IDIOMATIC INTEGRATED CIRCUIT DESIGN

Neil Bergmann

EX BIBL. UNIV. EDINBURGEN.

Ph.D.

University of Edinburgh

1984

ABSTRACT

An examination is made of the capture, storage and instantiation of well-known, generalised structures used in the design of MOS integrated circuits. These structures are called "idioms".

The capture of an idiom for translating from a high level language specification to a complete digital signal processing system, called the FIRST silicon compiler, is examined.

A system is presented which allows and encourages the capture of a large number of idioms at the cell level of IC design. This system is based on a purely textual language, VIRGIL, which captures circuits and idioms at the sticks level, in terms of a set of structural components laid out on a so-called virtual grid. The language supports parameterisation, selection and repetition as textual operations, and also allows idioms to be composed from a set of leaf cells which are joined by simple abutment.

An algorithm is presented for the conversion of virtual grid circuits into mask level representations, and in so doing the notion of a quasi-virtual grid is introduced.

A new style of CMOS design, called "generalised CMOS", is introduced, which allows the design of circuits which could be fabricated in a wide range of CMOS technologies. An idea for a method of converting existing mask level circuits into other technologies, called "sticks extraction", is presented.

A prototype implementation of a system to support the capture of idioms using the VIRGIL language, and their subsequent instantiation including conversion to mask geometry is discussed, and examples of idioms which have been captured by this system are presented.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, David Rees, for his guidance, support and encouragement throughout the period of my study for this thesis. Thanks must also go to the many people within the University of Edinburgh who assisted me during this time, most especially John Gray, Irene Buchanan, Gordon Hughes, Hamish Dewar, Peter Denyer and David Renshaw.

I am grateful to the University of Edinburgh for providing the facilities to conduct this research, and to the Association of Commonwealth Universities for their fianancial support under the Commonwealth Scholarship scheme.

DECLARATION

I hereby declare that this thesis has been composed by myself and that the work described in this thesis is my own.

[redacted] 24.5.84

CONTENTS

1: Introduction....................................9

1.1: The Need for Research into CAD for VLSI.........9

1.2: CAD in VLSI Design Capture......................10

1.3: Current CAD Tools for Design Capture...........13

1.4: The Notion of Design "Idioms"..................19

1.5: The Scope of this Thesis.......................21

2: The Current State of the Art.....................25

2.1: Structured VLSI Design.........................25

2.2: VLSI Design Languages and Programming Languages..........................27

2.3: Design Languages and Verification..............28

2.4: VLSI Design Languages & the Control of Complexity.......................31

2.5: Composition of VLSI Designs....................33

2.6: Methods of Circuit Description.................37

2.7: Sticks Compaction..............................41

2.8: Technology Independent Design..................43

2.9: Cell Libraries.................................45

2.10: Silicon Compilers.............................46

3: The F.I.R.S.T. Silicon Compiler..................48

3.1: Background to the FIRST Silicon Compiler.......48

3.2: An Innovative Architecture.....................49

3.3: Software Support for an Integrated Design Environment.................52

3.3.1: Language Compiler..........................52

3.3.2: Simulation..........................58

3.3.3: Physical Layout..........................61

3.4: An Example System..........................65

3.5: Conclusions..........................70

4: An Idiom Description Language..........................72

4.1 The Virtual Grid..........................72

4.2 Idioms and Instantiations..........................77

4.3: VG - A Virtual Grid Circuit
Description Language..........................78

4.3.1 Notes About the Language Description........80

4.3.2 Alphabet..........................81

4.3.3 Identifiers..........................81

4.3.4 Reserved Words..........................82

4.3.5: Numerals and Expressions..........................82

4.3.6: Statements..........................84

4.4: VG Leaf Cells..........................84

4.4.1 Leaf Cell Header and Terminator..........................84

4.4.2: Leaf Cell Component Specification..........................85

4.4.3 An Example Leaf Cell Description..........................96

4.5: VG Composition Cells..........................96

4.5.1: Header and Terminator Statements..........................99

4.5.2: Basic Composition Operators..........................99

4.5.3: Rotation and Reflection..........................105

4.5.4: Port Hiding & Renaming..........................108

4.6: VIRGIL - An Idiom Description Language........113

4.6.1: Parameterisation..........................114

4.6.2: Selection....................................117

4.6.3 Repetition...................................118

4.6.4: Qualifiers..................................120

4.7: Circuit Connections and their Validity........121

4.8: An Example Idiom..............................133

4.9: Summary.......................................143

5: Sticks Compaction of the Virtual Grid...........145

5.1: Background....................................145

5.2 Mask Level Representation on a

Quasi-Virtual Grid.................................148

5.3: Translation to a Quasi-Virtual Grid...........155

5.4: Device Templates..............................159

5.5 Coordinate Mappings............................167

5.6: Translation of CMOS Circuits to Mask Level....173

5.7: Sticks Extraction.............................183

5.8: Summary.......................................196

6: A Prototype Implementation of

the VIRGIL System..................................198

6.1: System Overview...............................198

6.2: A Cell Librarian..............................198

6.3: Text Analysis.................................200

6.4: Internal Representation of a Virtual Grid.....201

6.5: Analysing Leaf Cell Definitions...............202

6.6: Analysing Composition Cell Definitions........203

6.7: Outputting the Virtual Grid...................204

6.8: Conversion from Virtual Grid to CIF 2.0.......204

6.9: User Interface....................................205

6.10: Some Implementation Issues..................207

7: Example Idioms.......................................210

7.1: Shift Register......................................210

7.2: Programmable Logic Array......................210

7.2.1 Introduction......................................210

7.2.2: Capture at the Mask Level..................213

7.2.3: Capture Using VIRGIL.........................216

7.2.4: Comparison and Discussion.................221

7.3: Parallel Multiplier................................223

7.3.1 A Hardware Algorithm..........................223

7.3.2: A Multiplier Circuit.............................229

7.3.3: A Multiplier Chip................................232

7.3.4: Discussion...234

8: Conclusions and Limitations.......................235

8.1: Conclusions...235

8.2: Limitations & Areas for Further Research......238

Appendix A: VIRGIL "<u>include</u>" File......................241

Appendix B: PLA Mask Level Leaf Cells................251

Appendix C: PLA VIRGIL Cell Definitions..............257

Appendix D: Mutiplier Cell Definitions..................276

References..283

1: INTRODUCTION

1.1: The Need for Research into CAD for VLSI

A revolution is now underway which is comparable in scale and importance to the industrial revolution. This new revolution is based on the emerging field of Information Technology. One of the cornerstones of Information Technology is the provision of cheap, readily accessible computing power. It is only through the extensive use of Very Large Scale Integrated (VLSI) circuits that this power can be realised. The design of integrated circuits is then an area of immense interest, and will remain so for the forseeable future.

VLSI circuits are among the most complex systems designed by man. Mead [Mead 81] compares the design of an integrated circuit using the technology available by the end of this decade with the task of designing an urban density road network the size of an entire continent. Even with current leading edge technologies, managing the mere complexity of a VLSI circuit is the most pressing problem facing design engineers. Research into methods of handling this complexity is one of the most important tasks facing the academic and industrial communities. The provision of sophisticated Computer Aided Design (CAD) systems does and will continue to play a major role in the management of VLSI designs.

1.2: CAD in VLSI Design Capture

There are several rather distinct steps which can be identified in the design of VLSI circuits. These steps cannot be treated in complete isolation, rather each step should be considered in relation to the others.

Firstly, there is the specification of the intended function of the system to be designed. Next is the translation of this specification into a description of a physical realisation of that system. Next is the verification that the intended realisation performs the specified function. Finally there is device fabrication. The area of interest in this thesis is the translation of some specification into a description of a physical realisation, here called the design capture phase.

An integrated circuit is fabricated using a set of masks to define the patterns which will appear on the different physical layers of the circuit. The masks are in turn fabricated from some machine readable description of the geometric patterns which comprise them. Within the research and academic communities, a standard language for describing such patterns, CIF 2.0 [Mead 80], has now been established. The design capture phase may then be considered as the production of a CIF description of an IC from some initial specification.

The production of a set of mask descriptions is not a straightforward task, but rather it is a complicated and error prone process of multiple translations, each into a finer level of detail. A typical set of such translations might be as follows.

(1) Translation of a general product specification, which may be a rather imprecise natural language description, into a formal behavioural specification, such as a set of input / output relations.

(2) Design of an algorithm to implement the behavioural specification.

(3) Choice of a hardware architecture to support the algorithm, and thence production of a system block diagram.

(4) Mapping of the block diagram into a two-dimensional floorplan of a chip.

(5) Production of logic diagrams to implement each hardware block.

(6) Translation of logic circuits into a topological arrangement of transistors and their interconnections.

(7) Translation of this topological arrangement into final mask geometries.

Although such a description of the design process is perhaps oversimplified, it does serve to illustrate the number of different levels at which descriptions of a design exist, and the number of translations, each into a finer level of detail, which must be done to produce the final mask geometries. Such translations can be performed manually, or with the aid of CAD tools. Since the final design description exists as a machine readable CIF description, the designer's intent must at some stage be captured by a suitable CAD system. Once captured, the CAD system may then perform one or more of the above translations to yield the final mask level description.

Ideally, it might seem that this design capture should occur at as high a level as possible, so as to reduce the number of error-prone translations which must be performed manually. However, none of the translations are by any means straightforward. While it is probably true that CAD tools exist to perform all of the translations in some form or other, none of these tools are yet able to match the flexibility and ultimate area efficiency of good manual design. At present, the major advantages of automatic translations are that they are far quicker and less likely to contain design errors, so reducing both the length and number of design iterations necessary. They are therefore most used in low and medium volume applications, where design cost is the major portion of overall device cost.

The tradeoffs which exist between automated and manual design are sufficiently varied for different applications that CAD tools which capture designs at many different levels are still used. As long as CAD tools do not perform as well as manual designers, such tradeoffs will continue to exist. One motivation for this thesis is to examine aspects of manual design and incorporate these into automated design systems, to make the latter more competitive.

1.3: Current CAD Tools for Design Capture

A great variety of CAD tools for design capture already exist, and it is useful to examine some of these tools to place the work of this thesis in its correct perspective.

The most basic method of converting VLSI designs into machine readable form is by digitizing hard copy representations of mask geometries. Such a process is very error-prone, and is not viable for designs of VLSI complexity.

A more direct method of capturing designs at the mask level is by the use of a graphics workstation. Here the designer is able to create and manipulate shapes representing the actual mask geometries using some form of graphics editor. Such workstations allow the definition of individual cells, and also the composition of these cells to produce complete chips.

Design capture by graphics workstations is now quite a mature field, with many commercial manufacturers offering such systems for sale [Werner 83]. Design at the mask level gives the designer maximum flexibility to decide on the final circuit geometries. However, the designs produced at this level are not constrained to represent valid circuit constructs, and the detection and correction of errors in the mask definitions is a time-consuming task. Often such errors are not detected until prototype devices are fabricated.

Design directly at the mask level also suffers from another major disadvantage. The individual cells which make up a design are described absolutely in terms of their mask geometries. Such cells are considered to be "hard", in the sense that their structure is fixed. A change in one cell will often require changes to be made in many adjoining cells.

A solution to this problem is to design cells to be "soft", i.e. to be able to adapt automatically to their surroundings. The simplest method of achieving this is to describe cells, and indeed the whole chip, not as a set of mask geometries, but rather as a computer program which produces those geometries. If cells are carefully described in terms of their relationships with neighbouring cells, then when a change is made to one cell, and the program run again to produce a new design, all dependent cells will also have changed to match. Design by program also allows inherently programmable structures, such as PLA's to be easily described.

This type of IC design language can be produced by adding a set of routines to draw mask shapes onto an existing programming language. Such a language is called an embedded IC design language. This approach allows the full data and control structures of the original language to be used. Useful features can be added incrementally to the library of available routines to continually enrich and improve the language. An excellent example of such an approach is the language ILAP [Hughes 83], developed at Edinburgh University, and embedded in the programming language IMP [Robertson 83].

The amount of parameterisation which can be introduced into a design description using an embedded language is virtually unlimited, even to the extent of producing a silicon compiler [Bergmann 83]. However, the design is still being specified in terms of the final mask patterns, and the responsibility still rests with the designer to ensure that these patterns represent valid circuits.

An alternative to embedded IC design languages is the use of a special purpose design language. In this case special syntactic structures are used to specify mask features. Such descriptions tend to be clearer and more concise than embedded language descriptions, however special purpose languages are usually not as rich in control and data structures. In addition, they require special compilers to be written for them. Examples of such languages are SILT [Davis 82], and SCALE [Marshall 84].

Some special purpose languages attempt to constrain the designer in the mask descriptions which can be described, in an attempt to reduce the possibility of design errors. Such languages are also able to provide special syntactic structures to aid in the composition of designs. SCALE [Marshall 84] is an example of such a language.

Many of the problems of mask level design can be attributed to the fact that a circuit is not constructed from arbitrary mask shapes, but rather from a set of items with real circuit significance, notably transistors, wires, contacts and bonding pads. It is therefore sensible to allow a designer to design in terms of these structural components [Buchanan 80], rather than the mask shapes which comprise them. Also it is usually not the absolute position of such items which is important, but rather their topological arrangement with respect to each other. Design at the level of a topological arrangement of structural components is popularly referred to as "sticks" design [Williams 77], or alternatively as symbolic design.

Since sticks designs are concerned primarily with the relative placement of components, and not their absolute positions, sticks cells are inherently "soft" cells. Thus the speed and convenience of graphical entry can be used without the disadvantages of "hard" cell descriptions. An example of a primarily graphics based sticks system is MULGA [Weste 81b]. Sticks descriptions can also be language based. ABCD [Rosenberg 82] is an example of a sticks language.

Since sticks descriptions do not specify the exact positions of structural items, the design is no longer tied to a single fabrication process, with its accompanying design rules. Rather, the conversion from sticks to mask geometries can incorporate these design rules, and so allow a single description to be fabricated using several different technologies.

At levels of abstraction above sticks level, the designer is removed from many of the topological details of the final design. The design system itself determines these details. This can be done in two ways - either the system restricts the designs to a limited class of topological arrangements which are built into the system, or the system deduces a suitable arrangement from scratch, using sophisticated algorithms to do so. The fundamental principles by which such arbitrary arrangements should be determined are not well understood, and in general systems to produce arbitrary layouts do not perform particularly well.

Design at the logic level, in terms of boolean logic function primitives, is at present mostly restricted to standard cell systems. Here the designer is presented with a library of cells to perform various logic functions, which can then be connected together to produce complete chips. DUMBO [Wolf 83a], is an example of a system which can lay out arbitrary logic functions.

At the highest level are systems which take behavioural descriptions of complete systems, and produce complete chips which implement the specified behaviour. At present, all such systems - called silicon compilers - produce chips only within a limited architectural framework. Often such systems are also restricted in the class of behavioural descriptions they can translate. Examples of such systems are Bristle Blocks [Johannsen 79], FIRST [Bergmann 83], Model [Gray 82], MacPitts [Siskind 82] and UNIT [Deas 83].

1.4: The Notion of Design "Idioms"

It has been shown that design capture can occur at any one of a large number of levels of abstraction. However all design capture mechanisms share a common feature - capture is achieved by the use of a fixed set of design primitives. These primitives vary from complete functional units in the case of a silicon compiler down to simple geometric shapes in the case of mask level design. Nonetheless, the design process remains one of composing elements from a set of available primitives together to give a system which exhibits the required behaviour.

One of the principle arguments in this thesis is that the composition of primitives to give solutions to specific problems relies heavily on the use of known constructs. These known constructs are called idioms, and the aspect of design concerned with their use is called idiomatic design. Idioms are in some sense the embodiment of a designer's experience.

Examples of idioms abound in IC design. At the mask level, structural primitives such as transistors and contacts could be considered as idioms constructed by overlaying particular shapes on particular layers. These idioms are so important that almost all mask level design systems include them in their set of design primitives.

It has already been explained that IC design involves translations into finer and finer levels of detail. Idioms can be seen as known ways of making these translations. The Gate Array can be considered an idiom which maps a structural description in terms of interconnected transistors, onto a regular form of mask geometry. The PLA is an example of an idiom which maps from a set of boolean equations into mask geometries.

Idioms are equally valid whether the translations they represent are done manually or by CAD tools. In both cases, the larger the repertoire of idioms, the better the final design is likely to be. The aim of this thesis is then not just to investigate means of capturing designs but rather to investigate means of capturing idioms, so that these are available for use in many different designs.

1.5: The Scope of this Thesis

As described earlier, idioms exist at many different levels in the IC design process. For the purposes of this thesis, two areas of particular interest have been chosen.

The first of these is the investigation of idioms for the automatic production of complete integrated circuits - in other words, silicon compilers.

The second area of particular interest is in that of so-called "cell design". At this level, the designer produces the mask level layouts to implement individual logic functions. The division of a complete system into such cells is the final step in a top-down decomposition of a specification, and the design of these cells represents the first stage in a bottom-up implementation of that specification.

The study of idiomatic design at both these levels draws much from the ideas and principles developed in related fields of VLSI research. In chapter 2, the current state of research in these related areas will be examined, and some of the ways in which idiomatic design furthers many of these ideas will be noted.

Idioms at the two levels of particular interest in this thesis - cell idioms and silicon compiler idioms - differ not only in the level of the design hierarchy at which they exist, but also in the manner in which such idioms may be captured.

A silicon compiler represents the capture of a single, quite complex idiom. In chapter 3, a suite of software, written by the author, to capture a single idiom for the production of signal processing chips will be examined. It will be shown that by the production of an integrated software environment to specifically support this one idiom, a quick and economical method for designing one class of systems can be developed.

At the leaf cell level, it is possible not just to capture single idioms, but to design a system for the capture and instantiation of a large number of useful idioms by the production of an appropriate idiomatic design system. The design and investigation of such a system forms the basis for much of the remainder of this thesis. Such an investigation is valuable for several reasons.

Firstly, this area of design already makes heavy use of idioms based on designer experience. There is therefore a need for a system which can formalise the collection and retrieval of such idioms.

Secondly, the structure of idioms at this level is relatively straightforward. The idioms are in some sense recipes for producing mask level circuits to implement particular functions. Methods of describing mask level descriptions are well understood. Indeed, the design of mask level descriptions of IC cells is one of the most intensively investigated areas of IC design. This has the twin advantages that the investigation of idiom capture at this level can benefit from principles used in more traditional design systems, and also that any new ideas investigated for use in idiomatic design are likely to have application in the wider field of custom cell design.

Because of this possibility of the wider applicability of idiom capture techniques, it has been decided that, where possible, new and novel methods of design description will be investigated. In this way the work of this thesis can contribute not only to the novel field of idiomatic design, but can also explore ideas with applicability to more traditional design styles.

The most fundamental decision to be made in the production of the idiomatic design system is the way in which idioms are to be described. It is argued that design capture at the "sticks" level (i.e. joint structural and topological level) has overwhelming advantages over other levels of design description, such as mask level. Furthermore, it will be shown that a so-called "virtual grid" [Weste 81a] provides an attractive and elegant method of sticks level circuit description. To these ends, a novel sticks level circuit description language has been devised, and this is described in detail in chapter 4.

Since the ultimate output from the idiomatic design system is mask level descriptions, a translator from sticks to mask level is needed. Such "sticks compactors" are presently a topic of keen research interest. A sticks compactor which combines some well established ideas about compaction with some novel and original ideas is presented in chapter 5.

A prototype implementation of an idiomatic design system has been developed, and this is described in chapter 6, along with a discussion of some of the software engineering issues raised in the production of such a suite of software.

Several specific examples of cell idioms which have been entered into the idiomatic design system are described in chapter 7. Some conclusions about the work of the thesis, plus some possible areas of further research are presented in chapter 8.

2: THE CURRENT STATE OF THE ART

The work of this thesis draws from many fields of VLSI design, most of which remain areas of active research interest. In this chapter, it is intended to review the current state of research in these areas, and where possible relate them to the work of this thesis.

2.1: Structured VLSI Design

VLSI circuits differ from other methods of digital system implementation in several important ways.

The most fundamental feature of VLSI circuits is their overwhelming complexity [Mead 81]. Circuits containing over 200,000 transistors are now being designed and fabricated, and the number of devices which can be accommodated in a single integrated circuit is expected to continue to increase exponentially for the foreseeable future [Noyce 77]. The prospect of designing circuits of such complexity is even more daunting when it is remembered that even one single design error may render an entire circuit useless.

Another fundamental feature of VLSI circuits is that interconnections often have an equal or greater influence on circuit size and performance than do functional elements such as transistors [Sutherland 77]. There is an inherent wiring "cost" associated with communication within an integrated circuit which must be taken account of during design. It is therefore important to plan the way individual sections of the circuit will be composed together early in the design process. This includes both the relative positions of the various parts of a circuit ("floorplanning") and the way in which connections will be made between these parts.

About 1970, the need for a more methodical approach to writing computer software became apparent as computer programs became increasingly complex. This need resulted in the development of "structured programming" techniques [Wirth 71b] [Dijkstra 72] [Knuth 74] [NATO 76] [Alagic 78] [Yourdan 78], which subsequently lead to the development of languages designed to support these techniques, such as Pascal [Wirth 71a] [Jensen 74], Modula [Wirth 77a] and ADA [DoD 78] [ADA 79].

The rapidly increasing complexity of VLSI circuits has resulted in the development of a similar, structured approach to VLSI design. The publication, in 1980, of a book by Mead and Conway [Mead 80], which presented a simple, structured approach to VLSI design, was an important force in bringing about the widespread acceptance of this new methodology.

2.2: VLSI Design Languages and Programming Languages

The structured design approach to VLSI is often thought of as a "computer science" approach, since it borrows heavily from many ideas inherent in structured programming. Just as structured programming lead to the development of new programming languages, so structured VLSI design has lead to a great deal of research into new VLSI design languages.

However, although new programming languages were quickly developed and widely accepted as a result of the development of structured programming, the development of VLSI design languages to support structured VLSI design are still mostly in the research stage, and no single language has yet to gain any wide acceptance outside the institutions where it was developed.

This can largely be attributed to some fundamental differences between the nature of computer programs and VLSI circuits. The most important of these is that in computers, memory is arranged as a one dimensional set of locations, all of which are more or less equally accessible. This corresponds well to the one dimensional nature of computer program text.

On the other hand, an IC is a two dimensional structure, where only physically adjacent points are accessible from any position in the circuit. The placement of the elements in an integrated circuit can have a major effect on circuit size and performance, since interconnecting elements must be physically joined by a wire which consumes space and has an inherent propagation delay.

The best way to represent the two dimensional aspects of a VLSI circuit in an essentially one dimensional textual language is by no means certain, and some of the diversity of VLSI design languages can be attributed to this uncertainty. It is argued that the development of VLSI design languages is fundamentally a more difficult problem than the development of programming languages, and in the following sections some of the features which affect the design of such languages are discussed.

2.3: Design Languages and Verification

Since the fabrication of a silicon chip is a lengthy and expensive process, and since chips cannot, in general, be modified after fabrication, it is essential that every effort be made to ensure that a design is correct before fabrication begins. This is called verification, and the ease with which design verification can be done is dependent on the design language used.

At the highest level, design languages for use with silicon compilers provide a functional description of a circuit. This functional specification can be simulated directly to determine the performance of the complete system at a functional level. Examples of such languages are the input language for FIRST, described in chapter 3, and also the language MODEL [Gray 82], which has its own special simulator called EXERT.

Design languages at the mask level have traditionally tended to describe only the physical geometry of circuits. In order to simulate a circuit it is necessary to know the structure of a circuit. This can be done by either extracting the electrical circuit from the circuit layout using computationally expensive CAD software, or alternatively, producing a separate structural description of the circuit by hand. The latter alternative is particularly unattractive since there can be undetected inconsistencies between the two circuit descriptions.

Buchanan [Buchanan 80] developed a language, ICSYS, which allows joint structural and physical descriptions of a circuit. A consistent design representation is thus ensured without the need for an expensive circuit extraction from the physical geometry. The idiom description language VIRGIL, which has been developed as part of this thesis and which is described in chapter 4, includes both structural and physical information by describing circuits in terms of structural primitives which are implicitly interconnected according to their relative positions.

At present, the only method readily available to show that a circuit meets some behavioural specification is by simulation. Using a joint structural and physical design description can help to ensure that the simulation adequately reflects the performance of the physical circuit, however simulation can only ever show that a circuit performs correctly for those combinations of input values which have been exercised.

If both the circuit and the desired behaviour can be described using some formal calculus then it may be possible to formally prove that a circuit meets its specification. Milne [Milne 83a] has designed such a language, called CIRCAL, and Gordon [Gordon 81] has investigated similar ideas.

Furthermore, if a silicon compiler could be designed which accepted the behavioural specification of a circuit as its input language, then if the same formal proof methods were used to prove that the transformation (between specification and circuit) that is implemented by the silicon compiler is correct, then <u>all</u> circuits designed using the compiler could be guaranteed to meet their specification [Milne 83b]. Such a breakthrough would eliminate the need for circuit simulation.

This formal approach is similar to that of formally proving computer programs correct, and the difficulties which have been encountered in the widespread application of program proving techniques indicates that the widespread application of similar techniques to VLSI circuit design may still be a long way in the future.

2.4: VLSI Design Languages & the Control of Complexity

One principle of structured VLSI design is to introduce sufficient hierarchy into a design that the amount of information which must be handled at any one time is not too great. It is obviously easier to adopt such a hierarchical design style if this methodology is specifically supported and encouraged by the design language being used.

Even simple geometric design languages such as CIF 2.0 [Mead 80] are capable of describing a design hierarchically by the definition of symbols, which may in turn be composed from previously defined symbols. Such symbols are very much like procedure calls in a high level programming language, except that no parameterisation of symbols is allowed.

More sophisticated languages such as SCALE [Buchanan 82] not only support a design hierarchy, but also allow the parameterisable definition of cells. Languages which can describe designs parameterisably can describe not only circuits but also idioms. The language VIRGIL is specifically for the description of idioms and as such has special support for both a hierarchical design style and parameterisation.

It is argued that textual design languages, such as VIRGIL, provide a powerful tool, not only for controlling design complexity, but also for providing additional design information which can help ensure the correct composition of designs.

Graphical design systems usually only support the simple repetition of fixed cells as an aid to building large regular structures. Textual languages, since they can easily accommodate selection, repetition and parameterisation, can describe far more general structures. Designing structures sufficiently generally that they can be used many times in one or more designs can be a valuable aid in reducing design complexity, and thus design time and design cost.

Graphical systems may facilitate the rapid entry of the physical aspects of a circuit design, but it has already been mentioned that it is also desirable to capture structural design information, rather than to attempt to infer structural details from the physical design.

Specifically, one often wants to name certain items in a design, such as ports for connection to other parts of the circuit, so that during design composition only like named ports are connected together. One might also like to include other useful information, such as whether outputs from one part of the circuit are restored or non-restored logic levels, and similarly whether inputs are designed to accept restored or non-restored levels. Textual languages allow the simple and natural specification and manipulation of such structural design information as well as the physical design details.

2.5: Composition of VLSI Designs

An essential feature in controlling design complexity is the use of hierarchy [Rowson 80]. Like SCALE, VIRGIL separates the design hierarchy into cells which contain only structural primitives, called leaf cells, and cells which contain only instances of other cells, called composition cells. This distinction helps to highlight what, in the author's opinion, is a fundamental difference between the tasks of designing leaf cells and of composing them together to give a complete circuit.

This composition of cells to give larger cells is a fundamental and frequently performed operation in a hierarchically designed system. Researchers with close contacts with industry already report that the composition phase of design now takes far longer than leaf cell design [Smith 83]. As circuits become more and more complex, so the proportion of time spent composing cells together will tend to become increasingly greater. The design process is again greatly eased if the design language being used specifically supports composition as a fundamental operation.

As mentioned earlier, there are two main factors which influence a composition operation - the relative position of cells and the interconnections between them. In some cases, it is possible that the definition of one of these also defines the other. In CIF, for example, composition is achieved by placing instances of cells at specified physical positions, relative to a common origin. If, in placing the cells, geometry from one cell touches or overlaps geometry from another cell then connection is made between the cells.

In the case of CIF, there is no guarantee that placing cells so that they touch or overlap produces correct connections between them. Rather, the resulting layout must be examined either manually or automatically to determine if connections have been made correctly. Often incorrect connections can go undetected until circuits have been fabricated.

A better approach is to ensure that connections are only made as intended by the designer by explicitly checking connections as cells are being composed. To do this, the points where connections may be made to a cell must be identified. These are variously called "ports" or "pins". Such information is specifically included in languages, such as VIRGIL, which support joint structural and physical design descriptions.

Next, the interconnections between ports of various cells must be specified. One method of doing this is by explicitly naming pairs of ports which are to be joined - this approach is used in SCALE, for example.

An alternative approach is to implicitly connect all pairs of ports along edges of cells which are placed next to each other (i.e. "abutted"). This is the approach used in VIRGIL. To avoid the problems of incorrect connections, pairs of ports are checked before connection to ensure that they match both in type (such as polysilicon, metal or diffusion) and in name. If ports do not match, then the composition of the cells containing them fails. The only concession made to this rule is that cells may be "stretched" so that matching ports align.

Other languages support more complex operations to assist in correctly joining ports together. In the proposed language SILVER [Rees 83], such operations are referred to as coercions. Examples of coercions would be the automatic inclusion of contacts between ports on different layers, and also the automatic insertion of routing networks between cells where ports cannot be joined by simply abutting the cells.

Other researchers [Lengauer 84] have also developed systems which apply such coercions to automatically aid the completion of cell composition operations.

Such sophisticated coercions are not included in VIRGIL, since it is argued that as VIRGIL is being used primarily to describe well known idioms, the nature of interconnections are already known and can be described precisely in the circuit description.

The only composition operation in VIRGIL is the abutting of cells together to produce larger cells. Relative placement of cells is achieved by specifying the way the cells are to be abutted. This contrasts with some other languages, such as ABCD [Rosenberg 82] where placement is done by instancing cells at specified positions. Part of the work of this thesis is to evaluate a system where all composition is in terms of simple abutment.

2.6: Methods of Circuit Description

Circuit description methods can be categorised both by the manner in which circuits are described (graphically or textually), and by the level at which they are described (mask level, sticks level, gate level etc.).

Some discussion of the various types of circuit description methods has already been presented in chapter 1. Tools for design at the mask level are now reasonably mature and well understood. However, design styles at higher levels of abstraction are still an area of active research interest.

The major area of interest in this thesis is the capture of idioms at the cell level. It is desired, not only to capture the functional aspects of these idioms, but also the topological aspects of their layout. For this reason, design styles which do not allow topology to be captured, such as gate level descriptions, are not considered further here.

At mask level, the designer is free to create an arbitrary collection of shapes on the various layers. The object of most recently developed design styles is to somehow constrain this freedom so that the designer is less likely to introduce design errors, and also that the amount of information needed to specify a design is reduced.

An early excursion into this field was the development of coarse-grid layout systems [Gibson 76] [Clary 80]. Here the designer could choose from a selection of rectangular "tiles" containing fragments of geometry, and by arranging these in a rectangular array, a complete design could be built up. Tiles would typically be about the size of the worst case tolerances in a given technology - say two or three lambda, in Mead-Conway terms. A major advantage of such systems is that tiles can be represented by different ASCII characters, allowing graphical design on normal alphanumeric terminals. Coarse grid systems, however, do not really provide any structural information about a circuit - they merely ease the task of describing its physical geometry.

A similar, alphanumeric based approach is the Bell Laboratories "gate matrix" design style [Lopez 80]. This imposes a structured design style on the user, which also allows the mapping of the alphanumeric grid of the circuit description onto a reasonably dense, variable pitch physical grid. Layouts exhibiting "hand-packed" densities are claimed, but this is most probably due to the structured design style rather than the design system which implements it.

Languages such as SCALE [Buchanan 82] and ICSYS [Buchanan 80] allow the specification of joint structural and physical design descriptions at the mask level.

In a sticks based system, the designer is removed from the physical details about widths and separations, and designs in terms of structural primitives such as transistors, wires and contacts. Sticks based systems are thus not only design rule free but they also implicitly provide a joint structural and physical design description.

There are two methods by which sticks level design can be described. One method is by so called gridless design systems [Williams 78] [Dunlop 80] [Lengauer 84]. Here the designer specifies the relative positions of components, and their interconnections. This information constitutes a set of "constraints" which any mask level equivalent of the sticks description must meet. Within these constraints, components may be moved around so as to achieve a compact layout.

The other method is to arrange components using a so-called "virtual grid" [Weste 81a] [Rosenberg 82]. Structural components are laid out on a two dimensional grid, but unlike coarse grid systems, the grid lines have no implicit correspondence to physical positions. Rather, it is only the relative positions of components which are implied by their positions on the grid which is strictly important. Thus, relative positions can be easily expressed without the need to reduce a circuit to a set of positional inequalities which must be both determinate and consistent, as is the case in most gridless sticks systems.

It is argued that the virtual grid provides a very neat and elegant method of representing sticks level designs which captures joint structural and physical design information. For this reason a virtual grid based sticks level design language has been chosen for the description of idioms.

