BIRGMAN: a compact portable symbolic V.L.S.I. editor after INGRED, but with relative size relationship displayed

Robert Atkinson
University of Wollongong

Follow this and additional works at: https://ro.uow.edu.au/compsciwp

Recommended Citation
https://ro.uow.edu.au/compsciwp/30

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
Abstract

This report describes the implementation of a compact and portable graphical editor for V.L.S.I. circuits. The editor provides a symbolic design approach whilst still allowing the designer to view realistic size relationships. First, the aims of the project will be identified, followed by a discussion of the factors which influenced these decisions.

Before embarking upon justification of the aims of the project we must understand the nature of V.L.S.I. design. To this end, an overview of V.L.S.I. is provided. Readers familiar with symbolic design methods may refer directly to the concluding section of this background material.

The next section describes the implementation of the project. Having defined the principal aims we discuss their implications, and arrive at a description of the project. This description covers the relationship between the editor and the JMRC (Joint Microelectronics Research Centre, University of New South Wales) package, the internal organisation of the editor and the user interface.

The user interface is described in general terms, and then a more specific treatment of individual commands is undertaken. Much of the complexity of this project is due to the use of domain-specific geometric relationships of V.L.S.I. circuit elements to assist the user in specifying correct designs. In addition this knowledge can be used to make V.L.S.I. layout quicker and easier.
BIRGMAN:

A compact portable symbolic V.L.S.I. editor after INGRED, but with relative size relationships displayed.

( Bob's Interactive Reasonable Graphical Microcomputer Aid for NMOS designers )

HONOURS PROJECT
Robert ATKINSON

November 1986

Department of Computing Science
University of Wollongong
Acknowledgements:

There comes a time in everybody's life when they have to be humble. Tomorrow. In the meantime I would like to offer my sincerest thanks to those people who ghost-wrote the project. But seriously, I appreciate the efforts of my supervisor, Mr Gary Stafford, who provided much of the intellectual stimulation that led to this project, and much help (and goading) throughout the year. Much of the direction of the project is due to the help of Dr Philip Mc Kerrow, who leads the V.L.S.I. interest in this department. Obviously, a good deal of credit must be given to the JMRC team, who supplied the Department with their comprehensive suite of V.L.S.I. tools, and a hands-on demonstration.

Other people whose help was invaluable include Mr Peter Arnold (whose awesome all-night fights with the BELLE tools I was privileged to behold), Mr Philip Maker, Mr Micheal Shepanski and Mr Colin Atkinson.

Finally, I would like to thank the staff of the Department of Computing Science for their support, understanding and suppressing their giggles throughout the year, and Dr Greg Doherty especially for coming to my rescue with equipment money in spite of the inconvenience at the time.
CONTENTS

0. INTRODUCTION 5
1. AIMS of the PROJECT 6
2. BACKGROUND 7
   2.1 V.L.S.I. Design 8
      2.1.1 The salient features 8
      2.1.2 The problems 10
      2.1.3 The solutions 11
   2.2 'BELLE': a textual approach 12
   2.3 'KIC': a graphical approach 14
   2.4 Symbolic V.L.S.I. Design - the JMRC Package 16
   2.5 Why another V.L.S.I. editor ?? 18
3 IMPLEMENTATION 20
   3.1 Introduction 20
   3.2 Shaping forces 20
      3.2.1 Restrictions. 21
      3.2.2 Desires. 22
   3.3 Decisions - an honours project is born! 23
   3.4 Overview :-
      3.4.1 How INGRED and BIRGMAN relate 24
      3.4.2 The internal Organisation of BIRGMAN 26
   3.5 Data Organisation 27
      3.5.1 The object domain -
         3.5.1.1 Leafcells and Modules 28
         3.5.1.2 Ports and Connections 28
         3.5.1.3 Devices 30
3.5.1.4 Instances 32
3.5.2 Files and Libraries 33
3.5.3 Virtual Data Structures, and Caching. 34
3.5.4 The Global Editing Environment 36

3.6 The User Interface 37
3.6.1 The nature of commands 37
3.6.2 File based commands 39
3.6.3 Window manipulation commands 40

3.7 Leafcell Editing 43
3.7.1 The Leafcell Palette 43
3.7.2 Placement rules 44
3.7.3 Context-dependent relocation 45
3.7.4 Wire placement 47
3.7.5 Port placement 48
3.7.6 Parameter changing 48

3.8 Module Editing 49
3.8.1 The Module Palette 49
3.8.2 Cell Instantiation 50
3.8.3 Connections 52
3.8.4 Other Commands 55

4 PORTABILITY 57
5 CONCLUSION 60
REFERENCES 62
APPENDIX A. C.I.F. Sample
APPENDIX B. BELLE Sample
APPENDIX C. SILO Sample
APPENDIX D. The AMIGA terminal protocol
0. INTRODUCTION

This report describes the implementation of a compact and portable graphical editor for V.L.S.I. circuits. The editor provides a symbolic design approach whilst still allowing the designer to view realistic size relationships. First, the aims of the project will be identified, followed by a discussion of the factors which influenced these decisions.

Before embarking upon justification of the aims of the project we must understand the nature of V.L.S.I design. To this end, an overview of V.L.S.I is provided. Readers familiar with symbolic design methods may refer directly to the concluding section of this background material.

The next section describes the implementation of the project. Having defined the principal aims we discuss their implications, and arrive at a description of the project. This description covers the relationship between the editor and the JMRC (Joint Microelectronics Research Centre, University of New South Wales) package, the internal organisation of the editor and the user interface.

The user interface is described in general terms, and then a more specific treatment of individual commands is undertaken. Much of the complexity of this project is due to the use of domain-specific knowledge to aid silicon design. We can use knowledge about geometric relationships of V.L.S.I. circuit elements to assist the user in specifying correct designs. In addition this knowledge can be used to make V.L.S.I. layout quicker and easier.
1 PROJECT AIMS

This project was born out of a desire to design the processor for a multiprocessor message-passing machine, as an honours project. A survey of the facilities available at the time quickly showed the infeasibility of this in the year allowed. However, it did highlight the need for V.L.S.I. design tools suitable for large design projects. Our research into the availability of such tools led us to examine the package under development at JMRC. We decided to adopt the 'symbolic' approach of the JMRC tools, and in fact the JMRC package has been obtained by this institution recently. Unfortunately, the computing facilities required to support this package make it unsuitable for teaching purposes. Nevertheless, we thought the labour-intensive process of specifying design layouts could be supported on cheap, available microcomputers in laboratory, home, or classroom environments. The computationally massive tasks could be processed in batch on the department's mini-computer facilities.

Thus, the aims of this project are to implement a graphical editor for V.L.S.I. layout, such that it:

1. integrates with JMRC's extensive support tools and libraries,
2. is suitable for porting across a range of cheap micro-computers,
3. supports symbolic design methods,
4. allows the user to see real sizes, and
5. provides a more convenient interface for the designer.

Since layout is a graphical process, a graphical interface is required. Such an interface involves using some form of positional
input, such as a mouse or graphics tablet, and identifying the action specified. Many programs merely allow commands to be selected this way, as an alternative to the keyboard. Such an approach limits the amount of detail, and hence power of the commands, requiring the user to perform unnecessarily complex sequences of simple commands. These redundant commands serve only to make program code simpler, and results in an annoyingly pedantic environment. We can avoid seemingly useless actions by being sensitive to context, utilising more fully the information-intensive nature of graphics. Previous commands can establish a context in which to identify a new command. The context can be displayed in some convenient form. For example the interface can route wires, given that a wire has been selected previously (this fact is displayed by highlighting), and that two positions have been specified on the design field as consecutive commands. Moreover, the coordinates given can be used as a basis for finding another object nearby with which the wire will interact, and adjust the wire's position so that the interaction will take place correctly. This means that the user has to be accurate enough to resolve ambiguity rather than accurate in an absolute sense.

2 BACKGROUND INFORMATION

This section is provided to bring symbolic V.L.S.I. design into perspective. The need for software support for V.L.S.I. design is discussed, and the various generations of support tools are analysed. The last word is the marketing clause - why this project addresses the problems identified.
2.1 V.L.S.I. Layout

V.L.S.I. circuitry is created by depositing (actually a mixture of deposition and mask etching techniques are used) patterns of different material onto a silicon substrate. There are a number of different technologies, or ways of creating transistors in silicon. The nMOS technology is favoured for Multi-Project Chip (MPC) design, largely due to the relative simplicity of nMOS design methodology. It also provides quite good device density and speed. The examples given shall concentrate on nMOS design rules. The JMRC package presently supports nMOS only, and thus so does the editor. However, extending the project to support other technologies is quite feasible.

