receive screen co-ordinates selected by a mouse, generate rubber-band mouse actions, produce pop-up menus, prompt for input the same way the rest of the system does, and perform other I/O tasks as if they were originally built in as primitives. If all of the above changes were made, then EPLE would be easily extendable to handle new solutions to new problems.

5.3. The Current Procedural Environment

The current environment for procedural code development in EPLE was formed by joining STSC APL[29] with EPLE's display, trails, and database segments. The STSC APL environment consists of an APL interpreter and a session manager. It is the responsibility of the session manager to allow for the editing of functions, the saving and loading of workspaces, and the managing of the flow of results to and from the interpreter. The session manager is divided into two parts on the SUN workstation. One part of the manager is contained within the interpreter, while the other part of the manager is a separate process responsible for interfacing with the screen. STSC APL is started in an EPLE window by the user after EPLE is running. When EPLE starts, it displays a set of windows: a display composer window, a layer mapping window, a layer properties window, a command window, a colormap window, and a terminal window. The terminal window is the window from which the user must start STSC APL. When APL is running it forks - unix's term for starting a second process - into two processes, the interpreter and screen interface. The screen interface of the session manager takes over control of the terminal window so that it may receive keyboard input and display output using the APL character set. The session manager is essentially an emulation of an APL terminal. The STSC APL program arranges before forking that the interpreter and the screen interface processes are to

be connected by two unix pipes (See Figure 5-9). These two pipes handle the flow of data to and from APL and the terminal emulator.

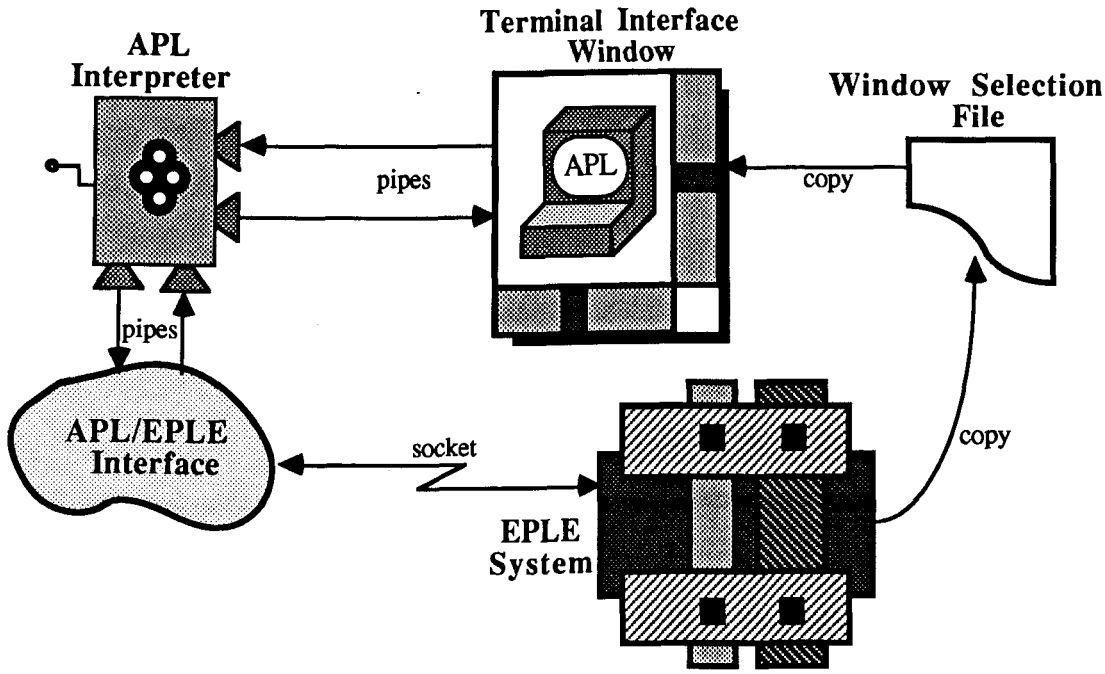


Figure 5-9.

Once APL is running the user must load the EPLE workspace. This workspace contains the extra functions needed to extend APL as described in the earlier part of this chapter. One of the functions in this workspace is called STARTAPLEPLE. This function is niladic and only needs to be run once at the beginning of an APL/EPLE session. What the function does is start an APL external process. An APL external process is a process which runs in the unix environment but treats its standard input and standard output as special streams of encoded values. The standard input packet to an APL external process is the encoded function name of an external function and the left and right arguments with which the function should be invoked. The standard output packet from the external process to APL is an encoded representation of the result vector produced by the function that was invoked. When the user executes the function STARTAPLEPLE in APL it starts the external process which is responsible for the communication of object creations, object modifications, and object queries to and from EPLE.

The elaborate interconnection of processes shown in Figure 5-9 is required because of the order in which EPLE and APL must be started. For the terminal emulation window to exist, EPLE must be running. This means that APL must be started after EPLE is started. Since STSC APL only provides pipes to external processes and all external processes must be children of the APL process, then EPLE can not be directly connected to STSC APL as an external process because EPLE is already the parent process of APL. This requires that a socket - a unix domain bi-directional link established using a previously agreed upon name - be used to communicate creation and query request back and forth between APL and the EPLE database. When APL encounters a function which requires access to the EPLE database an external APL function is called. All current external functions are contained within the APL/EPLE process. The APL/EPLE process decodes the values sent to it by APL and passes the request on to the EPLE database through the socket. Once EPLE has performed the requested operation the results are sent back through the socket. The APL/EPLE external process then re-encodes the response from EPLE into a valid APL result and returns it to APL through the outgoing pipe. When APL receives the result from the external process it continues evaluation of the current expression from the point where evaluation was suspended.

The only links in Figure 5-9 remaining undiscussed are the copy links to and from the Window Selection File. This file normally holds the contents of an area of visible screen text which was selected using the mouse during normal interactions between the user and the SUN terminal interface. Once an area of text has been selected, the user may direct the system to use the selection - by using the STUFF item from the pop-up menu - as if it was keyboard input typed at the cursor location within a window. EPLE uses the Window Selection File by placing into it the text of the code generated from the translation of a graphical trail. Once the translated code is placed into the Window Selection File, it may be inserted into an APL function by using the same STUFF operation the user would normally use for copying text. The reason the STUFFing operation works in the APL screen interface window is because the window was configured as a TTY window by EPLE before it was taken over by APL. This means that there is code running

as part of EPLE which supports the request for the pop-up menu and which handles the copying of the file into the standard input stream of the window.

The environment that the user currently sees is made up of two collective parts, the EPLE environment and the STSC APL environment. This has its disadvantages. The APL environment uses an APL character set, while EPLE uses a more standard character set. This can lead to some confusion. For example, the layer names 'N+' and 'P+' when entered in APL must be typed as 'N÷'and 'P÷' respectively. The names must be entered in this way so that when APL's characters are mapped to EPLE the names are interpreted correctly. Also, since the internals of the APL interpreter could not be modified it resulted in a looser coupling between the EPLE environment and the interpreted environment. Problems like this and the need for future additional language changes will require the replacement of the current STSC APL environment with an environment which is more fully integrated with EPLE. The immediate advantage of using STSC APL is that it allows for the concept of a vector based interpreted language to be demonstrated without the need to construct a language interpreter and editing subsystem. It also shows, as an added benefit, that the interpreter subsection of EPLE could be removed from the main program and run as a separate process. This has several ramifications when considering how the EPLE project might be integrated with the Structured Architecture Machine (SAM)[31, 32, 33] project to form a high powered CAD system for VLSI design† .

5.4. The Desired Procedural Environment

The desired procedural environment should be highly integrated with the remainder of the EPLE system. The editing environment, which is currently part of the interpreter, should be specialized to handle the task of effectively inserting or replacing pieces of code with the code produced by graphical trail translation. The editor would be more effective if it was located in the main segment of EPLE. Doing this would allow the editor to be extended to form the interface for

† SAM would be the interpreter engine behind a high speed CAD system based on procedural design.

communicating with APL and for handling the typed commands directed at the remainder of the EPLE system. In addition, it would allow greater control to be exercised over which version of code is associated with a cell. Placing the editor interface in the main part of EPLE would produce a more self-contained feeling. No longer would the system feel like different components glued together. There would exist one common interface, not two as exists in the current EPLE prototype.

Other environment features which would be added to an integrated environment would be the addition of code debugging tools. These tools would not be directed at debugging complex code, as are normal code debugging tools, but would instead be used to help identify pieces of code associated with parts of a circuit. For example, break-points could be set waiting for named objects to be generated or examined, or possibly the code could be single stepped through as the user watches the creation of the objects produced by the code. These additions would greatly simplify the task of modifying a cell. If the user could point at a graphical object to be modified and then re-evaluate the code until the point at which the object was created, high-lighting the line of code where execution terminated, it would remove all of the current searching which occurs to find the correct line of code. In addition, the system should be extended to allow for immediately evaluated expressions, which produce objects not defined to be part of a cell, to be displayed. This would allow for experimental placement of objects before the code for an object was really inserted into the definition of a cell. This feature in conjunction with an editing environment which maintains a script of previous interactions - from which previous text may easily be extracted - would allow for experimentation with selective inclusion of the desired results in new cell definitions.

It is impossible to totally define or direct what the environment of EPLE will look like. What should be a guiding force is the idea of a system where textual changes are made with the least effort and minimum delay. This means that the system should help in the identification of the code which produced a selected object, help in the creation of code representing new objects, and

provide methods for generating objects like PLA's and register arrays. In the long run the users will identify which changes which must be made to produce the desired type of system.

Chapter 6

Conclusions & Other Considerations

From the 16,000 lines of C code developed in an 8 month period the first version of EPLE has been born. EPLE is still an infant - may it be said a new born - but it is ready to begin to improve the current development environment it was meant to replace. Given time it may evolve into a complete tool, playing an integral role in the design of not only custom chip projects but in the development of complex parameterized generators. The most complete part of the current EPLE system, the display and hardcopy output sections, will soon replace the aging VLSIplot. This simple change alone should greatly reduce the awkward nature of the current design environment and that in itself is enough to consider the EPLE project a success.

The initial goal of EPLE, to produce an interpretive interactive graphical procedural VLSI design environment which has a closed cycle from graphics to text and back, has been successfully demonstrated. Trails appear to be a successful short-hand method for graphically describing procedural fragments of code and it has been shown that their translation can be done very efficiently. In addition, a new naming strategy was defined which allows for the hierarchical referencing of objects. Also demonstrated was the power and flexibility provided by a uniform approach to defining all objects through the use of vector expressions. These results are rewarding and indicate that development and refinement of EPLE should continue.

The current prototype of EPLE still has several deficiencies. One is the extremely loose coupling of the interpreter to the remainder of EPLE and the other is the lack of an integrated editor. The loose coupling of the interpreter caused because the code for the interpreter could not be modified, resulted in a slower than expected speed for the evaluation of user's code. The slow-down is mainly due to the APL external process which translates and communicates results between APL and EPLE. Replacement of the current APL interpreter with either a modified APL

or a custom interpreter would remove many of the current bottlenecks which exist. The editor, in this version of EPLE, is simply the editor which comes with STSC APL. The result is that the editor works well in an APL environment but it does not provide the type of support that is needed to make optimum use of EPLE's graphic section or EPLE's trails. Replacement of both these sections with custom code should be made a priority in the production of the next version of EPLE.

A different alternative when considering the production of a new EPLE system would be to take advantage of the Postscript[38] interpreted graphic environment. A complete Postscript environment was not available at the time the EPLE project began, but is now. Postscript has many desirable features. It is interpreted, it represents graphical objects naturally, it maintains a data description which is easily extendable, and it is becoming the defacto standard for communicating with display hardware. It is hard to say what type of performance or design compromises would have to be made to implement all of EPLE in Postscript, but it is definitely an area which should be examined.