Some other sticks based systems are either extensions of existing languages such as Pascal [Lengauer 84] or Lisp [Pettengill 83], or else they can be accessed via a procedural interface from such a language [Weste 81b]. Thus, such languages are capable of describing idioms as well as circuits. The VIRGIL language is not an extension of any other language, rather it is a special purpose language for the description of idioms. It is felt that the use of a special purpose language allows idioms to be described more naturally and more concisely than is possible with embedded languages. The manner in which parameterisable features have been added to VIRGIL is novel, and allows very general selection and repetition operations to be performed.

2.7: Sticks Compaction

Once a sticks description has been entered into a design system, it must at some stage be converted into mask geometries - a process called "sticks compaction".

Ideally, one would like the final mask arrangement to be optimally compact, but the determination of such a layout has been shown to be NP-complete with respect to the number of components being arranged [Schlag 83]. In most cases this is computationally too expensive, and so sub-optimal compactors are usually used.

The simplest method of compaction is so called 1-dimensional compaction, used in several design systems [Williams 78], [Dunlop 80]. In this method, components are placed as closely together as possible in the x-direction, and then as closely together as possible in the y-direction.

For gridless sticks systems, such compactors have been shown to exhibit complexity O(N**1.5), where N is the number of circuit components [Zinszner 83]. Weste [Weste 81b] describes a very simple algorithm which performs 1-dimensional compaction on a virtual grid with complexity O(N). An algorithm based on this approach has been implemented in this thesis.

The sticks compactor developed in this thesis combines this algorithm with some novel ideas about methods of specifying process-specific design rules. Some interesting results about the way in which interconnections between components at mask level can be realised are also presented. Perhaps the most important results produced during the work on compaction are those to do with the technology independence of CMOS circuits, which is discussed more fully in section 2.8.

Sticks compaction is an area of quite active research interest, and investigation of especially efficient or clever sticks compactors is outside the scope of this thesis. Some of these developments are nonetheless worthy of mention, since they could be applied to the VIRGIL system if it were to be developed beyond the research stage.

Improvements to the basic algorithm which has been used for virtual grid compaction can provide a constant factor speed up to the computation time needed for compaction [Boyer 83]. A similar approach can be employed to allow for hierarchical sticks compaction, which could give very considerable improvements in computation time, especially for very regular structures [Rosenberg 84]. Both these improvements are primarily to do with the time required for computing the compaction, and not with improving the density of the final mask level description.

Such density improvements are the goal of so-called 2-dimensional compaction algorithms. Sometimes, in order to place components closer to each other in one dimension, it is necessary to move them apart in the other dimension. 1-dimensional compactors cannot detect this. An algorithm for detecting these situations and dealing with them has been presented by Wolf et al [Wolf 83b]. Similar optimisations can be made manually in the VIRGIL system, by the designer identifying and modifying critical areas of the circuit.

The compactor developed for this thesis is described in detail in chapter 5.

2.8: Technology Independent Design

One of the principles upon which the work of Mead and Conway [Mead 80] is founded is the production of a simple set of conservative geometric design rules for NMOS technologies, based on a single scalable constant, lambda. Designs based on these rules can be scaled simply by changing the value of lambda. It is estimated that these rules are sufficient to allow design down to about 1 micron feature sizes. Sequin [Sequin 82] has proposed a similar set of lambda based rules for CMOS processes.

By their very nature, sticks designs are design rule free, and so are even more applicable to producing designs which will remain valid as critical dimensions shrink. Sticks designs can also accommodate variations between design rules for processes which do not conform to lambda based rules.

Design styles which allow a circuit to be implemented in several different processes within a general class of fabrication technology - say one layer metal, one layer polysilicon NMOS with buried contacts - are here called "process independent". Often such styles are called "technology independent", but here that term is reserved for design styles which can describe designs that are valid in several different general classes of fabrication technologies. Specifically, in section 5.6 of this thesis, a design style called "generalised CMOS" is introduced which allows circuit descriptions which can be implemented in four different classes of CMOS technologies, viz. n-well, p-well, twin-well and SOI (silicon-on-insulator).

At sticks level, the only significant difference between the four CMOS technologies mentioned above is the location of substrate wells (or in SOI, island dopings). There are several structures which are specific to certain technologies, such as island-island contacts in SOI, but these can be avoided.

To design technology independent CMOS circuits, it would be best if wells were not included at all in the design description, but rather included automatically if they are needed in a particular technology. An alternative is to include both p-wells and n-wells, and them remove one or other if they are not needed in the particular technology. The former seems a better solution, since the designer is freed from the need to describe wells at all.

The ways in which wells are handled in various sticks systems are not often described in detail or even mentioned. Zinszner et al [Zinszner 83] describe a sticks system which can specifically handle wells, but it seems that one must must explicitly include the specific well structure of the class of CMOS circuit being implemented - i.e. the design descriptions are process independent, not technology independent.

In chapter 5.6 of this thesis, an algorithm for automatically generating the specific well structure needed for a particular technology is presented.

2.9: Cell Libraries

At present, almost all integrated circuits are still custom designed. In some cases, a library of useful cells is maintained, but these are almost invariably stored as mask geometries.

Recently, a library of such cells has been published [Newkirk 83]. Many of the cells described are part of larger parameterisable structures. Although the cells themselves are described in a standard language (CIF), the way in which they are composed to form these larger structures is described somewhat informally in the documentation. This is undoubtedly due to the lack of any standard procedural description language.

With the exception of a few very important and regular structures, such as counters, PLA's, ROM's and RAM's, design systems tend not to have a large repertoire of well known idioms available for instantiation. Partly, this is because with conventional, embedded mask level design languages, the capture and debugging of such idioms is a reasonably time consuming task.

The system presented in this thesis specifically supports and facilitates the concise description of idioms, and so will hopefully encourage the capture of a large number of useful structures.

2.10: Silicon Compilers

Apart from the work on the capture of cell idioms, some work has been done in this thesis on the capture of an idiom at the silicon compiler level.

Unlike cell design, silicon compilation is only a very new field, and there is little published work in this area. Mostly, this presents the results of compilers which have been developed to meet specific needs of particular projects.

The first silicon compiler was "Bristle Blocks" [Johannsen 79], used to produce datapath chips.

MacPitts [Siskind 82] implements digital systems as a datapath plus controller, and includes specific support for concurrent datapath operations where possible.

Model [Gray 82] is a silicon compiler which allows arbitrary logic descriptions to be implemented as gate arrays.

The FIRST silicon compiler, described in chapter 3, is a compiler for bit-serial digital signal processing systems, and is a typical example of the first generation of silicon compilers.

3: THE F.I.R.S.T. SILICON COMPILER

In this chapter, a specific idiom for the production of complete VLSI systems from high level specifications is examined. More especially, the software to support this "silicon compiler" is discussed.

The FIRST silicon compiler idiom was devised by Dr. Peter Denyer, while the library of primitive functional cells was produced by David Renshaw. The author was responsible for writing the software to support the silicon compiler, i.e. the software which captures the idiom.

The work in this chapter was presented, in a slightly different form, at the 3rd Caltech Conference on VLSI in March, 1983 [Bergmann 83].

3.1: Background to the FIRST Silicon Compiler

The FIRST silicon compiler (Fast Implementation of Real-time Signal Transforms) has been developed as a cooperative project between the departments of Electrical Engineering and Computer Science at the University of Edinburgh, in order to allow the rapid investigation and implementation of VLSI digital signal processing systems.

The FIRST system is built around an underlying bit-serial signal representation as proposed by Lyon [Lyon 81], and systems are implemented as hard-wired networks of pipelined bit-serial operators. A typical flow diagram for a system suitable for implementation by the FIRST compiler is shown in figure 3-1.

3.2: An Innovative Architecture

The hardware implementation of a FIRST circuit consists of a network of interconnected bit-serial operators, laid out according to a relatively fixed floorplan. Each bit-serial operator is implemented as a separate function block, which is, in turn, assembled from a library of hand-designed leaf cells. A typical leaf cell might comprise, say, a single bit-slice of a given function, and the complete operator would then be arranged, both logically and physically, as a linear systolic array of these individual bit-slices. In this way, the logical size and exact function of each function block can be easily varied by selecting different numbers and types of leaf cells. For example, figure 3-2 shows two possible configurations for a bit serial multiplier - one which uses an 8-bit coefficient and rounds the least significant product bit, and the other which uses a 12-bit coefficient and truncates. (Note: the multiplier design uses 2-bit systolic array elements).

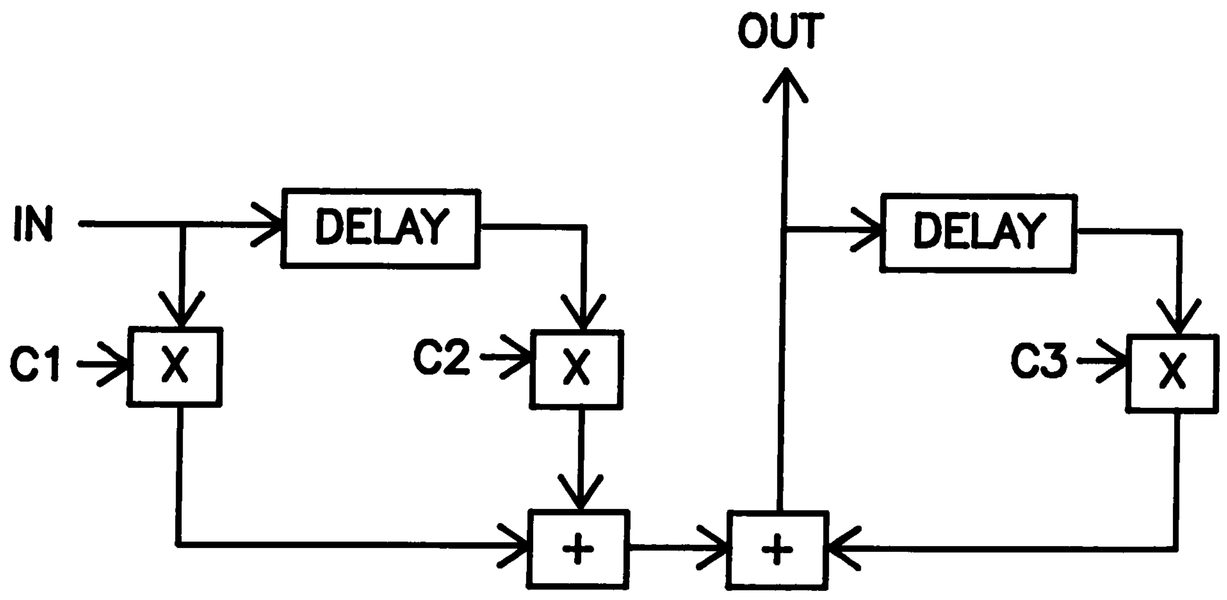
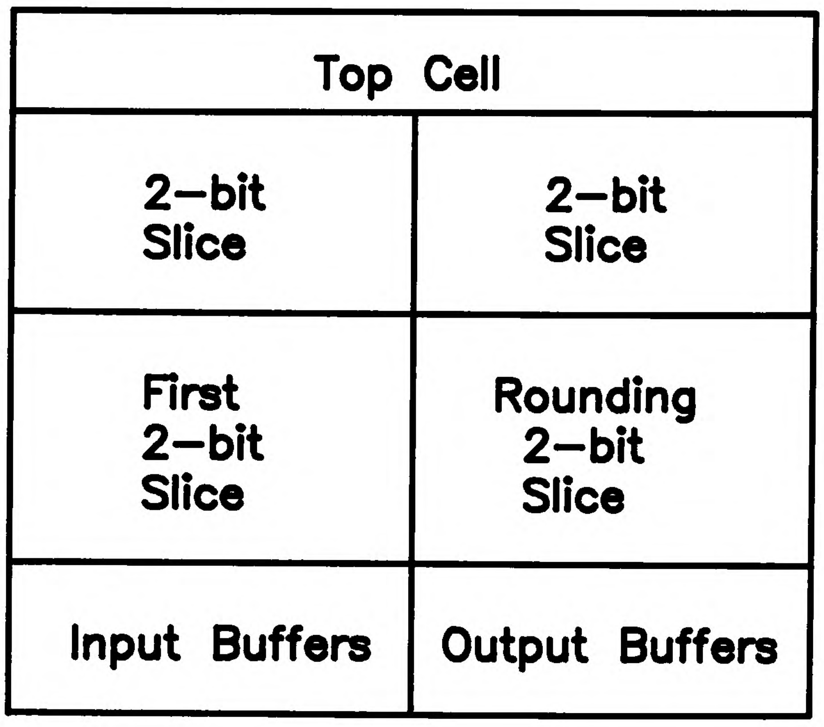
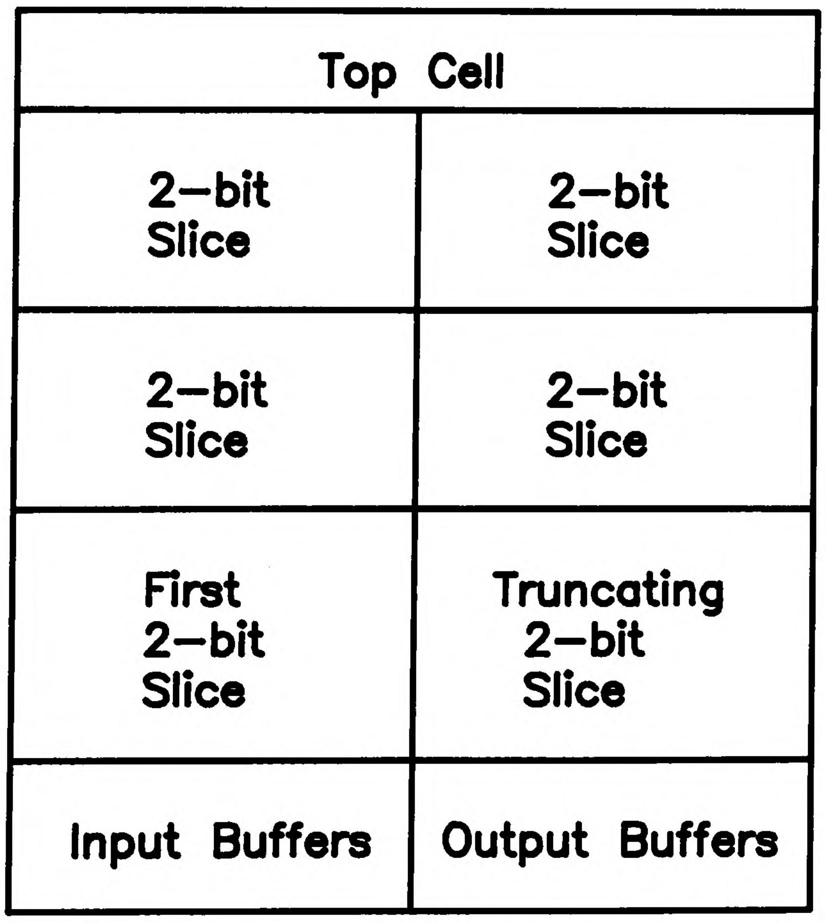


Figure 3–1: A Signal Processing System Suitable for FIRST Implementation



(a)



(b)

Figure 3–2: Two Possible Multiplier Configurations
(a) 8 bit coefficient, product is rounded
(b) 12 bit coefficient, product is truncated

EX BIBL. UNIV. EDINBURGEN.

The function blocks on a chip are arranged in two rows along either side of a single, central communications channel. Interconnections between function blocks and connections to bonding pads are all made within this channel. A typical floorplan is shown in figure 3-3. Some silicon area is wasted by this approach, since function blocks may differ in height. Typically, this area is about 20% of the total chip area, which, since it is not active area, has only a linear effect on good die/wafer yield.

Bonding pads are arranged more or less evenly around the chip periphery. After some thought it was decided to allow the pad order to be user controlled, in order to improve PCB level wiring management.

3.3: Software Support for an Integrated Design Environment

The software support for the FIRST system consists of a small suite of programs which are able to provide the designer with a complete, specialised design environment. The structure of this environment is shown in figure 3-4, and each of the major components is described below.

3.3.1: Language Compiler

The only design input available to the FIRST user is the FIRST high level language. This language provides a

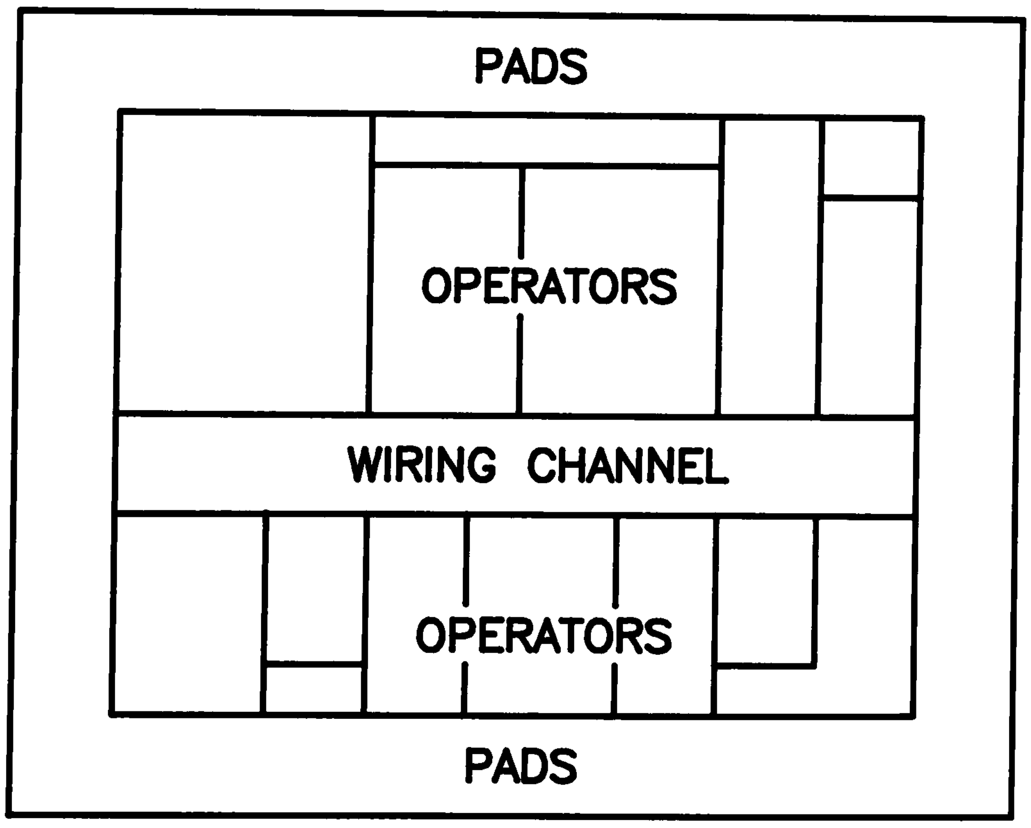


Figure 3—3: Typical Floorplan

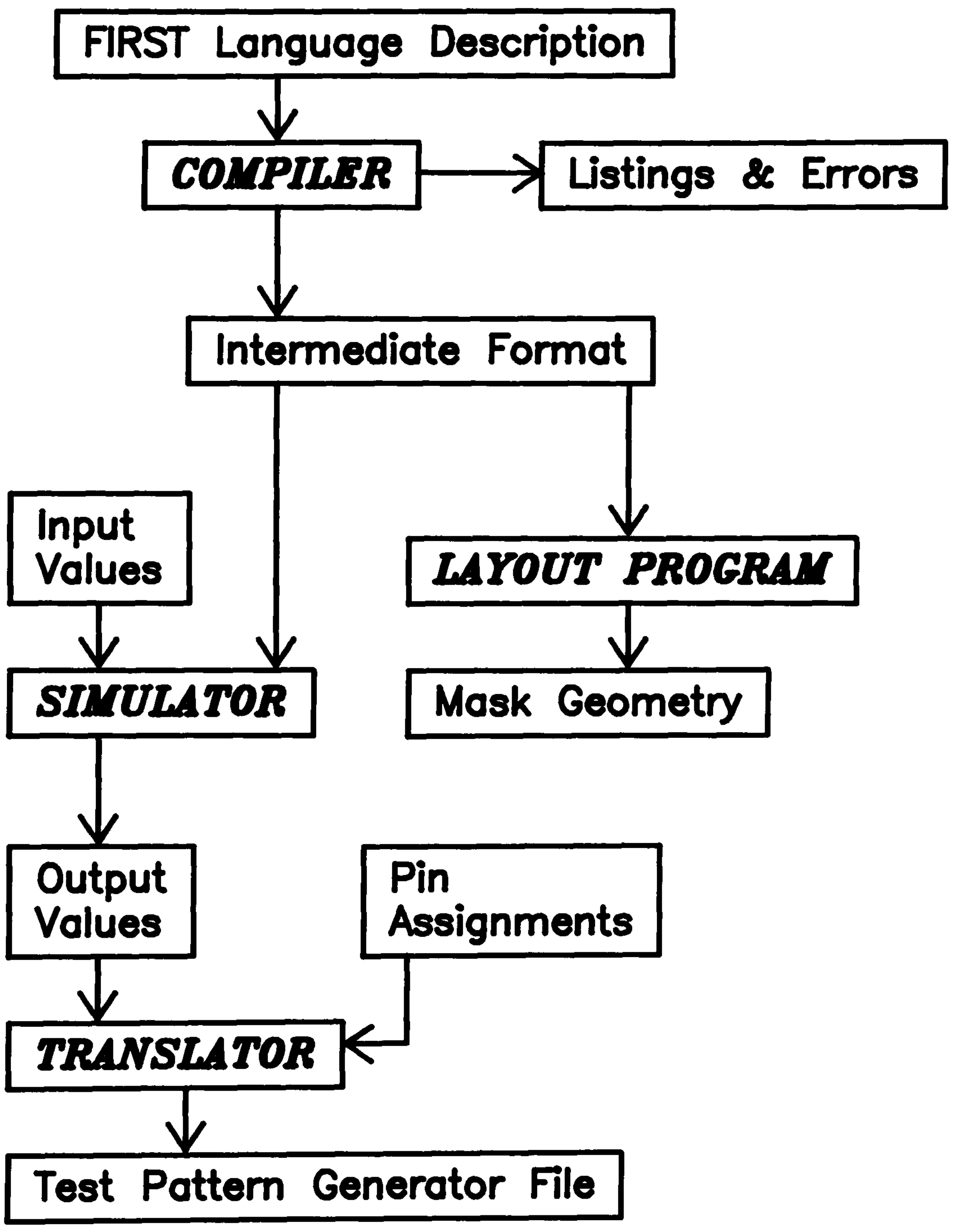


Figure 3–4: The Software Environment

structural description of the circuit under consideration, in that it describes the function blocks (i.e. bit-serial operators) which are present in a circuit, and their interconnections. In addition, because the structural design primitives have been chosen to correspond to functional design primitives, it also serves as a de facto functional description of a circuit. The language is able to capture the designer's intent in a form which closely matches the designer's logical conception of a system. This is a major advantage over designing with proprietary parts, where the designer must translate from his or her logical conception of a system into a quite different structural realisation, often in terms of ill-matched and inconvenient functional units.

The FIRST language identifies four distinct data types needed to build a circuit description:

(1) OPERATORS, corresponding to arithmetic and logical functions.

(2) CONSTANTS, which are integer expressions corresponding to the values of parameterisable operator attributes.

(3) SIGNALS, corresponding to network nodes which carry sampled-signal data.

(4) CONTROLS, corresponding to network nodes which carry timing information such as 'START OF WORD' and 'START OF FRAME'.

A circuit is a collection of parameterised operators, each with a set of input and output ports which are connected to control or signal nodes. Interconnections between different operator ports are made implicitly by connecting several ports to the same node. Each operator invocation, corresponding to a line of the FIRST language description, contains the following information:

- the operator name.

- the values of any parameterisable attributes.

- ordered lists of the names of the nodes to which input/output ports are connected.

These different data types are syntactically separated in an operator invocation, which is in essence a mathematical expression in prefix notation. The general form is:

```
NAME [params] (ctls in -> ctls out) sigs in -> sigs out
```

e.g.

```
 ADD [DELAY+2,0,0] (LSBTIME) A,B,CARRYIN -> SUM,CARRYOUT
```

The FIRST language also allows the definition of common, parameterisable sub-circuits as user-defined operators. These operators may then be invoked any number of times in exactly the same manner as primitive operators. Operator calls are grouped into CHIP definitions, corresponding to physical integrated circuits. These CHIP definitions may then be grouped into SUBSYSTEM definitions, and finally into a single SYSTEM definition. Thus the design language spans the whole hierarchy from structural primitives, to complete multi-chip signal processing systems.

The language compiler reduces this hierarchy into a list of primitive operator invocations, with node names replaced by unique node numbers. The resultant description is called the FIRST intermediate format, and it is this description which is used by later phases of the design software.

A high level language design interface allows considerable error checking to be performed, e.g. type checking, undefined names, incorrect number of operator port connections etc. The relatively constricted syntax of the language helps to avoid many of these errors in the first place.

The form of the language has been derived from the structural design language for the MODEL gate-array design system [Gray 82]. The language compiler was built using the lexical and syntax analysis phases of an existing teaching compiler, SKIMP [Rees 80], with a custom "code generation" section added. The result is a very simple single pass, recursive descent compiler. Since FIRST circuit descriptions are typically only about one page in length, the simplicity and ease with which the compiler can be altered have far outweighed any considerations about run-time efficiency.

3.3.2: Simulation

Two different simulators have been produced in the development of the FIRST system. The earlier simulator was clock driven. On each clock cycle, every operator in the system would be simulated in turn, using the present binary values on its input nodes along with any stored internal state to produce new values on its output nodes at the next clock cycle. Once all operators had been so invoked, the clock would advance by one cycle, and the process repeated. External inputs to the simulator were entered via an external data file. Similarly the values on any nodes could be output to another data file for subsequent inspection.

The operation of this simulator was a direct algorithmic interpretation of the physical circuit - so much so in fact that the functional definitions of operators could often be written such that each logic equation in the functional description would have a direct hardware counterpart in the physical realisation. This proved useful in determining if a proposed hardware realisation of an operator did, in fact, implement the desired function, and also gave a great deal of confidence that the simulator provided an accurate model of operator behaviour.

However, because this simulator was using a sequential, word oriented machine (i.e. the computer) to directly simulate the operation of a highly concurrent, bit-oriented architecture, the computational effort required for a thorough simulation of a large system was unacceptable.

For this reason, another simulator has been designed which simulates a system at a higher level of abstraction. This simulator is event driven, and simulates the operation of individual operators on a word by word basis. The values on nodes, which in reality consist of serial bit streams, are modelled as discrete words of data occuring at discrete time intervals. When a new word of data reaches a node, an event is said to have occurred. Events are described in terms of the node to which they refer, the new value of that node as a result of the event, and the time at which the event occurs.

The scheduling of events is handled by keeping all pending events on an event queue. All events due to occur at a given time are removed from the queue, and the values of the associated nodes are updated to reflect their new values. All operators which have any of these nodes as their inputs are activated. The new values which have arrived at their inputs will generate new values at their outputs. These new values are modelled as further events scheduled to occur at some later time, as determined by the latency of the particular operator.

In general, new values should arrive at all inputs of a particular operator at the same time. If not, this usually implies that the designer has made an error in matching the latency of the various signal paths leading to this operator. When such mismatching occurs, the simulator issues a warning, and continues the simulation. The operation of the simulator in such cases will not truly echo that of the physical system, which is itself unlikely to be a valid circuit. Once all such timing "bugs" have been eliminated (and this simulator provides a powerful tool for such debugging) the designer can have a high degree of confidence in the simulation results.

Inputs and outputs to the system are again via external data files. These data files are essentially lists of events relevant to the system input or output nodes. Since the system is being modelled at a higher level than in the clock driven simulator, the computing effort required for a given simulation has been reduced by about an order of magnitude.

A program also exists within the FIRST suite to convert the simulator's output into a form suitable for use with an automatic test pattern generation system.

3.3.3: Physical Layout

The task of the FIRST layout program is to produce a physical realisation of the system implied by a FIRST language description. This process proceeds in distinct phases, according to a strict layout strategy.

For every invocation of a predefined operator in a FIRST language description of a system, a corresponding function block appears in the physical layout. Each function block is assembled from the appropriate leaf cells as described above. Once constructed, operator blocks are placed along "waterfronts" at the top and bottom edges of a central wiring channel. The layout program uses the criteria of minimum chip area to decide on a suitable arrangement of the blocks. The size of operator bounding boxes typically covers quite a range of sizes. The overall size of the chip, however, depends on four main factors:-

- the height of the tallest block on the top row
- the height of the tallest block on the bottom row
- the width of the wider of the two rows
- the size of the wiring channel

The first three factors depend only on which blocks are placed on the top row, and which are placed on the bottom, and the placement algorithm first decides on this subdivision. Without rigorously attempting to explain the algorithm, the basic idea is to place all the tall blocks on the top row, and the short blocks on the bottom row, where the division between tall and short is chosen so that the total area is minimized. Perturbations are then made to this arrangement to reduce the area further by making the rows more equal in width. The size of the wiring channel is considered constant during these calculations.

Next, the arrangement of the blocks within each row is decided. The width of the wiring channel, and hence total chip area will differ with different arrangements. Rather than use an algorithmic method for determining a good arrangement, the blocks are placed left to right within each row in the same relative order that they were invoked in the FIRST language description of the system. Since a designer may be expected to write this description such that the general flow of information is from one operator to the next down the page, such a strategy should lead to closely coupled operators being placed close together on the chip. Wiring channel area is only a relatively small part of the overall chip area, and so it was not considered worth the effort, both in terms of programming time, and run time, to calculate any more optimal arrangement.

All routing between operators is done in the single central wiring channel. Inputs and outputs of all operators are available along the channel waterfront. A very simple two layer router is used, with metal lines running horizontally, and diffusion lines running vertically. The input and output ports of operators are restricted to points on a fixed grid, with the grids for the two sides being offset. This ensures that connections can be made between any two ports with a single horizontal wire and two vertical wires, i.e. without dog-legs. Figure 3-5 shows a section of a typical wiring channel.

Finally bonding pads are arranged around the chip, and ancillaries such as power, ground and a global two-phase non-overlapping clock are added.

The layout program has been written using an embedded IC design language called ILAP [Hughes 83], based on LAP [Locanthi 78], but using IMP as a host language. IMP [Robertson 83] is a high level Algol-like programming

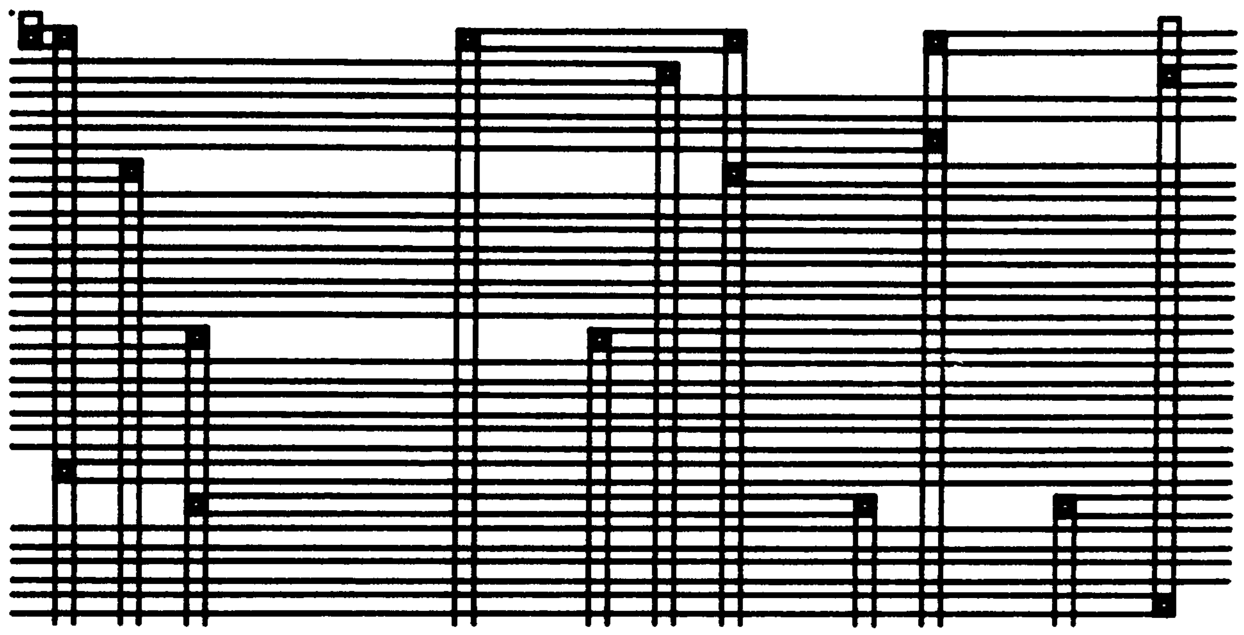


Figure 3–5: Wiring Channel Section

language used extensively within the Edinburgh computing environment.