2.1.1 The Salient Features of nMOS V.L.S.I.

NMOS circuitry is created by the interaction of three layers, diffusion (or 'thinox'), polysilicon and metal. The layers are separated by insulating layers of silicon dioxide. Transistors (which more or less act as switches) are created by crossing polysilicon and diffusion layers. The metal layer form low-resistance connections between parts of the circuit. To connect the layers it is necessary to 'cut' the silicon dioxide insulation, so that metal can be laid down to form a connection. The layout is specified as masks defining where each material is to be deposited. The silicon foundries accept a code known as C.I.F. (Caltech Intermediate Form) which specifies in absolute coordinates the positioning of various shapes in the masks.
The coordinates used are in an arbitrary unit, the \textit{lambda} ($\lambda$). This means that the layout can be made with whatever value of $\lambda$ that the fabrication can support. In most cases it is possible to design circuit that will work over a wide range of $\lambda$.

Somewhat unfortunately the circuitry thus formed does not behave as absolute on/off logic circuits. There are various resistances, capacitances and consequent delays and signal degradation. It is important to be able to change the sizes of transistors (the overlap between polysilicon and diffusion) and the width of connecting wires in order to maintain voltages at logic levels. It is possible to define a number of rules-of-thumb to ensure that circuits will behave properly. Other rules specify minimum distances between objects in order to avoid electrical interaction.

The reasons for the popularity of nMOS technology include the simplicity of these design rules. The nMOS design methodology set out by Mead and Conway has gained universal acceptance. By following these rules it becomes relatively easy to create a circuit element correctly.

The rules specify:

(i) minimum widths of tracks  
(ii) minimum separations between objects on the same layer  
(iii) minimum extensions beyond overlap regions to ensure that the overlap will be adequate  
(iv) size of overlap region for contact cuts

In addition there are some rules defining when it is necessary to 'boost' signals to ensure that the correct logic level is maintained.
2.1.2 The Problems

The above mentioned rules make it easy to design a handful of logic gates, and specify the C.I.F. code. However, the complexity of V.L.S.I. is the source of its usefulness. Typically a circuit might contain 50,000 transistors. Every one has to be correctly specified and connected in order for the circuit to work. Keeping track of coordinates in a design of this size is a daunting task.

The only way to deal with complexity is by reducing it to simplicity. Thus a complex circuit is made by connecting simpler circuits. Very often modules are made by repetition of simple elements. For instance a register bank is a repetition of 16-bit registers, and each register is 16 repetitions of a 1 bit latch. High regularity is very important in making circuit design easier and quicker. Obviously a designer does not wish to specify the coordinates of each masking area in each repetition, when for the most part the relationships remain unchanged. Repetition of even very simple operations will introduce errors, apart from being soul-destroying.

Another important factor is the cost and turnaround time of fabrication. It is not feasible to use trial-and-error methods. Before entrusting funds to fabrication one must have confidence that the chip WILL work. The 'black box' approach allows libraries of components to be designed and debugged. This allows a much more 'top-level' design to be carried out, which reduces complexity considerably, and hence decreases the amount of time needed to arrive at correct designs.
2.1.3 The Solutions

The problems due to the complexity of V.L.S.I. circuitry are just those problems best addressed by computation. What the designer needs is a way of producing the mask specifications from a higher-level description of the circuit. The higher level description could incorporate the regularity of some circuitry, and the use of standard units. Such things as inverters and I/O pads are obvious targets for repetition throughout a design.

Often combinatorial logic can be implemented by PLA's (essentially just tables) instead of networks of gates. The layout of these can be automated - freeing a designer of much tedium. Such tools can also provide error-free circuitry without the extensive effort associated with checking a design.

Libraries of circuit elements thus significantly reduce design effort. The other massively complex task the designer faces is checking the electrical properties of the proposed chip. Electrical properties include logical correctness, timing, power consumption, noise immunity and correct operation. Design methodology can make it easier to specify a layout that will not have anomalous electrical behaviour, however logic and timing need to be analysed. For non-trivial designs computer simulation is essential.

There exist a number of tools which simulate electrical properties of circuits (for example SPICE). Support tools can automatically convert V.L.S.I. layouts into equivalent electrical specifications. Simulation makes heavy demands upon computer
resources, however it can be performed as a batch operation.

The aim of support tools is to make it as easy as possible to design correct chips. Given that the checking of design rules and circuit simulation can be automated, it becomes important to make mask specification as easy as possible. The remainder of this overview is devoted to describing the approaches used by successive generations of V.L.S.I. design tools.

2.2 'BELLE' (Basic Embedded Layout Language)

BELLE is a textual layout language. This means that masks are specified in some convenient textual form, which is then converted to the detailed C.I.F. (or similar) format. BELLE provides constructs to define repetition, the use of sub-circuits and some degree of parameterisation. It makes it possible to define small circuits, and then to join instances of these. These constructs are made available by embedding BELLE in the PASCAL programming language. Thus we have circuit elements defined as subroutines, which can be instantiated into larger circuit elements by high-level language calls. Repetition is provided by loop constructs, which together with the use of parameters can specify adjacent instances of cells, and the connections to be made. BELLE has a number of predefined PASCAL procedures, such as:

- `define('CELL');` starts a new cell definition
- `layer(poly);` sets the current layer
- `box(x1,y1,x2,y2);` draws a box in the current layer
- `diffcut(x,y);` places a cut at x,y
wire(width,x,y) \quad \text{starts a wire at } x,y
\quad x(\text{dist});y(\text{dist}) \quad \text{extends the wire in the } x \text{ or } y \text{ direction by } \text{dist}

The arguments to these procedures can be constants, or PASCAL variables, modified within a loop to place adjacent copies of the specified objects.

BELLE provides a more convenient language to work with than the simple geometric mask layouts. It does however have drawbacks. Perhaps the best way to illustrate these is to examine the BELLE design process, as elucidated by Pucknell and Eshraghian ("Basic VLSI Design"):

"1. Beginning with an initial sketch ... the circuit must then be laid out on graph paper to aid in digitization. This process usually takes more than one attempt to generate an acceptable layout.

Hints for layout.

(a) Use colored pens to help distinguish between masks.

(c) Liquid paper .. is almost a necessity to remove those occasional mistakes.

(d) Don't worry too much about achieving very dense layout packing in early attempts at design. Densely packed layouts are more likely to contain design rule violations than are sparse layouts.

(e) Always check and recheck your layout for design rule violations... Some violations are very difficult to detect ... It is wise to have at least one other person check your layout since a fresh outlook may spot elusive errors. "
There are two main difficulties with BELLE - the need for hand
digitization and the difficulty of finding design rule violations.
Moreover, design rules have to be hard-coded into a design, there is no
flexibility for different rules. The first step in chip design is to
draw the layout. If this were done in an interactive CAD environment
there would be no need for hand digitization, which brings us to the
second generation of tools - graphical editors.

2.3 'KIC' - An Interactive Graphical Editor

The next stage in V.L.S.I. design tools was the use of
interactive graphics to bypass the problems of manual drawing and
calculation of coordinates. Such an editor is KIC, developed by the
University of California at Berkeley. There are other similar systems,
such as RIDE (University of Adelaide), but KIC is focused upon for
two reasons: it has been used in the past by students in this
department, and it is included in the JMRC package.

KIC supports a hierarchical design. Instances of sub-circuits can
be invoked, and placed in the viewing window (The term 'window'
will be used to refer to the area of the design currently displayed. It
will conform to a section of a graphics terminal screen). Instances
are made of generic cells, stored in files under the name of the cell.

The majority of KIC commands place geometric objects onto the
window. Objects can be many different shapes: boxes, polygons,
wire, lines, circles, arcs and donuts. The object is placed in the
current layer - KIC supports 25 different layers - at any angle (with
some restrictions). For most V.L.S.I. design works this is unnecessary -
most fabrication houses do not support arbitrary angles for mask items, and the number of layers is set by the technology being used. Nevertheless, if strange geometries are required, tools such as KIC are the best solution.

Obviously, a graphics editor relieves us of much of the effort in specifying a design, once we know what it looks like. It facilitates the use of libraries of sub-circuits, since we can place it on the screen, and then connect it appropriately. There are limitations, however. If cells have been designed to fit each other specifically they can be placed next to each other correctly, and arrays of instances formed quickly. Unless this is the case, each cell has to be manually connected to its neighbours, which means that each connection has to be placed manually: there is no way of knowing where and how to connect cells automatically, since they are just geometric objects. This implies that libraries will tend to contain many different sizes of the same type of cell, or that each instance will have to be modified to fit its neighbours. KIC does provide a stretching facility, but this is the prerogative of the designer. Furthermore, upon instantiation of a large cell, the designer must then pay attention to all the individual connections that have to be made. A single mistake at this stage could stop the design functioning correctly. Design rules can be checked automatically, and KIC allows problems to be highlighted in the window. Missing or misplaced connections cannot be found as readily.