After the changes just mentioned above have been made the next problem facing the EPLE system is not how to expand it, but where to start. A complete VLSI development environment, not just a layout tool, consists of many different programs. These range from layout systems like EPLE to design rule checkers, behavioral model simulators, schematic capture systems, logic generators, comparitors, and floorplan administrators. To develop a complete environment requires many many man years. This in itself will probably indicate the direction EPLE will follow. EPLE will have to be made to co-exist with the other various tools that currently exist. One option would be to force EPLE to co-exist with a much larger system which already has many of the tools needed. Electric[5, 6] or Magic[7] would be possible environments into which to integrate EPLE. Electric would probably be the better alternative because it was designed to be expanded and source code and documentation are available. The determining factor of how

successfully EPLE could integrate with Electric will be the ease with which the interactive formation of trails could be done in Electric. It may only be possible to produce output from EPLE which is acceptable as Electric's input.

Another area which must be examined, which is not addressed by EPLE, is the organization of cells, cell code, and all the different views which exist of cells. Electric provides a mechanism where all the information about a library of objects is kept within a single file. This provides a clean interface to the library because the system always knows where objects are and how they have been modified. The organization which Electric uses to organize its cells is not acceptable though unless EPLE can be totally integrated with Electric, which after an initial examination does not seem possible. This probably means that EPLE may have to co-exist with Electric and many other programs. EPLE's method of organizing cells is inherited from the system it is replacing. This is necessary because EPLE needs to access cells produced by the earlier system. The current method for organizing cells is to place each cell in its own file and to place the file somewhere with the unix file system. The locations of valid places for EPLE to look for cells in the file system are determined by examining a path search list.

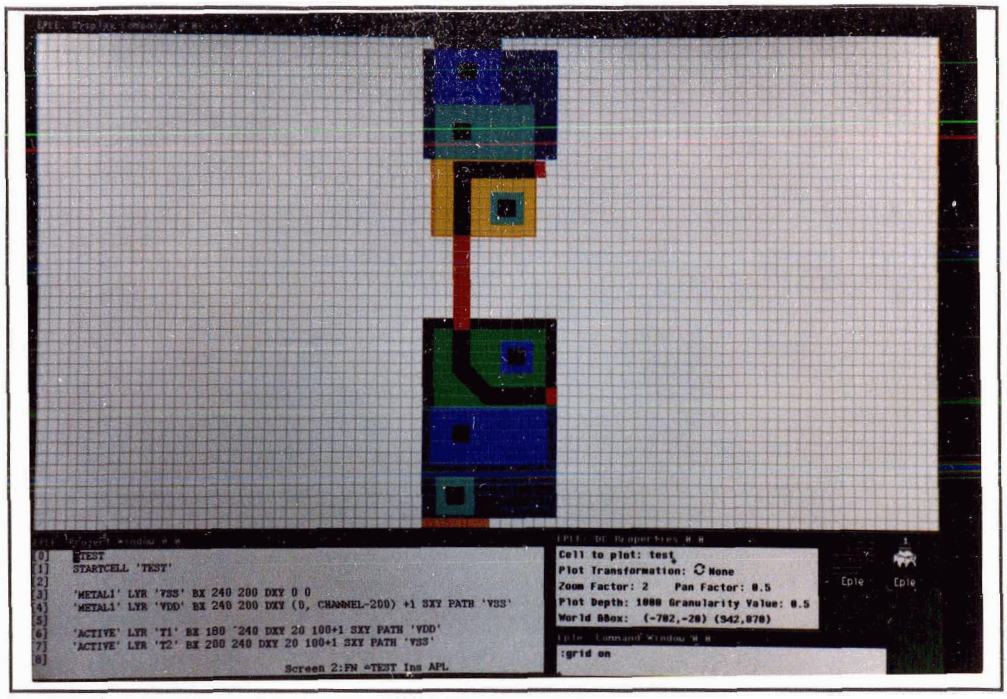
What needs to be provided is a library management system which understands how cells, schematics, simulation results, behavioral models, procedural code, symbolic sticks representations, truth tables, documentation, floorplans, etc., are related, grouped, owned, and changed. The library management system must be accessible through both subroutine calls and through a user interface. This will allow programs which have been developed or will be developed to be interfaced to a common underlying management system; while still allowing a user to use the tools which currently exist by allowing him to communicate changes made to cells to the librarian through a librarian program.

It is my hope that EPLE will continue and that it will not be swallowed up by larger systems and totally forgotten. It is my belief that only through experimentation with procedural design will we find a way to create a true high-level silicon generator. This is not to say that all work should be done procedurally, as can be seen when examining the creation of a leaf level cell. Sometimes its much easier to use an interactive editor, but on the other hand why should a user be forced to create a PLA by hand when it can be coded once procedurally. I hope that EPLE will grow to become a powerful procedural code generation system within a larger integrated collection of other tools.

Appendix

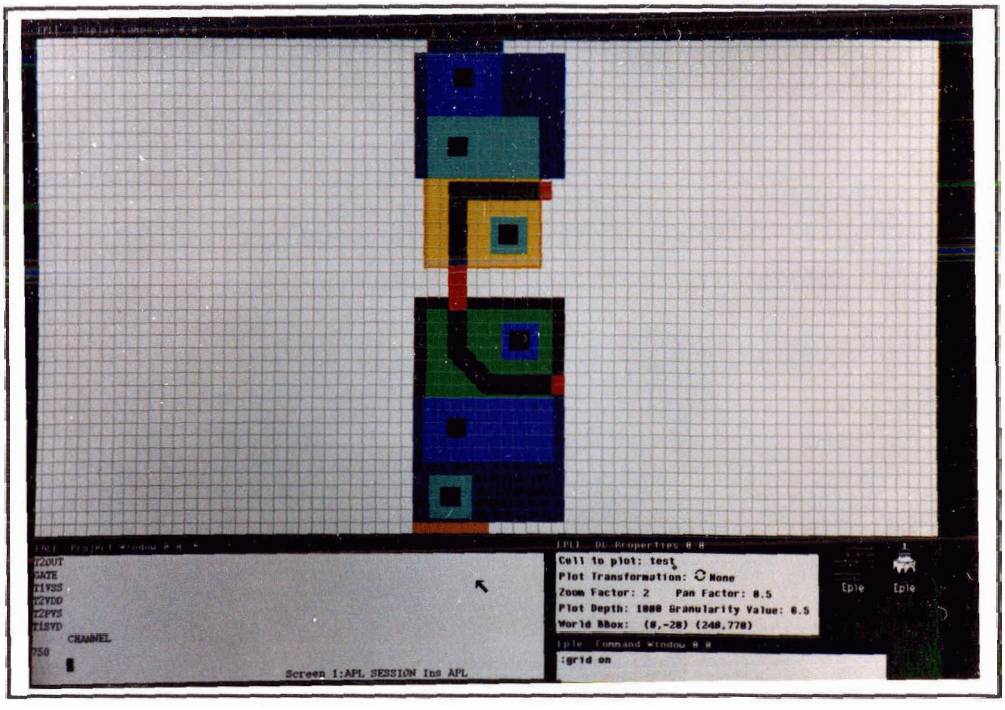
This appendix contains a series of photographs[†] of EPLE in operation. Many of the photographs are related and show an operation occurring at several different stages. Along with each photograph there is a short description of what is being demonstrated and/or what has changed from the previous photograph. The photographs selected to be in this appendix show a wide range of operations and attempt to give the reader an impression of what EPLE really looks and feels like. Some of the photographs show EPLE constructing new cells; while other photographs demonstrate the different ways the system can display results. Additional photographs show the regular form of common VLSI structures for which procedural code could or is used to produce the cells.

[†] All photographs were taken using a Minolta XE-7 with a Vivitar 70-210 zoom lens and ASA 100 film with a 2 second exposure at f22.

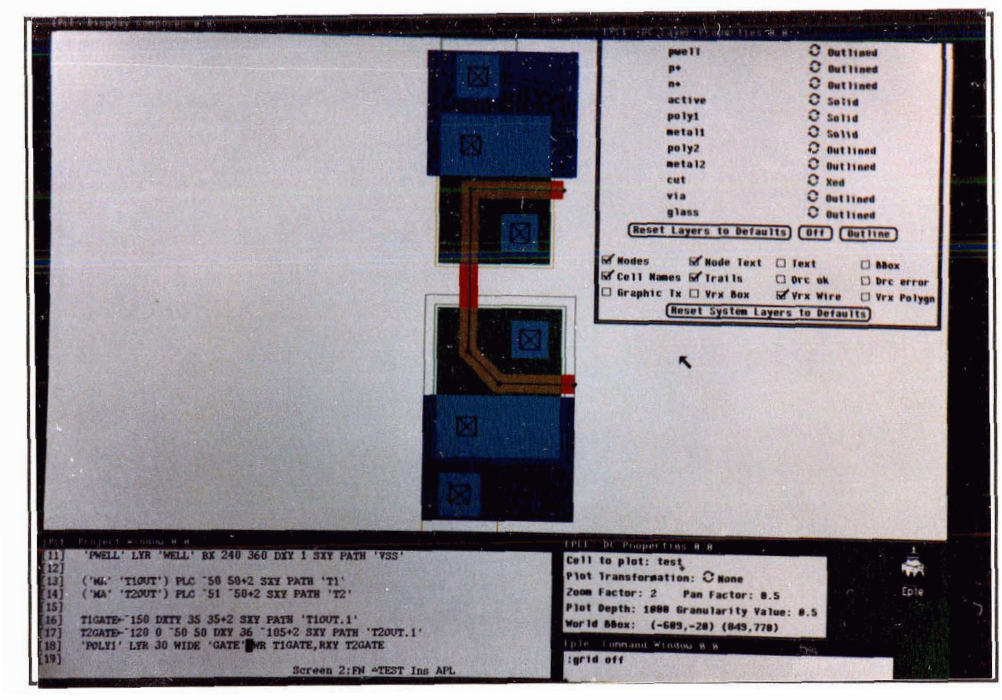


Shown above is EPLE running with four of its windows open and two closed to their iconic form. The large window at the top is the Display Composer window. It is in this window that trails are constructed and cells are displayed. The window below and to the left of the Display Composer is the Project window. This window is the user's interface to STSC APL's editor and session manager. It is in this window where the user edits and invokes execution of procedural code. The two windows labeled DC Properties and Command window are used to control what the system plots and how the system performs other operations. The DC Properties window is used to set the name of the cell to plot, zoom factors, pan factors, plot depth, etc.; while the Command window is used to enter long textual commands. The two iconic windows control, from left to right, what layers are to be displayed and what colors are to be used for each layer.