3.4: An Example System

As an illustrative example of a FIRST circuit, consider a chip to implement a simple four-stage, cascadable FIR filter section. A flow diagram of such a section is shown in figure 3-6. For illustrative purposes, consider it divided into two "TWO STAGE" sections as shown in figures 3-7 and 3-8. From these flow diagrams, an implementation in terms of FIRST operators could be derived to give the circuits shown in figures 3-9 and 3-10. Note that the final delay element in TWO STAGE has been made parameterisable in order to allow the final TWO STAGE section on the chip to have a slightly shorter inter-stage delay which compensates for the delay in going off-chip to the next stage in a multi-chip system. Also note that a network of timing (CONTROL) signals has been added in order to give 'start of word' information to the various bit-serial operators. These signals are shown in figures 3-9 and 3-10 as broken lines.

The FIRST language description of a single section is shown in figure 3-11, while the resultant layout is shown in figure 3-12. If such a chip was fabricated using a 5 micron NMOS process, it would be approximately 5mm x 5mm,

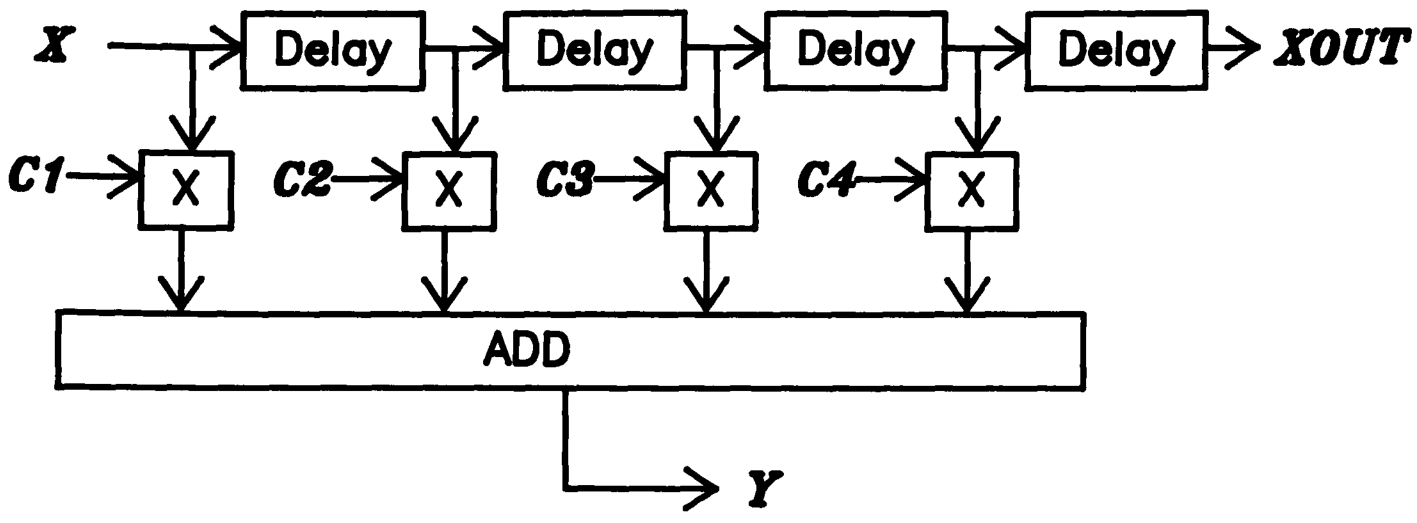


Figure 3-6: Flow Diagram

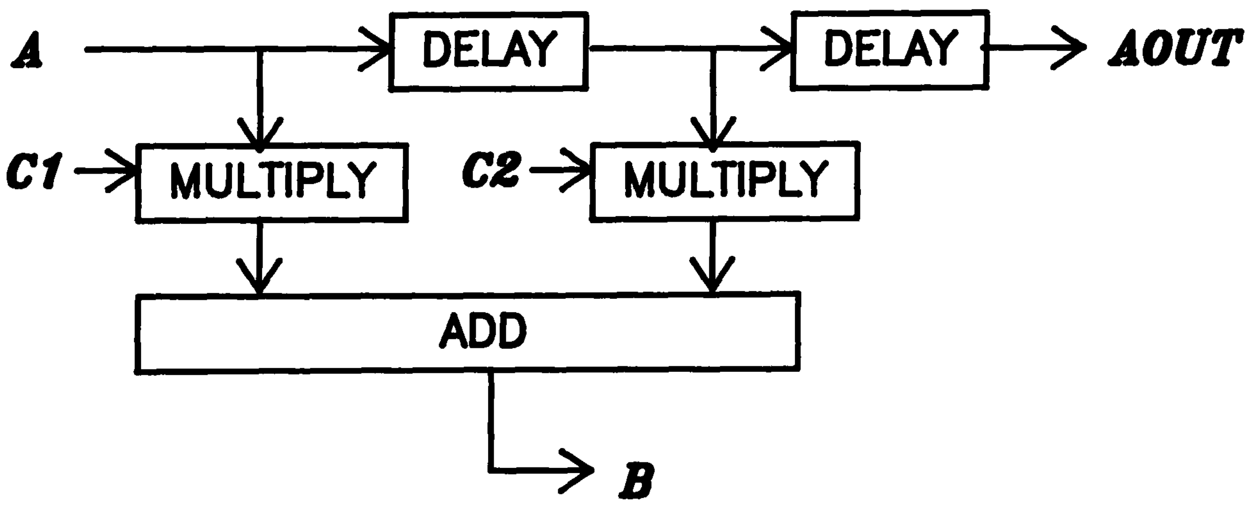


Figure 3-7: TWOSTAGE sub-circuit

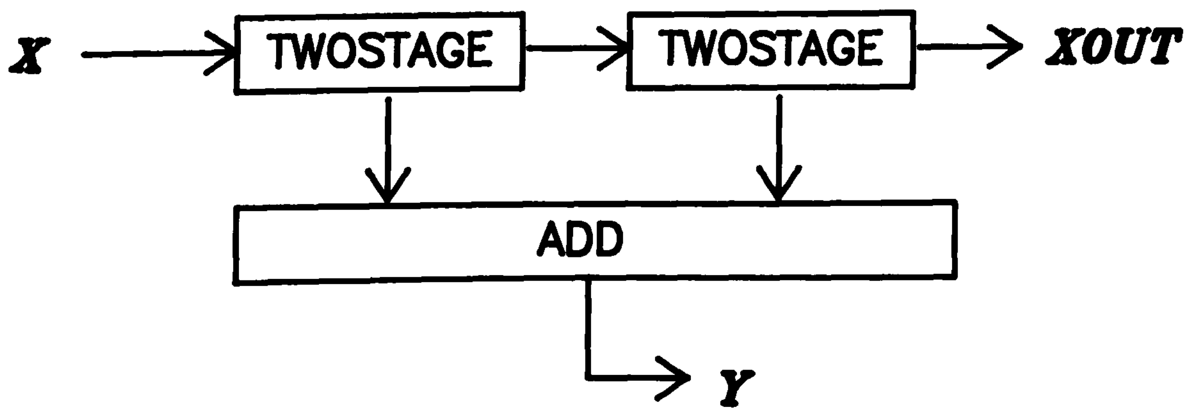


Figure 3-8: Simplified Flow Diagram

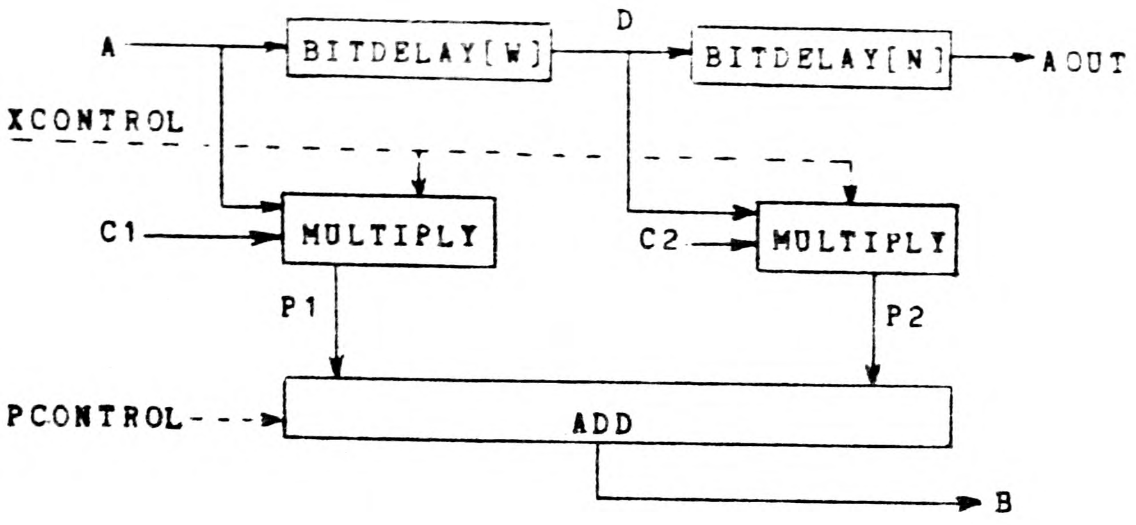


Figure 3-9: Flow Diagram for TWOSTAGE[N]

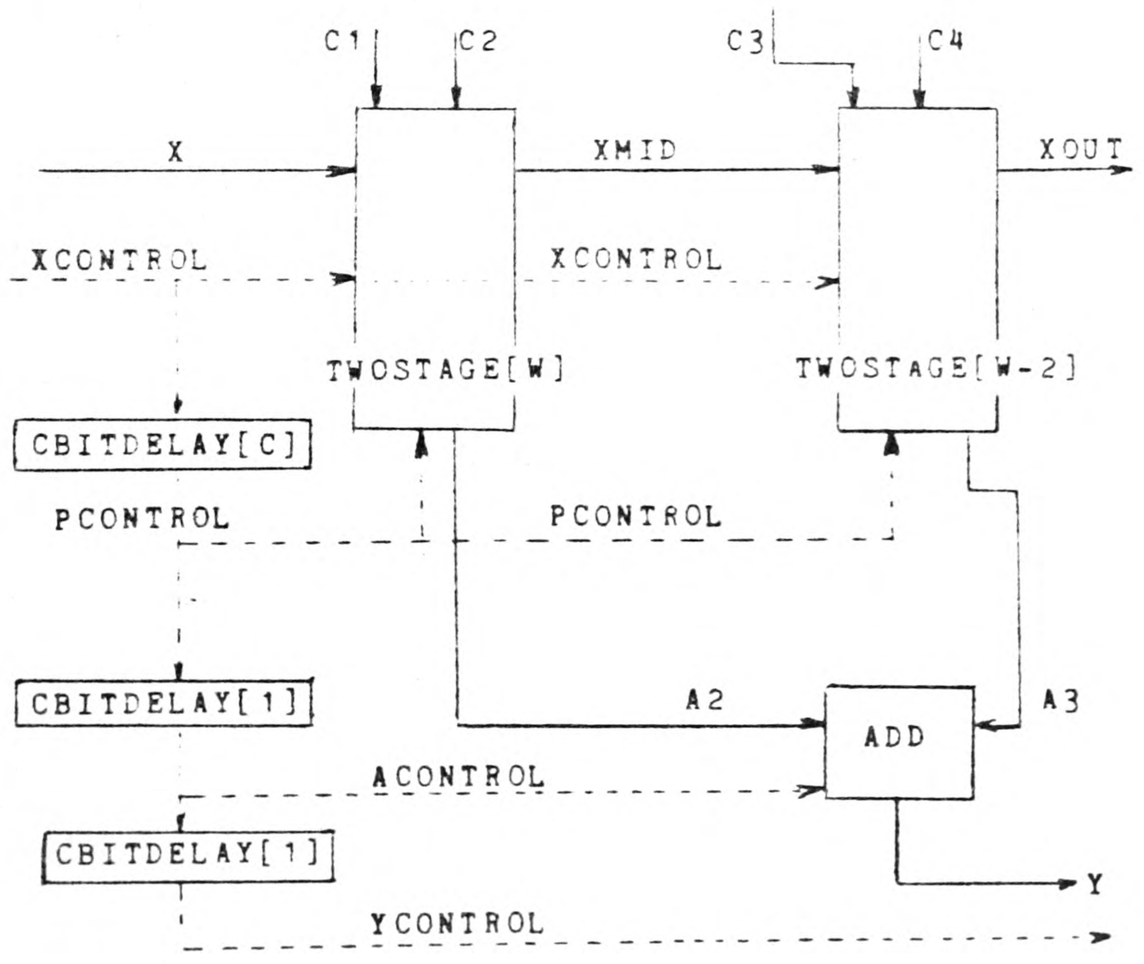


Figure 3-10: Complete Flow Diagram

```
FIRST COMPILER - Copyright Denyer,Renshaw,Bergmann 1982
SOURCE FILE: FIR

! Global Constants
CONSTANT wlth=10,            ! Data word length
         cbits =10,          ! Coefficient word length
         d = cbits/2,        ! Multiplier latency
         truncate = 0        ! Type of multiplier

! Define simple forms of ADD and MULTIPLY

OPERATOR Adder (c) a,b -> sum
  Add[1,0,0,0] (c) a,b,gnd -> sum,nc
END
OPERATOR Multiplier (c) a,b -> p
  Multiply[truncate,cbits,0,0] (c->nc) a,b -> p,nc
END

! Two stages of an FIR filter, n=2nd stage delay

OPERATOR TwoStage[n] (xctl,pctl) a,c1,c2 -> aout,b
  SIGNAL d,p1,p2
  Bitdelay[wlth] a -> d
  Bitdelay[n] d -> aout
  Multiplier (xctl) a,c1 -> p1
  Multiplier (xctl) d,c2 -> p2
  Adder (pctl) p1,p2 -> b
END

! Define the whole chip,including pad  ordering
CHIP FIR (xxctl -> yyctl) xx,d1,d2,d3,d4 -> xxout,yy
  CONTROL pctl, actl, xctl, yctl
  SIGNAL xmid,a2,a3,x,y,c1,c2,c3,c4,xout
!!!!Specify order of bonding pads
  PADORDER VDD,xx,xxout,yy,GND,
           xxctl,yyctl,CLOCK,d1,d2,d3,d4
!!!!Equate external bonding pads to internal nodes
  Padin (xxctl->xctl) xx,d1,d2,d3,d4 -> x,c1,c2,c3,c4
  Padout (yctl -> yyctl) xout,y -> xxout,yy
!!!!Specify operators to be included
  TwoStage[wlth] (xctl,pctl) x,c1,c2 -> xmid,a2
  TwoStage[wlth-2] (xctl,pctl) xmid,c3,c4 -> xout,a3
  Adder (actl) a2,a3 -> y
  Cbitdelay[d] (xctl -> pctl)
  Cbitdelay[1] (pctl -> actl)
  Cbitdelay[1] (actl -> yctl)
END
ENDOFPROGRAM
```

Figure 3-11: FIR Filter Listing

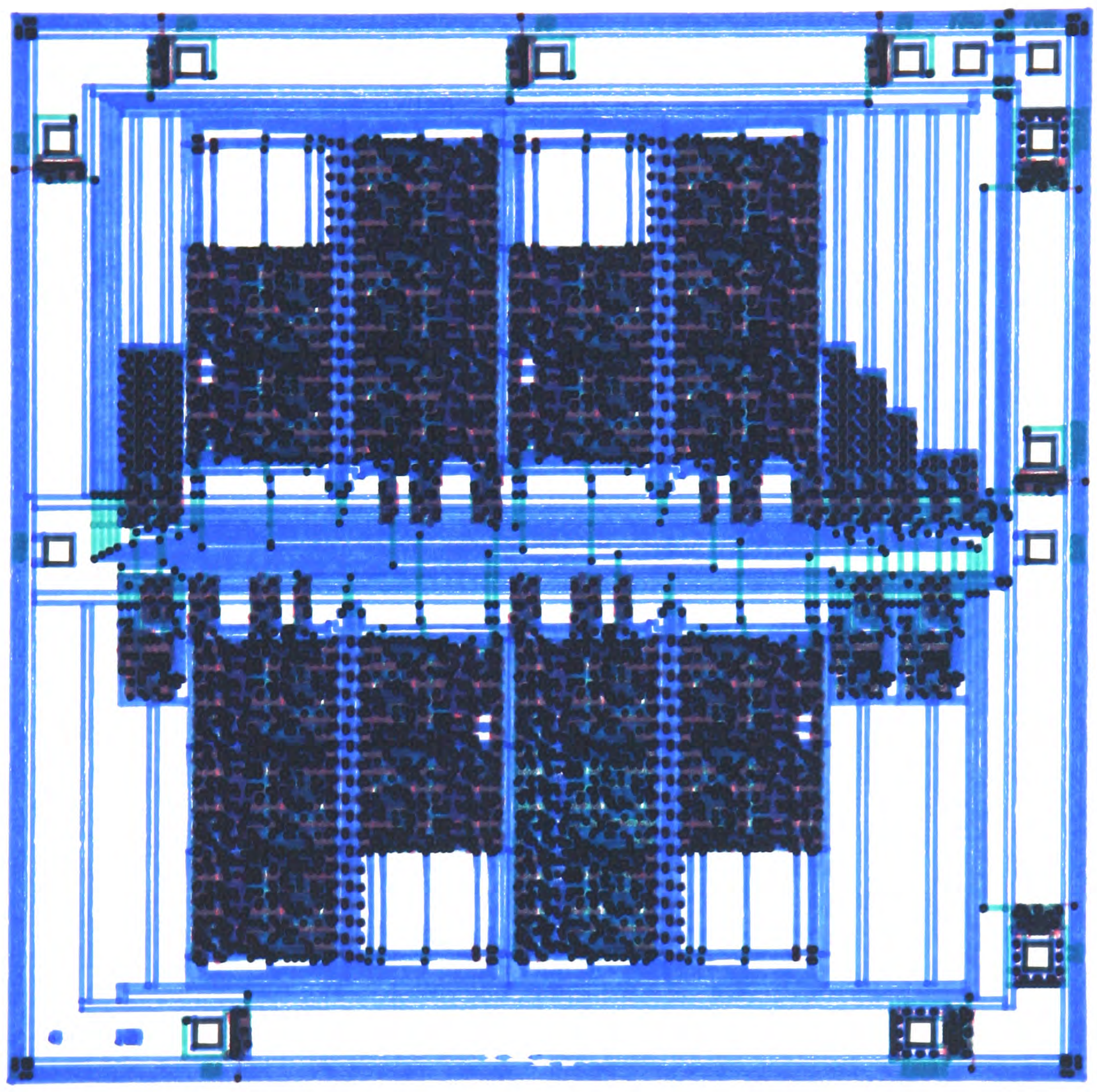


Figure 3–12: FIR Filter Layout

and have a clock speed of 8 MHz, corresponding to a sample rate for 10 bit samples of 800 kHz.

At present, test chips are being fabricated to test and debug the primitive operator cells, which have been designed using traditional custom design methods. FIRST has been used to design and simulate several different systems, but none of these have yet been fabricated, mostly due to delays in debugging primitive operator cells.

3.5: Conclusions

The FIRST system is not simply an automatic layout system, rather it provides a complete design environment for bit-serial signal processing systems. It allows designers with no previous IC design experience to exploit silicon as an implementation medium.

A simple, high level structural design language provides a consistent circuit description for both simulation and layout. Complete signal processing systems can be functionally simulated both quickly and accurately, encouraging designers to explore and compare a wide variety of possible circuit solutions without the need to produce hardware prototypes. When a design has been thoroughly simulated, the same structural description can be used to produce a hardware realisation which will exactly match the simulated performance. This physical layout process neither requires nor allows human intervention, precluding the possibility of designer-introduced layout errors.

By using an innovative architectural floorplan, this automatic and error-free silicon compilation can be achieved at the expense of a relatively small increase in overall silicon area, while at the same time reducing design time and cost by at least an order of magnitude in comparison with conventional custom IC design.

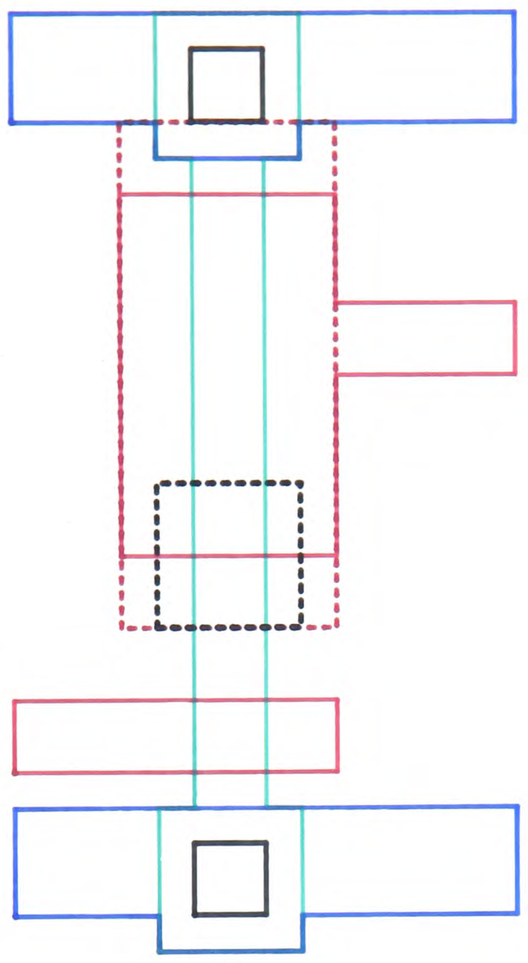
All of the FIRST software has been specifically designed for the capture of this one idiom. The savings in design cost for suitable systems is more than enough to justify this effort. The advantages of capturing just this single idiom provide strong motivation for the development of a more general system for the capture of a whole range of idioms. The design and investigation of such a system forms the basis for the remainder of the work of this thesis.

4: AN IDIOM DESCRIPTION LANGUAGE

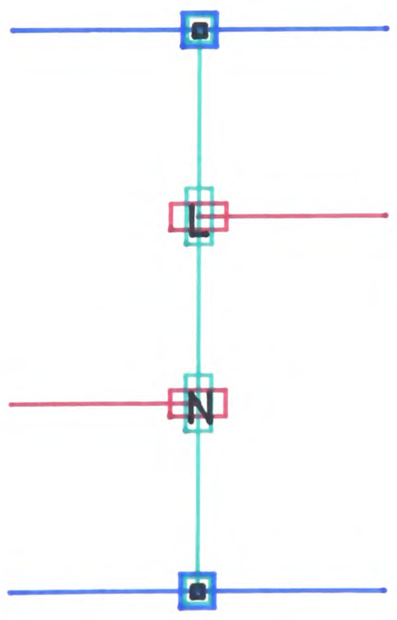
4.1 The Virtual Grid

Integrated circuits are usually designed in terms of the mask layers from which they are fabricated. From combinations of these mask layers, the basic circuit primitives, such as transistors, wires and contacts, are constructed. A structural design system allows the designer to specify a circuit entirely in terms of these circuit primitives, rather than in terms of the mask layers used to build the primitives. A structural design system may also allow the designer to specify the relative topological arrangement of design primitives, in which case it is referred to as a "sticks" system [Williams 77]. Figure 4-1 shows inverters represented at both the mask level and sticks level.

Mask level descriptions can be described as sets of polygons existing on the various mask layers. The polygons can be defined in terms of the physical coordinates of their vertices. A sticks based representation, however, does not impose any physical coordinates on the elements within it. Rather it only imposes relative spatial orderings on components, and also specifies connections between components. An elegant method of describing these orderings and connections is the virtual grid, as reported by Weste



(a) Mask Level



(b) Sticks Level

Figure 4–1: Equivalent Representations of an Inverter

[Weste 81a].

A virtual grid consists of a two-dimensional grid of coordinate points. Circuit primitives are then placed on this grid - transistors and contacts are placed at grid points, and wires are placed on the line segments joining grid points. Only wires parallel to coordinate axes are allowed in the type of virtual grid considered in this thesis. The relative ordering of components on a virtual grid is implied by the relative ordering of the virtual grid points on or between which these components lie.

The actual coordinates of grid points are not significant, it is only the relative topological orderings which they impose on circuit components which are important. For example, figures 4-2(a), 4-2(b) and even 4-2(c) describe topologically similar circuits. In practical virtual grid systems use can be made of the extra twists and bends in the circuit in figure 4-2(c). Such bends can be used by the designer to convey advice such as: "When this circuit is converted to mask level descriptions, a better layout might be achieved by including these bends in these wires". The ability to include "hints" like these is a useful feature in a design sytem, since it allows the designer to aid the efficiency but not hinder the correctness of automated translation between different levels of representation.

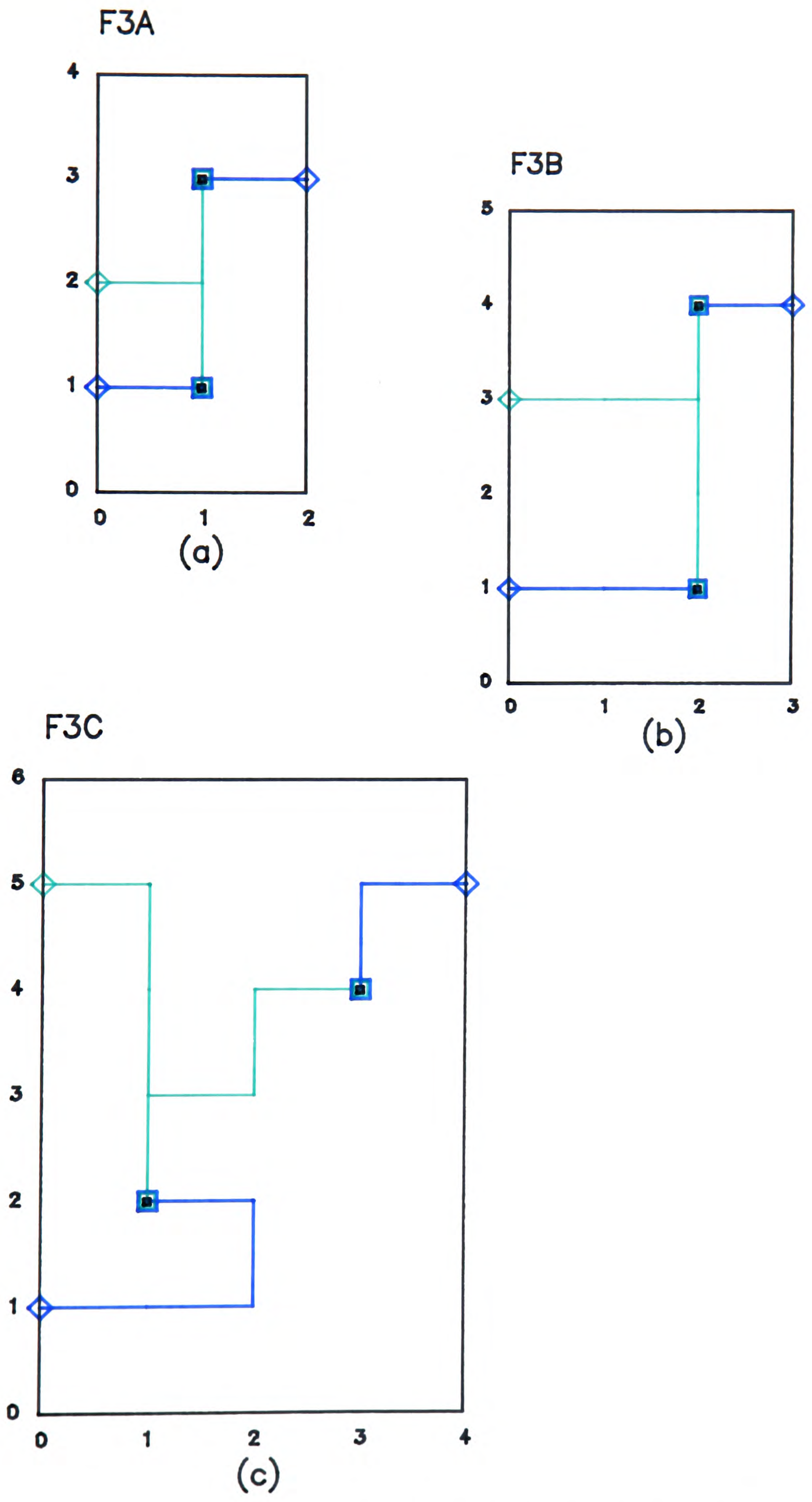


Figure 4–2: Topologically Similar Circuits

One of the most important features of a virtual grid representation is that it has a natural textual equivalent. The virtual grid coordinates allow the position of items in this type of sticks level description to be easily specified. The ability to describe circuits textually is considered to be an important aid in controlling the overwhelming complexity of VLSI circuits, as has already been mentioned in section 2.4. Textual descriptions allow features such as selection, repetition and parameterisation to be added to a design description. In addition, textual based representations can use the same editing and filing facilities as are used for ordinary computer programs - in fact most of their advantages are related to their similarity to computer source code, and many ideas can be borrowed from the lessons learned in structured programming.

A textual, virtual grid based system has then been chosen as a suitable base on which to build an idiomatic design system. As the description of this system is presented, more advantages of using such a system will become apparent.

Certain conventions have been adopted throughout this thesis for the graphical representation of both mask level and virtual grid circuits. The following colours are used to represent the various mask layers and corresponding virtual grid features:

Red	- Polysilicon
Blue	- Metal
Green	- Diffusion
Black	- Contact
Dotted Red	- Implant
Dotted Black	- Buried Contact
Dotted Green	- Substrate Contact
Chained Red	- N well
Chained Green	- P well

Virtual grid devices are described graphically by symbols, which should be largely self explanatory. Ports are shown as diamond shapes of the appropriate colour. Different types of transistors are labelled with the following letters:

N	- n channel
P	- p channel
L	- load device
D	- depletion mode

4.2 Idioms and Instantiations

An idiom is not merely a useful circuit, but rather it is the manner in which a whole range of circuits can be generated. Each of the individual circuits which is embraced by an idiom is called an instantiation of that idiom.

It is important to be aware of the fundamental differences between idioms and their instantiations. An instantiation corresponds to a single circuit, which in this thesis will be described using a virtual grid. Such a circuit could be graphically displayed, or equally well, described textually. However, an idiom is a recipe for generating such circuits, and the parameterisable features of an idiom do not lend themselves to simple graphical representation. For this reason, a purely textual notation has been developed for the description of idioms.

This has been done by first developing a language, called VG, for the description of virtual grid circuits, and then adding additional features to produce another language, called VIRGIL, for the description of idioms.

4.3: VG - A Virtual Grid Circuit Description Language

In designing a language to capture circuits, it was recognised that even relatively simple circuits are often best described hierarchically. Typically, a circuit is decomposed into a number of simple cells, which are then individually designed and composed together to give the complete circuit. The simple cells are called leaf cells, because they are the leaf nodes in the tree representing the hierarchical decomposition of the circuit. Cells which are generated by the composition of smaller cells are called composition cells, and correspond to non-leaf nodes in the tree.

The reason for using such a hierachical approach is to reduce the complexity of the design task. People have a natural limit on the amount of data which can be simultaneously comprehended, and it is by keeping within this limit that design systems are best utilised by human designers. By representing a circuit hierarchically, the amount of information which must be handled at any one time is controlled. Such an approach has long been recognised in the design of computer software, and forms the basis of structured programming.

VG supports such a design hierarchy, by describing designs in terms of leaf cells and composition cells. Leaf cells describe a set of primitive structural components which are laid out on a virtual grid. Composition cells describe the way in which leaf cells are composed together to form complete circuits. There are no mixed cells which contain both structural primitives and instances of other cells. Such a restriction serves to highlight the differences between the cell design and cell composition phases of circuit design.

4.3.1 Notes About the Language Description

The following is not meant to be a rigorous definition of the VG language. The language is meant to be, initially at least, a private research tool, rather than a generally available facility. What follows is more in the nature of a description of the features of the language, rather than a definition of it.

In general, the language shares many of the syntactic features of high level languages. Where necessary, the syntax of parts of the language will be described using the conventions proposed by Wirth [Wirth 77b]. Basically, these are as follows.

Productions are of the form:

```
        phrase = definition
```

Repetition, zero or more times, is denoted by braces {..}.
Parts of a definition enclosed in square brackets [..] are optional.
The '|' character is used for selection.
Parentheses (..) are used merely for grouping.
Terminal symbols, i.e literals, are enclosed in double quotes.

4.3.2 Alphabet

The language is written using the ASCII charcter set. Lower case letters are considered equivalent to their upper case counterparts. Spaces may be inserted anywhere except within identifiers and numerals (see below). Comments may be placed anywhere except within identifiers and numerals, and may consist of any text, except '}', enclosed within braces {..}.

4.3.3 Identifiers

Identifiers consist of any combination of letters and digits, beginning with a letter. Identifiers may optionally include a single '.' which divides the identifier into a root (before the dot) and an extension (after the dot). Examples of legal identifiers are:

```
    FRED
    AReallyLongAndComplicated.One
    A1234.9p
but not:
    Bil#6                 {Contains illegal charcter '#'}
    4Minute.Warning       {Starts with a digit}
    Nimble Toadstool      {Spaces not allowed}
```

4.3.4 Reserved Words

The following words have special significance within the language, and are not available for use as identifiers.