In essence these problems are all due to the rigid geometry of KIC designs. If it were possible to specify how elements were connected rather than where they were placed then the connection
process could be automated. Automation not only makes it easier for
the designer to make connections, he is less likely to neglect them, and they will be made with adherence to the design rules.

Another disadvantage of rigid geometry is that it must comply with specific design rules. It is difficult to change it to comply with, or take advantage of, different sets of rules dictated by changing fabrication techniques. On the other hand, there are potentially cases where the designer wishes to ignore design rules - to speed up critical areas of the circuit. It is possible to reconcile these considerations by allowing rigid geometry as a special case, in a more flexible 'symbolic' environment.

2.4 Symbolic VLSI Design - the JMRC Package

Symbolic V.L.S.I. design is based upon the concept of connecting devices, instead of specifying the actual geometries of each layer mask. The geometries can be generated at a later stage - and modified if necessary by a geometric editor. This has the advantage of abstracting the design rules from the circuit specification. The mask generation is performed according the adopted set of design rules - so the designer is freed of this onerous task, and cannot introduce errors this way. Past projects in this department have illustrated the difficulty of eliminating design rule violations without software assistance.

The post-processing of chip designs includes the automatic stretching of cells in order to make connections correctly. This is achieved by an algorithm known as corner-stitching, in which the
cells are first placed in their minimum dimensions, and then stretched to make connections with those already placed. Placement thus starts in a corner, so all new placements can only affect known regions of the design. Thus the algorithm involves backtracking to re-space the two strips of design that intersect in the region covered by the new cell. The JMRC tool that performs corner-stitching is called JIGSAW and can produce interactive graphical output.

There are several important implications of automatic spacing. The first is the potential for producing inefficient designs. In practice it tends to produce designs 10-20% larger than geometric methods. This is not particularly excessive, considering the reduction in turnaround time that is possible. Moreover, scrutiny of the spaced design can reveal just where space is being wasted. The package allows the use of geometric cells, which cannot be manipulated internally. These can be used when exact sizes are important, or to reduce space requirements. The relative placement of sub-circuits might also be changed to minimise space requirements.

JMRC symbolic designs are specified in a textual language called SILO (see Appendix C). There are four basic types of SILO cell: leafcells, modules, cifcells and routecells. Leafcells contain the connection details for low-level devices, transistors, wires and cuts. Modules contain instances of cells, and the connections between them. Cifcells contain a reference to a CIF file, and information on the connection ports and size. Routecells are special cases of leafcells containing no transistors, and are generated automatically by routing software.

Leafcells are made by specifying the types of transistors, and
the connections between them. Topology is represented by a virtual grid, as opposed to a lambda grid. Using a virtual grid relative placement is specified, not physical distances. The JIGSAW program will calculate the appropriate spacing for individual grid lines in order to maintain the topology without violating design rules.

The importance of allowing automatic spatial adjustment rests in the capability of maintaining libraries of useful components. A cell can be stretched to place it's ports ( inputs and outputs ) anywhere on the boundary consistent with the topology and design rules. Thus the library needs only a copy of a generic cell type in order to provide cells with any particular spacing of ports. Cells can be transformed on instantiation by rotation and mirroring. The ports of each instance are then connected together, which provides the information used to stretch them during CIF generation.

2.5 A New Editor For Symbolic Design

The symbolic editor provided with the JMRC package ( INGRED ) provides a menu driven graphical interface. It displays devices as fixed size icons on a virtual grid. This is consistent with a purely symbolic approach, however it tends to obscure some of the important information. Whilst the size of the cell will not depend upon the distances actually left between objects, it will depend upon the topology chosen. There are generally many ways of specifying the relationships between devices in order to make equivalent electrical circuits. In order to make the best choice of topology the relative sizes of objects are important data. Since the sizes are known
(because a transistor's length/width ratio is specified) they should be displayed if possible.

There are some difficulties with displaying relative sizes. With a grid, coordinates need only be translated to the nearest grid position in order to perform an operation, such as inserting an item. Collisions between objects can be found by checking single points. Using relative sizes, displaying a grid no longer makes sense, and coordinates have to be referenced to nearby objects. To maintain the ease of use of a grid environment demands much more complexity internal to the program, but it does give the user a realistic spatial context in which to make topological decisions.

INGRED commands are chosen from an extensive menu, which makes explicit distinction between operations on different types of device. Thus deleting a transistor and a wire are separate commands. This leads to an extensive menu, requiring a large graphics screen, and could be rationalised at the expense of a more complex program. The lack of expensive graphics terminals in this department led to the examination of implementing an editor on a cheap color microcomputer for teaching purposes. INGRED, which runs under UNIX and a device-independent graphics package is perhaps not the easiest of programs to squeeze into the limited memory available on many of the cheaper microcomputers.

From these considerations a need for a small, portable INGRED-like editor arose. It was decided that the advantages of showing relative dimensions outweighed the extra complexity, and that the INGRED interface would serve as a reference point for building a convenient editor.
3 IMPLEMENTATION OF THE EDITOR

3.1 Introduction

Having justified the major design decisions, a symbolic approach showing relative sizes, and suitability for microcomputer workstations, we will analyse their implications. How these created other desires and imposed restrictions will be discussed, and the resulting implementation described. A brief overview of the relationship between the project and the JMRC package, will be followed by a brief discussion on the internal organisation of the project. The data structures and memory management are described in some detail, as they are the key to the compactness achieved in the implementation. Ease of operation requires consistency in form of the various editing features. The command paradigm is presented before the individual commands are explained. Since there are a large number of parameters to control, there are quite a few distinct editing commands, and only a cursory description of the means used to effect the changes is provided in the interests of conciseness.

3.2 Shaping Forces

The major aims of the project have led to a number of other considerations, some of them related to ease of use, and other designed to minimise the effect of certain restrictions. These influencing factors are presented here to place the project's final
shape in a correct perspective.

3.2.1 Restrictions

The decision to create a portable editor supportable by a range of microcomputers had a wide-ranging influence on subsequent design criteria. As a starting point, the C language was chosen because it is efficient and supported by most of the envisaged target microcomputers. For portability we had to forego any of the built-in features of individual machines, and make a bare minimum assumption about the graphics and operating system support available. This required the project to become involved in quite low-level I/O details.

Furthermore, the size of the program's code and data segments had to be consistent with the memory capacity of most cheap microcomputers. This restriction precluded the use of much of the standard library routines. For example, the ubiquitous 'printf' routine, of which only a few features are ever used, tends to add about 10-12Kb to the code requirements. This was felt to be an unacceptably large amount.

Yet another restriction is the quality of graphics available. The extensive menu of INGRED requires at least 30 lines of text. This is unlikely to be supportable on most microcomputer screens. A rationalisation of menu commands was therefore demanded. As well, the KIC features allowing the colour and pattern of layers to selected interactively are not supportable in general.
3.2.2 Desires

Running large programs in a microcomputer environment is often a painful experience to those spoilt by minicomputer performance. The most obvious cause for dissatisfaction is the speed (or lack thereof) of disk I/O. Thus one of the desires of the designer was to minimise this aspect, by appropriate choice of file formats and an intelligent use of available memory. To this end, the data structures had to be designed to facilitate minimal disk I/O and memory usage.

Apart from the environmental considerations, there was a desire to make an editor that compensated for the relative difficulty of specifying placement accurately through graphical means. An interface that made good guesses as to the user's intentions was called for. The emphasis on good is important. Software that second-guesses the user poorly is frustrating to use. The desire was to make as good a guess as was possible, without ever making an inappropriate choice.

One other desire that arises from the nature of symbolic V.L.S.I. design is the abstraction of design rules. The actual design rules are not necessary to edit symbolic designs. Any contextual decisions made to disambiguate commands should not be based on the exact spacing rules of any one methodology. Instead it is sufficient to use placement rules to resolve topological ambiguities. Future versions of the editor might allow generalised placement rules to be specified, and used during design. Real-time interactive composition (spacing) could also be supported.
3.3 Decisions : A Honours Project is Born!

The shaping forces above led to the adoption of the following specifications :

(i) The project is to be a graphical editor for nMOS V.L.S.I. circuitry, to produce symbolic topologies. The ability to support other technologies will not be a prime consideration at this stage, but will be taken into account for later development.

(ii) The final output will be suitable for use by the JMRC tools.

(iii) The editor shall display relative sizes of objects, as an aid to efficient layout.

(iv) The project is to be written in C, with minimal assumptions regarding I/O support.

(v) The project will not exceed 64K code and 64K data segments (as minimum requirements)

(vi) Additional memory is to be exploited if possible.