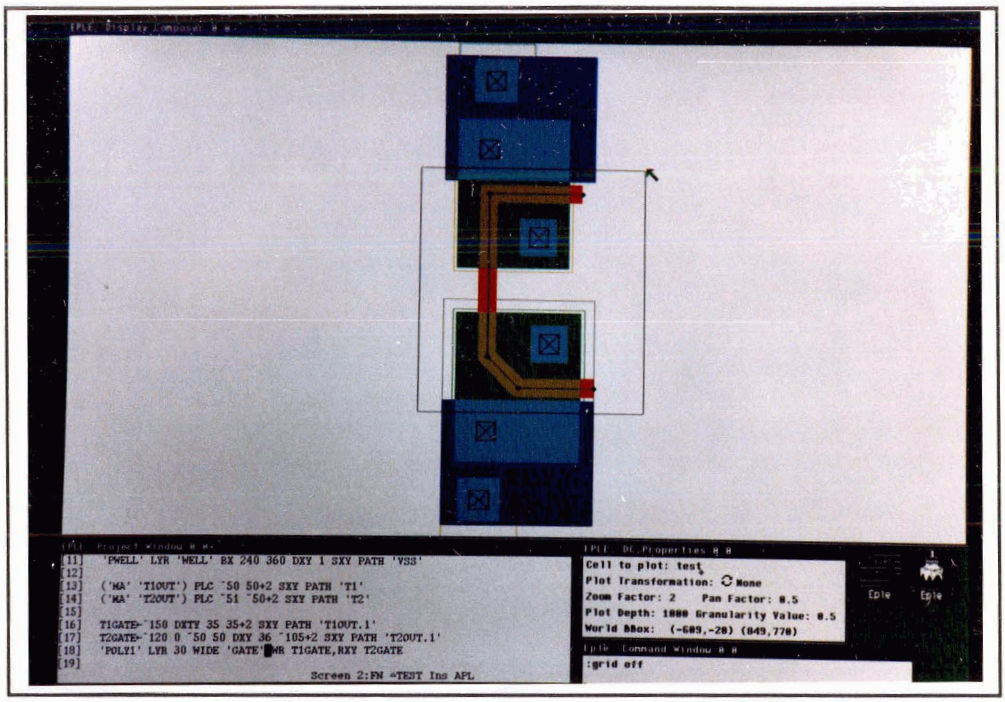
Drawn in the Display Composer window is the image for a simple inverter with the grid selected so that the user is able to visually determine the size and spacing of objects. Visible in the Project window is part of the code which produced the cell. Notice the variable CHANNEL in line 4. It defines the spacing of the power rails.



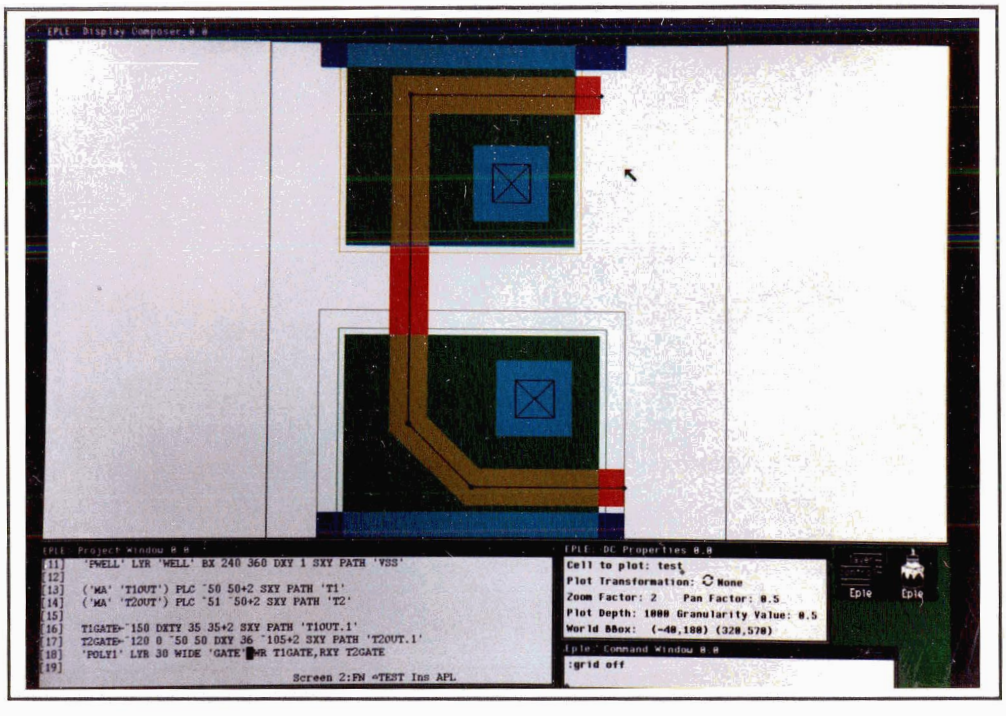
This photograph is the same as the previous except that the variable CHANNEL has been changed from 850 to 750 units and the procedural definition of the cell has been re-evaluated. When the cell is redisplayed, after being re-evaluated, it is clear that the total cell height has been reduced by 100 units. This can be done because all of the objects in each transistor are defined relative to their respective power rail.



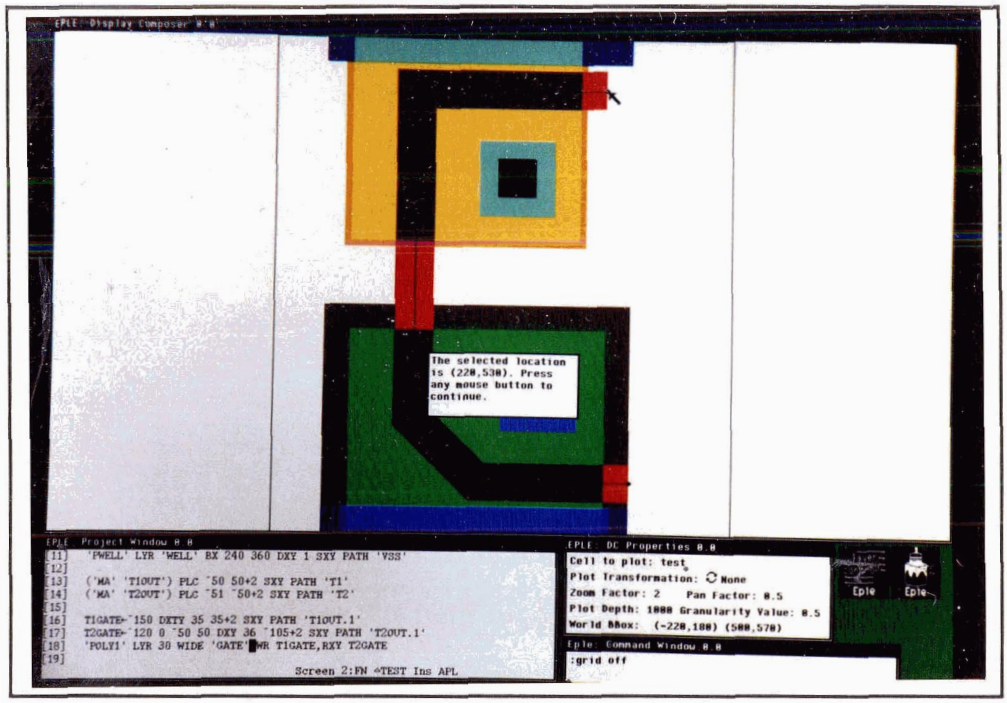
Here we see the same inverter as in the last photograph except this time the grid has been turned off and the layer control window has been opened. In addition the code for the cell is being edited in the Project window. Notice that in the layer control window that Vrx Wire is selected. This has caused the wire, which forms that gate of the inverter, to be displayed such that the points which define the wire are visible. Each defining point in the wire is displayed as a dot and the dots are connected to show the path of the wire. In the Project window the user has positioned the textual cursor on the line of code responsible to the construction of the inverter's gate. The two previous lines compute the position of the gate in the two transistors respectively.



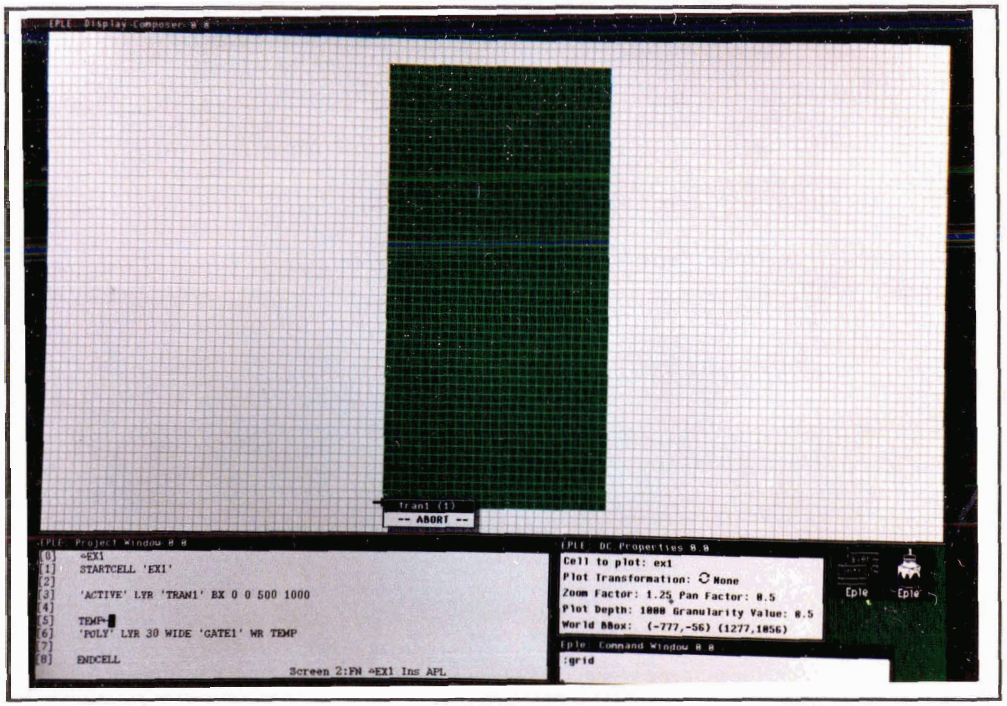
Here the user has selected the selection box mode from the command menu (done by pressing the right mouse button while the shift key is down) and has selected the outlined area on the screen. The area is selected using the left and middle mouse buttons. The left mouse button defines the lower-left corner of the outlined area and the middle mouse button defines the height and width of the selected area. The selection box may be used to select a new area to view, as a ruler to measure distance, or a sub-area to display on a plotter.



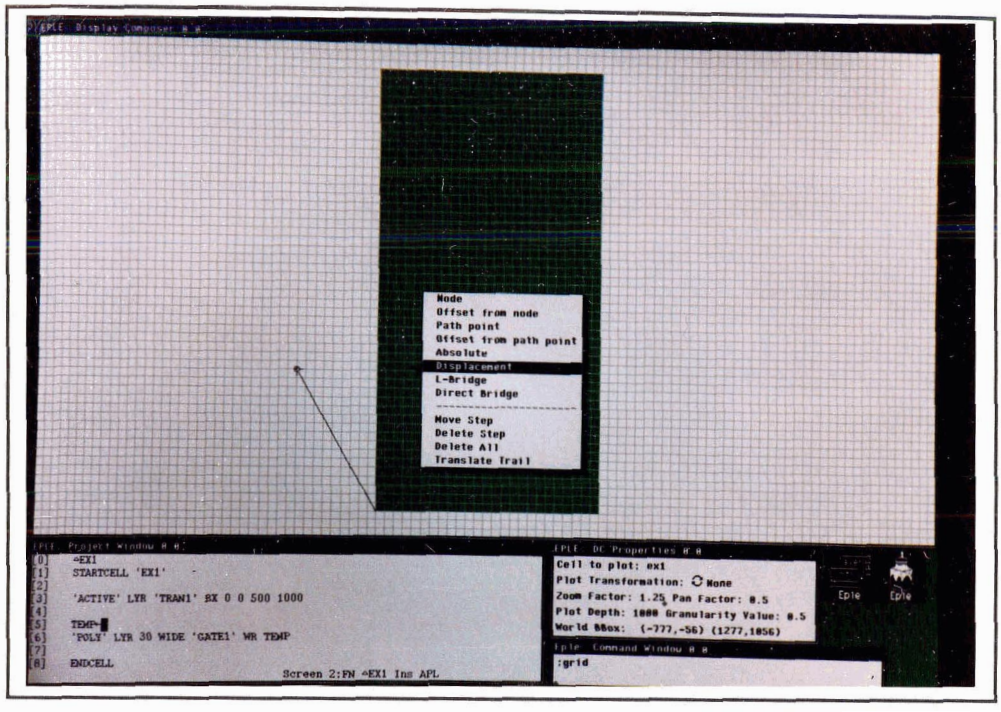
This shows the area outlined in the previous photograph after the selected area was used to defined the new viewing area. To use the outlined area of the selection box as a new view the user must depress the home key and the shift key at the same time. Depressing the home key alone causes the displayed view of the cell to be changed so that the entire cell is visible.