IF	THEN	ELSE	FOR	REPEAT	TRUE	FALSE
BOOLEAN		ARRAY	OF	INTEGER	INX	INY
LEAF	COMPOSITION		CELL	END	BURIED	BUTTING
DWIRE	PWIRE	NWIRE	MWIRE	DM	PM	NM
PSUB	NSUB	PPORT	NPORT	DPORT	MPORT	

4.3.5: Numerals and Expressions

Numerals consist of any combination of digits, and are interpreted as base ten non-negative integers. Negative integers are written as expressions.

Expressions may yield either boolean (true or false) values or integer values. They conform to the normal rules of arithmetic, including the use of parentheses to alter the order of evaluation. The following operators are available in decreasing order of precedence; operators on same line have equal precedence.

Integer Operators: Take integer operands

Yield integer result

Unary -

*, /

+, -

Relational Operators: Take boolean or integer operands

Yield boolean result

>, <, >=, <=, =, # (not equal)

Boolean Operators: Take boolean operands

Yield boolean result

NOT

AND

OR

Integer operands may be either numerals or integer valued expressions. Boolean operands may be one of the predefined boolean constants TRUE and FALSE (where TRUE>FALSE) or boolean valued expressions.

It is a general feature of the language that expressions may be used whenever a value is required.

4.3.6: Statements

A VG language description consists of a number of statements. Statements are written one to a line. Statements may be continued onto the next line by ending the current line with a hyphen '-', which is otherwise ignored.

4.4: VG Leaf Cells

Leaf cells are constructed from a set of structural components positioned on a virtual grid.

4.4.1 Leaf Cell Header and Terminator

In order to define a leaf cell, it is first necessary to define the virtual grid on which it is based. This is done by beginning each leaf cell with a header statement of the form:-

header = "LEAF" "CELL" identifier "=" bounds

where

bounds = "(" llx "," lly "," urx "," ury ")"

identifier is the name of the cell

llx, lly, urx, ury are all integer expressions which give, in order, the X and Y coordinates of the lower left corner of the virtual grid and the X and Y coordinates of the upper right corner.

The last statement in a leaf cell description is:

```
    terminator = "END"
```

e.g., the statements

```
     Leaf Cell TEST = (0,0,2,3)
     .
     .
     .
     End
```

would enclose the statements defining a leaf cell called TEST, with components laid out on the virtual grid shown in figure 4-3.

4.4.2: Leaf Cell Component Specification

Virtual grid circuits are specified in terms of their structural components, and so a suitable set of such components must be chosen. The components chosen will depend on the underlying fabrication technology which will be used. For the purposes of this thesis, it was decided to choose a set of primitives which would allow the description of NMOS and CMOS digital circuits. It was also decided to restrict the system to technologies with a single layer of metal and a single layer of polysilicon.

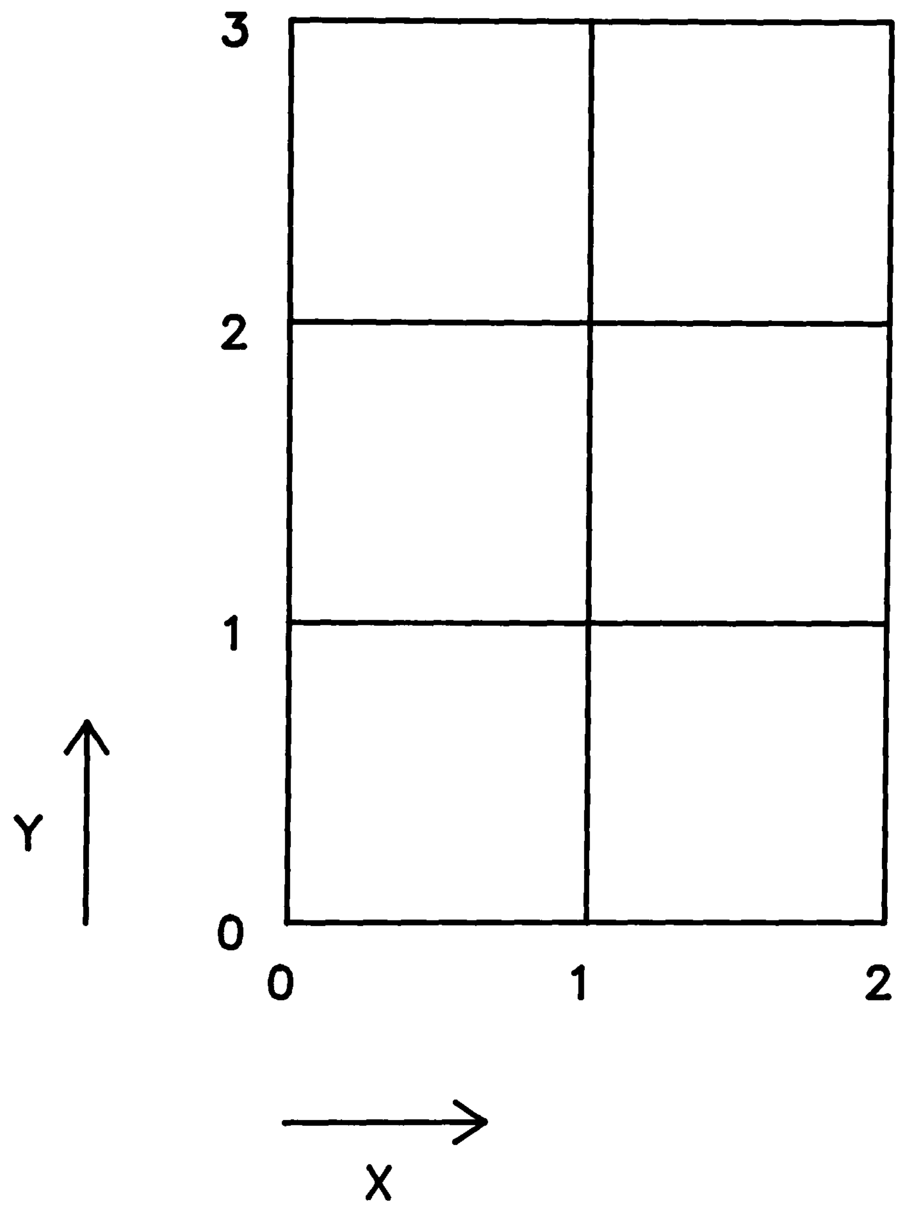


Figure 4–3: A Typical Virtual Grid

The primitives which are available for use in VG descriptions are identified by individual names. The primitives may be divided into four classes, as detailed below.

Wires

Wires are used for interconnection between other components. Wires are placed between grid points, and they automatically connect components on those grid points. The exact manner of this connection is described later. Four types of wire are available:

MWIRE	a wire on the metal layer
PWIRE	a wire on the polysilicon layer
NWIRE	a wire on the n-type diffusion layer
DWIRE	a wire on the p-type diffusion layer

Contacts

Contacts allow wires of different types to be connected to each other, and allow metal wires to be connected to the silicon substrate. The following types of contact are available:

PM	PWIRE to MWIRE
DM	DWIRE to MWIRE
NM	NWIRE to MWIRE
BURIED	NWIRE to PWIRE
BUTTING	NWIRE to PWIRE and MWIRE

PSUB : MWIRE to p-type substrate

NSUB : MWIRE to n-type substrate

Transistors

The following types of transistor are available:

PTYPE : p-channel enhancement-mode transistor

NTYPE : n-channel enhancement-mode transistor

DEPLETION: n-channel depletion-mode transistor

LOAD: n-channel depletion-mode transistor with common gate-source connection, used as NMOS "pullup"

Ports

Ports are virtual grid items which do not have a physical realisation. Ports are used to identify where external connections may be made to a cell. Only wires which are connected to ports may extend to the edge of a leaf cell. There are four types of ports, corresponding to the four types of wires:

MPORT: for MWIRE

PPORT: for PWIRE

NPORT: for NWIRE

DPORT: for DWIRE

Items are placed on the virtual grid by specifying the X and Y coordinates of their position. Coordinate pairs are written as integer expressions enclosed in parentheses and separated by a comma. In practice, the expressions are almost invariably simple integer values.

Ports, contacts and transistors are specified by statements of the form:

```
     statement = item_name "@" coordinate
```

e.g.

```
        NPORT @ (0,1)
        PTYPE @ (2,4)
```

The simplest form of wire specification is for wires which run parallel to the X or Y axis, and between adjacent grid points:

```
statement = wire_name "@" coordinate "->" coordinate
```

e.g.

```
          NWIRE @ (0,1) -> (1,1)
```

The same form of statement is used for wires which extend in the same direction for more than one grid unit, e.g.

```
          NWIRE @ (0,1) -> (2,1)
```

which is exactly equivalent to the two statements:

```
          NWIRE @ (0,1) -> (1,1)
          NWIRE @ (1,1) -> (2,1)
```

Wires may be specified with more complicated paths. Such wires are described by breaking the path into a series of sections parallel to one or other of the coordinate axes. This more general form of wire statement is:

```
statement = wire_name "@" coordinate "->" coordinate
                                 { "->" coordinate" }
```

where a statement of the form

```
  wire @ (X1,Y1) -> (X2,Y2) -> (X3,Y3) -> .. -> (Xn,Yn)
```

is equivalent to

```
      wire @ (X1,Y1) -> (X2,Y2)
      wire @ (X2,Y2) -> (X3,Y3)
      .
      .
      wire @ (Xm,Ym) -> (Xn,Yn)
```

e.g. the path shown in figure 4-4 could be described by any of the following sets of statements:

```
    NWIRE @ (0,1) -> (2,1) -> (2,2) -> (4,2)
```

or

```
    NWIRE @ (4,2) -> (2,2) -> (2,1) -> (0,1)
```

or

```
    NWIRE @ (0,1) -> (1,1) -> (2,1) -> (2,2)
    NWIRE @ (4,2) -> (2,2)
```

Labels can be added to statements specifying component positions, and take the form:

```
        statement = identifier ":" rest of statement
```

e.g.

```
      input : nport @ (0,1)
```

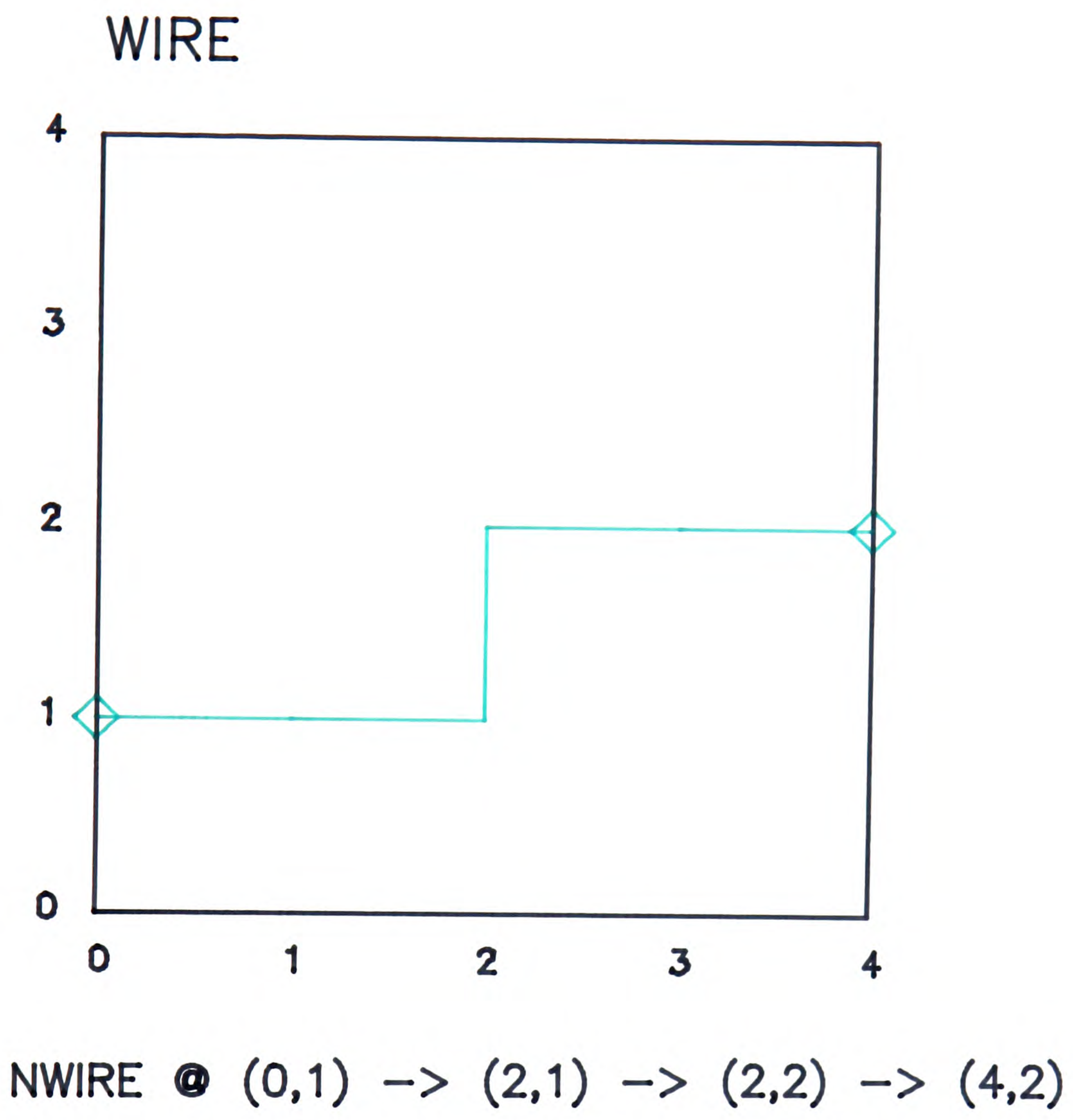


Figure 4-4: A Virtual Grid Wire

The label can subsequently be used to refer to the grid item with which it has been associated, and also to refer to the coordinates of that item. In the case of wires, the label refers to the first coordinate in the wire path. Labels can then be used instead of explicit coordinates in subsequent statements, e.g.
Given the statements:

```
Contact1: nm @ (3,2)
Contact2: nm @ (3,4)
```

then the statement:

```
Mwire @ contact1 -> contact2
```

could be used instead of:

```
Mwire @ (3,2) -> (3,4)
```

Forward references to labels are not allowed in the current implementation.

Coordinates can be specified relative to some other coordinate. Such relative coordinates are specified by:

```
relative_coordinate = base "+" offset
```

where both base and offset may be labels or explicit coordinate pairs.

A relative coordinate:

```
(basex,basey) + (offsetx,offsety)
```

is equivalent to the coordinate

```
(basex+offsetx, basey+offsety)
```

The circuit in figure 4-5 could thus be described by the following statements:

```
point1: nm @ (1,2)
point2: nm @ point1 + (1,-1)
mwire @ point1 -> point1 + (1,0) -> point2
```

Such constructs enable related components to have their absolute position within the grid determined by the absolute position of some single base point (in the above case 'point1').

It is often necessary to have some control over the physical properties of various components, most notably the relative lengths and widths of transistor channels. In this language, values may be specified for various physical attributes by adding a list of parameter assignments to the name of a grid item. This has the form:

```
item = item_name [ "(" parameter_list ")" ]
```

where

```
parameter_list = assignment { "," assignment }
```

and

```
assignment = identifier "=" expression
```

e.g.

```
ntype(l=2,w=3)
```

Each assignment has the effect of giving the value of the expression to the attribute associated with the

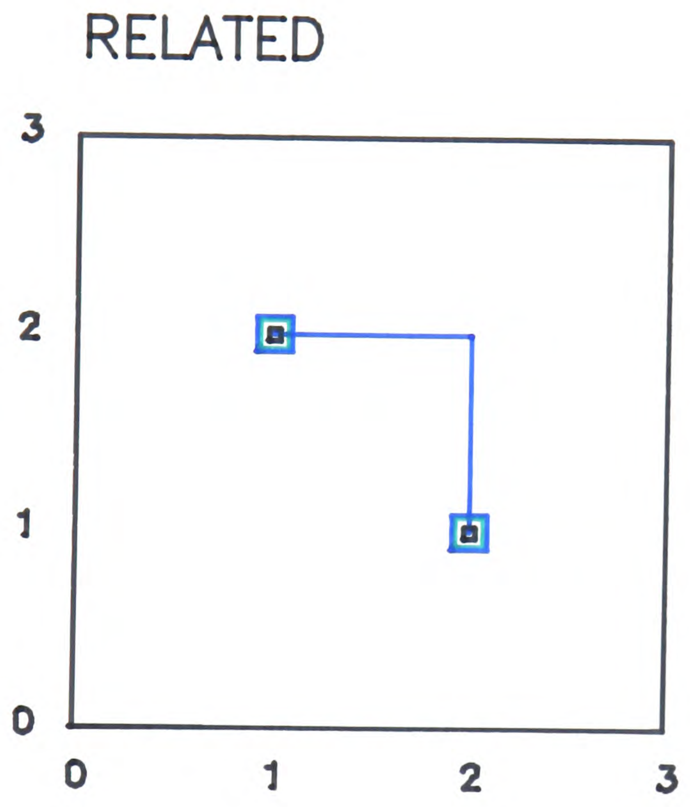


Figure 4—5: A Group of Related Components

identifier. If parameters are not specified, they implicitly take on default values. The available parameters are:

(a) For all types of transistor

w : channel width

l : channel length

Channel sizes are all in multiples of the minimum size, default values for both parameters are 1, except in the case of load transistors where the default for l is 4.

(b) For load transistors

Connections to load transistors cannot always be made unambiguously, since the source and drain connections appear on the same layer, and are not interchangable. It is therefore sometimes necessary to specify the direction in which the source connection will be made. This is done by specifying a value for the parameter SOR. Possible values are 0,1,2,3 and these have the effect of placing the source connection facing east, north, west, south respectively. Component orientation is discussed more fully in section 4.7.

(c) For MWIRE

It is sometimes necessary to make metal wires wider than the minimum possible width. This can be done by specifying a value for the parameter 'W'. The value of the parameter is the width in integral multiples of the minimum width. The default value is 1.

Users might also wish to make wires on other layers wider than the minimum possible width. The prototype implementation of VIRGIL currently only supports the specification of widths for metal wires but such specification could easily be extended to other wires if necessary.

4.4.3 An Example Leaf Cell Description

Figure 4-6(a) shows a typical VG leaf cell description, in this case a shift register cell. Figure 4-6(b) shows a graphical representation of the same cell.

4.5: VG Composition Cells

By a series of successive compositions, a collection of leaf cells can be arranged so as to produce a complete circuit. Composition cells define the nature of these compositions.

```
Leaf Cell  SHIFT = (0,0,4,4)
{A single stage of a shift register}

{Ground Lines}
gnd.e: mport @ (4,0)
gnd.w: mport @ (0,0)
gnd.s: nport @ (1,0)
mwire @ gnd.e -> gnd.w

{Power Lines}
vdd.w: mport @ (0,4)
vdd.e: mport @ (4,4)
vdd.n: nport @ (1,4)
mwire @ vdd.e -> vdd.w

{Inverter}
nm @ vdd.n
nm @ gnd.s
nwire @ vdd.n -> gnd.s
load @ vdd.n + (0,-1)
pulldown: ntype(w=2) @ gnd.s + (0,1)

{Input signal}
in: pport @ (0,1)
pwire @ in -> pulldown

{Clock Line}
clock.n: pport @ (2,4)
clock.s: pport @ (2,0)
pwire @ clock.n -> clock.s

{Pass transistor and output signal}
pass: ntype @ (2,2)
contact: buried @ (3,1)
out: pport @ (4,1)
nwire @ (1,2) -> pass -> (3,2) -> contact
pwire @ contact -> out

END
```

Figure 4–6(a): Textual Representation of Shift Cell

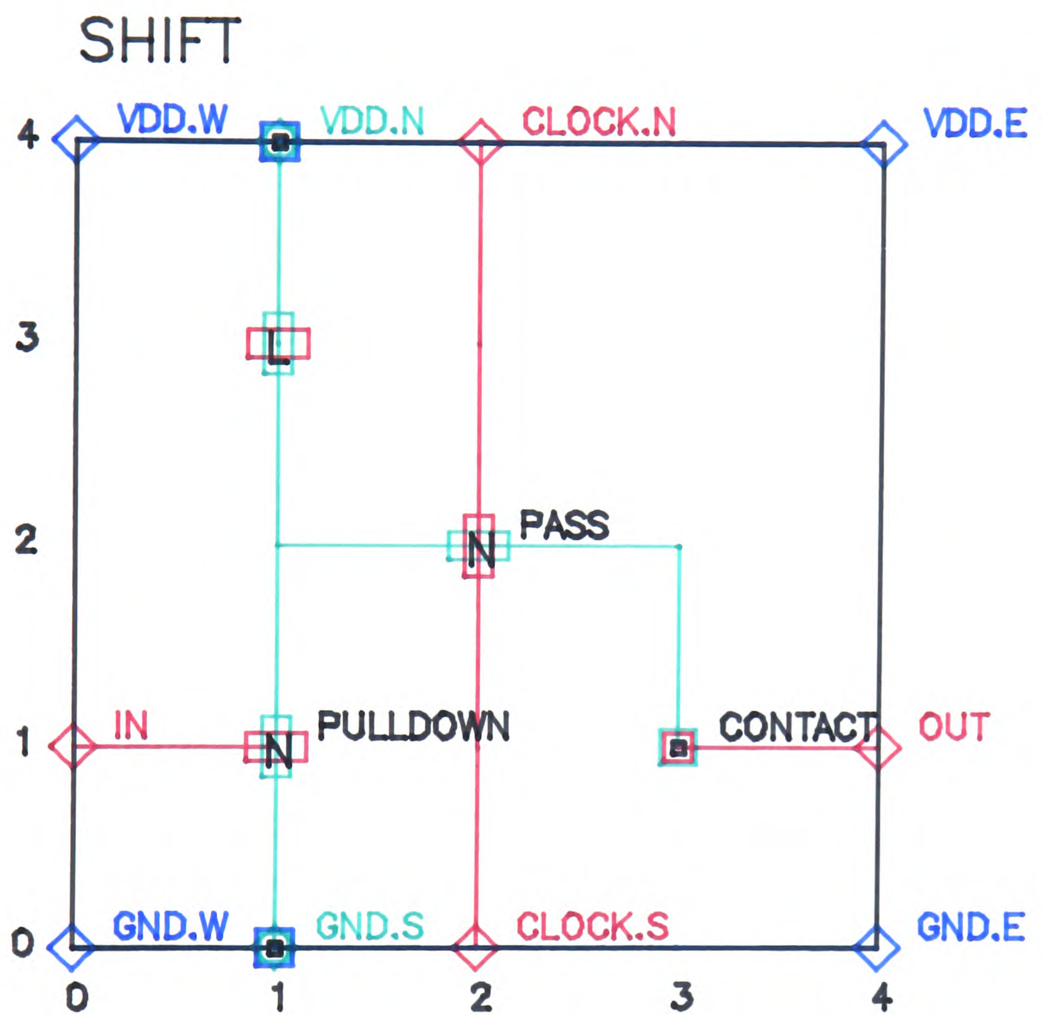


Figure 4–6(b): Graphical Representation of Shift Cell

4.5.1: Header and Terminator Statements

All the composition operations needed to produce a complete circuit are grouped together in a COMPOSITION CELL. Such a cell is delimited by a header of the form:

```
      header = "COMPOSITION" "CELL" identifier
```

and by a terminator of the form:

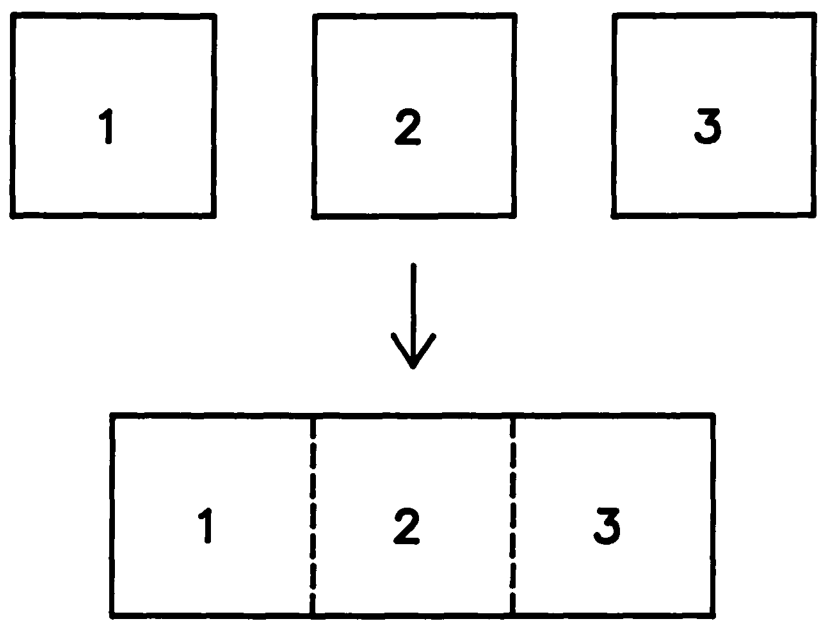
```
      terminator = "END"
```

4.5.2: Basic Composition Operators

The simplest form of cell composition is by abutment, i.e. by placing cells next to each other, so that matching connections line up. All composition of cells in VG is done by abutment. Such abutment may be done in the vertical or horizontal direction as shown in figures 4-7 and 4-8.

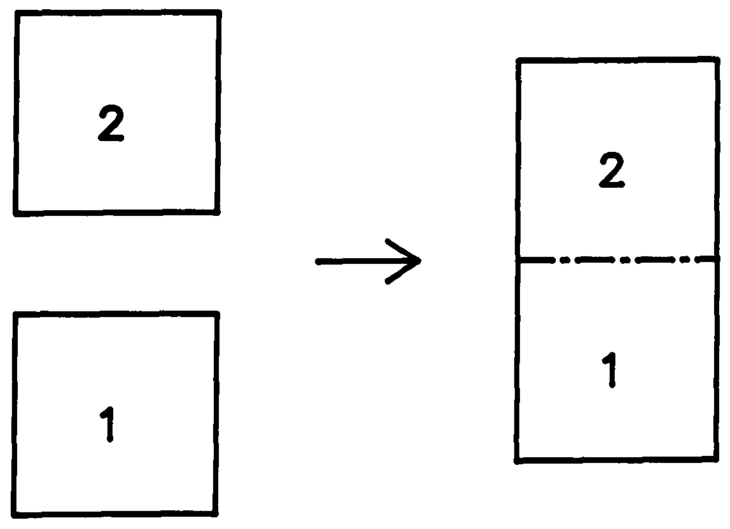
When cells are joined by abutment, they combine to give a single larger cell. Abutment of more than two cells can be considered as a series of simple two-cell abutments. The composition constructs in the VG language supports composition by abutment through the use of two composition operators:

```
>> : compose by horizontal abutment
^^ : compose by vertical abutment
```



NEWCELL = >> CELL1 >> CELL2 >> CELL3

Figure 4–7: Horizontal Abutment



NEWCELL = ^^ CELL1 ^^ CELL2

Figure 4–8: Vertical Abutment

The form of these composition statements is:

```
statement = identifier "=" cell_call { cell_call }
```

where

```
cell_call =  operator   cell_name
operator = ">>" | "^^"
```

Such a statement has the effect of creating a new cell, called an intermediate cell, with its name given by the identifier, which is composed of the cells, identified by cell_name, abutted left to right in the order given, for horizontal composition, or bottom to top for vertical composition.

Thus the statement:

```
Newcell = >> cell1 >> cell2 >> cell3
```

would give the composition shown in figure 4-7.

Similarly, a vertical composition statement :

```
Newcell =  ^^ cell1 ^^ cell2
```

would give the composition shown in figure 4-8.

Multiple instances of a single cell can be composed together, e.g.

```
OneBitShift = >> SHIFT >> SHIFT
```

Previously constructed intermediate cells, as well as leaf cells, can be composed together to give new intermediate cells. Leaf cells and intermediate cells may be mixed in the same composition statement. Vertical and horizontal composition cannot be mixed within a single statement.

The definition of a complete circuit then consists of a number of LEAF CELL definitions and a single COMPOSITION CELL definition. The various levels of hierarchy in the composition of cells to produce a complete circuit are supported by the various intermediate cells defined within the composition cell. The last intermediate cell defined in the composition cell is taken to be the definition of the complete circuit.

The composition of cells shown in figure 4-9 would be defined by the following composition cell:

```
Composition Cell Example
  cell6 = >> cell1 >> cell1
  cell7 = ^^ cell6 ^^ cell2 ^^ cell2
  cell8 = >> cell3 >> cell7
  cell9 = ^^ cell8 ^^ cell4
  example = >> cell9 >> cell5
End
```

Note that it is not possible to describe a composition of cells such as that in figure 4-10 as a set of vertical and horizontal compositions, however such compositions would rarely be encountered in practice.

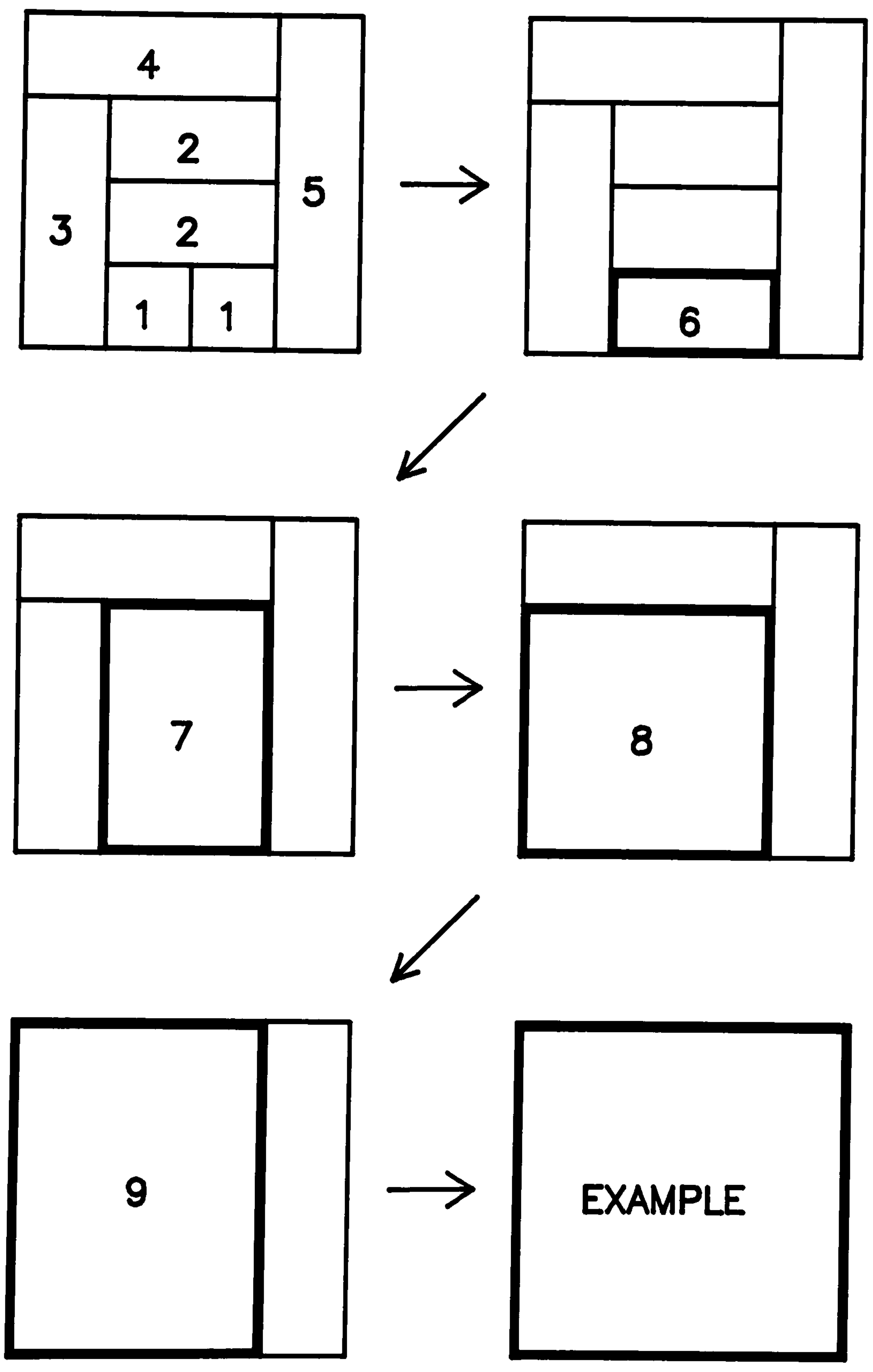


Figure 4–9: A Series of Successive Compositions to Produce a Complete Circuit

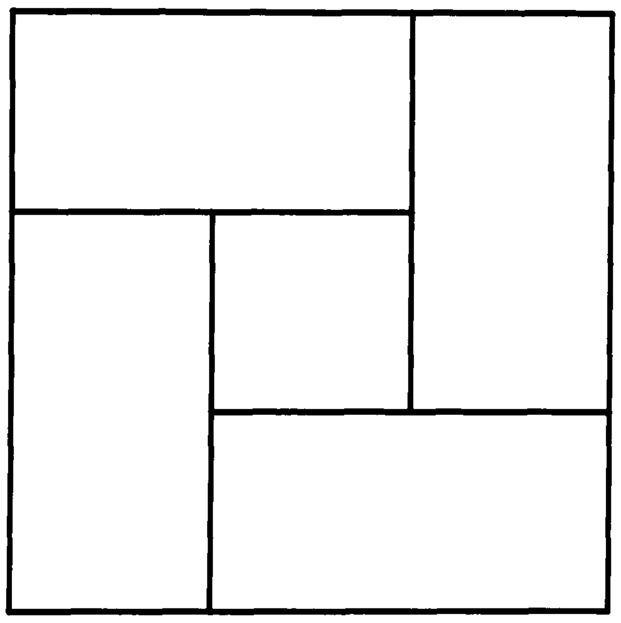


Figure 4-10: A Composition of Cells not Describable in VG

4.5.3: Rotation and Reflection

It is often desirable to be able to rotate or reflect cells before composing them. This is supported by a more general form of composition statement:

```
   statement = identifier "=" cell_call {cell_call}
where
   cell_call = operator cell_name [rotation] [reflection]
   operator = ">>" | "^^"
   rotation = "@" integer_expression
   reflection = "INX" | "INY"
```

The amount of rotation, if any, is in integral multiples of 90 degrees anticlockwise. Thus '@ 3' would correspond to 270 degrees anticlockwise rotation. Reflection, which is always done after any rotation, may either be about the X axis (INX) or about the Y axis (INY). Figure 4-11 shows some examples of rotation and relection of a single cell called CELLA.