(vii) File formats should be consistent with internal data structures, to minimise disk I/O and program size.

(viii) Cell designs are to be converted between the textual form of SILO and the compact format.

(ix) Libraries of cells shall be supported.

(x) Designs shall be hierarchical, ( consistent with SILO ), and this hierarchy should be visible on the display.

(xi) Caching of cell designs shall be implemented to reduce disk I/O.

(xii) The display of large designs, involving more cell definitions than can be accomodated in memory simultaneously will
be supported.

(xii) Only the top level of the current hierarchy will be modifiable by the editor commands.

(xiii) No features specific to individual machines shall be assumed as available.

(xiv) A minimum of graphics and file primitives will be used.

(xv) Screen use shall be parameterised to allow for different sized screens.

(xvi) The menu sizes shall be consistent with the resolution of the envisaged microcomputer graphics screens.

(xvii) The interface will make the best use of the information present in context to minimise interaction. This implies second-guessing the user in some cases, which will be performed as much as possible with consistent correctness.

The final form of the project adheres to these design decisions. We shall now present a discussion on the actual workings of the project: it's interaction with the machine, the JMRC package and the V.L.S.I. designer.

3.4 Overview

3.4.1 How INGRED and BIRGMAN Relate.

Since INGRED is the JMRC tool for producing symbolic nMOS designs, we wish BIRGMAN to produce a similar format, and then use exactly the same procedure for processing the resultant chip
specification. The JMRC documentation covers this more fully than could be attempted here, and is suggested as a prior reference to anybody contemplating V.L.S.I. design using either editor.

The symbolic layout specifications so produced are spaced by another program. This spacing is a computationally involved task, and it is envisaged that it would be done by a powerful minicomputer. The program that achieves this, JIGSAW, makes heavy demands upon both the processing power and available memory. At some stage a version might become available that will run on microcomputers, just as this editor has been created. In the meantime a V.L.S.I. design course would probably use the JIGSAW facility as a batch processor.

To use SILO files BIRGMAN requires the cell to have been spaced by the JIGSAW program. This program spaces virtual grid lines to get the minimum size of the cell consistent with design rules. If the cell has not been spaced, the only information available is the grid lines, which is insufficient for the display of relatively sized objects. This spacing could be abstracted from the object sizes by the format converter, but this would merely be a duplication of effort. The other advantage of this decision is that the BIRGMAN set of tools is then completely independent of any particular set of design rules.

Two separate programs are provided to convert files between the compact form used by BIRGMAN and the textual format of SILO. The existence of separate programs, and formats is due to a number of reasons. Because we are constrained to a relatively small program that minimises disk access the SILO format is unsuitable. The textual representation requires lexical scanning and parsing every time it is needed, and is difficult to reduce disk accesses without extensive
buffering. Since we are making minimal assumptions about the operating system, we would be required to perform this buffering, making heavy demands upon our tightly restricted space requirements. The amount of code required to perform the scanning and parsing would also restrict the interface possibilities.

The separate tools provide a convenient method for communication between a workstation and a remote host, since the workstation will not need access to the Ascii encoded SILO, the tools can be set up to address a communications port, if the operating system is not capable of redirecting output.

The file organisation is similar to that used by JMRC tools. Each cell exists in a file of the same name as the cell, and a directory is used to collect all the cells used in a chip. Libraries exist in separate directories. Should such a simple arrangement be incompatible with the operating system of the workstation then a 'virtual file' system could be built on top of individual files. For a more detailed discussion of these considerations see the section on 'Portability' in this report, and the 'BIRGMAN Technical Report'.

3.4.2 The Internal Organisation of BIRGMAN

The project can be divided into a number of distinct sections, which interact minimally with each other. In most cases the interaction is through a number of global parameters, describing the current editing environment. Some sections are responsible for changing the environment, others merely use it to provide output. Briefly the editor is composed of:
(i) Menu handling and command selection.
(ii) File manipulation routines
(iii) Memory management
(iv) Display routines
(v) Screen manipulation commands
(vi) Object selection
(vii) Object manipulation commands
(viii) A global editing environment.

The editing environment includes the specification of the cell currently being edited, screen display parameters, and object manipulation context. Many of the editing commands are window oriented, their effect based upon the object context. At any one time only one object can be selected, and subsequently the type of that object is used to distinguish between editing functions. This concept will be expanded more fully below.

3.5 Data Organisation

A general overview of the data that the editor has to manipulate, and the methods used will be given here. For a more comprehensive treatment refer to the technical report. Firstly we shall discuss the objects that are to be manipulated, and then the way they are represented internally to facilitate this. After this we will briefly describe the file organisation, followed by the memory management techniques used to save disk accesses and allow editing of large hierarchical structures.
3.5.1 The Objects

3.5.1.1 'Leafcells' and 'Modules'

The basic unit of chip specification is the cell. Cells of two types can be edited, leafcells and modules. They differ in internal representation, although they are connected identically. The cells form a hierarchy: cells can contain smaller cells. Thus the distinction between the two types is born: modules contain instances of cells, leafcells contain devices. Combinations of devices and instances are not supported by the symbolic representation, SILO. This is consistent with modular design methods.

Leafcells contain all the details about connection of devices, and thus occupy the bottom level of the tree (hence the name). All the nodes in the hierarchy are modules. Design is conducted in a top-down fashion, and implemented from the bottom-up. Having decided what leafcells are required, they are then built, and connected to form units with more and more complex functions. At the final stage, the large units are joined together as appropriate. The physical layout of the functional units is not important. Connections can be routed if the exact 'pin out' does not match what is required, or the lowel level cell can be modifed (one or more instances).

3.5.1.2 Ports and Connections

The common feature of leafcells and modules is the method of interconnection. This is acheived via the use of ports. A port is a
named object that lies on the boundary of a cell, and is internally connected as appropriate. All connections are made to ports. A port has a name, which must be unique to the edge upon which it is placed. Thus a port is identified by port name, cell name and side. In addition, a port has the attributes of \textit{width, position} and \textit{layer}. The layer is the actual material on which the connection will be made, and the width is the width of the track that will be used. The position parameter describes the position along the edge of the cell. In the SILO representation this position is stored relative to the parent cell. This includes those ports on cells internal to a module.

\textbf{Connections} are made between ports of the same layer type. They can either be made by \textit{abutting} or \textit{routing}. In the case of abuttment the cells are spaced to make connected ports align. If the connections are to be routed, then a special cell containing only \textit{wires} (a \textit{routecell}) is created at some later stage, and that is then abutted as appropriate.

Ports and connections are stored in contiguous arrays so that single disk operations can be made for them. Arrays can be extended dynamically to allow expansion, doubling the size each time there are no vacant positions. Deletion is handled by copying the last item in the array into the hole occupied by the deleted item. There is no attempt made to keep items sorted in any way, in the interests of code size. Keeping spatial coordinates sorted is the only way that could offer any advantages, and this proves to be of limited value. The number of items in any given array will never preclude simple searching of the whole array, and any sorting would simply increase
code complexity. A more compelling reason not to sort the elements is the fact that we are keeping dynamically reallocatable arrays. Because of this, we cannot maintain pointers into the array, but instead keep indices, which will always be valid. Sorting would involve address table sorting, or modification of existing pointers. Either would involve more code and data than the potential benefits could justify.

3.2 Devices

The devices described here are specific to nMOS V.L.S.I. circuitry. Initially we shall present the different types of device, followed by the attributes of each, and finally the ways that they interact (in a symbolic design methodology).

Transistors are the workhorse of V.L.S.I. They are the switching elements that make logic possible. Physically they are created by crossing a diffusion channel with a polysilicon gate. In symbolic design a transistor is described as a single unit instead. No transistors other than those explicitly specified can exist. This protects the designer from making circuits which behave unpredictably due to unintended transistors. There are two types of transistors, enhancement and depletion mode. A depletion mode transistor has an implant layer at the junction of poly. and diff. This changes the sense of the logic level required to activate it. A transistor has important electrical properties associated with the ratio between length and width of the overlapped region. There are three spatially distinct coordinate pairs to specify a transistor in
symbolic V.L.S.I. layout: source, gate and drain. The gate is the polysilicon region, whilst the source and drain are the diffusion extensions on either side of the gate.

**Contact cuts** are connections between layers. Normally layers do not make electrical connection because of the insulating material between them. A cut specifies a hole in this insulation. There are three types of cut: **diffcut**, **polycut** and **buttconn**. A fourth, the **buried contact** is not yet supported by the support tools. The editor has, however, been written to accomodate such extensions relatively easily. Polycuts and diffcuts allow connection from those layers to the metal layer. They require only a single coordinate for position. A butting connection is a connection between polysilicon and diffusion layers, made through the use of metal, thus:

![Butting Contact (buttconn)](image_url)

**Figure 1**

**Wires** are the other type of device. They are used to connect transistors together and to the ports on the edge of the cell. The contact cuts are generally used to connect wires on different layers.
together. A wire has two coordinates associated with it, specifying the endpoints. When a cell is stretched wires also stretch so that they still connect to the same devices. Wires also have the attributes of width and layer. The design rules specify minimum widths for wires, depending upon which layer is used.