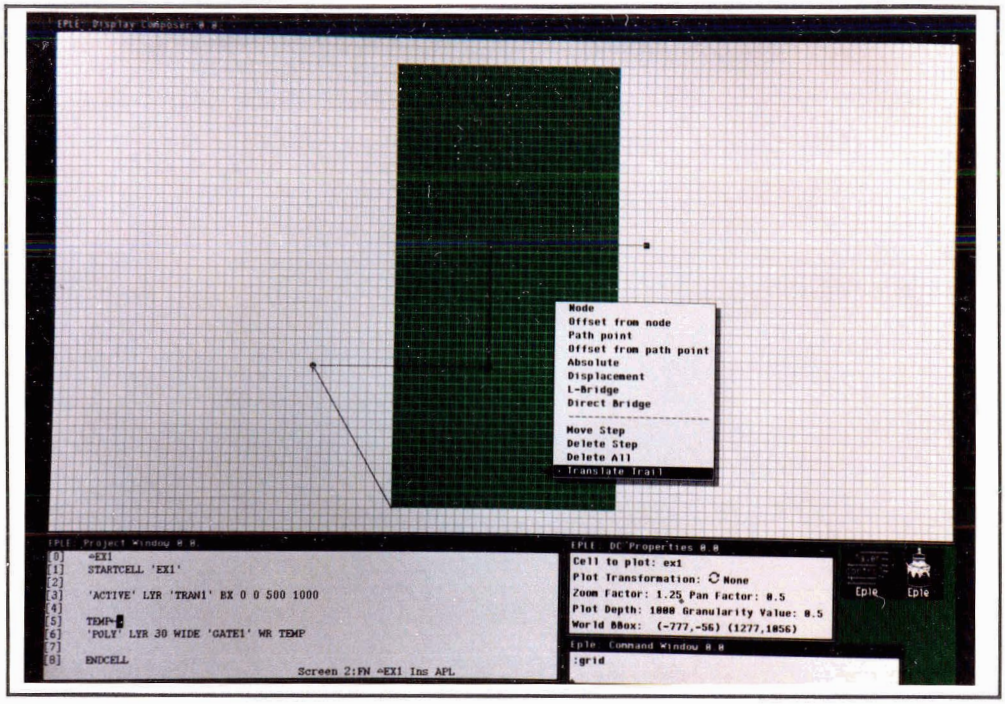
The user may determine a co-ordinate within a plot several ways. He may evaluate an expression produced by translating a trail, request the co-ordinates of the selection box, or enter into location mode. Location mode can be selected from the command menu and the command menu is displayed when the user depresses the right mouse button and shift key. By depressing the left mouse button, while in location mode, the user can determine the world co-ordinates of the cursor as it is moved around the screen. When the user releases the left mouse button a message is displayed indicating the last co-ordinate location displayed. In addition to the message the co-ordinate values are placed into the STUFF buffer used by suntools so that the user may insert the co-ordinate into his code.



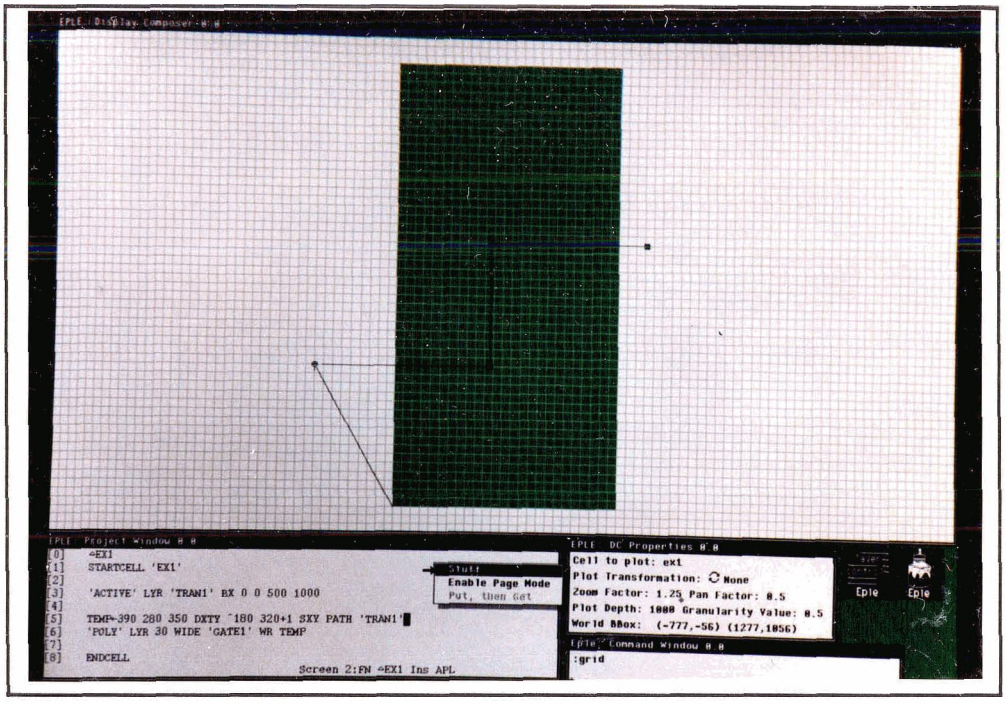
This photograph shows the start of a new cell. The project window has the beginning of the cell's code. In line three the user has defined an area of 'active'. Line six defines the gate of the transistor which will be formed from the values contained in the variable TEMP. TEMP is currently undefined on line five. The cell has been named 'EX1' in line one so the 'cell to plot' is set to 'ex1' in the adjacent window. The user has entered into trails mode by selecting TRAILS from the command menu (right mouse button down and shift key down). He is currently defining a path point reference step with offset to begin at the first point in the object 'tran1'.



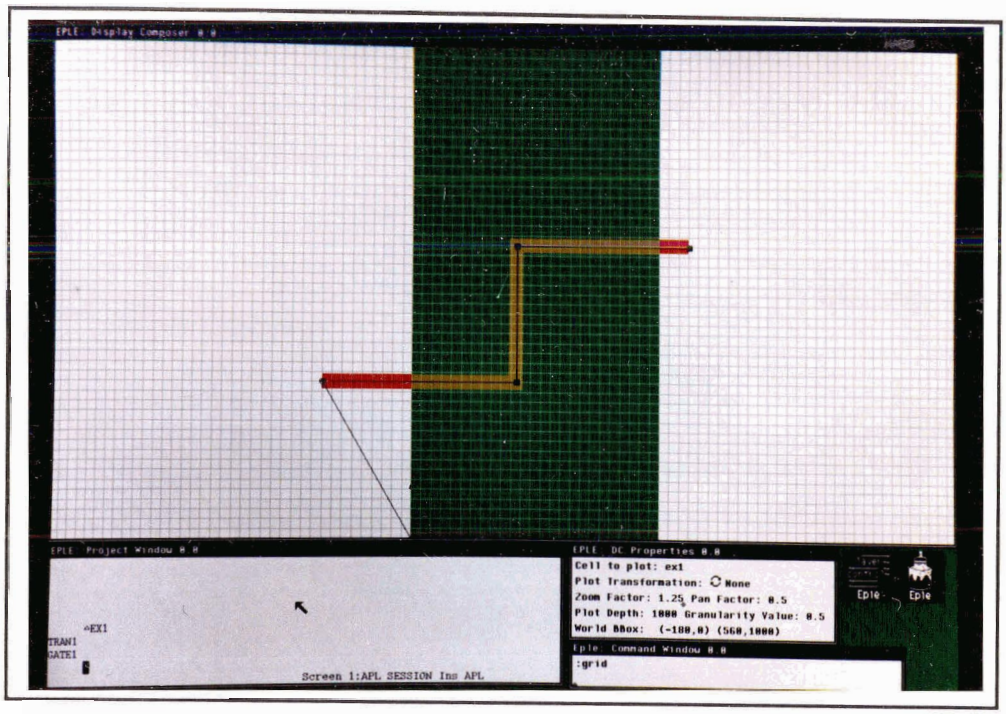
The user has finished stretching the rubber-band offset for the path point reference step and has clicked the mouse at its desired location. Note that the line extending from the corner of the 'active' - point 1 of 'tran1' - to the reference step marker has a small arrow at its head. This arrow indicates that the line is not a relative connection between two steps but is instead an offset from a known location. The user is now about to select the Displacement operation from the TRAIL menu. Once in Displacement mode the user may define steps relative to other steps which already exist by clicking the mouse over a starting step and then stretch a rubber-band line until a new connected point is formed by a second mouse click. Rubber-banding continues from the last new points until the operation is aborted by pressing the middle mouse button.



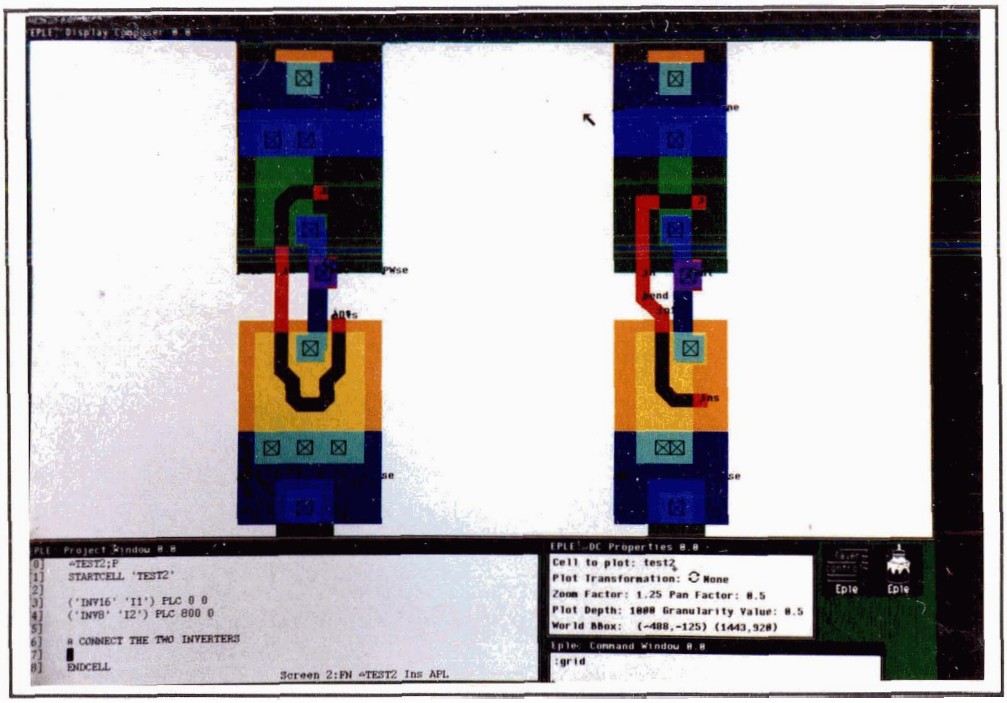
In this photograph the completed trail can be seen. The trail's reference step is the path point offset from the first point in 'tran1'. The path of the trail is formed by a series of three displacement steps. The user is about to select Translate Trail from the Trail menu. Once this mode is selected the user may indicate the desired end of the trail where translation should begin by clicking the mouse over the desired step. When the translation has occurred the user may complete line five in the editor window.