A composition statement corresponding to the arrangement (of the same CELLA) shown in figure 4-12 would be:

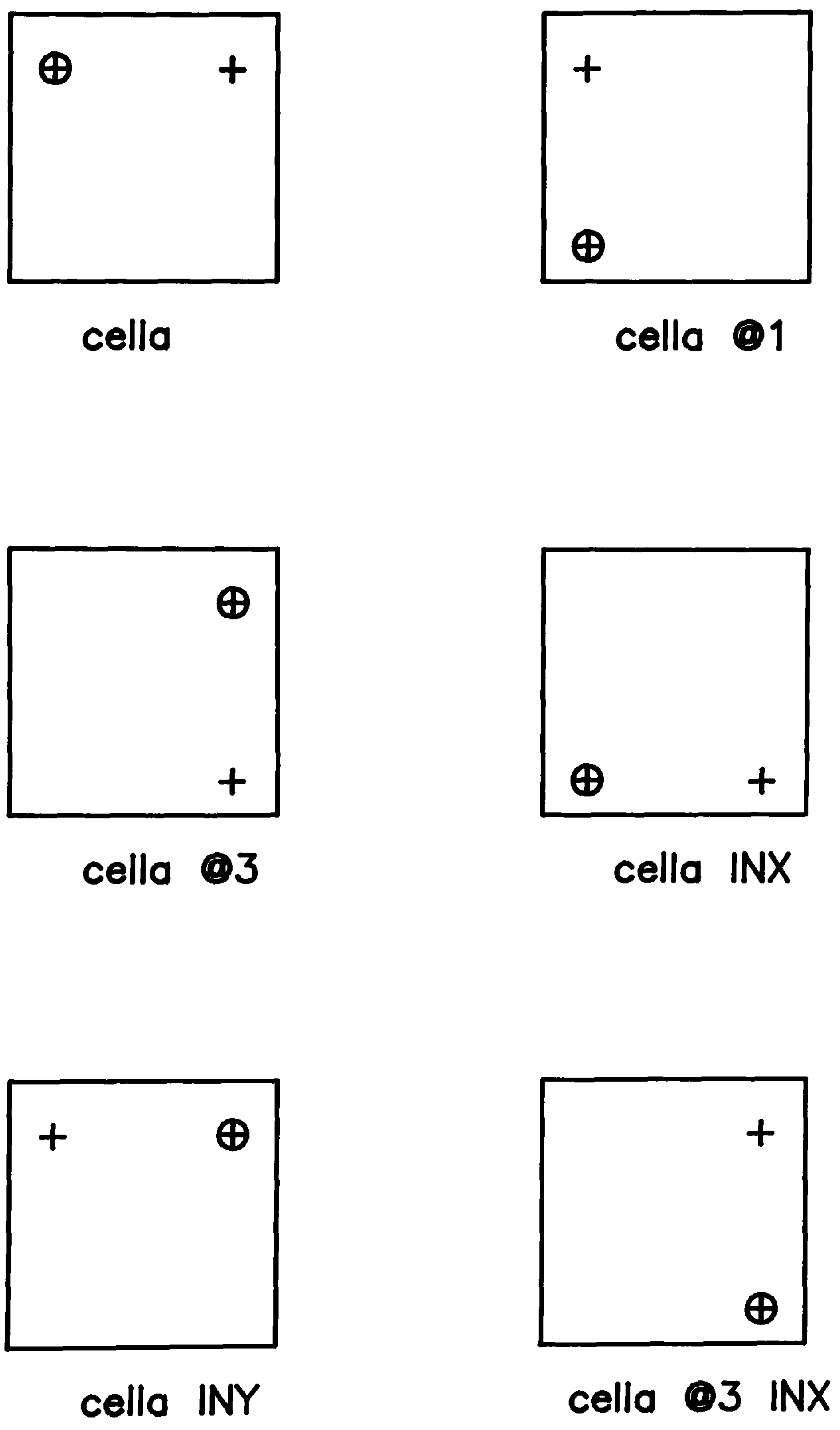
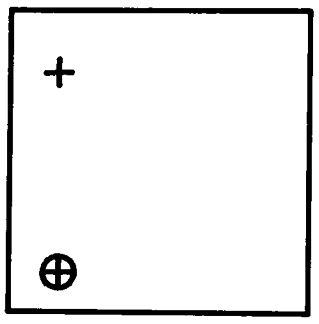
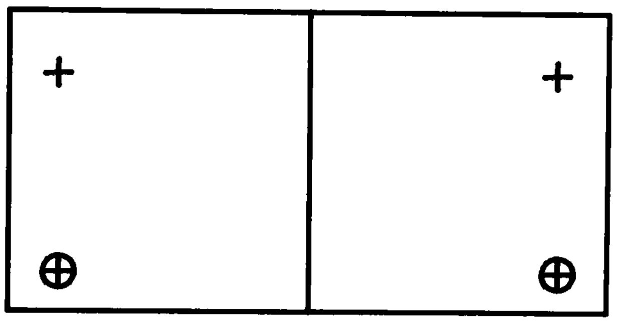


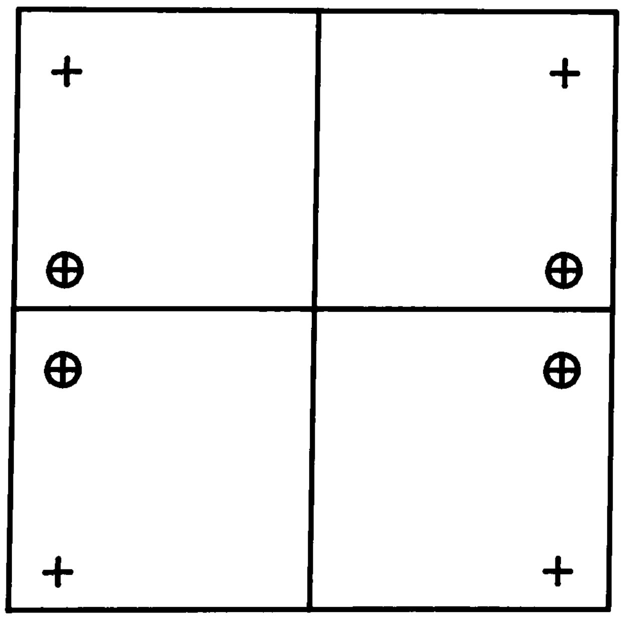
Figure 4-11: Examples of Rotation and Reflection



cellb = >> cella @ 1



cellc = >> cellb >> cellb INY



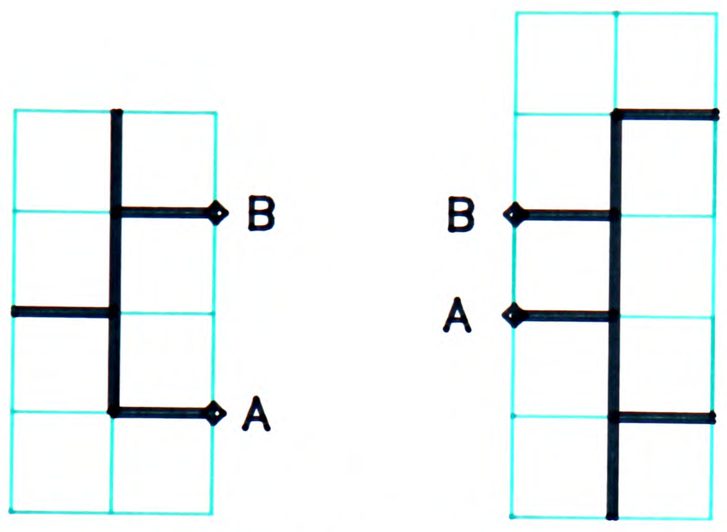
square = ^^ cellc INX ^^ cellc

Figure 4—12: Composition of Rotated and Reflected Cells

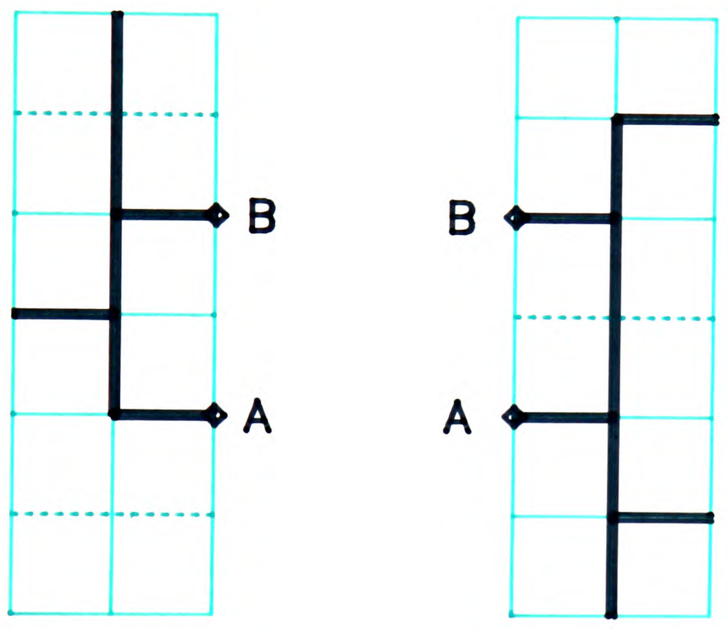
```
Composition Cell Square
 cellb = >> cella @ 1
 cellc = >> cellb >> cellb INY
 square = ^^ cellc INX ^^ cellc
End
```

4.5.4: Port Hiding & Renaming

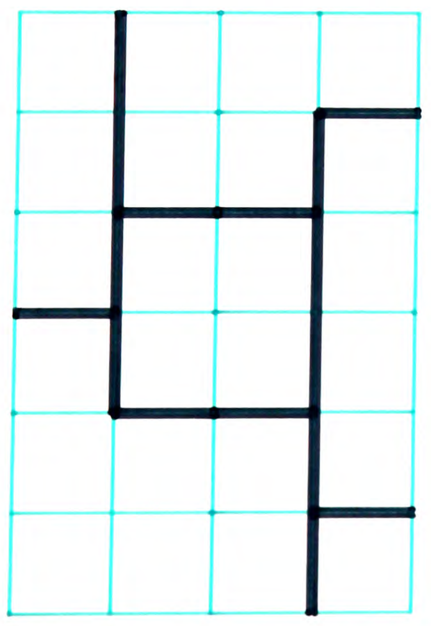
The most important task in composing cells together is to ensure that connections are made correctly between the cells. The points where connections may be made to leaf cells have already been explicitly defined by the definition of PORTS. It is by matching ports that correct connections between cells are made. For correct matching, the ports must match both in type and position. Since components have been laid out on a virtual grid, it is only the relative ordering of components, and thus only the relative ordering of ports to be matched which is significant. The virtual grids for the connecting cells can be stretched, by the inclusion of extra grid lines, to bring ports into alignment, as shown in figure 4-13. Stretching can also be used to match the overall length of the abutting sides of the cells. This ensures that the union of the two rectangular cells is itself a rectangle.



(a) Cells to be Composed



(b) Ports Aligned and Cell Heights Matched by Adding Grid Lines



(c) Cells Joined

Figure 4–13: Cell Stretching During Composition

In addition to the constraint that connections must be between ports of the same type, it would be advantageous to have some additional constraints which help ensure that connections are made correctly. Such a constraint has been made in VG, and it is that matching ports must match in name as well as position and type. In order to allow some flexiblity port names are said to match if the roots of the port names match (remember, names can have the form root.extension). Unnamed ports match any port of the correct type.

It is sometimes desirable not to make a connection to a port, e.g. a cell might have multiple ports for connection to power, only one of which need actually be externally connected. The port matching process, however, seeks to match all ports on the abutting edges of cells. To avoid a port being considered for such matching, it can be hidden, using the port hiding construct [Cardelli 81].

It is also sometimes useful to connect together ports with different names. Thus the language allows ports to be renamed before composition so that they will subsequently correctly match when connected. Both port hiding and port renaming are achieved by adding a renaming list to the composition construct. The most general form of the composition statement is then:

```
statement = identifier "=" cell_call {cell_call}
```

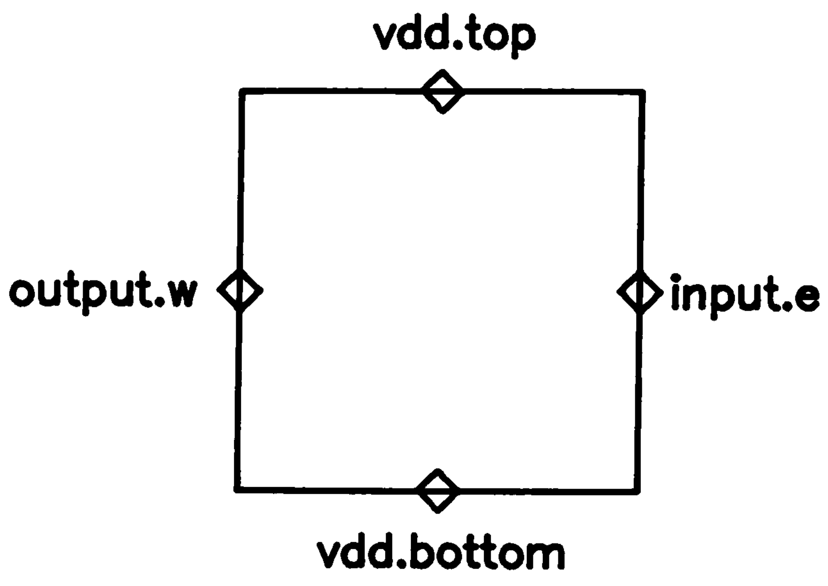
where

```
   cell_call = operator cell_name {renaming}
                    [rotation] [reflection]
   renaming = "/" port_name [ "=" new_port_name]
```

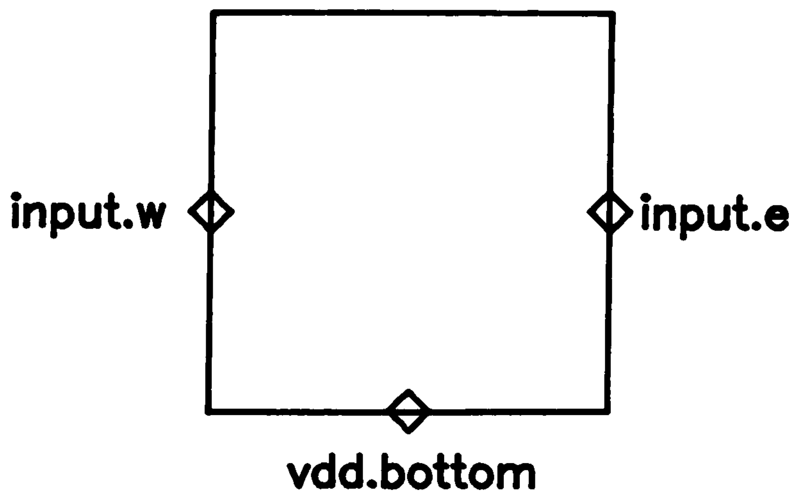
The effect of each renaming is to either change the name of the port called port_name to new_port_name if this is present, or to hide the port if no new_port_name is specified. Figure 4-14 shows the effect on the ports of a cell by certain renamings.

When two cells are composed, the ports visible on the new cell are all those ports on the old cells except those on the abutting edges. Thus joining instances of the same cell may lead to an intermediate cell with several instances of the same port name. Therefore port renaming and port hiding might not be able to be applied unambiguously to intermediate cells. These operations are therefore only allowed to be applied to instances of leaf cells. If port hiding or renaming is required with intermediate cells, then it must be done to the constituent leaf cells as they are being composed to give these intermediate cells.

The ideas of port hiding and renaming are similar to those used in more abstract circuit description languages such as "Sticks and Stones" [Cardelli 81] and CIRCAL [Milne 83a].



CELLA



CELLA /VDD.TOP /OUTPUT.W=INPUT.W

Figure 4-14: Port Hiding and Renaming

4.6: VIRGIL - An Idiom Description Language

The similarities between textual circuit descriptions and programming languages have already been noted. It is by adding programming language constructs, such as selection, repetition and parameterisation, to the virtual grid description language VG that the idiom description language VIRGIL will be developed.

Before deciding on a method for adding such constructs, it is worth noting a fundamental difference between VG and programming languages. VG is not "executed" in the sense that a programming language is executed. There is no flow of control in a VG description, rather there is a static correspondence between the textual description and the virtual grid circuit it describes. An instantiation of an idiom corresponds to a particular virtual grid circuit, and so also corresponds to the textual description of that circuit. Instantiating an idiom can then be thought of as the production of a textual description of a circuit in a language such as VG.

Since the form of an idiom instantiation will be a textual description, selection, repetition and parameterisation are implemented as basically textual operations. The process of idiom instantiation is then similar to that of "macro expansion" in some programming languages, especially assembly languages.

4.6.1: Parameterisation

If an idiom is to give different circuits for different sets of input data then there must be some method of specifying the values of this input data. This is done by the use of parameters, which are identifiers which appear in a VIRGIL idiom description, and which may take a range of different values. Parameters may be applied to both leaf cell and composition cell definitions.

Parameters may be one of two basic data types - integer and boolean. Parameters for use in composition cells may also be arrays, of up to six dimensions, of these basic types. Array subscripts may be of either basic data type.

Parameters are declared in the cell header statements. At the time they are declared they may be assigned a default value. If a default value is supplied for an array it applies to all elements of that array. The form of the header statements is then:

```
  header = "LEAF" "CELL" cell_name [parameter_list]
          "=" bounds
and
header = "COMPOSITION" "CELL" cell_name [parameter_list]
where
  parameter_list = "(" parameter {"," parameter} ")"
  parameter = identifier ":" type ["=" default_value]
  type = simple |
         array "(" dim {"," dim} ")" "OF" simple
  simple = "INTEGER" | "BOOLEAN"
  dim = expression ".." expression
  default_value = expression
e.g.
  Leaf Cell Test (m:integer=4, n:integer=5) = (0,0,m,n
or
  Composition Cell Ctest (j:integer=4,
                          a:array(1..j) of integer)
```

Parameters can be used in subsequent parts of the same statement in which they are declared, or in any subsequent statement as part of an expression. Array elements are accessed by constructs of the form:

```
 element = array_name "(" expression {"," expression} ")"
e.g.
        A(i,2)
```

Given the declarations above, statements of the form:

```
MWIRE @ (0,0) -> (0,n) -> (m,n)
```

would be allowed. However, the main use of parameters is in the control of selection and repetition constructs, which are described later.

When a cell is instanced, such as a leaf cell being instanced in a composition cell statement, it is possible to supply values for its parameters. This is done using the same sort of construct as was used for supplying values for component parameters, viz.:

```
cell = cell_name ["(" assignment {"," assignment} ")"]
```

where

```
assignment = identifier "=" expression
```

e.g.

```
Test(n=4,m=5)
```

Since it would be clumsy to assign values to an array in this manner, arrays are not allowed in leaf cell descriptions.

If a parameter which is defined within a cell is not given a value when that cell is instanced, then it takes the default value supplied. Since parameters are explicitly named when they are given values, parameters can be listed in any order.

Assuming the above parameter definitions for the leaf cell TEST, possible composition cell statements might be:

```
    Cella = >> test(m=3,n=2) >> test(n=6,m=3)
    Cellb = ^^ test ^^ test(m=1)
    Cellc = >> test(n=2*j-1,m=j) >> test(n=a(1))
```

Values for parameters in composition cells determine the particular instantiation of an idiom, and are supplied by the design system user.

4.6.2: Selection

In the selection construct, portions of text are selected to be either included or not in the circuit definition, depending on the value of a boolean expression. The form of this construct is:

```
if = "[" "IF" expression "THEN" text1 ["ELSE" text2] "]"
```

If the expression evaluates to true, then text1 is chosen, if the expression to false, then text2 is chosen if it is present. The chosen piece of text is substituted for the entire selection construct.

E.g., assuming the existence of a boolean parameter P1, the following statement:

```
   [if p1 then mwire @ (2,2) -> (2,3) -
          else pwire @ (2,2) -> (2,3) ]
```

would be replaced by, if P1 were true:

```
       mwire @ (2,2) -> (2,3)
```

and if P1 were false by:

```
       pwire @ (2,2) -> (2,3)
```

The above selection could also be written:

```
 [if p1 then mwire else pwire] @ (2,2) -> (2,3)
```

The text which is selected can be as long or short as desired, provided identifiers, reserved words or numerals are not split by the selection construct. The above text could NOT be written as:

```
[if p1 then M else P]wire @ (2,2) -> (2,3)
```

Multiple lines of text are allowed, and line breaks in the text are significant. Thus:

```
      [if p1 then mwire
       else pwire] @ (2,2) -> (2,3)
```

would give, for p1 = true:

```
          mwire
           @ (2,2) -> (2,3)
```

which is incorrect.

4.6.3 Repetition

In the repetition construct, a given piece of text is repeated a given number of times. The repeated text is substituted for the entire repeat construct. The form is:

```
for = "[" "FOR" identifier "=" for_list "REPEAT" text "]"
where
      for_list = for_item {"," for_item}
      for_item = expression |
                     (expression ".." expression)
```

The for_list defines a set of values which are assigned, in turn, to the identifier. This identifier is a local variable which is declared by its use in the FOR construct, and whose scope is limited to the FOR statement. A for_item may be an expression or a pair of expressions denoting a range of values. For such ranges, the identifier is assigned the value of each element in the range, starting with that given by the first expression, up to that given by the last. E.g.

```
        [ for i = 1,3,1,2..5,3..0 repeat text ]
```

would result in 'i' taking the following values:

```
        1,3,1,2,3,4,5,3,2,1,0
```

and text would be repeated 11 times.

The identifier associated with the repetition construct may appear as part of expressions within the text, and for each repetition it will be replaced by the appropriate value. Thus:

```
        2  [for i = 3..5,-2..0,12 repeat + i]
```

would give the text

```
        2 + 3 + 4 + 5 + -2 + -1 + 0 + 12
```

As for selection, line breaks in text are significant.

Selection and repetition constructs may be nested, provided nested repetition statements have different associated identifiers. This can lead to quite complex statements, e.g.:

```
      ABC = [for i = 1..3 repeat -
              >> cella -
              [if i=1 then (first=true)] -
              [if i=3 then (last=true)] ]
```

would give

```
      ABC = >> cella (first = true) -
             >> cella >> cella (last=true)
```

4.6.4: Qualifiers

It is often desirable to build up rectangular arrays of cells by a pair of repetition statements, one of which builds up the rows, and another which composes the rows to give the whole array. The exact composition of individual rows might vary from row to row, so a separate intermediate cell would be needed for each. In such cases it would be nice to have an "array" of rows so each row could be called by the same name, but with a different "index". Such a notion is supported by the use of intermediate cell name qualifiers. Intermediate cell names can be subscripted by an integer expression, called a qualifier, using a construct of the form:

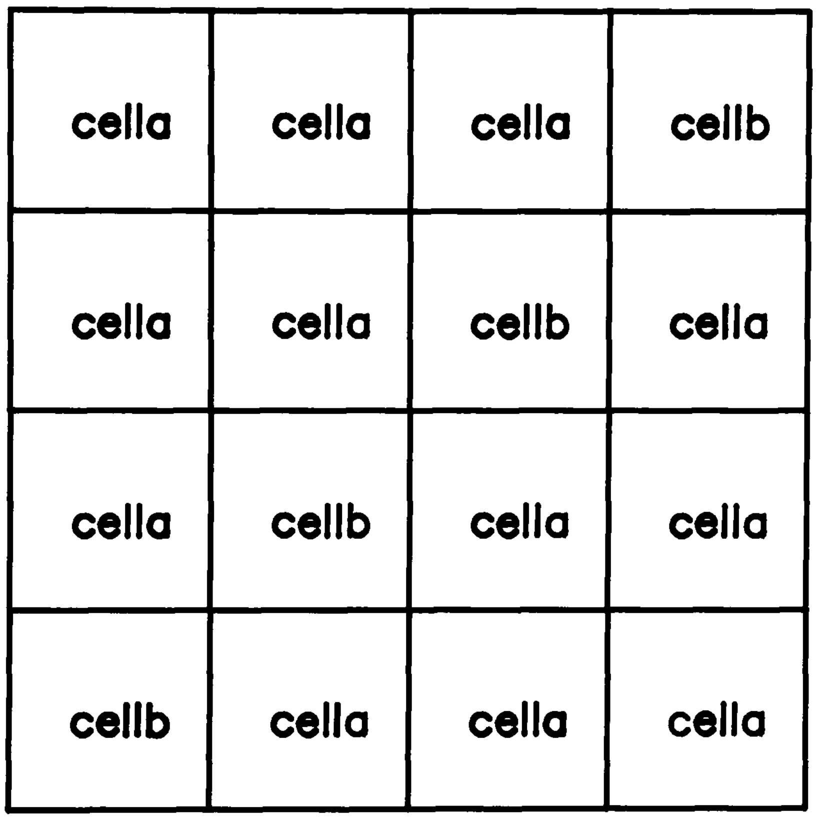
```
    qualified_name = name "_" expression
e.g.
          cella_i,  cella_i+1, cella_6
```

The composition shown in figure 4-15 could thus be constructed by the following statements, with N taking its default value of 4.

```
Composition Cell Block (N:integer=4)
  [for i = 1..n repeat
    row_i = [for j = 1..n repeat -
              >> [if i=j then cellb else cella] ]
   ]
   block = [for i = 1..n repeat ^^ row_i]
End
```

4.7: Circuit Connections and their Validity

The previous sections have described how components may be arranged on a virtual grid to produce a circuit. However, arbitrary arrangements of circuit elements do not always yield valid circuit constructs. By examining the way interconnections are implicitly made between components, it is possible to identify many invalid circuit situations. By checking a circuit for such invalid situations, many possible errors can be identified early in the design cycle.



cella	cella	cella	cellb
cella	cella	cellb	cella
cella	cellb	cella	cella
cellb	cella	cella	cella

Figure 4–15: A Composition of 'cella' with 'cellb' on the Diagonal

Items which are placed at grid points may be considered as multiport devices. Device ports are referred to as pins. Each pin exists on a certain layer, and on a certain side of the device. Each point within a virtual grid has up to four immediate neighbours, in the directions of the coordinate axes. Wires can exist between a given grid point and any of its nearest neighbours. Correct connections between wires and devices are made by insisting that wires which approach a grid point from a certain direction must connect to a pin on that particular side of a device at that grid point. e.g. a metal wire approaching from the right side of a grid point must connect to a metal pin on the right side of some device which exists at that grid point.

To allow the above requirement to be met it is necessary to introduce the notion of null devices. There is one type of null device corresponding to each different type of wire. A null device of a certain type has one pin of the corresponding type on each of its four sides. The null device connects together any wires connected to its pins. A null device of a particular type implicitly exists at a grid point if there is a wire of that type approaching the grid point, and no other devices at that grid point have any pins of that type. The usual physical realisation of a null device is a minimum size square of wire of the appropriate type.

As an example, consider a metal wire running from point (1,1) to point (3,1). Also assume there are no grid items explicitly placed at point (2,1). Then, at point (2,1), a metal null device implicitly exists. Both the metal wires approaching point (2,1), viz. one from the left and one from the right, now connect to pins of the null device, and the requirement that all wires approaching the point be connected to pins of the correct type is satisfied.

Another useful notion, which helps development of a consistent analysis, is that a VIRGIL port implies that a wire of the same type as the port is approaching the grid point from outside the cell. Thus ports can be thought of as a special type of wire, rather than as grid items.

The available types of grid items are then categorised as:
Transistors: NTYPE, PTYPE, DEPLETION
Load Devices: LOAD
Substrate Contacts: PSUB, NSUB
Interwire Contacts: PM, NM, DM, BUTTING, BURIED
Null Devices

A single grid item may be available in a number of different "arrangements". An arrangement is defined by the pins available on each of the four sides of the grid item. In the case of active devices, viz. transistors and load devices, different pins connect to different parts of the device. These different parts are given different names, "gate", "source" and "drain". In a load device, there is no pin named "gate", since all pins named "source" are internally connected to the gate of the device as well as the source. Pins are identified by the type of wire that can connect to them, the part of the device they connect to, and the side of the device on which they are available. In the case of null devices and contacts, all pins are connected to effectively the same point, named "common".

Figure 4-16 shows diagrams of all the possible arrangements for each different device. Note that the drain and source connections of transistors (not load devices) are symmetrical, thus halving the number of functionally distinct arrangements of these devices. A rotation of one arrangement is considered a separate arrangement.