All the devices are stored internally in the same template, so that they can be treated the same way during duplication, insertion and file access. The devices for a cell are stored in a contiguous array in the same way as ports and connections.

3.5.1.4 Instances

Instances are the copies of generic cells that are connected together to form a module. Each instance has all the external connection detail of the generic cell associated with it. An instance thus has a list of ports, a transformation and a bounding box associated with it, in addition to a name unique within the module, and a cell type (the name of the cell type). The bounding box serves to locate the instance within the module. If the module has been spaced by JIGSAW then the bounding box will be the stretched bounding box. The transformation is a 2x2 matrix, with two non-zero elements on a diagonal. This means that it can describe mirror transformations and 90 degree rotations, or any combination of them. There is one more field associated with each instance: whether or not it is 'uncovered'. An uncovered instance means that the associated cell type is displayed (after the appropriate transforms) within the bounding box of the instance. Thus uncovering instances gives rise to
an acyclic graph of cell relationships within the memory. This has implications later, when we discuss memory management within the program.

Instances are stored in arrays like the other data elements, with an associated port list allocated for each.

3.5.2 Files and Libraries

Each different type of cell has a unique name within an integrated circuit design. In the JMRC package (under UNIX) each cell is stored in a file under the cell name. All the cells used in a design are stored in a single directory. Since most microcomputer operating systems of today support tree structured file systems, the same approach is used by this project. The section on 'Portability' discusses alternatives if the host file system cannot be used in this way.

Each instance, within a module, refers to a cell by name. This avoids pointless duplication of data within the file system, but does have some implications. If a cell is changed, then all instances of that cell are changed. To change a single instance, it is necessary to change the type (cell name) of the instance, duplicate the cell being modified, under the new name, and then edit it.

Within each file is an indication whether the cell is a leafcell or module, the name of the cell, and the bounding box of the cell. Following this are the contiguous arrays of ports, devices, instances and connections. These are exact copies of the internal data structures, which reduces I/O time and code size. Because of this it
is necessary to produce the cells locally to the machine. The two format conversion tools provide the facility for transporting cell designs. The positional data stored in each cell is relative to the bounding box of the cell. The cell is stored without any transformation details.

BIRGMAN supports the use of libraries of cells. When a cell is to be edited, it is extracted from the current library. If no library is currently selected (from a menu), then the current design directory is searched. The editor will not allow a cell within the current design directory to be overwritten accidently. In this implementation only a set of flat (single level) libraries is supported, but extending the file handling to support multi-level libraries is envisaged.

3.5.3 Virtual Data Structures, and Caching.

The user of floppy disks becomes acutely aware of redundant file accesses. Take, for example, the following scenario: a user saves text file 'A' of 20K bytes size, then edits 'B' (also 20K), and then edits 'A' again. If the user has to wait five seconds for 'A' to reload, then he has every right to begrudge paying for the megabyte of memory in the machine. If, on the other hand, he is unable to load 'B' because all the memory is used to save 'A', then dissatisfaction will arise. BIRGMAN, with an acyclic graph of complex data structures, each representing a file, needs a more sophisticated approach to reducing disk accesses than a simple text editor.

In BIRGMAN the same mechanism is used for intelligent caching and allowing only parts of the graph to be resident. Whenever a cell is
loaded or created, it is placed in the cache. If it is required subsequently, then the reference count associated with it is incremented. These accesses are generally due to the cell being *uncovered* during editing. When the design is drawn, and the uncovered cell is in the window, then the cache is searched for the appropriate cell. If it is not there, then the cell is loaded from the disk. If the cache is full, then the cell with the *lowest reference count* is purged, and replaced. Purging involves freeing all the memory associated with it.

There is a restriction in the choice of which cell is to be purged, however. All the cells that form the path through the hierarchy to the cell being drawn must be inviolate, because the design is drawn recursively, and the cell data for each node in the path is thus being accessed.

Purging also takes place if there is insufficient memory to load a new cell. The program can make use of all the memory available before needing to perform this action. Purging takes place on a cell by cell basis until enough memory is available. Because of this feature it is possible to display designs that are too big to fit simultaneously in memory. All that needs to kept is the root cell, and one other - being the cell currently drawn. This is why the term 'virtual data structures' is used.

As mentioned the cell with the lowest reference count is removed. This reference count need not be zero. If there is a need for space, then cells can be 'swapped out'. It is not likely that this will be necessary very often, but it does allow the display of complete chips on machines with little memory to spare. When a cell is loaded,
it has a reference count of zero. Covering all the instances of a particular cell will zero it's reference count, making it a likely candidate for removal. When a cell is dereferenced, then all of the cells referenced by it become less important. Dereferencing is therefore also a recursive process. A cell that is a relic of a past editing session will have a zero reference count, as will all of it's children which are not in use elsewhere.

One potential problem with caching is consistency. The user should always know what to expect when he retrieves a cell from the cache. In all cases, the cache should have the same data as the disk file. Therefore, if a design is 'dirty' (the changes have not been saved) after editing, then it is imperative that it is purged from the cache. In addition, if a cell is to be taken from a library it is not likely to be the same as the cached version. Thus the cache is only checked for cells from the current design directory. All other references cause the cache to be cleaned.

3.5.4 The Global Editing Environment

As well as the cell data, there are a number of parameters that are used to describe the view the user has chosen, and the editing context. The parameters used to display a design in the window are:

(i) The scaling factor (zoom)
(ii) The transformation matrix to be used
(iii) The origin of the current cell, in global coordinates.
(iv) The position of the viewing window in the current coordinate system.
(v) A flag to indicate if port names are to be shown

Parameters (ii), and (iii) are used to be able to recursively draw the cells in the hierarchy. The root cell has its origin at 0,0 and an identity transform. The transform is a cumulative transform of all the cells in the path to the cell being drawn. It is kept globally for efficiency, since it is heavily used.

The *editing context* is the data that is necessary for all the editing functions to have access to. It includes:

(i) a pointer to the **current cell** - the one being edited.
(ii) a *Dirty* flag - use to warn if changes are about to be lost
(iii) The editing **mode** - how to interpret a command
(iv) A default **object** - what to perform operations upon.

### 3.6 The User Interface

This section will describe the way commands are given by the user, and what has to be done by the program to interpret them.

#### 3.6.1 The Nature of Commands

The interface consists of a set of general commands, common to both leafcell and module editing, and a set of specific commands for each. The common commands are generally those used to control the editing session, and display context. The prime consideration of the interface design is to provide a "Do What I Mean" environment. Every piece of information that is available should be used to minimise the
input requirements from the user. To this end all commands that require additional input can be interrupted by another command. This is supported by having all commands return a command identifier as status to the controlling logic. If this status is a valid command it is performed immediately, otherwise the next input is waited upon.

All commands are interpreted on the basis of positional context. A command is therefore just a positional indication, such as pressing a mouse button. There are three active areas of the screen, the menu area, the display window and the **palette**. The palette provides a selection of objects to choose for insertion. The objects in the display window can also be used in exactly the same way. There can be one selected object at any time - this is highlighted and is used as the default object for most operations. The variations on this theme are discussed more fully under the sections on editing leafcells and modules.

The window commands are interpreted according to the **mode**. Some of the menu commands also require window coordinates as additional input. The two types of mode are **insertion** and **selection**. If there is already an object selected, then it is assumed to be a template for further instances. If a copy of the object cannot be placed at the new location, then it is assumed that insertion was an inappropriate action, and the editor reverts to **selection** mode. In selection mode the position is used to find another object. If it is successful then the new object becomes the insertion template. Pointing at the selected object de-selects it in most cases (except when it is appropriate not to do so for other reasons).

A great deal of effort has been made to make the commands as
suitable and easy to use as possible. The actual form of most commands has evolved through experimentation. Furthermore, it is assumed that user input will be inappropriate, and is therefore checked for validity before acceptance.

The three menus used are

(i) File and library cell viewing and selection
(ii) Leafcell editing
(iii) Module editing

The first menu is shown upon startup, and whenever the user wishes to edit a new cell. The second and third provide all the editing and display commands, of which those common to both are described in the remainder of this section.