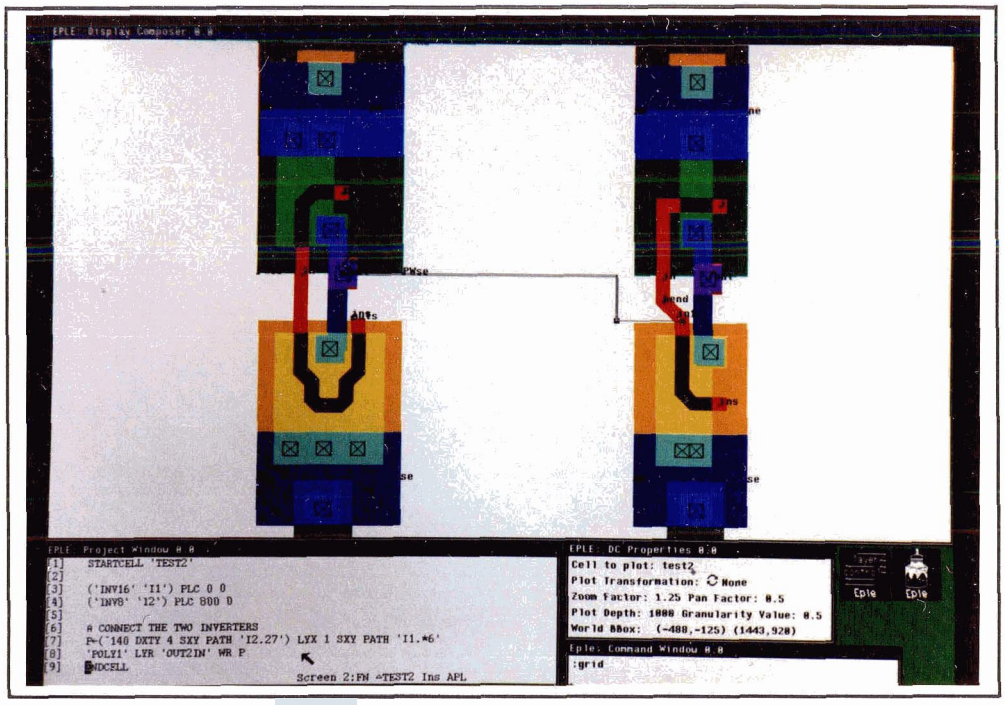
Line five in the Project window has been completed now by inserting the code produced by the translation. This photograph shows how the user selected the Stuff option from the Project window menu. Note that the Stuff operation has already been performed and that it is again being selected from the menu just to show the menu. Line five was completely generated by EPLE except for the assignment to TEMP. The code - read from right to left - gets the path points associated with the object 'tran1', selects the first point in the path, and then adds an offset of -180 320 to produce the location of the trail's reference step. The remainder of the expression, 390 280 350 DXTY, describes the displacement path which forms the body of the trail. When line five is evaluated TEMP will be assigned a vector of co-ordinate values which describe the position of the trail.



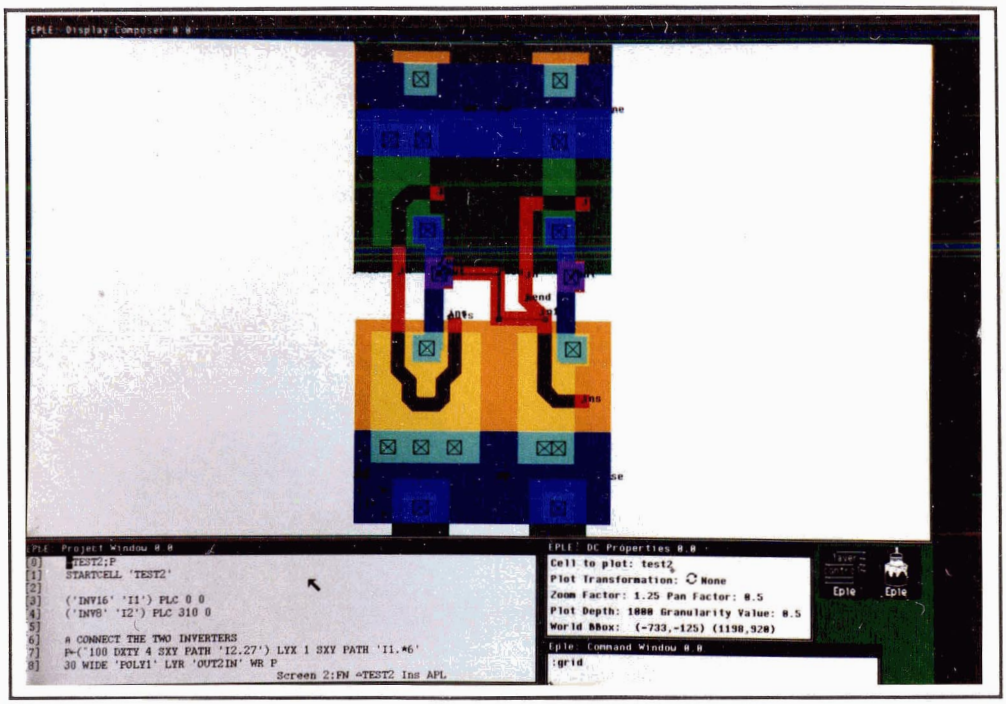
In this photograph the code has been evaluated and the cell has been displayed again. The project window is no longer being used as an editor, rather, it is now the link to the APL session manager. The user has evaluated the function describing the cell and APL has responded by displaying the names of each object created during the evaluation. Notice that even though the cell has been redisplayed and has changed, EPLE has maintained the trail drawn earlier because the reference step from which it is defined still exists. A counter-example is if the user deleted the object 'tran1' from the code (the larger active area) then the trail defined as an offset from 'tran1' would be removed when the cell was redisplayed. This is because there would be no way to determine the position of the displacement steps. Since the trail still exists the user may modify the trail if the result produced after the cell was evaluated was incorrect. The modified trail may then be translated and used to replace the code already on line five.



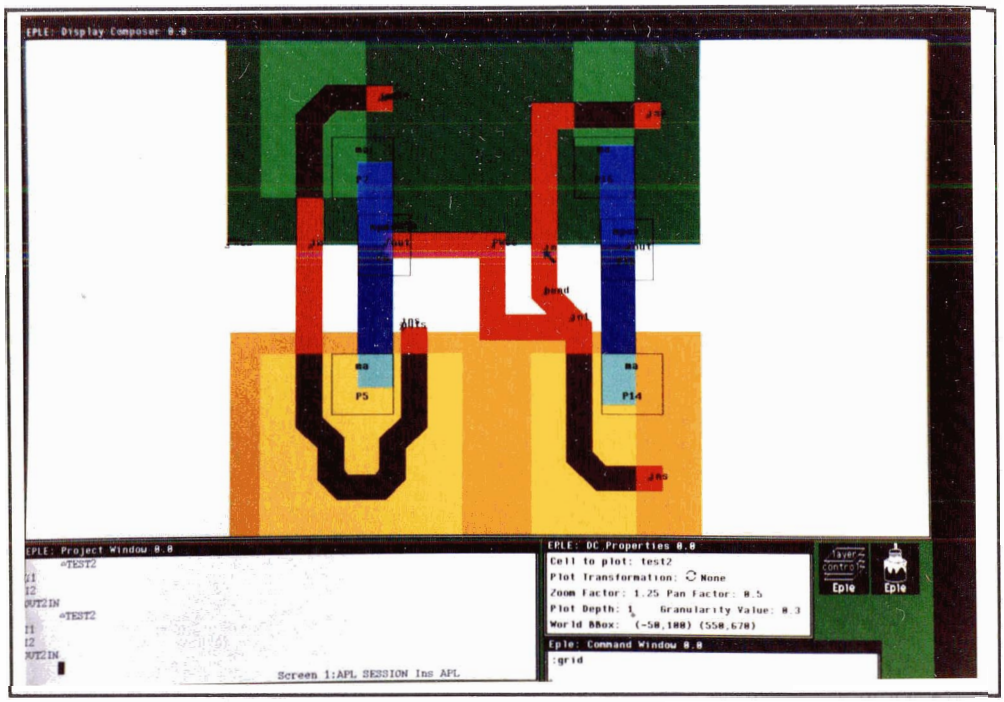
Here the user has begun to define a new cell composed initially of two inverters. The inverters are placed in lines three and four and given the names 'I1' and 'I2' respectively. This code has been evaluated and the results displayed in the top window. The user is now in the process of connecting the two inverters on line seven.



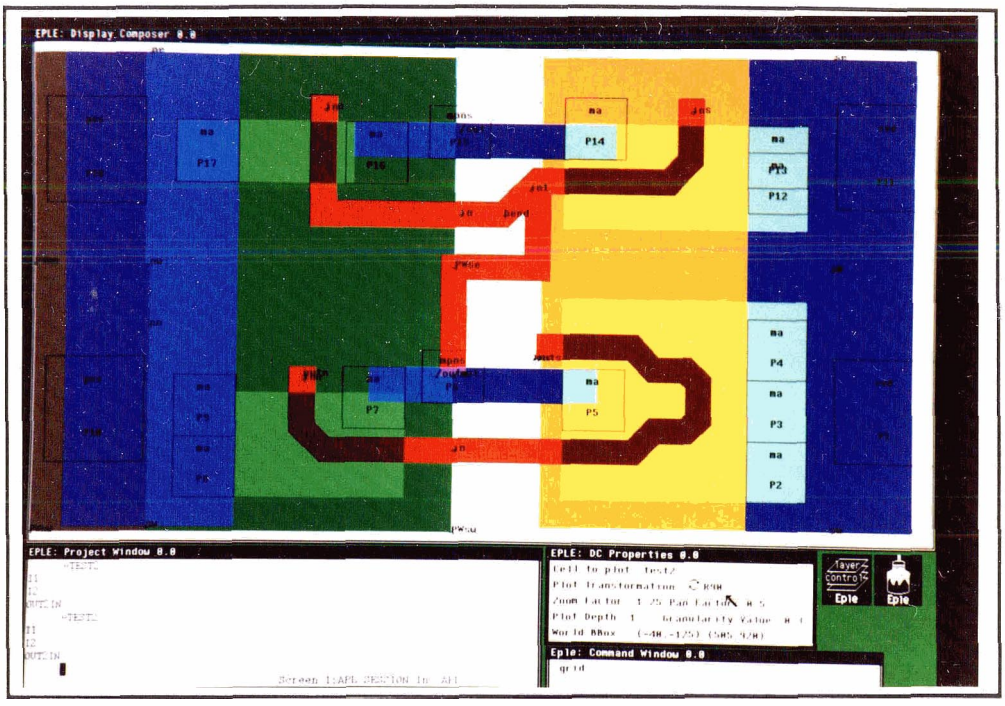
To make the connection between the two inverters the user has defined a trail. This trail is composed of two paths and a stretchable bridge which connects them. The path on the left is composed of a single path point reference step located at the placement point of the metal/poly contact. The path on the right is defined to begin at the fourth point in the gate of the inverter and to continue to the left by a fixed displacement. The two paths are joined by an L-bridge which will stretch to accommodate the movement of the cells. The trail has been translated and the resulting code inserted into line seven. Line eight creates the connection by using the results assigned to P in line seven.