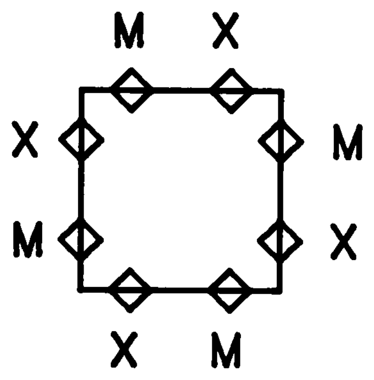
Given this set of arrangements it is possible to produce a relatively simple set of conditions to ensure validity of circuit constructs. Recall first the conditions under which a null device exists at a grid point. A null device of a particular type exists at a point if a wire of that type approaches the grid point, and no other device at that grid point has a pin of that type (on any side). Thus if a metal wire approached a

DEVICE ARRANGEMENTS

M = metal; P = poly;
D = ptype diffusion; N = ntype diffusion

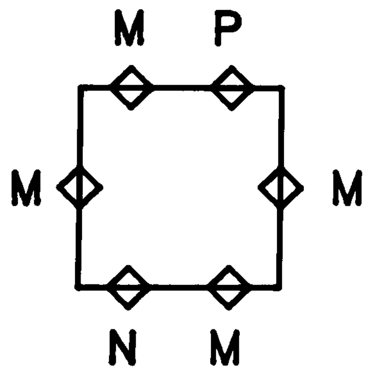
(1) PM : X <- P
DM : X <- D
NM : X <- N

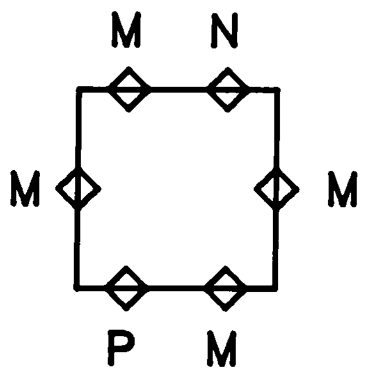
All pins connected to "common"

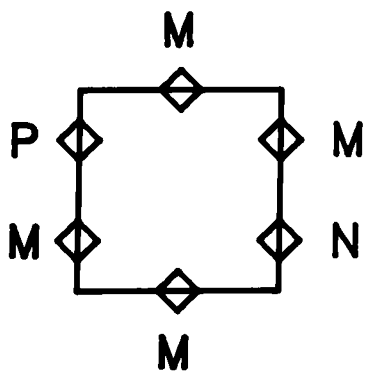


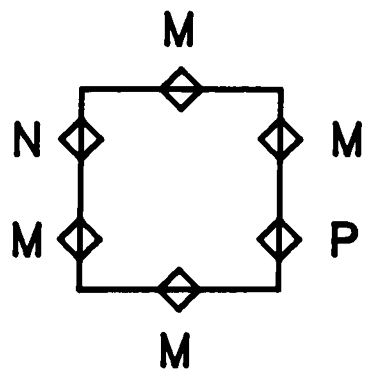
(2) BUTTING

All pins connected to "common"







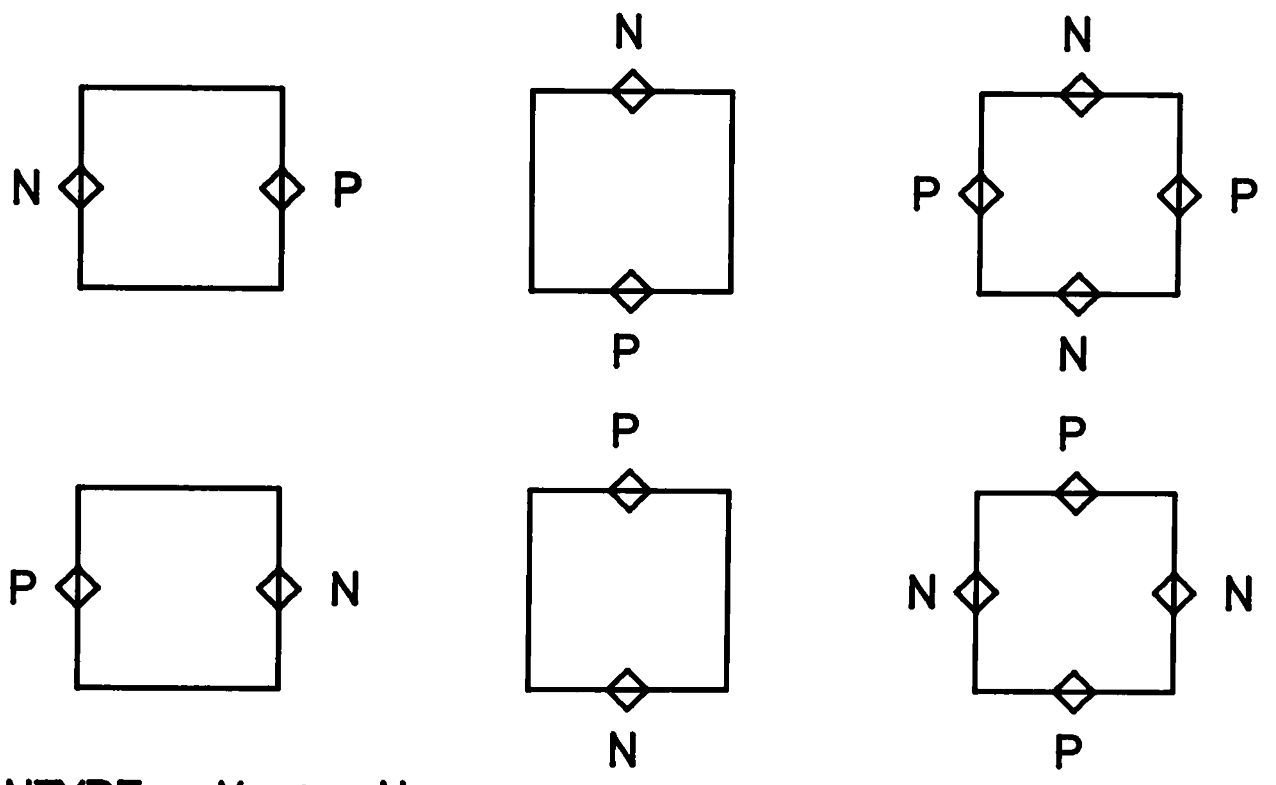


cont.../

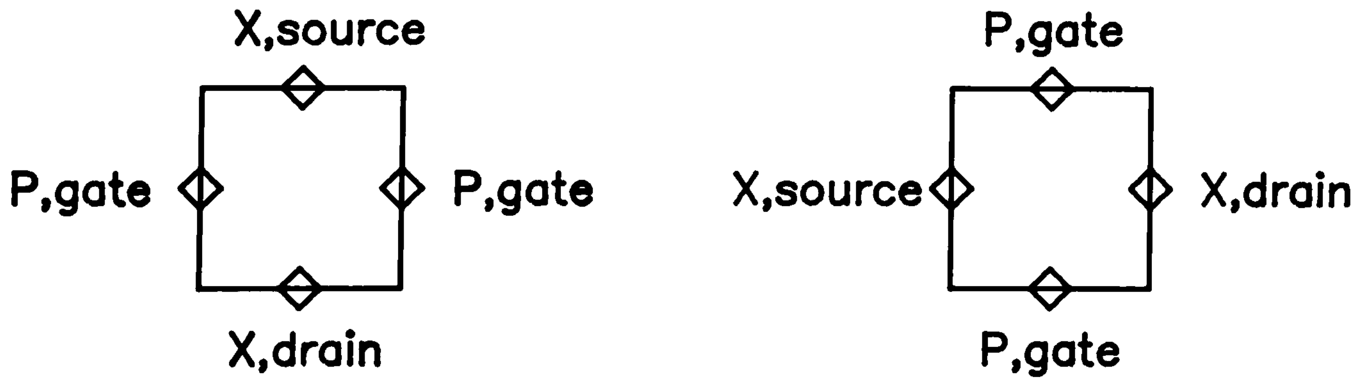
DEVICE ARRANGEMENTS (cont)

M = metal; P = poly;
D = ptype diffusion; N = ntype diffusion

(3) BURIED All pins connected to "common"



(4) NTYPE : X <– N
PTYPE : X <– D
DEPLETION : X <– N
"source" and "drain" may be interchanged

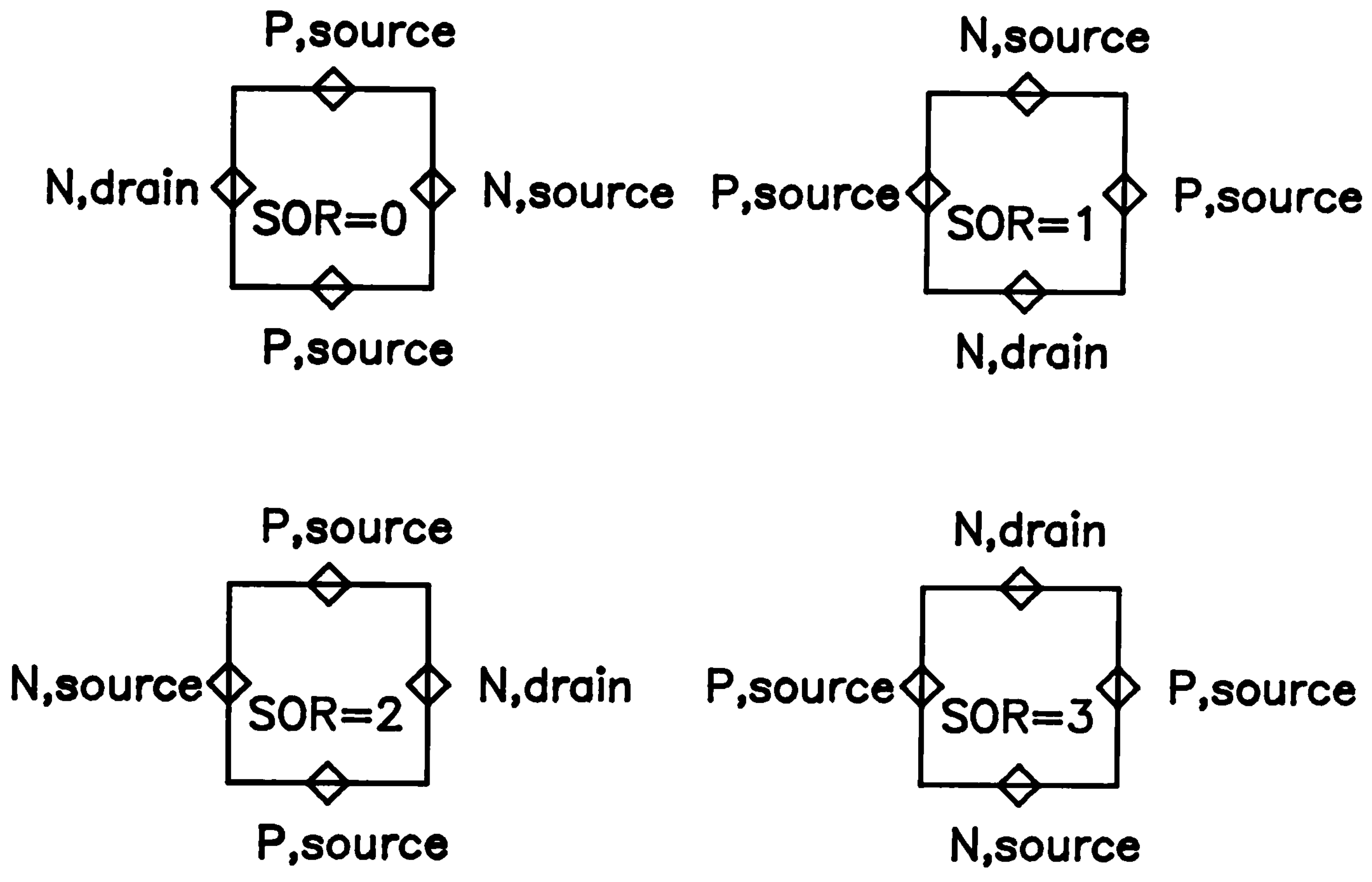


cont.../

DEVICE ARRANGEMENTS (cont)

M = metal; P = poly;
D = ptype diffusion; N = ntype diffusion

(5) LOAD
Appropriate SOR parameter values shown



(6) PSUB : X <- M
NSUB : X <- M

Null Devices
MWIRE : X <- M
NWIRE : X <- N
DWIRE : X <- D
PWIRE : X <- P

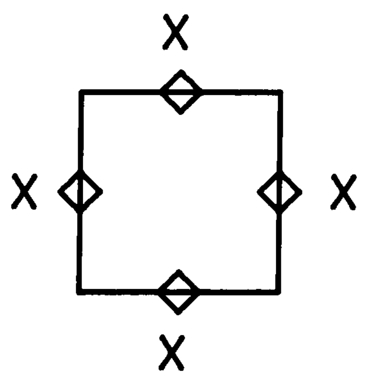


Figure 4-16: Device Arrangements

grid point where a buried contact existed, then a metal null device would also exist at that point since a buried contact has no metal wire pins.

If some set of devices (perhaps some null) exist at a grid point, and there is a set of wires which approach that grid point from specified directions, then these sets of components specify a valid circuit construct only if all the following conditions are met.

(1) It must be possible to choose an arrangement for every device, such that every wire can connect to a pin of the correct type and on the correct side of a device.

(2) For each device, there must be at least one wire connected to one member of each set of similarly named pins.

(3) There may never be more than two devices at any grid point. If there are two devices, then one device must be a metal wire null device.

(4) Every device, except substrate contacts, must have at least two wires connected to its pins. In the case of interwire contacts, at least two of these wires must be of different types. In the case of butting contacts at least one wire must be polysilicon and at least one must be diffusion. (Remember VIRGIL ports count as wires). A

single unconnected VIRGIL port is allowed as a special case.

Some examples of invalid circuit constructs and the manner in which the above conditions detect them are illustrated by the following examples, shown in figure 4-17.

(1) No arrangement of BUTTING can be chosen such that all wires can be connected to pins of the correct type, so condition 1 is violated.

(2) There is no wire connected to any of the "gate" pins, so condition 2 is violated.

(3) Poly crosses diffusion without a transistor. No explicit devices exist, therefore there is a poly null device and an n-type diffusion null device, so condition 3 is violated.

(4) No explicit devices exist at the grid point, therefore metal wire and polysilicon wire null devices exist. The metal wire null device has only one wire connecting to it, so condition 4 is violated.

The only interesting circuit construct which violates these simple conditions is a PM contact over a transistor gate. Such a construct is often disallowed in many MOS

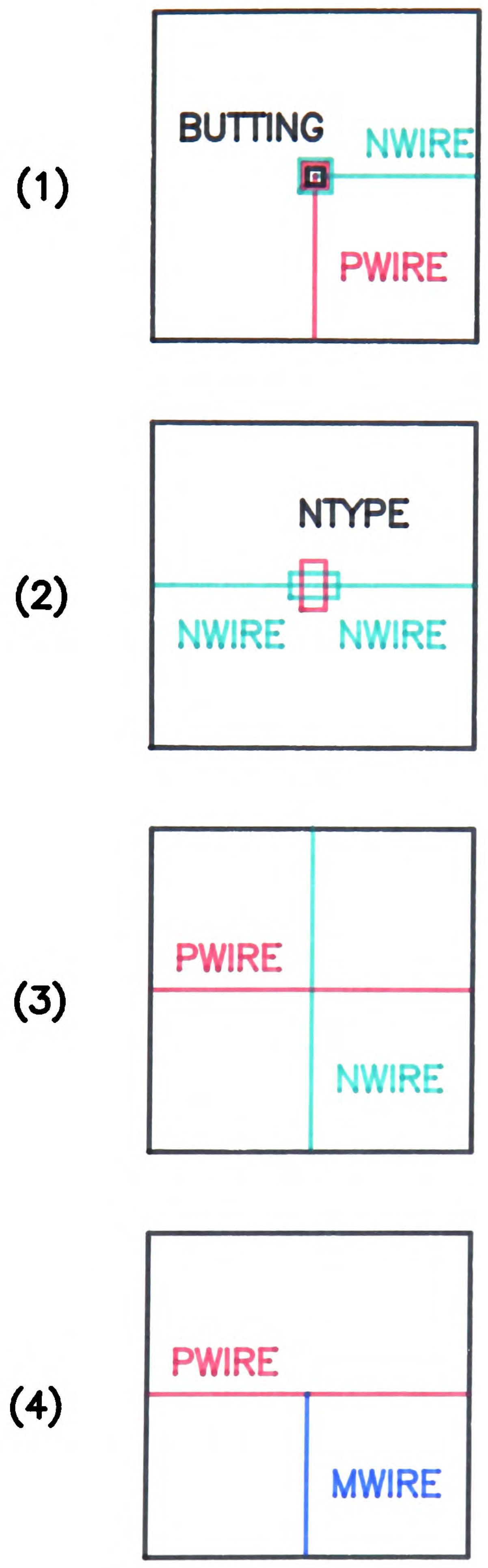


Figure 4—17: Invalid Circuit Constructs

design rules, and is unlikely to be allowed at all in small geometry technologies, so in many cases it is indeed an invalid circuit construct.

In a very few cases it is possible to find more than one possible arrangement of a device which satisfies the above conditions. In fact the only case where this happens is if two diffusion wires connect to the source and drain of a load device. In this case it is necessary to decide on which arrangement is chosen, since they are not functionally the same. This is done by specifying the direction of the source diffusion connection, using the SOR parameter described earlier. The four arrangements shown in figure 4-16 correspond to SOR values of 0,1,2,3 respectively. If unspecified, a SOR value of 0 or 3 is chosen (rather than 2 or 1 respectively).

In all other cases, a valid set of wires uniquely determine a device arrangement. Mostly, device arrangement is merely to do with the orientation of asymmetric devices. In the case of buried contacts, it also determines whether the contact is realised as a colinear or orthogonal buried contact.

4.8: An Example Idiom

A simple example of an idiom which illustrates many of the points raised in the preceding sections is now presented. More complicated examples will be examined in chapter 7.

Figure 4-18 shows a mixed notation description of a generalised two-phase shift register, including connection grids for power, ground and clocks. Figure 4-19 shows the subdivision of the generalised shift register into an array of VIRGIL leaf cells.

SHIFT is just the shift cell described in figure 4-6. SIDE, TOP and CORNER are wiring cells which make the necessary power, ground and clock connections. SIDE connects either power (on the right side of the array) or ground (on the left) lines together. TOP similarly connects different phases of the clock together at the top and bottom edges of the shift register. CORNER completes the rectangular array of cells, and allows common clock lines to pass out under the power line on the right of the cell.

The cells TOP and CORNER have been made parameterisable to allow connections to be either made or not made to certain clock lines. The VIRGIL descriptions

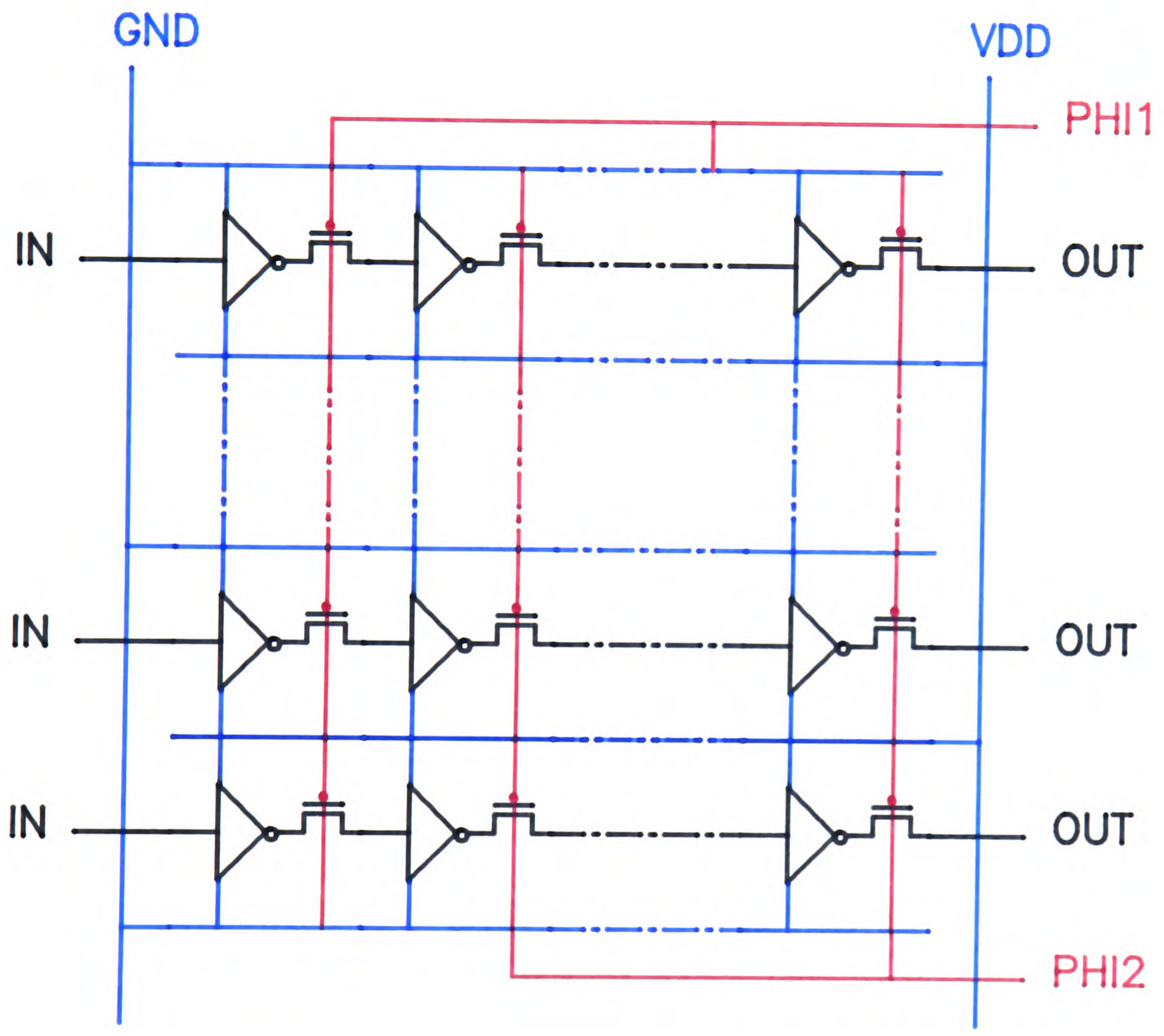


Figure 4–18: Shift Register Circuit

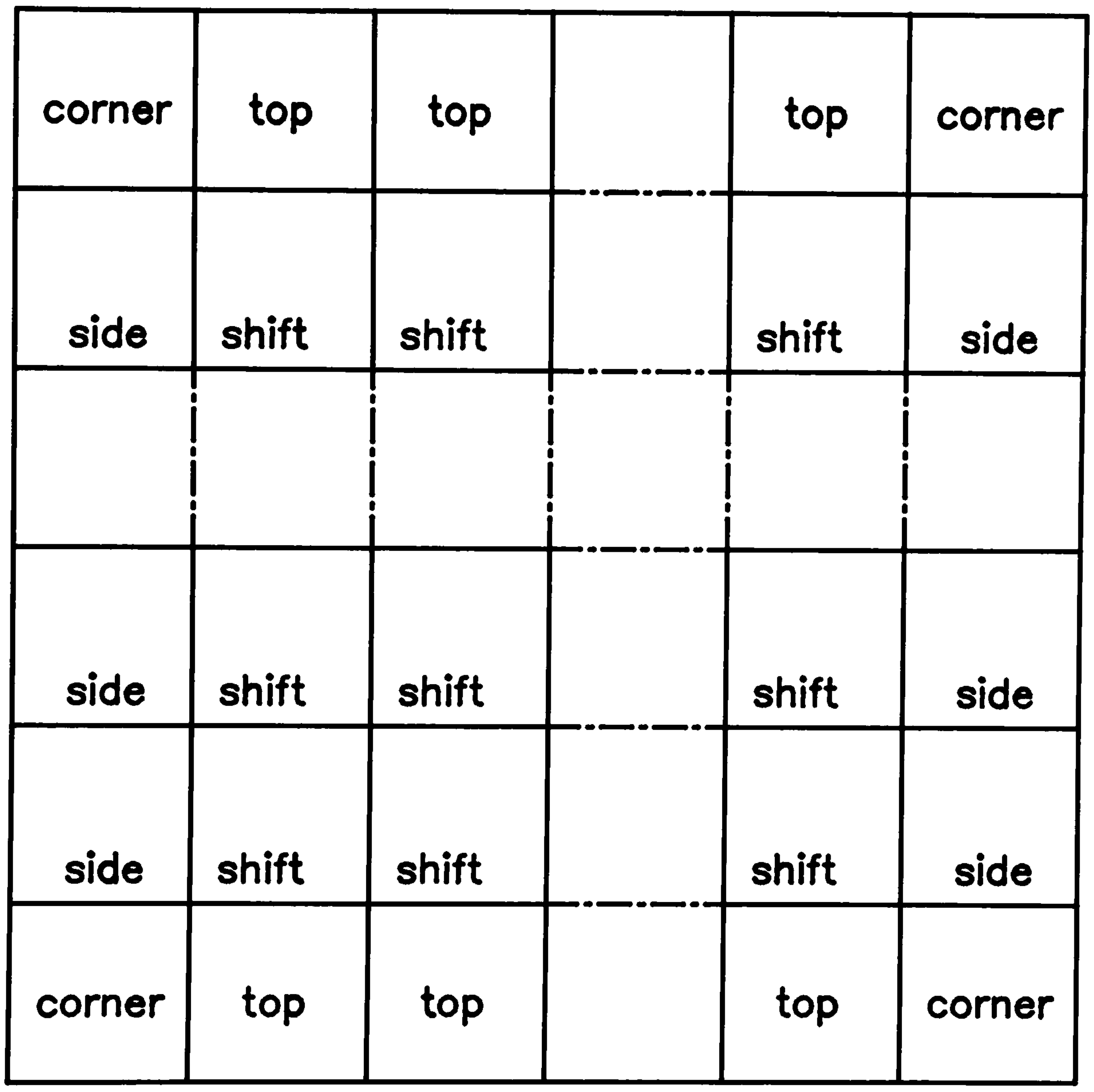


Figure 4–19: Shift Register Cells

of the various cells, (see figure 4-6 for SHIFT), and graphical illustrations of instantiations of the cells are shown in figure 4-20.

The manner in which the cells are composed together to produce a complete shift register is given in the VIRGIL composition cell definition shown in figure 4-21. Although quite simple to write, this cell definition is at first glance quite difficult to understand, and it is instructive to explain it in detail.

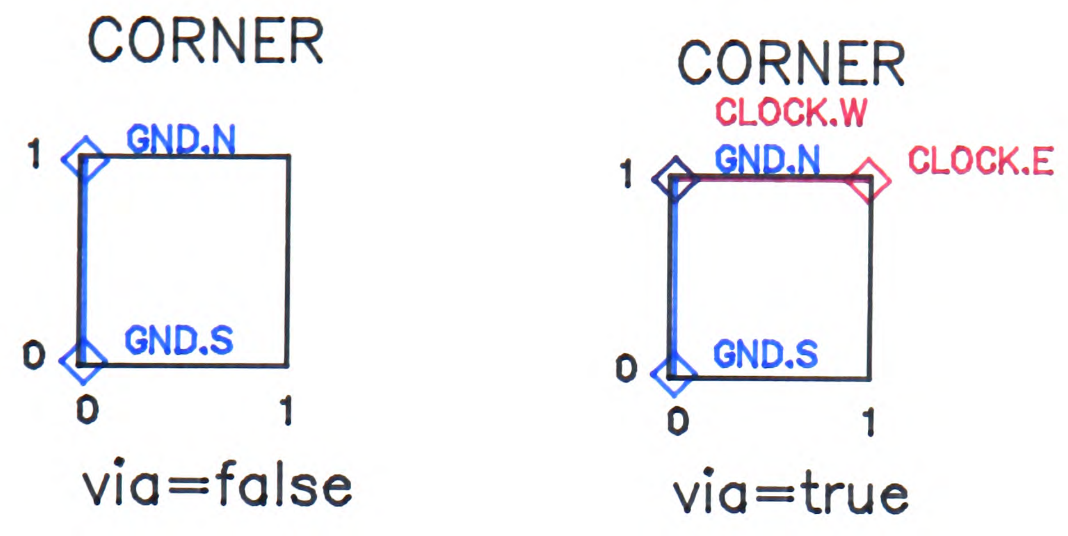
The shift register definition can best be explained by looking at one particular instantiation of the cell, in this case the instantiation with wide=2 and long=4.

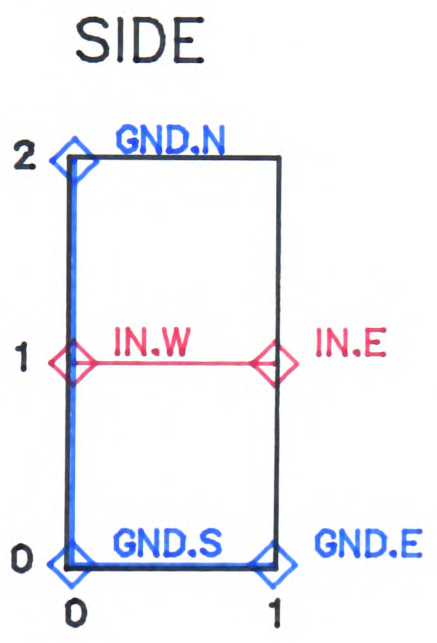
The cell definition makes extensive use of an arithmetic odd/even expression of the form (i-i/2*2=0) which yields TRUE if i is even and FALSE if i is odd. Such an expression is used to select every second element of some array for some special action, e.g. a common clock connection is made to every second phase of each row of the shift register. The textual form produced by this instantiation is shown in figure 4-22. The graphical representation of the complete shift register is shown in figure 4-23.

Note that certain ports, such as power ports on the left and ground ports on the right have been hidden so

```
Leaf Cell Corner(via:boolean=false) = (0,0,1,1)
  gnd.s: mport @ (0,0)
  gnd.n: mport @ (0,1)
  mwire @ gnd.s -> gnd.n
  [if via then
    clock.e: pport @ (1,1)
    clock.w: pport @ (0,1)
    pwire @ clock.e -> clock.w ]
End


Leaf Cell Side = (0,0,1,2)
  gnd.e: mport @ (1,0)
  gnd.s: mport @ (0,0)
  gnd.n: mport @ (0,2)
  mwire @ gnd.e -> gnd.s -> gnd.n
  in.e: pport @ (1,1)
  in.w: pport @ (0,1)
  pwire @ in.e -> in.w
End


Leaf Cell Top (join:boolean=false) = (0,0,3,1)
  clock.e: pport @ (3,1)
  clock.w: pport @ (0,1)
  clock.s: pport @ (2,0)
  nport @ (1,0)
  pwire @ clock.e -> clock.w
  [if join then pwire @ clock.s -> clock.s + (0,1) ]
End
```

Figure 4–20(a): VIRGIL Leaf Cell Descriptions





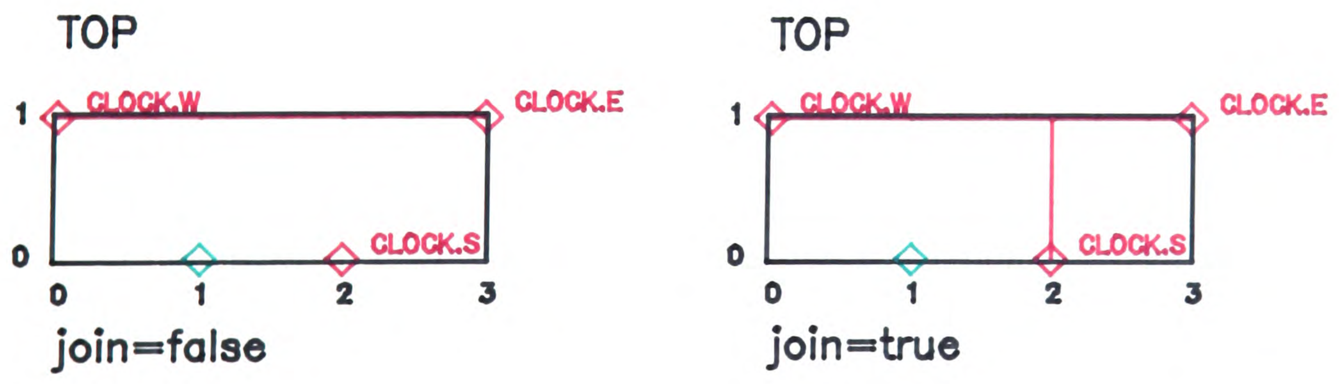


Figure 4-20(b): Graphical Representations of Leaf Cell Instantiations

```
Composition Cell SR (wide:integer=2, long:integer=4)

  Row = >> Side -
        [for i=1..long repeat -
        >> Shift/out=in [if i=1 then /vdd.w] -
                  [if i=long then /gnd.e] -
        ] >> Side/gnd.s=vdd/gnd.n=vdd/gnd.e=vdd  @ 2

  TopRow = >> corner [for i=1..long repeat -
           >> top (join = (i-i/2*2 = 1)) -
              [if i=1 then /clock.w] ] -
           >> corner (via = true)/gnd.s=vdd/gnd.n=vdd iny

  BotRow = >> corner [for i=1..long repeat -
           >> top (join = (i-i/2*2 = 0)) -
              [if i=1 then /clock.w] inx ] -
           >> corner (via=true)/gnd.s=vdd/gnd.n=vdd @ 2


  SR = ^^ BotRow [for i=1..wide repeat -
       ^^ row [if i-i/2*2 = 0 then INX] ] ^^ TopRow
 END
```

Figure 4-21: Shift Register Composition Cell Definition

```
COMPOSITION CELL SR

ROW =>> SIDE -
    >> SHIFT / OUT=IN / VDD.W  -
    >> SHIFT / OUT=IN  -
    >> SHIFT / OUT=IN  -
    >> SHIFT / OUT=IN / GND.E  -
    >> SIDE / GND.S=VDD / GND.N=VDD / GND.E=VDD @ 2

TOPROW =>> CORNER  -
     >> TOP (JOIN=(1-1/2*2=1))/ CLOCK.W  -
     >> TOP (JOIN=(2-2/2*2=1)) -
     >> TOP (JOIN=(3-3/2*2=1)) -
     >> TOP (JOIN=(4-4/2*2=1)) -
     >> CORNER (VIA=TRUE) / GND.S=VDD / GND.N=VDD INY

BOTROW =>> CORNER -
     >> TOP (JOIN=(1-1/2*2=0))/ CLOCK.W  INX -
     >> TOP (JOIN=(2-2/2*2=0)) INX -
     >> TOP (JOIN=(3-3/2*2=0)) INX -
     >> TOP (JOIN=(4-4/2*2=0)) INX -
     >> CORNER (VIA=TRUE) / GND.S=VDD / GND.N=VDD @ 2

SR =^^ BOTROW ^^ ROW ^^ ROW  INX ^^ TOPROW

END
```

Figure 4–22: Textual Form of Composition Cell Definition after Instantiation

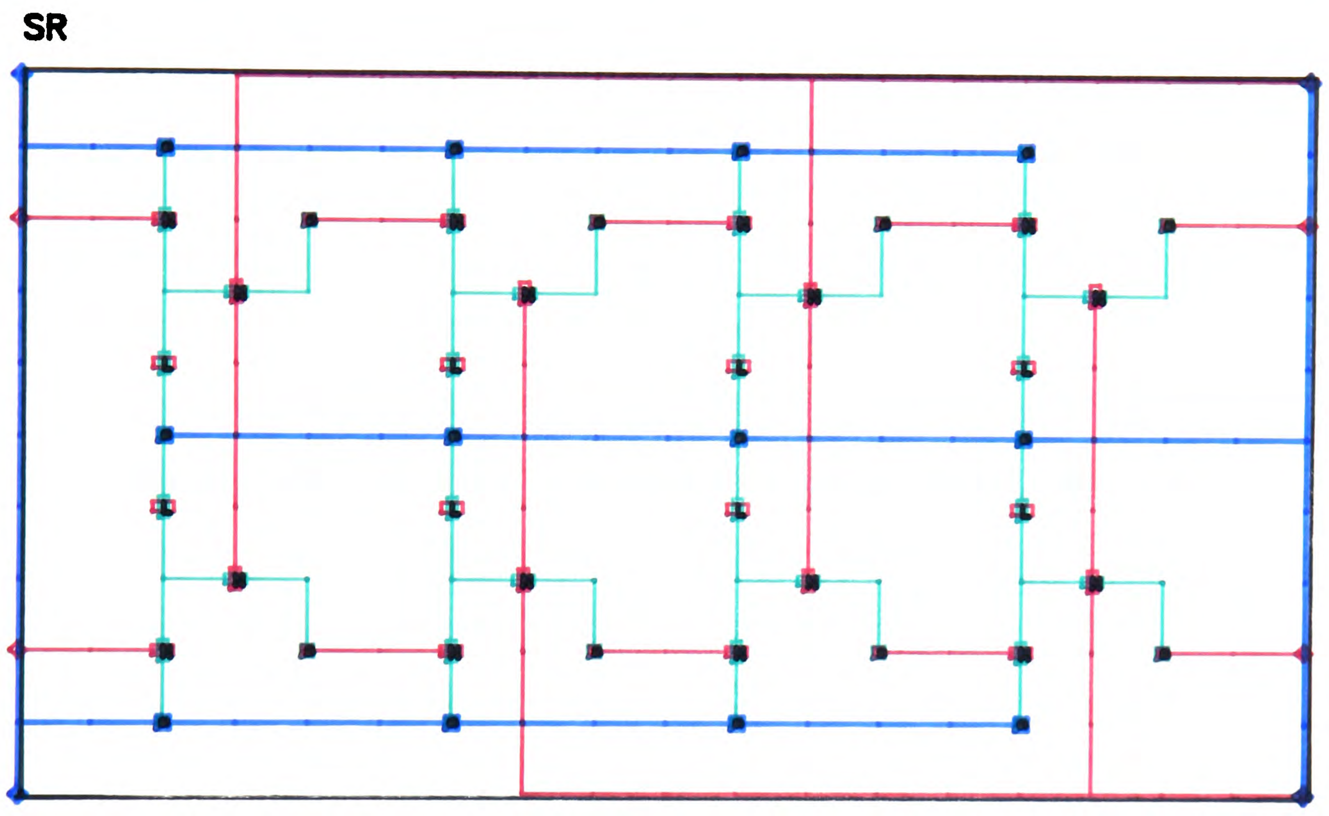


Figure 4–23: Graphical Representation of a Shift Register

that no connections are made to them. A different method has been used to hide unconnected clock lines. Such lines are connected to ports in TopRow and BottomRow which are then internally unconnected to the common clock wire.