3.6.2 File-based Commands

The cell selection menu has four commands:

(i) 'quit' - used to leave the editor.
(ii) 'edit cell'

This command prompts the user for a cell name. If the current library is not the design directory, then the cell is copied from the current library to the design directory, and opened for editing. If a cell of the same name exists already then the user is notified and no action taken. If the named cell cannot be found then an appropriate message is generated and the user returned to the menu. If the cell has been found and loaded correctly then the user is shown the appropriate editing menu, and the design is displayed in the window.
(iii) 'new leafcell'
(iv) 'new module'

These two options are used to create new cells. The relevant editing menu is displayed, but no file is generated at this stage in the proceedings. Any cell of the same name is purged from the cache. Thus creating a new cell and then not saving it will ensure that the cell is purged from the cache, should the replacement of floppy disks cause the current design directory to be changed.

The rest of the file selection menu is devoted to a list of the libraries available. Wherever possible pathnames will be supplied as command line arguments, but some operating systems may force interactive specification. When the cell selection menu is invoked, the current default library will be highlighted. Selecting any library as a menu command will cause that to become the default library, and will generate a list of the elements within. The library manipulation routines are the only ones that make use of any operating system facilities in the current implementation. The creation and updating of directories may be undertaken by the program if necessary in future implementations.

There are also some file based commands available from the editing menus as well. These are:

'next' - next is used to return to the cell selection menu. If the 'dirty' flag is set the user is prompted to save any changes. If the changes are not saved the cache entry is purged, so that the cache remains consistent with the files. Since 'next' is a drastic action (in an editing sense) it must be selected twice to activate. This ensures
that it will not be selected accidently.

'quit' - 'quit' is treated exactly as 'next', except that it exits from the program rather than returning to the cell selection menu.

'save' - the changes made to the cell are saved to a file in the current design directory. If a file already exists then the user is informed, and asked whether or not to replace it. 'Save' has the effect of clearing the 'dirty' flag.

### 3.6.3 Window Manipulation Commands

The editing menus provide the following set of operation to adjust the viewing area to the designer's wishes:

'redraw' - Simply redisplays the viewing area. This is provided because some commands leave the screen untidy, especially wire insertion.

'locate' - this displays the cell so that it just fits into the viewing window. This is useful in case the design was 'lost' to view through careless use of the other commands. 'Locate' works by comparing the ratio of the bounding box dimensions and the window dimensions, to decide whether to use horizontal or vertical size to scale the design into the window. Having found the appropriate scaling factors the viewing window is placed according to the bottom left-hand corner of the cell. The scaling factor is adjusted to allow a margin around the cell for the sake of neatness.

'zoomin' and 'zoomout' are used to display selected areas of the cell. If an object is currently selected, then both zoom commands
will locate the viewing window so that the object fills the viewing window in one dimension. If there is no default object they both accept pairs of inputs, specifying the corners of a rectangular region. If the region has a zero dimension then it is ignored. Either diagonally opposite pair of corners is satisfactory. 'Zoomin' displays the selected area so that it fills as much of the window as possible, whilst 'zoomout' displays the current window in the area specified. In order to perform a zoom operation the relative aspect ratio of the area being 'zoomed' to the window is calculated. From this, the new 'zoom' or scaling factor is calculated, with bounds checking. The zoom factor is stored as an integer, so that no floating point operations are ever done to transform positions between coordinate systems. The actual significance of the value was chosen to allow a zoom of 1 to display a device of minimum size using the whole window. The user should be able to 'bottom out' before the picture becomes meaningless, which occurs when fewer than about 4 \( \lambda \) is displayed in a single dimension. There is a parameter which scales the zoom so that it need never take fractional values. This constant describes the maximum number of pixels that can be used to display a single \( \lambda \) unit. The maximum level of zoom is set at 32000, which allows a chip of about 150,000 \( \lambda \) in each dimension to be displayed. After calculating the new zoom factor the position of the window origins have to be calculated. In the 'zoomin' case they simply become the bottom left of the zoom rectangle. 'zoomout' on the other hand requires the calculation of where the zoom rectangle would be in the new coordinate system, and translating the current window origins by the same distances
outwards. The zoom process is performed for each pair of coordinates specified in the window, until a position outside the window is specified. Crosshairs are drawn on the screen to show where the the rectangle will be. If the two points are very close, no zooming takes place. Thus, if the user has indicated a point, he can ignore that by indicating it again, without leaving the zoom facility.

The 'pan' command is considerably simpler: it centres the display around the point in the cell indicated. Successive window positions cause repetition of the panning process. Like the zoom commands, panning is terminated by a non-window command entry.

The final command, 'settings' allows the changing of some of the display parameters. These are displayed at the bottom of the screen. One parameter that can be changed is the 'Display names' flag. This indicates whether or not portnames should be drawn upon the cell. Displaying portnames becomes somewhat messy when the cell is not drawn large enough to distinguish ports. This flag is toggled. The other settings that can be changed are the 'zoom factor' and the name of the cell. Entering either of these values can be overridden by any other command.

3.7 Leafcell Editing

Leafcells contain devices and ports. Although these are quite simple objects, there are a number of rules for describing the legality of interaction between them. Being relatively small objects, it is difficult to place them accurately with any consistency, without using a very close-up view. To overcome this involves searching the
nearby design space for clues regarding just where objects should be placed.

3.7.1 The Leafcell Palette

As has been mentioned, the palette provides a graphical way of choosing the next item to insert. Pointing at a palette item selects that item. If an object is already selected, it is deselected in favour of the new object. Pointing at a selected object deselects it, leaving the editor in insert mode. The objects visible in the display window are considered to be part of the palette, they can be selected in the same way. The editing method is to select an object, and by then pointing at a vacant place in the window, to insert a copy of that object, with identical parameters. The parameters of any selected object, from the palette or window, can be modified by the change menu command. The palette positions are parameterised internally to allow extension at a later date. Such extensions might be to include a second metal layer, and buried contacts. Although beyond the scope of this implemention, it should be quite possible to generalise the palette items, and the ways they interact. Thus the leafcell editor could support any silicon fabrication technology, and quite possibly other types of application. The palette items are permanently stored, as complete templates for new instances of them. When the item is to be placed in the design, a copy is 'massaged' (the coordinates relocated). This is then checked against the positions of devices already placed for layout rule violations.
3.7.2 Placement Rules

Most of the design rules specify minimum dimensions of devices, and separations between them. The actual values are not important here, however the consequent symbolic layout rules are:

(i) Objects on the same layer are either joined, in which case they share a coordinate, or else they must not touch.

(ii) Polysilicon and diffusion wires must not cross, unless there is a transistor at the junction point oriented such that the polysilicon wire crosses the gate at right angles to the diffusion joining the source and drain. The editor will disallow this.

(iii) Cuts cannot overlap other cuts, nor can a cut be placed where it interferes with another layer.

(iv) Polysilicon and diffusion must join at the appropriate ends of a butting connection.

These rules also govern the placement of ports. A port is placed to connect to a device on the same layer.

3.7.3 Context-dependant Relocation

The layout rules above make it more difficult to create a graphical editor, because they dictate that every device insertion
must conform. This means searching through the data structures, applying various tests of legality. On the other hand, there is a great deal of information available to us to help the designer. Since we have stringent rules for deciding whether a device placement is legal, we can use them to make it easier to place devices. This is the heart of the "Do what I mean" abilities of BIRGMAN. Whenever a coordinate is specified on the display a search is made for an object with a coordinate within a certain range of the indicated point. Since the absolute minimum width for an object is $2\lambda$, and the separation between non-connecting objects is not less than $2\lambda$, then a range of $3\lambda$ units is appropriate. Any or all of the coordinates may be relocated in this way. Consider the example below, when two transistors have been placed already, the target area indicated will cause the next to align properly (properly means that it is possible to make minimum width connections between them).

![Fig. 2 Coordinate relocation.](image)
The first version of the leafcell editor did not make this relocation, and was difficult to use as a result. The inclusion of this feature was a great improvement, allowing objects to be placed correctly with little concentration.

The placement of ports along the cell edge also makes use of the same strategy, with similar improvements in ease of use.

If, upon relocation, the new device collides with some object, then the legality of the combination is checked. Thus a transistor can be placed such that a poly wire connects to the gate, a diff to the source or drain, or under a metal wire. If the collision is illegal, then it assumed that insertion was not intended, and the item at the insertion point is selected. In practice this assumption seems justified.

The last point about device insertion is that it can occur outside the bounding box of the cell. This means that the cell must be expanded in order to include it. Upon expansion, all wires that go to the edge are stretched to maintain connections to ports. This feature also evolved out of experimental discoveries. Coupled with the relocation facility, and the small number of menu commands leafcell editing is a very quick and simple process that demands very little of the user beyond designing the topology of the circuit.