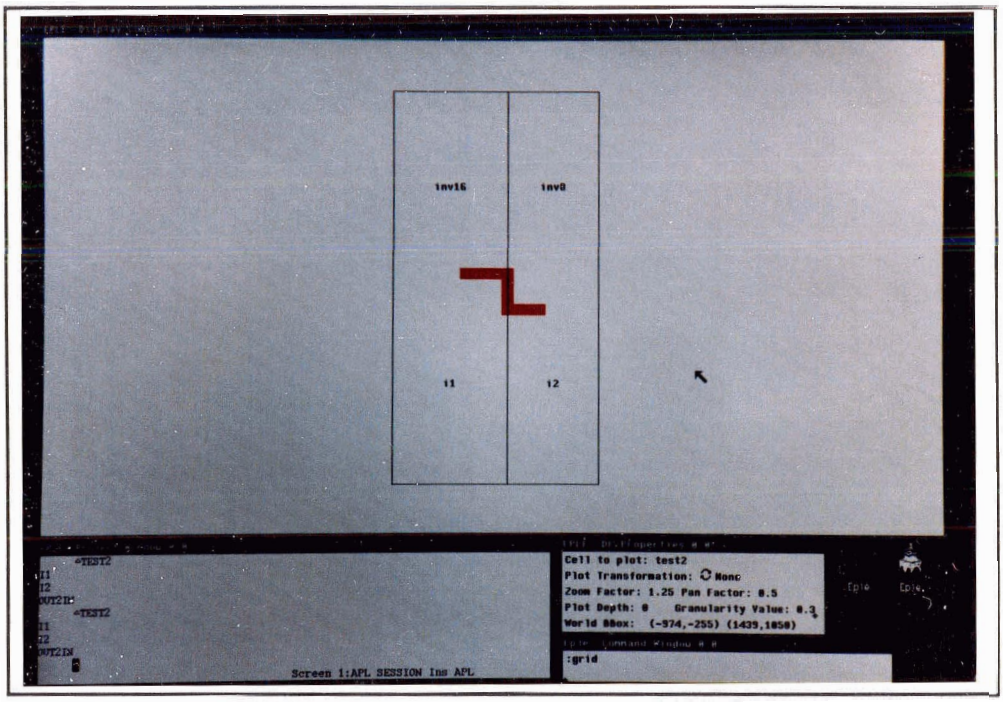
The user has decided that the two inverters were too far apart in the last photograph and has moved them closer together. This was done by modifying the placement position of the cell 'INV8' in line four of the code. Notice that line seven did not have to be modified because the bridge contracted to keep the two inverters correctly connected. Line eight was modified slightly by the user to change the width of the poly wire to 30 units instead of the default setting of 40 units.



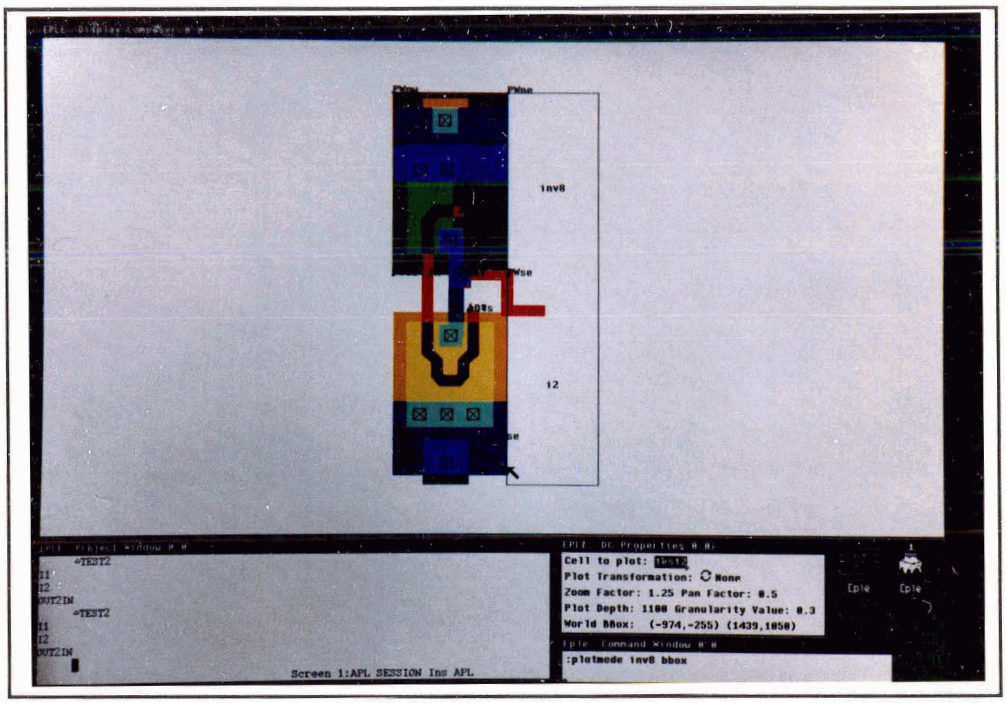
In this photograph the user has zoomed in on the center of the cell and has indicated that only the first two layers of the cell should be plotted. The plot depth is set by adjusting its value in the DC Properties window. It is currently set to 1, level 0 contains the objects in the upper most level. This means that subcells on level 1 will not be flushed out but will only be shown as bounding boxes.



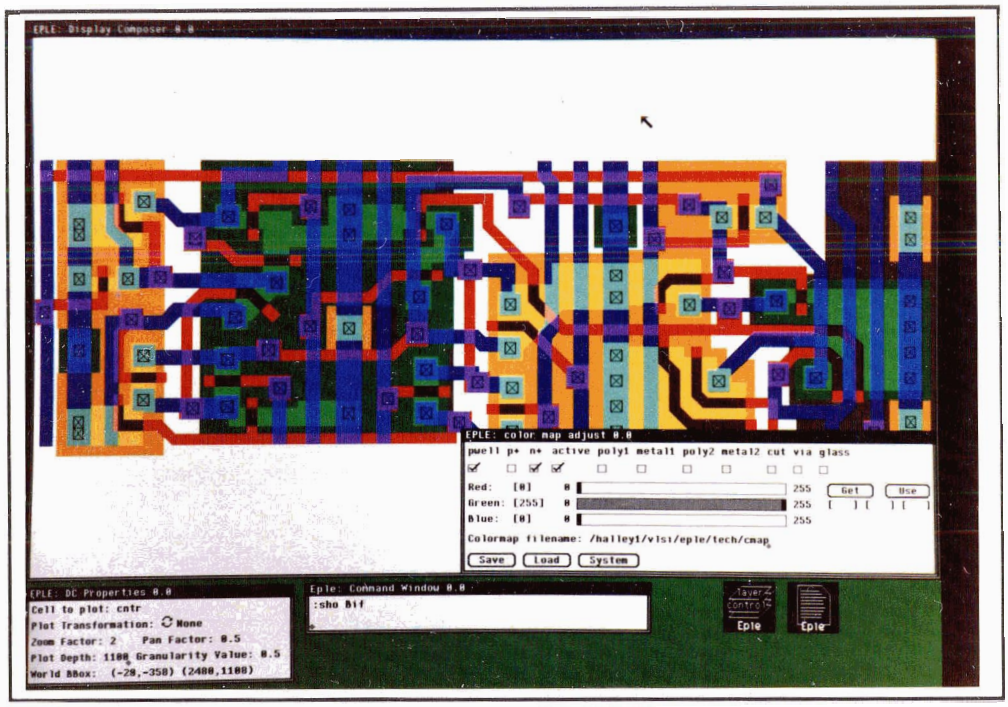
This is the same cell which was displayed in the previous photograph except that it has been rotated to make better use of the available display space. It was rotated by setting Plot Transformation in the DC Properties window to R90. To allow a user to view a cell in the orientation in which it may be placed, before it is placed, the Plot Transformation may be set to any one of the following transformations: None, R90, R180, R270, MX, MX-R90, MX-R180, MX-R270.



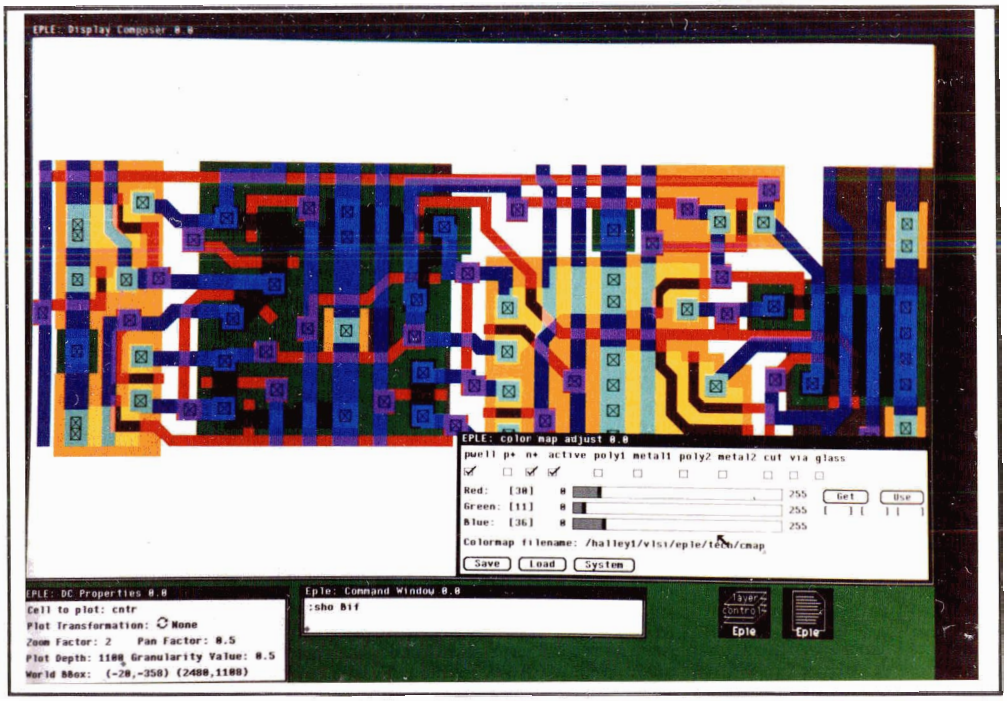
Here again is the cell that contains the two inverters which were previously constructed by the user. This time the user has requested that a plot depth of zero be used. This results in the display of the two bounding boxes for the cells and the wire connecting them. This allows the user to more readily identify the cell that an object is created in. It can be awkward in many cases to determine the cell an object belongs when several cells overlap.