The shift cell has been designed with power and ground connections at the top and bottom of the cell. When cells are composed, if alternate rows of cells are mirrored, it is possible to share power and ground lines, and the diffusion-metal contacts on them, between adjacent rows. This can be achieved in VIRGIL by placing the features to be shared on the very top and bottom grid lines, since these grid lines will be merged with those on the adjacent cell during composition. This merging can be seen in the VDD line in figure 4-23.

Another feature worthy of note is that of automatic wire trimming. Wires which are connected to ports subsequently hidden in composition operations are left "dangling" in the middle of the newly created cell. Such wires are automatically removed, so that the complete idiom instantiation still meets the validity requirements given in section 4.7. Examples are the clock and power lines in figure 4-23.

4.9: Summary

The VIRGIL language includes several features which distinguish it from other VLSI design languages, and which are worthy of further discussion.

Primarily, VIRGIL is a textual language for the description of idioms, but it can also describe circuits. This differs from most design systems which are primarily for the description of circuits, and often have a strong bias towards graphical entry of designs.

The syntax of VIRGIL has been designed to allow concise idiom descriptions, but not at the expense of being too cryptic. Examples are the use of syntactic "sugar" such as parentheses around coordinate pairs and arrows between coordinates in wire paths. While not strictly necessary, they help to make the design description more readable.

Selection and repetition have been implemented as textual operations, allowing selection and repetition down to the level of individual lexical entities. This allows for very concise idiom descriptions. For example, if a whole row of cells were identical except for some port hiding on the end cells, only the port hiding part of the cell call would need to be placed in a selection construct. Examples of this can be seen in the shift register description in figure 4-21.

The notion of device arrangements has been introduced. These provide a simple but powerful tool for the detection of invalid circuit constructs. Such detection is greatly aided by the structural information implicit in the design description. Such information is especially well handled by a virtual grid description, since the information needed to verify correct circuit constructs can be obtained by looking, independently, at each grid point and the grid lines leading from it. The detection of invalid circuit constructs is a valuable aid in quickly debugging a circuit description.

Device arrangements also allow, in most cases, the automatic orientation of non-symmetric devices such as transistors, and butting and buried contacts, meaning that the designer need only specify their position on the virtual grid and not their orientation.

Finally, because idioms are described at the sticks level, they are valid for a wide range of different fabrication processes. As will be shown in the next chapter, CMOS idioms especially can be designed to be valid in a wide range of different technologies.

5: STICKS COMPACTION OF THE VIRTUAL GRID

5.1: Background

Sticks compaction is the common name given to the translation between sticks level representations and mask level representations. Sticks are converted to mask descriptions by creating mask level equivalents of structural items in the sticks description, and arranging these items so that the relative topological orderings and interconnections implied by the sticks description are preserved, and also so that none of the minimum mask level spacing requirements (i.e. design rules) are violated. A brief discussion of the current state of the art in sticks compaction has already been presented in section 2.7.

Sticks compactors are now available which can efficiently produce acceptably dense mask level equivalents of sticks circuits. Given such compactors, it would seem far quicker and easier to design circuits at the sticks level rather than at the mask level, especially considering the other advantages of sticks design such as freedom from process-specific design rules. However, in practice, design at the sticks level has yet to gain any real degree of widespread industry acceptance as a production tool, rather than as merely a research topic. Some of the reasons for this lack of acceptance are worthy of brief investigation.

One reason for this is seen by the author to be that the largest contribution to reductions in circuit area are not made by clever compaction of a fixed topological arrangement. Rather, the greatest savings are made by changes in the topological arrangement of components once the circuit has been initially laid out and the parts of the circuit preventing further compaction have been identified.

Furthermore, a human designer, who has a good global overview of a circuit, can often see areas where slight changes in one area of a circuit can give a globally more compact circuit. However, it may be impossible to convey this information to the sticks compactor, and so in frustration the design returns to design directly at the mask level.

The compactor designed for the VIRGIL system is intended to allow the designer to interact in the compaction process by including "hints" to the compactor in the virtual grid representation of a circuit, such as was mentioned in section 4.1. Changes in topology can be rapidly made, and their effect on circuit area rapidly gauged by use of automated compaction. In such cases, the main requirement is for predictable and controllable compaction. The designer must be able to gauge the effects which certain changes in the sticks representation will have on mask level circuit representation. Similar observations have been made by other researchers in the field of sticks design systems [Weste 81a].

The sticks compactor produced as part of the work for this thesis has been designed to be both predictable and controllable.

Obtaining the densest possible circuit realisation, and hence lowest fabrication cost per part is not always the primary consideration in circuit implementation. In designing circuits for low and medium volume applications, initial design cost is often the major portion of the overall cost of each device. Design at the sticks level can play a part in reducing this overall cost by allowing the rapid design of circuits which are both free of design rule infringements and which can be rapidly implemented in new technologies as these become available, even though this may result in less dense circuit layouts.

Just as most software is now written in high level languages, so it can be expected that VLSI circuits will increasingly be designed at levels of abstraction above the mask level. Using the same analogy, it can be expected that there will always be a place for design directly at the mask level, just as there is still a place for programming in machine specific assembly code.

5.2 Mask Level Representation on a Quasi-Virtual Grid

A virtual grid description of a circuit provides not only physical but also structural information about a circuit. It is also a "stretchable" design representation because as grid lines are spread further apart, perhaps by insertion of extra grid lines during composition, then it is only wires which are affected. These wires automatically become longer so as to maintain connections between the devices at their end points.

When converting to a mask level representation of a circuit, it would seem a pity to lose these attributes of a joint physical and structural design description, and of a stretchable design representation. For these reasons, a novel method of mask level circuit representation based on the idea of a virtual grid is introduced. This is referred to as the quasi-virtual grid.

Circuits, whether at the sticks or mask level, consist of a set of devices (transistors and contacts) and points of external connection (ports) interconnected by wires. At mask level, devices are constructed by overlaying several different mask layers in a specific way. The sets of overlaying mask shapes which comprise the various devices are called "templates". In some cases, all devices of a certain type (such as polysilicon to metal contacts) can be formed from the same template, while in other cases (such as buried contacts) there may be several different templates for the same type of device.

Devices in a circuit can then be described in terms of a template and a pair of coordinates representing the physical position of the template within the complete mask level circuit. Wires can be represented in terms of a type (layer and width) and a path followed by the centre line of the wire. This path can in turn be represented as the set of the coordinate pairs of the end points of the line segments comprising it.

Within a circuit, there will be a finite number of coordinate pairs needed to describe the positions of all the devices and the paths of all the wires in that circuit. These coordinate pairs can then be broken up into a set of X-coordinates {X1..Xn}, and a set of Y-coordinates {Y1..Ym}, such that Xi <= Xi+1 and Yi <= Yi+1. A set of lines, parallel to the Y-axis, can be defined by {x=Xi}. A similar set of lines, parallel to the X-axis are defined by {y=Yi}. All the coordinate pairs needed to specify the circuit appear at intersections of these lines, and so these lines are called the "lines of action" within the circuit. All significant points in the circuit can be mapped to a new, more canonical, integer coordinate plane by the simple mapping:

(Xi,Yj) -> (i,j)

This new coordinate system is called the quasi-virtual grid. A circuit described on a quasi-virtual grid is the same as a circuit on a virtual grid - devices at grid points, wires between grid points - plus some additional information, viz. a set of device templates, and a set of physical coordinate values corresponding to the quasi-virtual integer coordinates.

Consider, for example, the shift register cell shown in figure 5-1. This cell is designed using lambda based Mead-Conway design rules, with a lambda of 3 microns. The device templates for this cell are shown in figure 5-2 in graphical form. Device templates are included for the null devices which implicitly exist in virtual grid circuits (see section 4.7). Wire widths are all minimum allowable line widths. Figure 5-3 shows the lines of action in the shift register cell, and shows the physical coordinate to quasi-virtual coordinate mappings. Figure 5-4 shows the quasi-virtual grid corresponding to the shift cell.

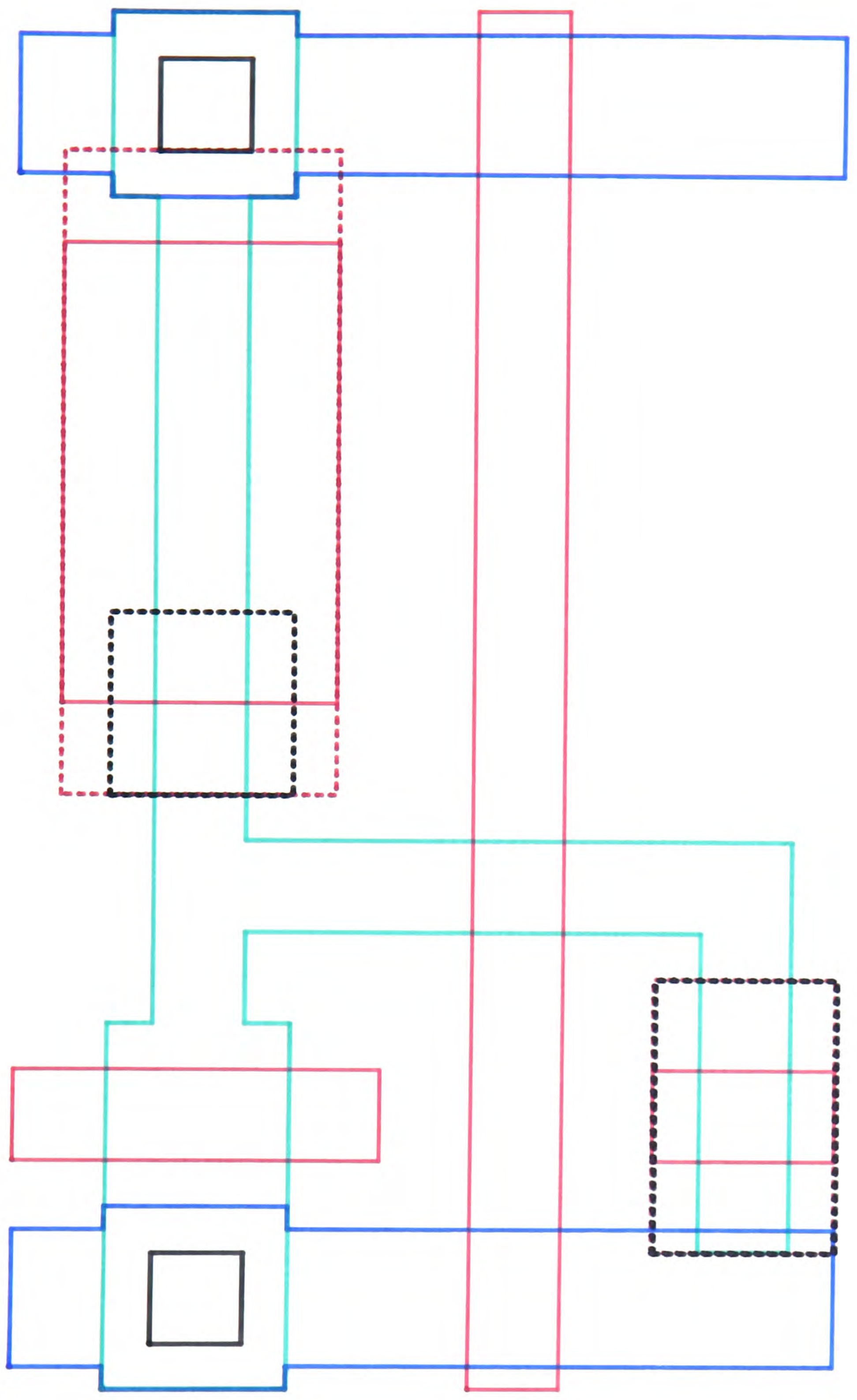


Figure 5-1: A Shift Cell at Mask Level

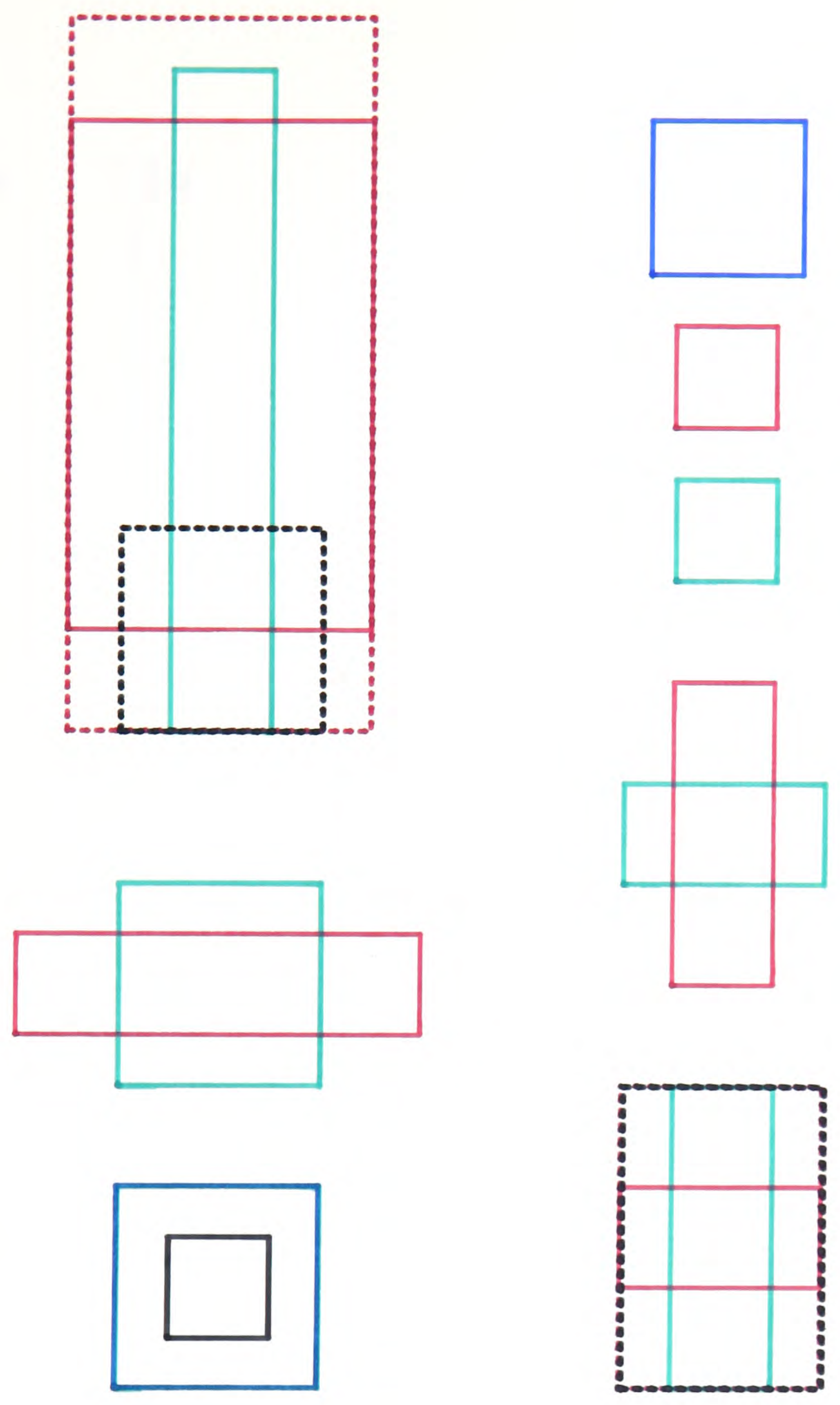


Figure 5–2: Device Templates in Shift Cell

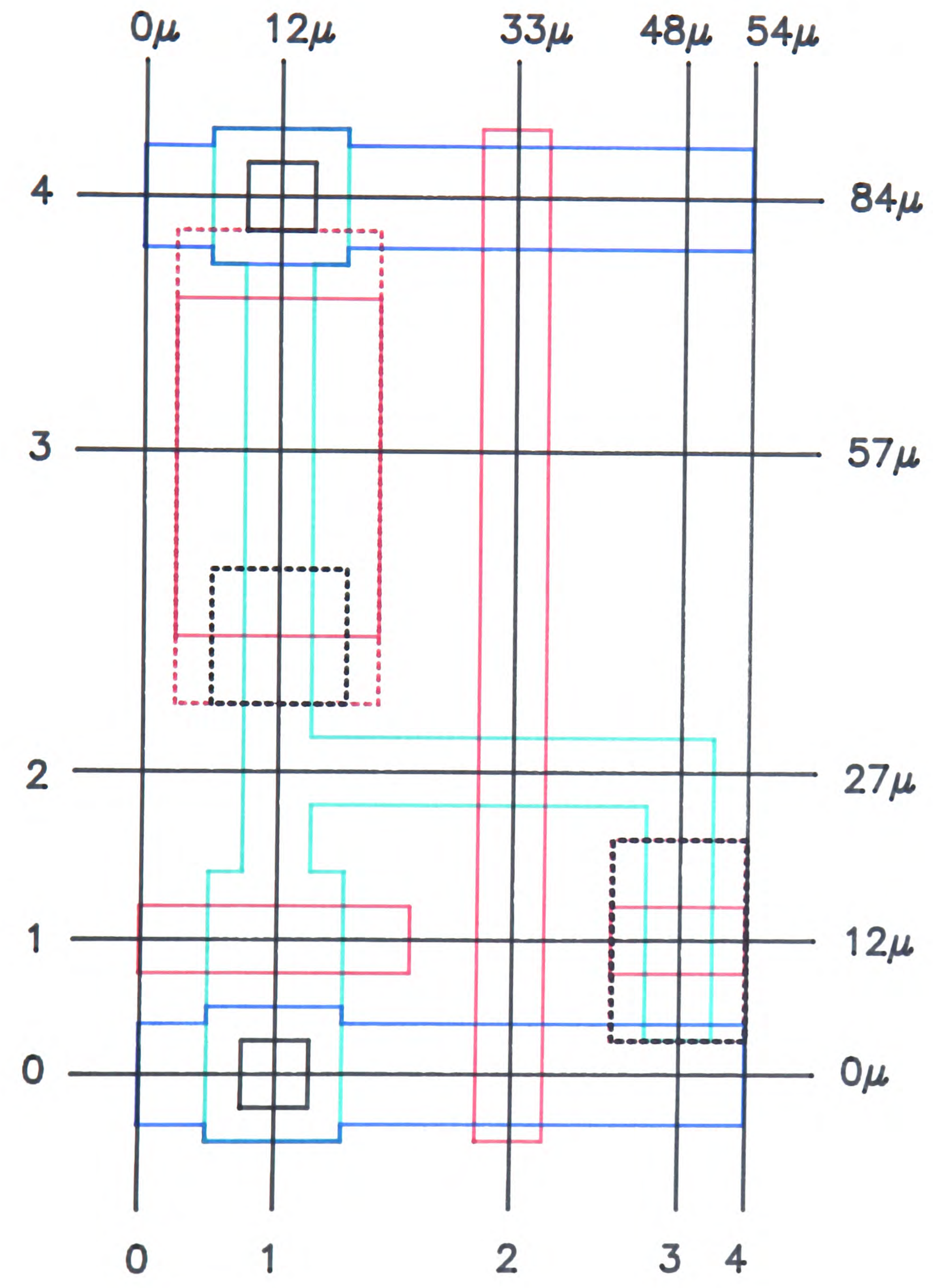


Figure 5–3: Lines of Action and Coordinate Mappings

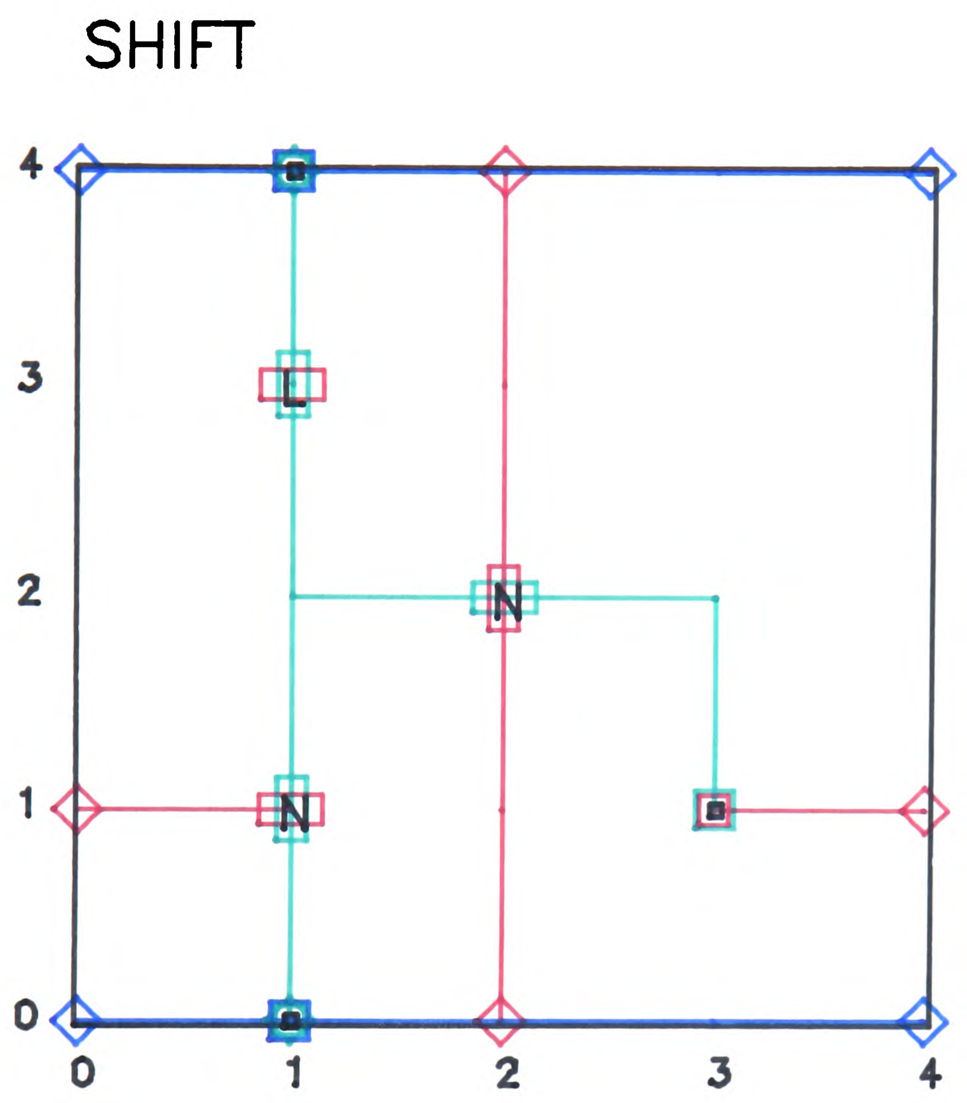


Figure 5-4: Quasi-Virtual Grid for Shift Cell

The quasi-virtual grid of figure 5-4 together with the coordinate mappings of figure 5-3 and the device templates of figure 5-2 then comprise the entire quasi-virtual representation of the mask level description in figure 5-1.

5.3: Translation to a Quasi-Virtual Grid

The quasi-virtual grid provides a way of representing mask level circuits which is especially convenient for translating from virtual grid "sticks" circuits. Since the quasi-virtual grid consists of a virtual grid plus device templates and coordinate mappings, translation from sticks to mask level can be achieved by simply adding these latter two components to an existing virtual grid.

It should be immediately emphasised that this simple translation is by no means the only possible translation which can be made. A virtual grid may be translated into a mask level representation which is based on a different quasi-virtual grid. It should be remembered that a virtual grid itself imposes no restriction, other than relative topological ordering, on the components within it. On the other hand, the quasi-virtual grid imposes the much stronger restriction that items with the same X or Y quasi-virtual coordinate are positioned at the same physical X or Y coordinate. This restriction is then also implicitly imposed on the original virtual grid if this simple translation is used.

This restriction is not necessarily a particularly harsh one, in fact this restriction gives the controllability and predictability of sticks to mask level translation which has previously been shown to be useful. This constraint is also the reason that virtual grid sticks compaction can be computed with time complexity O(N), rather than O(N**1.5) as is the case for gridless compaction, or virtual grid compaction without this constraint. A sticks compactor based on this simple translation has been developed for this thesis.

The quasi-virtual grid is primarily used here as a target for the sticks compactor, rather than as a design representation produced directly by a designer. As such, it is only necessary that the quasi-virtual grid is capable of representing those circuits which can be produced by the sticks compactor, and this leads to several simplifications.

Any geometric restrictions imposed on a virtual grid are also likely to be imposed on a quasi-virtual grid directly derived from it. Specifically, if only paraxial (i.e. parallel to one or other coordinate axis) wires are allowed on the virtual grid, then wires on the quasi-virtual grid will also all be paraxial.

Device templates need not be composed of purely Manhattan geometry shapes, the only restriction imposed by the Manhattan nature of the virtual grid is that connections between wires and devices will be in one of the four Manhattan directions. In practice, Manhattan geometry templates are quite adequate, and the use of purely Manhattan shapes simplifies the representation of templates.

Another simplification, which is less obviously justified, is to do with so-called "bent" transistors. In NMOS circuits particularly, the use of ratioed logic often results in the use of transistors with channels which are quite long or quite wide. These devices can often be fitted more neatly into a circuit by twisting the path of the channel. An example is shown in figure 5-5, where a load device with a length to width ratio of 16:1 is shown both "straight" and "bent".

In this compactor, only "straight" transistors are used, for several reasons. Firstly, interconnections between wires and devices can be greatly simplified if device templates conforming to certain conditions (which are fully discussed in section 5.4) are used. These conditions are not met by bent transistors.

Secondly, bent transistors are not really an issue in CMOS circuits, where ratioless logic is used. A good

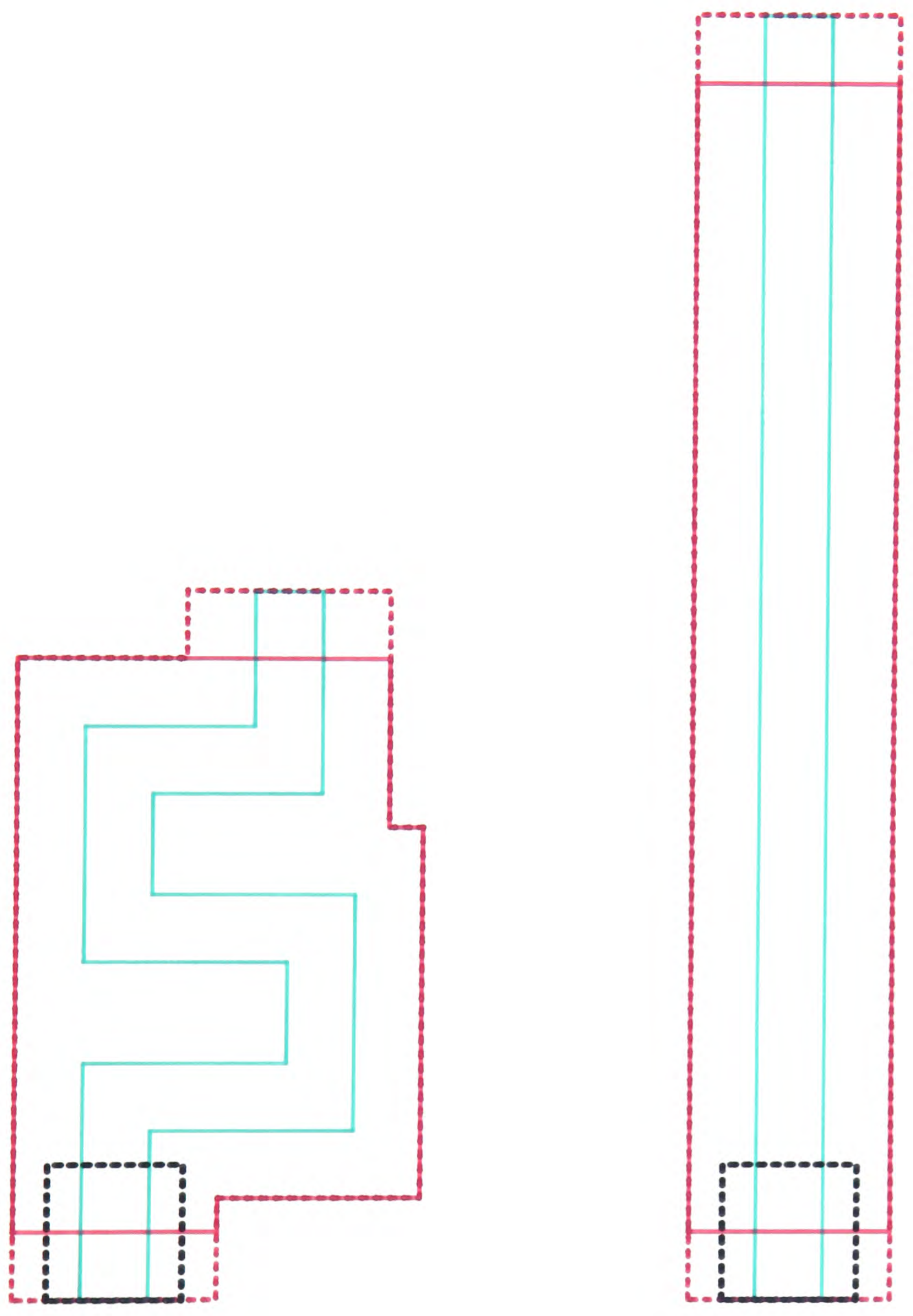


Figure 5-5: "Bent" and "Straight" Transistors

deal of the original work done here on compaction is specifically to do with CMOS circuits, and CMOS circuits are increasingly overtaking NMOS as the most widely used technology. It was felt that it was not worth spending too much time on subjects not directly applicable in CMOS circuits.

Finally, the provision of "bent" transistors really requires the bends to be arranged so as to fit into the surrounding circuit layout, if these bends are to be most effective in reducing circuit area. The generation of such "context sensitive" device templates is a far harder problem than the simple approach adopted here of having a single type of device template (i.e. straight transistors) which are always used.

In summary then, the compactor developed for this thesis has been deliberately made as simple as possible. It is hoped to show that these simplifications allow compaction to proceed in a very simple and predictable manner.

5.4: Device Templates

Device templates are the mask level equivalents of device "arrangements" mentioned in section 4.7. If templates are restricted to Manhattan geometry figures, then templates can be specified as a set of paraxial rectangles, each on a specific mask layer, and placed relative to some device origin at the centre of the device. Some of these rectangles exist on notional mask layers, which although not corresponding to physical mask layers, are still useful to include. An example is a layer called "active", which defines the transistor channel areas. It is simpler to specify these layers directly than to specify rules for how they can be derived in general. Some device arrangements (e.g. the four arrangements of BUTTING in figure 4-16) are simply rotations of a single arrangement, so in such cases only one template needs to be specified.

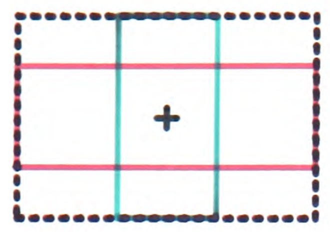
Some devices' shapes depend on parameters such as channel length and width. In these cases the templates are best specified parameterisably. Given the name of a device and an optional set of parameters, such a parameterisable definition returns the set of boxes on various mask layers which correspond to the exact template for the device.