3.7.4 Wire Placement

The placement of wires is somewhat different to that of other devices, for a number of reasons. For a start, wires need two
endpoints to specify them. Secondly, wires are supposed to collide with objects, in order to make connections, so selecting the object is inappropriate. Thirdly, one would like the ability to specify a series of points to be joined together, thus about halving the number of commands required to place individual wires. Finally, since insertion can expand the bounding box of the cell, the correct end(s) of each wire must be used for checking whether the expansion is necessary. If both ends lie outside the box, including one end might include the other, in which case the cell need only to be processed once.

Because wire placement disables the object selection facility, a special menu command 'deselect object' has been included to leave the wire insertion mode. Pointing into the palette can also be used the same way. The deselection command can be of use if the selected item disappears out of the viewing window, or has been reduced in size beyond visibility.

Wire placement requires the two endpoints to be specified. At each endpoint crosshairs are shown. Because only horizontal and vertical lines are allowed, diagonal lines are replaced by two orthogonal sections. The first, and longest of these is attempted first, and if successful crosshairs drawn at the intermediate point. The second section can be placed by pointing at the same place again. Should the longest section be illegal, then the shorter section will be attempted first.

Polysilicon wires can only join transistors when they cross the gate coordinates orthogonally to the diffusion layer. Diffusion wires must be collinear with the diffusion part of a transistor.
3.7.5 Device Deletion

The only device that can be deleted is the currently selected device, if it is part of the cell, not the palette. Upon deletion of a device, the remaining devices are checked for overlap with the deleted device, and redrawn if necessary. Deletion is provided as a menu command. The array is kept continuous by moving the last element into the 'hole'. This is a very cheap operation, and no other changes are necessary. Moving a few bytes during an interactive command is preferable to handling the possibility of missing elements in the list, every time an operation is performed that requires scanning the list.

3.7.6 Parameter Modification

The parameters describing the selected object can be displayed and modified by using the change menu option. If there is a choice of parameters, then the user is prompted for a choice. The parameter to be changed is displayed, and the replacement value taken from the keyboard. If there is no value supplied, or the value is inappropriate, then the parameter is unchanged.

3.8 Module Editing

The abilities needed to be able to create and modify modules include cell instantiation, port and connection specification. There are some features included to make most of the common tasks very
quick and easy to perform. To minimise the number of menu commands, there are a number of flags and defaults used, and displayed in the palette.

3.8.1 The Module Palette

The module palette contains the following items:

(i) **ports**
(ii) **transformations**
(iii) **multiple instance options**
(iv) **connection default**

The ports are the same as the leafcell ports, in all respects.

The transformations change the *default transformation* which is used when a cell is instantiated. The three possibilities are rotation ("rot"), mirror vertically (top -> bottom) ("my"), and mirror horizontally ("mx"). The transformations are cumulative, and are stored internally as a 2x2 matrix. The matrix is decoded to display the nature of the default transformation, thus the simplest form of the transformation is displayed. To make a change to the transformation, one of the three choices is pointed at.

The third set of palette items are controls over repeated instances. The two values, "Xrep" and "Yrep" are the number of rows and columns of cells to be placed at a time. The "Abut" flag determines whether the cells will be automatically connected upon
placement. Pointing at either value will generate a prompt for a new value.

The final item controls the connection commands. It offers a choice of abutting, routing or deleting connections. Pointing at the item will cycle through the possibilities.

3.8.2 Cell Instantiation

To place a cell upon the design area it is necessary to be in "Insert mode". This signifies that the editor knows which cell to insert. The two ways of selecting a cell type are by pointing at an existing copy (already in the design) or via a menu command, which generates a prompt.

Having located the relevant data for the cell to be instantiated, the editor allows the user to point at a place in the window where the lower left hand corner of the transformed copy is to be placed. The values of "Xrep" and "Yrep" determine how many to place. The default transformation is used for the individual elements in the array, not the whole array. The effect of the transformation is cumulative: there is no need to know the transformation of an instance to be copied, only how to make the copy, relative to the original. This is a much more convenient way than absolute transformations, because the designer can only see the relative transformation that he wishes.

After transformation, the bottom left corner of the new bounding box will be coincident with the point located by the user.
Thus to transform an instance it is necessary to:

(i) premultiply the original bounding box by the transform
(ii) swap coordinates to maintain left, right, bottom, top order
(iii) translate the new bounding box back to where it belongs.
(iv) Transform the port data to suit the new instance.

The bounding box of the new instances (in total) is checked against the previously placed instances for collisions. Instances must not overlap. If an overlap occurs, the insertion is disallowed. If necessary, the bounding box of the design is expanded to fit the new components. The bounding box of each instance is calculated by translation of the first instance: only one matrix multiplication is used. Finally, the ports are manipulated so that they fall on the correct edge after transformation. The use of absolute distances (relative to the parent cell, not the instance) for ports complicates this only slightly, and reduces the computation required when scanning the port data for a 'hit' during selection.

Upon instantiation of an array of cells, each has to be given a unique name. This is performed automatically by appending a number to the cell type string. Thus when an instance is created, the existing instances of the same type are scanned for the largest numerical tag, and then this tag is incremented for each cell placed within the array. The user has control over the names of instances, but it is important to relieve them of full responsibility.

3.8.3 Connections

Having placed the internal components of the module, all that is
necessary is to connect them together properly. The JIGSAW composition system supports two types of connection - abutting and routing. An abutted connection is made by placing appropriately stretched cells next to each other, so that each port coincides. The implication is that no abutted connections may cross each other. A routed connection allows a more flexible connection schema. Routed connections are replaced by a Routecell (as mentioned in 2.4) which is the abutted. A routecell is only a leafcell with no transistors. Wires are permitted to cross internally, but each crossing increases the minimum width of the cell. Thus abutting cells takes less space between the cells, but one cell will be stretched. Routing, on the other hand, allows cells to remain unstretched, at the expense of increased space usage between them. As routing is performed by abutting anyway, abutting is computationally cheaper. The initial release of the JMRC package for evaluation does not include the routing software, however BIRGMAN allows for their existence, in anticipation of the updated routing tools.

Since most connections are between sides of cells, this is provided as a special facility. The abut edges command allows the user to specify two sides (by pointing within the instance, near to the side required) and attempts to join all the ports along the two sides. Before it proceeds, it is necessary to check the feasibility of connecting the two specified edges together. Consider the following diagram, of two instances within a module:
Fig. 3 Abuttal Strategy

Inspection of the diagram will make it obvious that the only pairs of edges that can be abutted (non-trivially) are \((A,E)\), \((G,I)\) and \((K,C)\). The checking has three checks to decide if two edges can be abutted:

(i) The two edges are different
(ii) They are not perpendicular
(iii) The box formed between the two edges does not intersect with anything.

If all the above conditions hold, the ports of each edge are scanned, and connections made between ports of the same layer (proceeding from the bottom or left end of each edge). If the layers of two opposing ports do not match then one of the ports is skipped. This way not all connections are necessary. Thus automatic abuttal will
cause the connections between ports in shown in fig. 3. (NB. the two different shades indicate different layers).

The editing context relevant to edge abuttal can be summarised by the following factors:

(i) The connections can be abutted, routed or broken depending upon the value of the Connect flag in the palette. Breaking connections requires only one edge to be specified.

(ii) Abuttal is an editing mode. To leave this mode it is necessary to select a palette item, or point twice at the same instance in the window, or use the instance menu option.

(iii) Transforming an instance causes all connections to it to be invalidated. Thus all connections can be conveniently broken by a null transformation.

The other connection mode is single connection. This follows the same pattern as edge abuttal, in terms of the editing context, with the exceptions:

(i) Connections are made between ports.

(ii) The program will wait for a port to be hit, or a non-window command to terminate the connection attempt.

(iii) Connections may be routed between any edges as long as there is nothing in between them. It may be possible in future for JIGSAW to be able to make such connections. In the meantime, it is the responsibility of the designer to be familiar with the composition system to be used and to restrain him/herself to suitable connections.

(iv) Any number of connections may be made to a port.
When deleting connections to a port (by pointing at the port in single connect & break mode) only one connection is broken per input event (i.e. mouse-click). This will not be noticeable in most cases, when there is only one connection per 'net'. It may prove useful when there are more than one. When a port or instance is deleted, all the connections to the port/instance are deleted.

Since routing requires a rectangular leafcell, it is possible to check for collisions when making connection by the same strategy used for edge abutting. The coordinates of the two ports form the corners of a rectangular region, when edges are being connected the corners are the edges of the overlapping portions of the two edges.

Making a single connection requires finding two ports. The current version of BIRGMAN allows the user to keep pointing until he hits a port. This is useful when the detail is very small on the screen. It may be advantageous to perform a limited search in the neighbourhood of a point to find a port, however this will be a relatively complicated process, more so than the equivalent technique used in leafcell editing. If program size permits after inclusion of all other features, and portability constraints, it will be attempted.