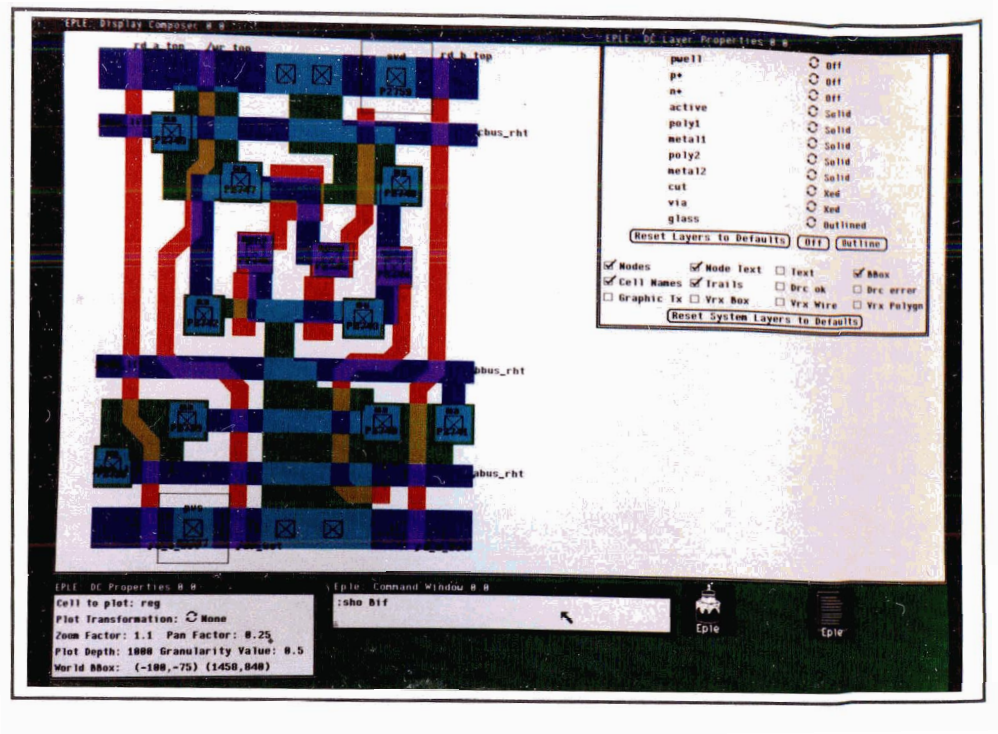
While this photograph may look similar to the previous picture it is not the same. Here the plot of the cell 'inv8' has been modified. In the Command window the user has indicated that all instances of the cell 'inv8' are to be plotted as bounding boxes. This is an extremely effective way to limit the amount of information plotted when large areas contain unneeded detail. For example, when interconnections are being made at a chip-wide level it is not important to see the details of every I/O pad. Using the plotmode command the user may turn all the instances of a given cell type on, off, or to bounding boxes.



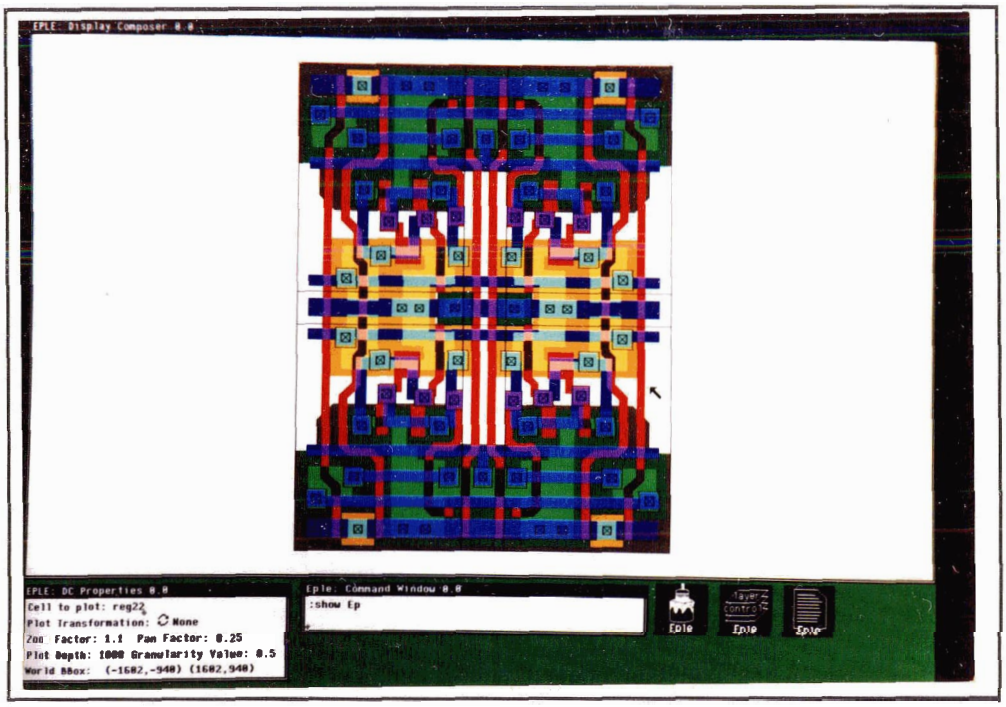
Here the user has opened the Color Map Adjust window. This window allows one to change the colors which are used to display the different layers and the interactions between layers. At the top of the Color Map Adjust window is a list of the different layers which may be displayed. The user may select the layer or layer interaction for which he wishes to adjust the color by placing a check in the boxes associated with that layer combination. From these layer selections, EPLE computes the color map index involved and displays the current red, green, and blue settings for it. The user may move the sliders by using the mouse. When a slider is changed the resulting colors on the screen are affected. The user may copy colors from one layer to the next by pressing the GET and USE buttons. The GET button copies the current slider settings into the space below the buttons. The user may then change the layer selection at the top of the window and then press the USE button to set the slider settings to the values below the buttons.



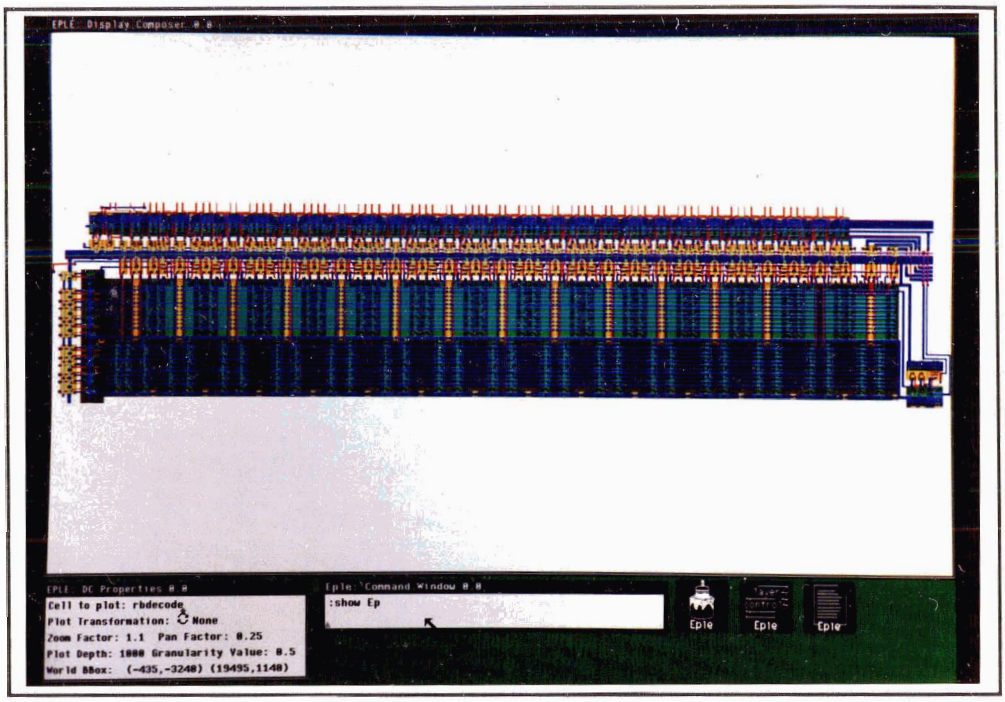
The user has modified the colormap in this photograph to more rapidly identify the areas associated with 'pwell' transistors. Notice how the color of the doped transistor areas in pwells has been changed to nearly black. The user can modify many layer combination colors and then restore them back to there original values. This may be done by reloading the colormap using the load button. If the user prefers the new colormap he may save it using the save button. It is also possible to have several different custom colormaps and to switch between them by entering the name of the colormap in the field Colormap filename.



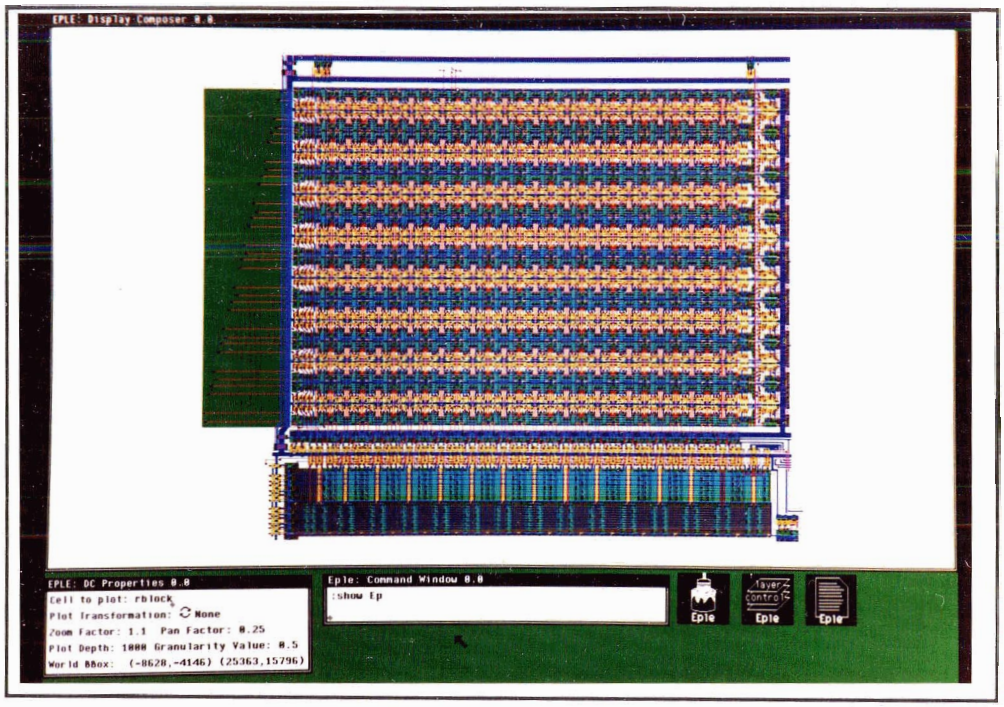
In this photograph the user is displaying the cell 'reg'. This is a simple 1 bit , 1 input, 2 output storage cell. The user has also opened the Layer control window and has selected the BBox switch at the bottom of the window. This has caused EPLE to display each subcell plotted with its bounding box, its cell name, and instance name displayed. This is different from setting the plot depth or the plotmode of a cell because the contents of the subcells are still displayed.



Here is an example of a cell which has been constructed to allow easy use within procedural code to produce arrays of storage cells. This is four instances of the cell shown in the previous photograph. Each instance has been mirrored or rotated to allow it to share a common power or ground supply rail. The BBox flag set in the last photograph is still on so the bounding box of each subcell can be seen. Note that the text has been removed because it would be to small to be useful. The point at which the text is removed may be set by modifying the Granularity Value in the DC Properties window.

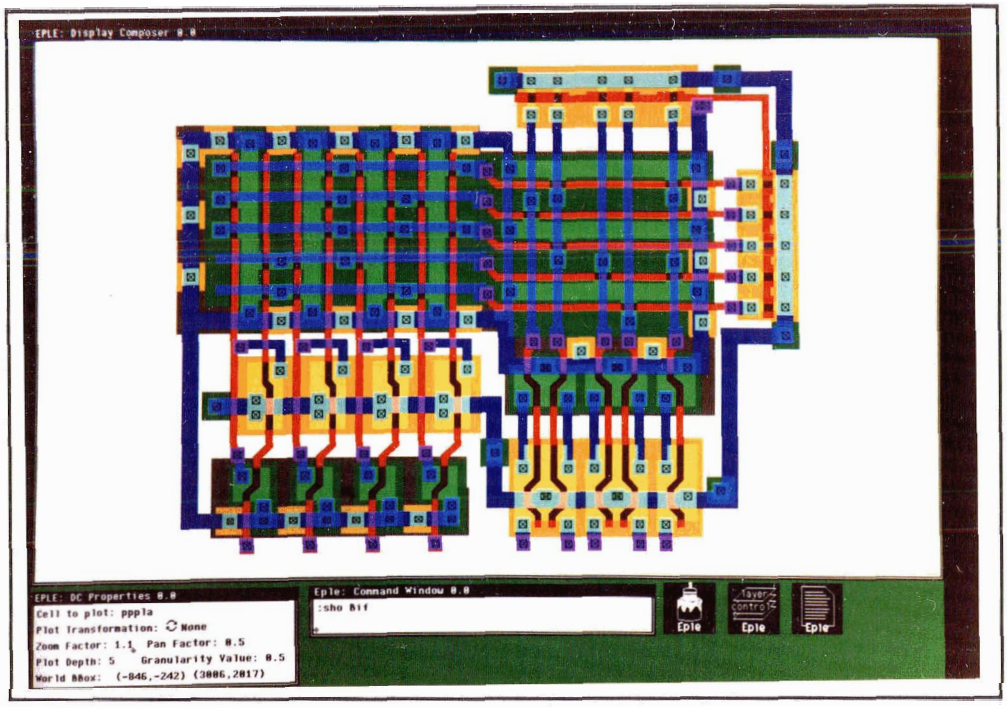


Here we have a two bus address decoder for a register array. Notice the regular algorithmic structure of the layout. This cell is currently generated using CDL procedural code. It is possible to imagine a generator which, when given the number of busses and the address decoding ranges, could automatically create a custom address decoder. While the current method of generating this block is not as automatic it is very close.

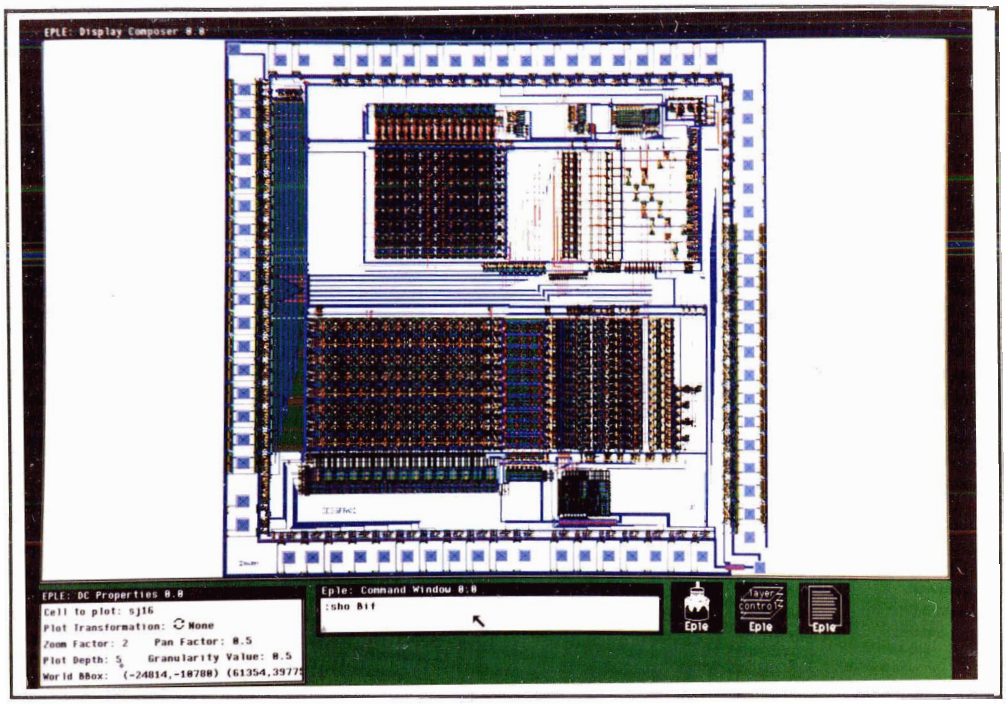


This is a complete register array with decoder which was produced procedurally using the cells shown in the two previous photographs. It is the desire of the EPLE project to be able to quickly create generators that produce this type of regular structured cell.

This is another cell from the data path of the SJ16 chip. This is the barrel shifter. It has been rotated 90 degrees in this plot but normally appears tall and thin. Notice its regular structure. This cell was also designed procedurally, which allowed its regular structure to be created algorithmically.



PLAs are yet another example of where procedural code can do a superior job. This PLA was created procedurally. The generator simply reads a truth table provided by the user and creates the need cell. This allows for rapid modifications to be made in the future because the user only needs to change the truth table, not re-layout the cell.



This is the SJ16. It was created using EPLE's predecessor CDL and VLSIplot. The total chip was created using procedural code. The register arrays, decoders, barrel shifters, PLAs, and address decoding lend themselves nicely to procedural layout due to their algorithmic nature. The other parts of the chip were awkward to create because there was no such algorithmic pattern to follow and in addition there was no graphical input description which could be used to short cut procedural layout process. EPLE has attempted to overcome this drawback by providing a powerful way of describing procedural code graphically.

Bibliography

1. C. Mead, L. Conway, Introduction to VLSI Systems, Addison - Wesley, Reading, Mass., 1980.

2. Steven M. Rubin, Computer Aids for VLSI Design, Addison - Wesley, Reading, Mass., 1987, Appendix C.

3. John K. Ousterhout, "The User Interface and Implementation of an IC Layout Editor", IEEE Transactions on Computer - Aided Design, Vol CAD - 3, No. 3, July 1984, pages 242 - 249.

4. Full - Custom / 3000 Tools Manual (VALE), Chapters 3 and 4, Metheus - CV Inc., Hillsbro, OR., 1985.

5. Steven M. Rubin, Computer Aids for VLSI Design, Addison - Wesley, Reading, Mass., 1987, Chapter 11 and Appendix F.

6. Steven M. Rubin, "Association of Textual and Graphical Circuit Descriptions and other Hard Problems in Computer Aided Design", Canadian Conference on Very Large Scale Integration, Technical Digest, Winnipeg, October, 1987, pages 5 - 16.

7. John K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, G. S. Taylor, "Magic: A VLSI Layout System", ACM IEEE 21st Design Automation Conference, 1984, pages 152 - 159.

8. Stephen Trimberger, "Combining Graphics and a Layout Language in a Single Interactive System", ACM IEEE 18th Design Automation Conference, 1981, pages 234 - 239.

9. Patrick A. D. Powell, Mohamed I. Elmasry, "The Icewater Language and Interpreter", ACM IEEE 21st Design Automation Conference, 1984, pages 98 - 102.

10. Patrick A. D. Powell, Mohamed I. Elmasry, "ICEWATER: A Procedural Design Language for VLSI", IEEE Transactions on CAD, July 1985.

11. M. Pulver, M. I. Elmasry, "Using IGLOO: A Constraint Based Layout Language for VLSI Design", Canadian Conference on Very Large Scale Integration, Technical Digest, Winnipeg, October 1987, pages 81 - 86.

12. Don J. Gamble, Dave J. McKie, edited (1987) John L. Simmons, "CDL for CMPT 490 / CMPT 852", School of Computing Science, Simon Fraser University, Unpublished, 1985.

13. R. J. Lipton, J. Valdes, G. Vijayan, S. C. North, R. Sedgewick, "VLSI Layout as Programming", ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983, pages 405 - 421.

14. N. H. E. Weste, "MULGA - An Interactive Symbolic Layout System for the Design of Interactive Circuits", The Bell System Technical Journal, Vol. 60, No. 6, July - August 1981, pages 823 - 857.

15. R.. Matthews, J. Newkirk, P. Eichenberger, "A Target Language for Silicon Compilers", IEEE Compcon, 1982, pages 349 - 353.

16. Jeffrey D. Ullman, Computational Aspects of VLSI, Computer Science Press, Chapter 7, pages 273 - 282.

17. W. Wolf, J. Newkirk, R. Matheus, R. Dutton, "Dumbo, A Schematic-to-Layout Compiler", 3rd Caltech Conference on Very Large Scale Integration, pages 379 - 393.

18. Jay R. Southard, "MacPitts: An Approach to Silicon Compilation", IEEE Computer, December 1983, pages 74 - 82.

19. D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan, R. A. Walker, "Automatic Data Path Synthesis", IEEE Computer, December 1983, pages 59 - 70.

20. Neil Bergmann, "A Case Study of the F.I.R.S.T. Silicon Compiler", 3rd Caltech Conference on Very Large Scale Integration, pages 413 - 430.

21. F. Anceau, "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits Specified by Algorithms", 3rd Caltech Conference on Very Large Scale Integration, pages 15 - 31.

22. Raul Camposano, "Structural Synthesis in the Yorktown Silicon Compiler", proceedings to VLSI '87 international conference, August 1987, Vancouver, pages 29 - 40.

23. L. I. Steinberg, T. M. Mitchell, "A Knowledge Based Approach to VLSI CAD the Redesign System", ACM IEEE 21st Design Automation Conference, 1984, pages 412 - 418.

24. H. Brown, C. Tong, G. Foyster, "Palladio: An Exploratory Environment for Circuit Design", IEEE Computer, December 1983, pages 41 - 56.

25. Warren Snyder, "SDL Layout Language", VLSI Design, Microtel Pacific Research Ltd., Burnaby, B.C., Unpublished, 1984.

26 B. W. Kernighan, D. M. Ritchie, The C Programming Language, Prentice Hall, 1978.

27. Don J. Gamble, edited (1987) John L. Simmons, "VLSIplot for CMPT 490 / CMPT 852", School of Computing Science, Simon Fraser University, Unpublished, 1985.

28. Leonard Gilman, Allen J. Rose, APL An Interactive Approach, Third Edition, John Wiley & Sons, New York, N. Y., 1984.

29. J. G. Wheeler, C. L. Kiernan, APL*PLUS / UNX SYSTEM USERS GUIDE, STSC Inc., Rockville, Maryland, 1986.

30. John A. Roach, "The Rectangle Placement Language", ACM IEEE 21st Design Automation Conference, 1984, pages 405 - 411.

31. Richard F. Hobson, "High - Level Microprogramming Support Embedded in Silicon", IEE Proceedings-E Computers and Digital Techniques, in press, 1988.

32. Richard F. Hobson, R. W. Spilsbury, W. L. Strange, J. D. Hoskin, J. L. Simmons, "Design Considerations For a New Memory Controller Chip", Canadian Conference on Very Large Scale Integration, Technical Digest, Winnipeg, October 1987, pages 149 - 154.

33. Richard F. Hobson, John Gudaitis, Jonathan Thornburg, "A New Machine Model For High - Level Language Interpretation", School of Computing Science, Simon Fraser University, CMPT TR84-18, 1984.

34. Thomas Lengauer, Kurt Mehlhorn, "The HILL System: A Design Environment for the Hierarchical Specification, Compaction, and Simulation of Integrated Circuit Layouts", Conference on Advanced Research in VLSI, M. I. T., Jan. 1984, pages 139 - 149.

35. T. Tokuda, J. Korematsu, O. Tomisawa, "A Hierarchical Standard Cell Approach for Custom VLSI Design", IEEE Transactions on Computer - Aided Design, Vol. CAD - 3, No. 3, July 1984.

36. W. H. Evans, J. C. Ballegeer, N. H. Duyet, "ADL: An Algorithmic Design Language for Integrated Circuit Synthesis", ACM IEEE 21st Design Automation Conference, 1984, pages 66 - 72.

37. Randy H. Katz, "Managing the Chip Design Database", IEEE Computer, December, 1983, pages 26 - 35.

38. Adobe Systems Inc., PostScript Language Reference Manual, Addison - Wesley, Reading, Mass., 1985.