3.8.4 Other commands

There are only a few other module editing commands, in keeping with the rationalisation of menu options. They allow deletion and modification of the selected object. If the object is a cell instance, the the parameters that may be changed are orientation and name. Orientation is changed by applying the default transform to the cell.
This should be set up prior to invoking the command. After any transform (including the identity transform), all connections to the cell are deleted. The transform will be disallowed if it would cause overlap of instances.

The other command, distinct to module editing, is the uncover command. This toggles the display of the selected instance between expanded (all internal objects displayed) and covered. When the object is covered, its name will be displayed, centred (on the visible portion of the instance) if there is sufficient room. If the object is expanded, then the internal components are displayed unstretched, because this shows how much the instance was expanded if it has already been composed. Uncovering many instances is quite feasible, for any sized design due to the implementation of 'virtual data structures' that do not all need to be resident simultaneously. The small size of the data will make this unnecessary in all but extreme cases. The penalty for uncovering too many instances lies in the amount of detail that must be drawn for each instance in the window, and the reduced response time. Fortunately, for the user, instances outside the viewing area are not processed, so that drawing time is proportional to the amount of detail in the viewing window only.

In order to draw the transformed cell correctly, it is necessary to change the environment transform, display origin, window origin and zoom factor temporarily. The cell can then be drawn with a recursive call to the main display routine. The environment transform is cumulative - this is the most efficient way of making the transformations at each level - and is saved and restored each time. This is far cheaper than calculating inverses and multiplying.
4 PORTABILITY

The current implementation of BIRGMAN resides under UNIX on a PYRAMID 90Xe, using a Commodore AMIGA 1000 as a graphics terminal. The graphics terminal emulator was written as a part of the development of BIRGMAN. The advantages of using a fast machine for development are well known, and much appreciated. The current size of the program is about 64K code, and 10K static data area, with around 2-3K of additional data required for each cell resident during editing. This is code for a 32 bit RISC machine. The porting of a preliminary version from an IBM PC/AT increased the code size from \( \approx 20K \) to \( \approx 39K \), so it seems that the target size of 64K code will be sustainable on all microcomputers. Equipment availability, and time constraints on a project of this magnitude have unfortunately precluded porting the program to a range of machines. The steps taken during development to facilitate this are discussed below:

(i) All data types are parameterised in a header file, so that machine word sizes can be handled simply. The data type NUM is a 16-bit (minimum) signed quantity, all other types are not critical.

(ii) Screen positions, and sizes are defined in a separate header file. Internal functions which rely on screen sizes use the parameters defined here. The palette, which is linearly arranged on the current implementation, can be set up in any form, as long as the two location macros Palx(i) and Paly(i) locate the coordinates of the palette items. The palette is, however, is the most ad hoc part of the screen usage, and any difficulties that arise will probably be due to poor screen resolution requiring modification to the palette.
(iii) The graphics primitives are very limited and easy to implement. All higher level graphics requirements are handled by BIRGMAN. The complete graphics terminal on the AMIGA uses 16K of code and data, of which most is used for serial line handling and buffering. The implementation of any of the graphics primitives is unlikely to cause space problems. The set of graphics and text primitives required is:

**OUTPUT**

(a) `Line(x1,y1,x2,y2,colour)` - draws a single pixel width line between `(x1,y1)` and `(x2,y2)`.

(b) `Block(x1,y1,x2,y2,colour)` - draws a filled block

(c) `Cursor(x1,y1)` - places the text insertion point to `(x1,y1)`

(d) `putchar()` - to place a text character at the cursor position

(e) `Textmode ([inverse or plain ] )` - how to draw characters.

**INPUT**

(e) `get_char()` - accepts a character from the keyboard

(f) `Get_command()` - sets `curx,cury` variables when an input event (mouse-click etc.) occurs.
It should be noted that the three text output primitives: (c), (d) and (e) can be combined if a more general primitive is implemented on the host machine. This is a trivial task, and most machines use the three primitives in the form given, which reduces the amount of data transferred during 'conversations'.

The 'colour' parameter is assumed to be unspecifiable for the machine, and is defined in the machine dependencies header file. The colours will be set up in the Init() function, and thereafter unchanged. The necessary colours are White, Red, Blue, Green, Yellow, Black. There are also colours used for Deletion and Selection which could be different colours, or one of the above. Deletion will probably be simply black, or it could a special colour.

The block primitive is always used with (x1,y1) being the corner with the smallest screen coordinates, so a very simple primitive can be written for this. It is advantageous for block filling to be patterned, but this is not essential unless a monochrome screen is to be used.

Text size is also used as a parameter in the program, and is specified in pixels in the definitions. The screen size and relative sizes of the menu, window and palette portions are also needed. The first implementation, before graphics facilities became available, used coloured character graphics with some success.

The other set of routines which may need modification are the file handling. File access uses only open(), close(), read() and write() which may need to be modified. The other, more difficult, problem is in generating library content lists. Under UNIX, the directory file is
searched for files with the ".com" extension (used for "compact form" representation instead of ".sl" for SILO or ".cif" for CIF). Some microcomputer operating systems provide very convenient alternatives, which can be used. The high level routine Get_name() is used for file selection, and this could use a supplied routine, or any form particularly suited to the application, as long as it returns the text name of the cell required. In some cases it may be necessary to maintain directories ourselves.

5 CONCLUSION

A new editor for nMOS V.L.S.I. design has been successfully implemented in the BIRGMAN program. BIRGMAN is a graphical editor for symbolic design specification, with the advantage that it displays true size relationships between objects. This increases the complexity of internal representation and display of the design, but it also makes it more difficult to align objects if they are not 'clipped' to a grid. This is overcome by using the grid concept implicitly, and performing local searches for an appropriate 'grid' to use. Such a strategy also allows a good deal of knowledge about nMOS design to be used to perform the action the user really requires (A "Do What I Mean" approach). A further requirement, in direct opposition to the increased complexity inherent in a geometric/symbolic hybrid editor, is the production of an editor suitable for teaching, and use where resources are limited. This demanded a program portable across a range of inexpensive microcomputer workstations. The program thus had to be of small size, have minimal requirements of the host
hardware or operating system, and be intelligent about the use of slow file storage devices. The target size of 64K code and 64K data segments has been met with a good deal of leeway, largely due to building all functions used from the very lowest level, to avoid the overhead of generalised library functions. In addition, it proved possible to implement disk caching and data overlaying to compensate for either slow disks or small memory, whilst allowing full use of available resources.

A small note about the performance of the system, from the sublime to the ridiculous, is appropriate here. A simple leafcell (an inverter) takes less than a minute or so to design and save, because each item is relocated automatically to join correctly. On the other hand, an array of 3000 D-type latches (making a chip of $≈19,000,000 \lambda^2$) was created without any difficulties, except the time to draw the sheer quantity of graphical detail when the whole cell was displayed. This is a chip several times larger than supportable by MPC technology. The BIRGMAN program is thus a robust and useable system. The two format converters (to and from SILO) allow all the supplied libraries to be converted (including CIFcells) and edited, and the SILO output after editing one of these library cells, and/or any number of conversion cycles, is equivalent to the original SILO file.
REFERENCES:

"Basic VLSI Design - Principles and Applications"
D.Pucknell & K.Eshraghian
Prentice-Hall (Australia) (1985)

"The C Programming Language"
B.Kernighan & D.Ritchie
Prentice-Hall (New Jersey) (1978)

The JMRC Documentation - especially :
"JMRC-UNSW VLSI Kit"
ingred (cad1),
jigsaw (cad1)
SILO (cad5)

"The KIC tutorial"
Ken Keller & Giles Billingsley
University of California at Berkeley
APPENDIX D : Graphics Terminal Protocol

Needing to write my own graphics terminal emulator for the AMIGA, and not intending to create a full-blown implementation of any particular terminal, I had a free hand in choice of communications protocol.

The protocol I used had a very simple objective, to minimise traffic for large numbers of simple commands. Each of the graphics commands were ESCape sequences, with base-26 numbers. Base 26 was chosen because the resolution of the AMIGA is 640 by 400, and $26 \times 26 = 676$. Thus only two characters were needed for an ordinate. It is easier to use a fixed two characters for a number than a variable 1-3 characters + a terminator. Possibly base-32 would be better (computationally cheaper certainly), but a readable format made it very easy to debug, and I can remember my alphabet over 60% of the time. UNIX makes it very difficult to support hardware handshaking, so the terminal is prompted every 10 graphics commands to respond with a single character, to allow it to catch up. Any other characters are ignored. This has the effect that any commands input while a design is being drawn will be ignored, saving either the complexity of working out their cumulative effect, or a string of 'experimental' commands being executed after the design has drawn. most designs are drawn very quickly, and this is not a problem.