Since wires are to go between devices to connect them together, it might seem that it is necessary to include in the templates the coordinates of the points on the devices to which wires may connect. Because of the simplifications mentioned in the previous section, it is possible to design templates in a manner so that this need is removed, viz. the templates are designed such that if a wire is to be connected to a device, then correct connection is made by connecting the end of the wire to the centre of the device. In other words, wires do not go merely between edges of devices, rather they go between the centres of devices. Wires can always be placed in this way such that they do not interfere with the device template.

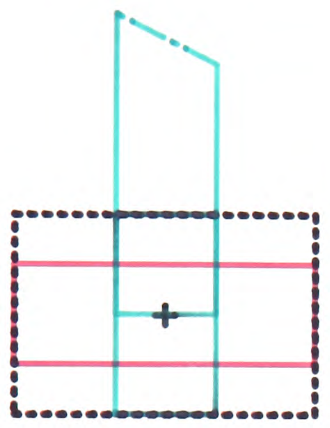
As an example, consider an orthogonal buried contact as shown in figure 5-6 (a). Wires can extend from the centre of the device in all four directions (figure 5-6 (b)-(e)) quite legitimately.

This property leads to the very useful result that wires can be placed on a quasi-virtual grid such that their centre lines extend between grid lines, without regard to what devices are placed at grid points. Note that these wires no not extend past the end-points of their centre lines (as, for example, CIF wires do), as illustrated in figure 5-7. Null device templates provide the overlap between the ends of orthogonal wires segments which is provided by the extended ends of wires in CIF.

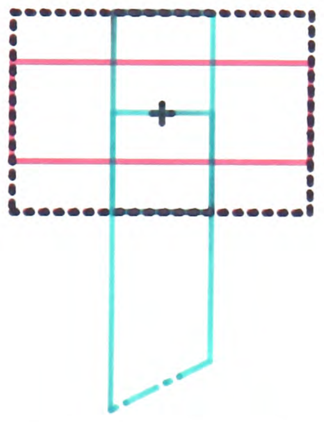
Figure 5-8 (a) shows a simple virtual grid, which can be divided into devices (5-8 (b)) and wires (5-8 (c)). These can be converted to mask representations independently (5-8 (d),(e)) and when combined (5-8 (f)) give the correct mask equivalent of the whole circuit.



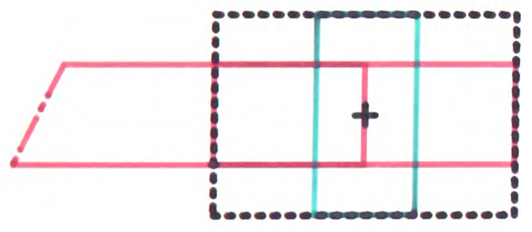
(a) Buried Contact Template



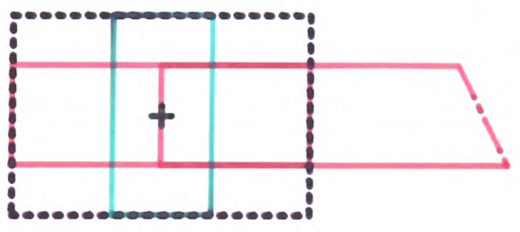
(b) Wire from North



(c) Wire from South



(d) Wire from West



(e) Wire from East

Figure 5–6: Wires Connecting to the Centre of a Device

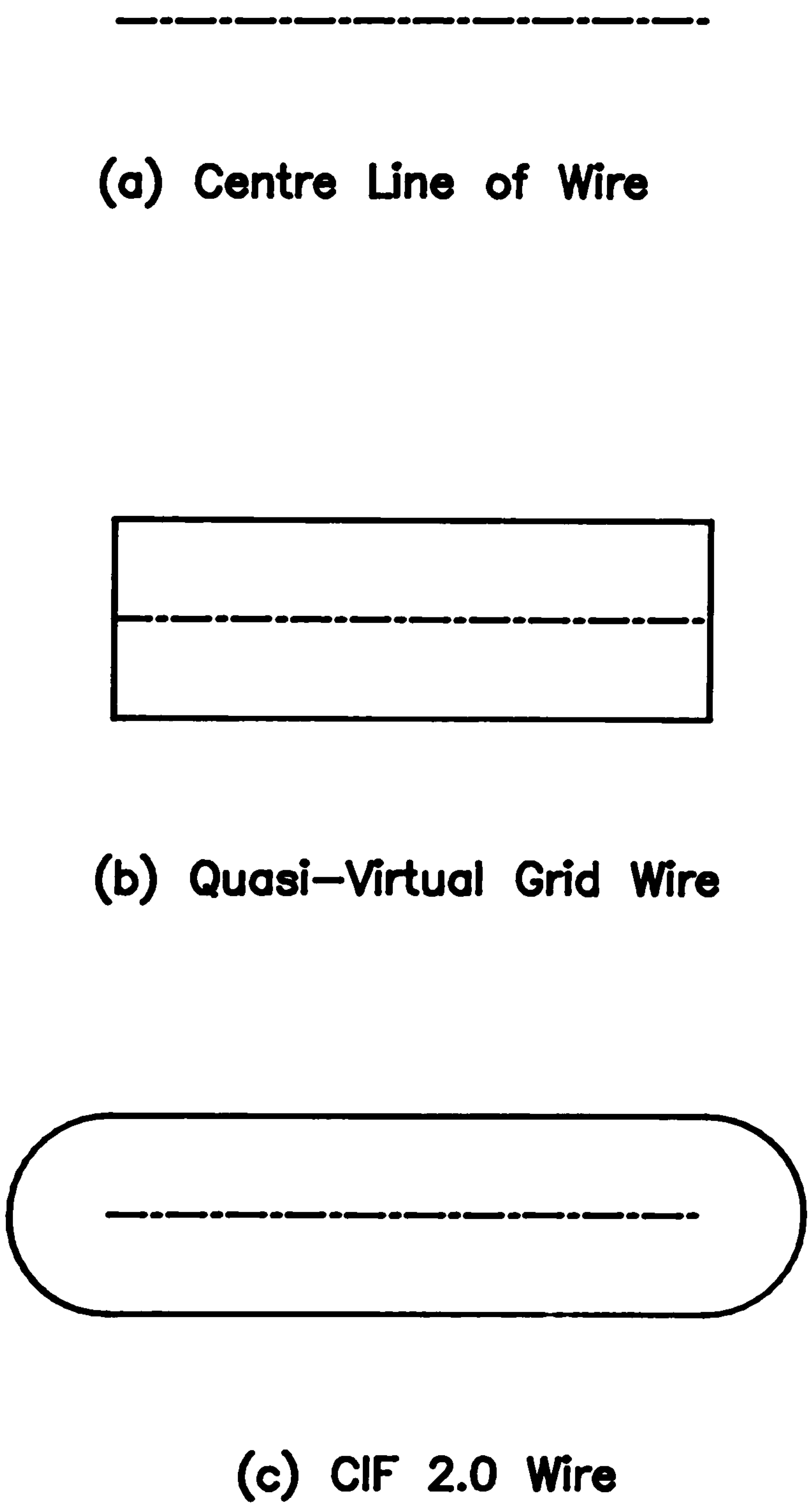
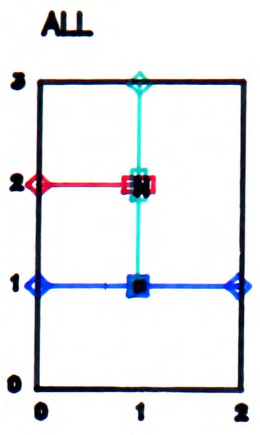
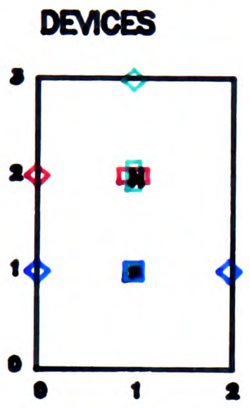


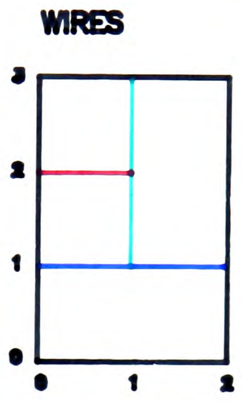
Figure 5–7: Wires in CIF and in Quasi–Virtual Grids



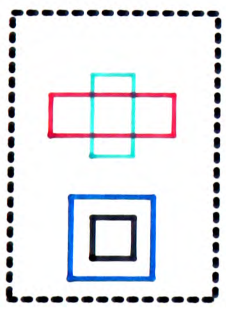
(a) Complete Virtual Grid Circuit



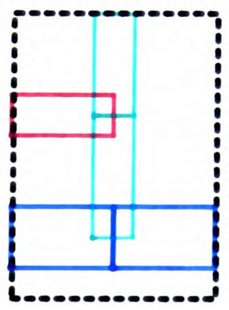
(b) Devices



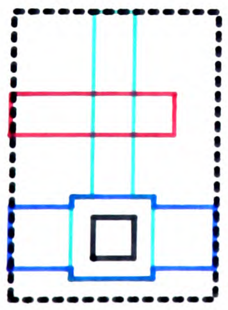
(c) Wires



(d) Mask Level Devices



(e) Mask Level Wires



(f) Complete Mask Level Circuit

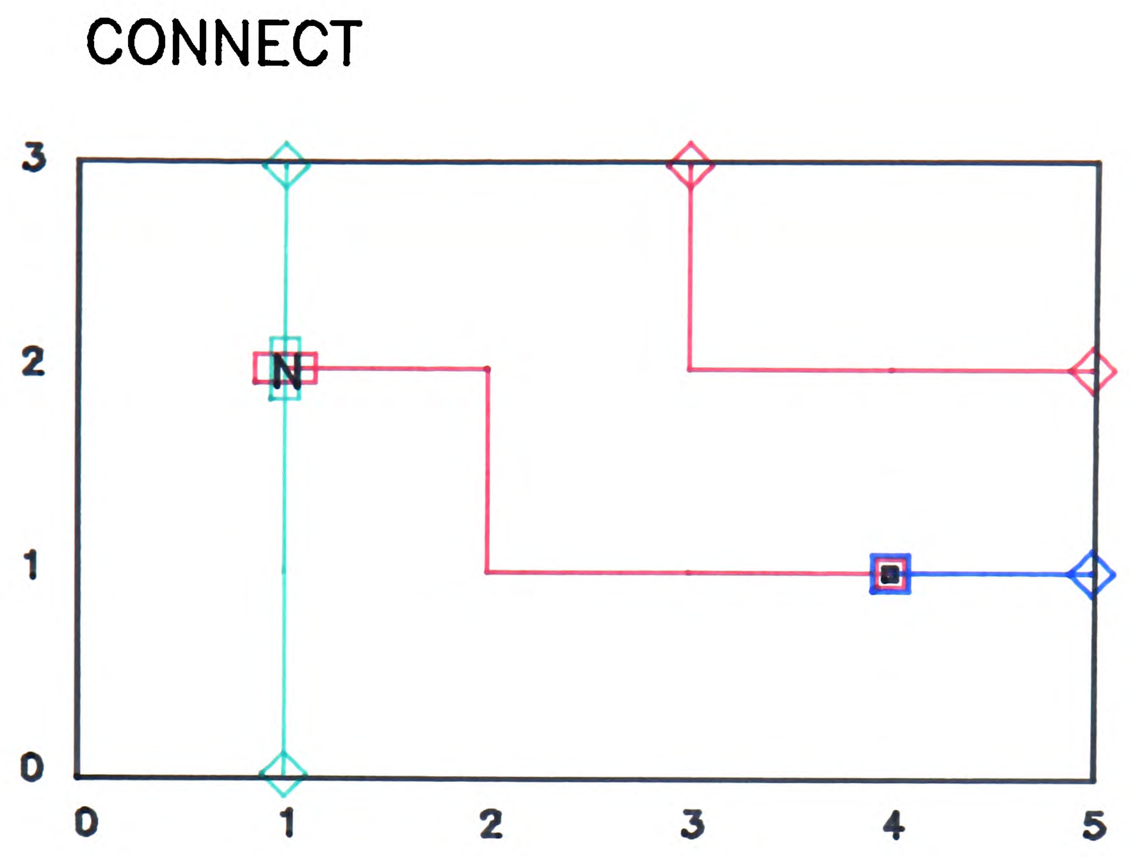
Figure 5-8: Independent Translation of Wires and Devices

Since null device templates are squares of the minimum possible size on various layers, the null device templates are sufficient to define minimum wire widths.

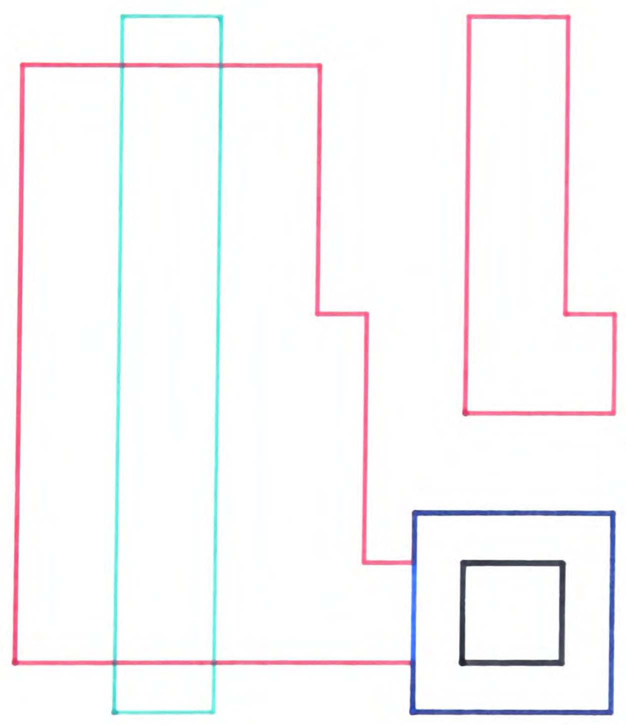
In some cases, it might be desirable to connect to points of a device other than the centre, especially where connection can be made over a wide area, such as the poly gate of a long transistor. In such cases, a wire which runs out of the centre of the device, and then along to the preferred connection point usually gives similar results. Figure 5-9(a) shows the way such a construct is represented on a virtual grid, and figure 5-9(b) shows the resulting mask level layout.

Device templates provide an elegant method of expressing all the design rules concerned with widths and lengths of items in a mask level representation. A list of minimum separations between layers completes the information required to specify all the design rules concerned with widths and separations for a particular fabrication technology. Separations such as active area to implant make use of the notional mask layers (in this case "active") mentioned earlier.

It is argued that a list of templates plus a list of layer separations provides a more elegant and more natural method of capturing the design rules associated with a particular technology than the specification of a



(a) Virtual Grid



(b) Mask Level Representation

Figure 5-9: Connection Desired to be Made to Bottom of Transistor Gate

large number of constants such as "poly gate over channel overlap" and "metal around contact hole overlap".

Some sets of design rules include rules which are not simple widths and separations. An example is "no coincident poly and metal edges". Such design rules are not conveniently handled by a set of templates plus a set of separations, and special code would need to be written for each special case. No such special cases are included in the simple compactor produced for this thesis. Note that device templates plus separations are sufficient to capture Mead-Conway design rules [Mead 80].

5.5 Coordinate Mappings

The next, and more difficult, phase of producing a quasi-virtual grid description is to find the mapping between quasi-virtual grid coordinates and physical coordinates. This amounts to finding the minimum physical spacing between grid lines such that no design rules are violated, and corresponds to what is normally called sticks compaction.

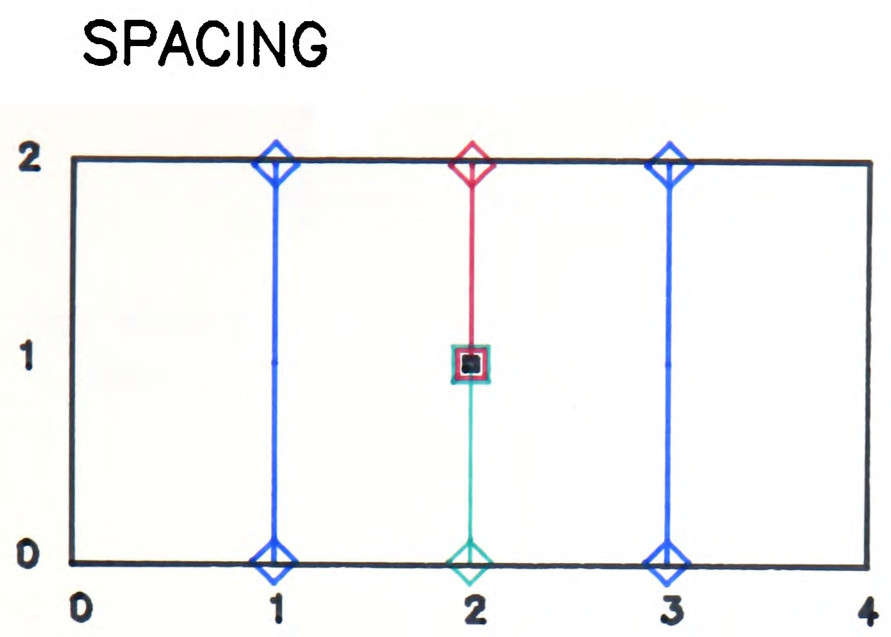
It is significant that wires have no direct effect on the grid spacing, and can be ignored during this phase of the translation. This is because the spacings between the devices which exist at the ends of wires are sufficient to ensure correct spacing between the wires themselves.

The spacing between vertical grid lines is determined first, and is achieved by assigning an X-coordinate to each grid line, in turn, from left to right. A grid line is first given the same X-coordinate as its predecessor. Each device on the vertical grid line is examined in turn. The minimum spacing between the device and any devices directly to its left (i.e. on the same horizontal line) is determined, and, if necessary, the X-coordinate of the current grid line is increased so that this spacing requirement is met. Notice that items on grid lines several grid lines away can affect the spacing, as shown in figure 5-10. In practice, all items closer than the maximum possible separation are checked.

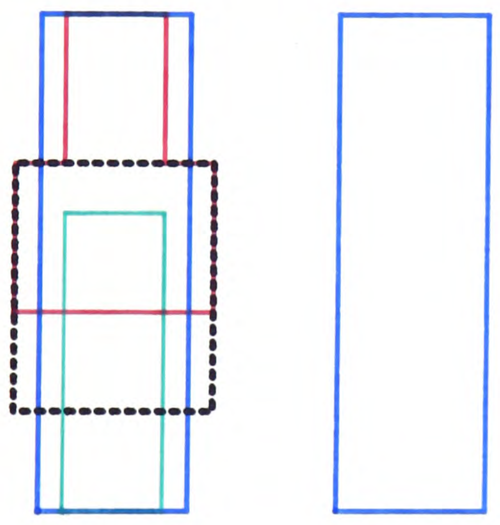
Y-coordinates are assigned to horizontal grid lines, from bottom to top, in the same manner.

Finally, diagonal spacings are checked. Devices are sufficiently separated diagonally if they are sufficiently separated either horizontally or vertically. Figure 5-11 shows two possible arrangements of a circuit which would satisfy diagonal spacing requirements.

The basic operation required to calculate grid spacings is that of finding the separation between two devices, given their relative positions (left, right,

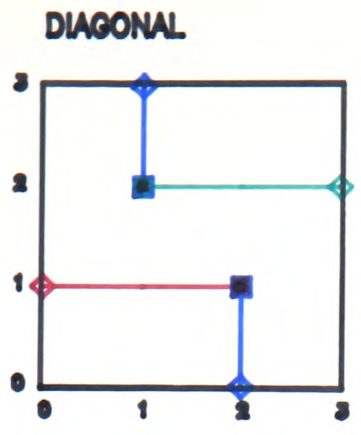


(a) Before Compaction

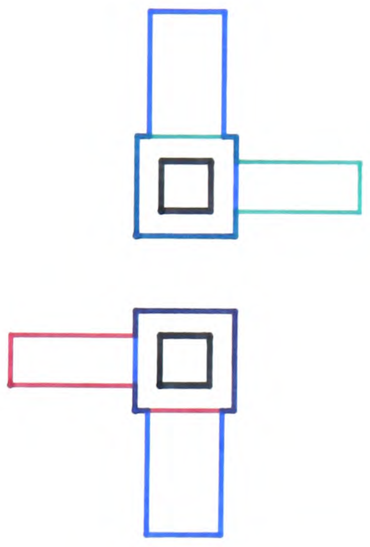


(b) After Compaction: Metal to Metal Spacing Dominates

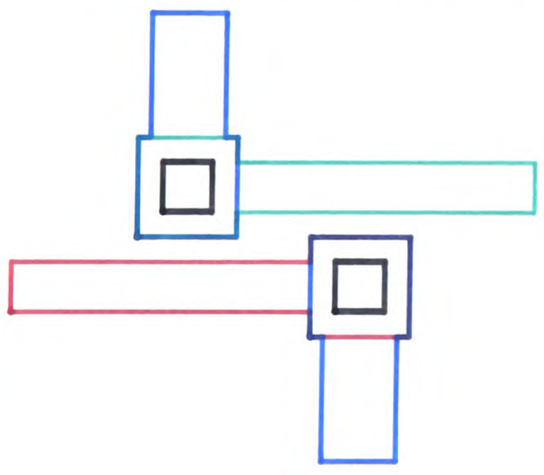
Figure 5–10: Influence of Items over Several Grid Lines



(a) Circuit with Diagonal Spacing
Requirement between Contacts



(b) Layout Providing Vertical Spacing



(c) Layout Providing Horizontal Spacing

Figure 5–11: Diagonal Spacing Requirement is Satisfied by Sufficient Horizontal or Vertical Spacing

above, below) and the layers on which they are connected. Figure 5-12 shows an example of two items whose separation depends on connectivity.

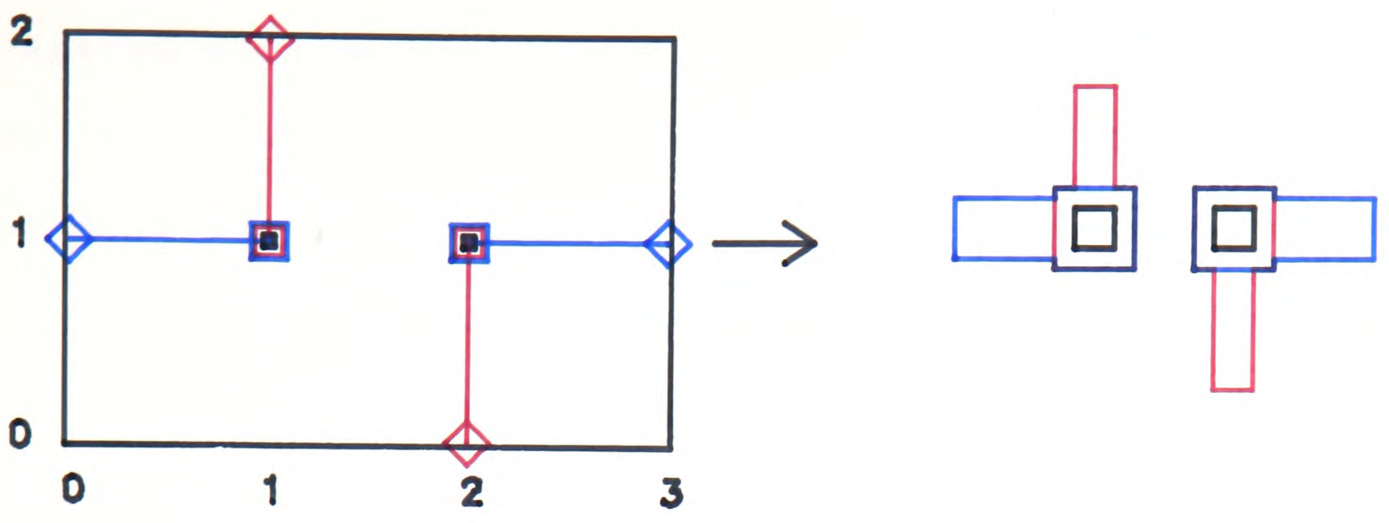
Note that in situations such as that in figure 5-12, it is necessary to know that, for example, metal and polysilicon are connected in a PM contact. This can be done by including in device templates a list of connected layers. In each device, all layers marked as "connected" are considered to be electrically connected.

Such connection information is sometimes rather artificial, e.g. in a PM contact, the "contact" mask is _not_ marked as connected, so that correct contact window to contact window spacing is maintained between adjacent contacts.

The only other information needed to calculate device spacings is a list of minimum separations necessary between the various layers.

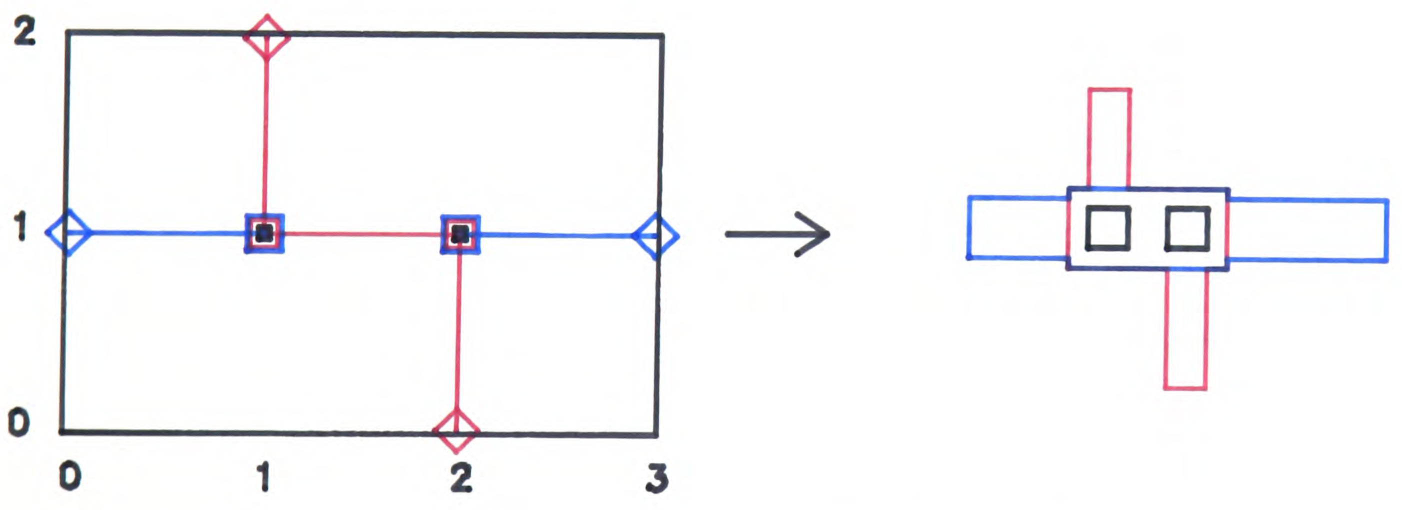
To find the separation between two devices, the connections between the different layers in each device must be calculated. If a wire of a certain type joins the two devices, then boxes on that layer in both devices are considered connected to the wire. If that layer is marked "connected" in the template of one device, then all other layers also marked "connected" are considered

UNCONNECTED



(a) Separation between Centres of Unconnected Contacts = 7 lambda

CONNECTED



(b) Separation between Centres of Connected Contacts = 4 lambda

Figure 5-12: Effect of Connectivity on Separation

to be connected to the wire.

Boxes on each layer in one device template are compared with boxes on each layer of the other. If the two layers on which the boxes exist are connected, or if there is no minimum spacing requirement between them, then this pair of boxes is ignored. Otherwise, the minimum spacing requirement imposed by this pair of boxes is the sum of the minimum separation between the layers and the size of each box in a direction towards the other, as shown in figure 5-13.

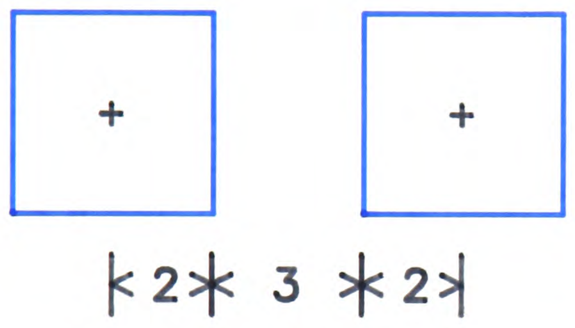
The largest spacing imposed by any pair of boxes determines the minmum spacing (centre to centre) between the two devices. Notional layers such as "active" allow spacings such as active to implant to be calculated.

A set of coordinate mappings as produced by the compactor represents a minimum set of spacings between grid lines. By increasing any of these spacings, a mask level circuit can be stretched, perhaps to allow connection to some previously designed mask level circuit.

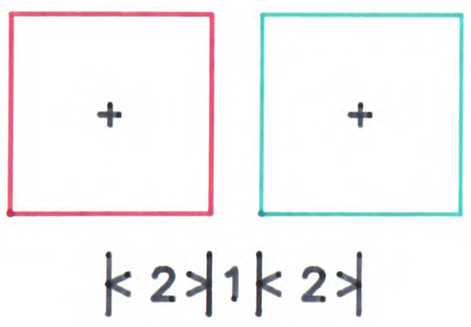
5.6: Translation of CMOS Circuits to Mask Level

The amount of design-rule independence offered by virtual grid design is far more important in the design

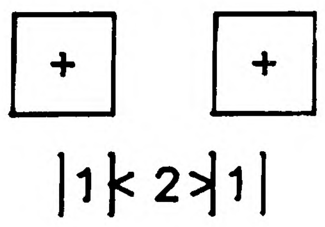
Calculation of Centre to Centre
Separation between PM and NM
(All Units in Lambda)



Metal To Metal
Spacing
= 2+3+2 = 7



Poly to Diff
Spacing
= 2+1+2 = 5



Contact to Contact
Spacing
= 1+2+1 = 4

Maximum Layer to Layer Separation = 7,
So Minimum PM to NM Separation = 7

Figure 5-13: Minimum Separation Calculation

of CMOS circuits than in NMOS. At present there are at least four major classes of CMOS technologies, none of which is particularly more favoured than the others. These are N-well, P-well, twin-well and SOI (silicon-on-insulator). A new design style, called generalised CMOS, is now introduced which allows the design of circuits which are valid in all of these technologies.

Circuits (and hence idioms) which are designed in generalised CMOS are able to offer the circuit designer a wider choice of possible target technologies. Furthermore, the choice of technology can be left to far later in the design cycle. A single circuit might also be implemented in two different technologies for different applications - say twin-well for performance and SOI for radiation hardness.

The conversion of CMOS circuits from virtual grid to mask level representation involves an extra step not present in the conversion of NMOS circuits. This is the insertion of wells. Wells are used to change the substrate doping in certain areas of a circuit so that the two complementary types of transistor (NTYPE and PTYPE) can be fabricated on the same substrate.

Wells differ from other circuit features in that they are not associated with individual devices, but rather with whole groups of devices. Where possible, wells around adjacent devices should be amalgamated. In particular, every distinct well region must usually contain at least one substrate contact. Although SOI does not have wells like the other technologies do, it does have island doping masks, which are similar.

Ideally, each well should contain as many adjacent devices as possible, but it should be no larger than is necessary to contain these devices, or else it might cause unnecessarily large separations between devices inside the well and those outside.

Wells are of two types. A P-well is an area of p-type diffusion, and can contain n-type (i.e. n-channel) transistors and n-type diffusion wires. An N-well is an area of n-type diffusion, and can contain p-type transistors and p-type diffusion wires. In single well processes, areas outside the well are as if they were in a well of the opposite type.

Design at the mask level forces a designer to select a specific CMOS technology. Even design at the sticks level often requires the specification of well areas, and so ties the design to a specific CMOS technology. Generalised CMOS circuits are just CMOS designs which contain no constructs specific to a single technology, such as wells. The sticks compactor developed for this thesis is able to automatically determine suitable well areas using a novel algorithm which is described below, and so is able to handle generalised CMOS circuits.

All virtual grid devices can be classified according to whether they can exist only in a P-well (class-p devices: NTYPE, NWIRE, NPORT, PSUB, NM); whether they can exist only in an N-well (class-n devices: PTYPE, DWIRE, DPORT, NSUB, DM); or whether they can exist in wells of either type (class-u devices: MWIRE, MPORT, PWIRE, PPORT, PM).

Each point on a virtual grid circuit can be similarly classified. If any class-n device exists at a point, then it is a class-n point, similarly for class-p. A point which has no class-p or class-n devices is a class-u point. Note that a class-p point or class-n point may contain class-u devices. Figure 5-14 shows a virtual grid circuit of a 4-input generalised CMOS nand gate. Figure 5-15 shows the classification of the grid points into class-n, class-p and class-u.

On the virtual grid, wells are constructed so as only to contain devices of the correct classes. P-wells may not contain class-n devices, and vice versa. Wells are described as sets of rectangles, perhaps overlapping, surrounding groups of grid points. Such rectangles are constructed so as to contain as many devices of the

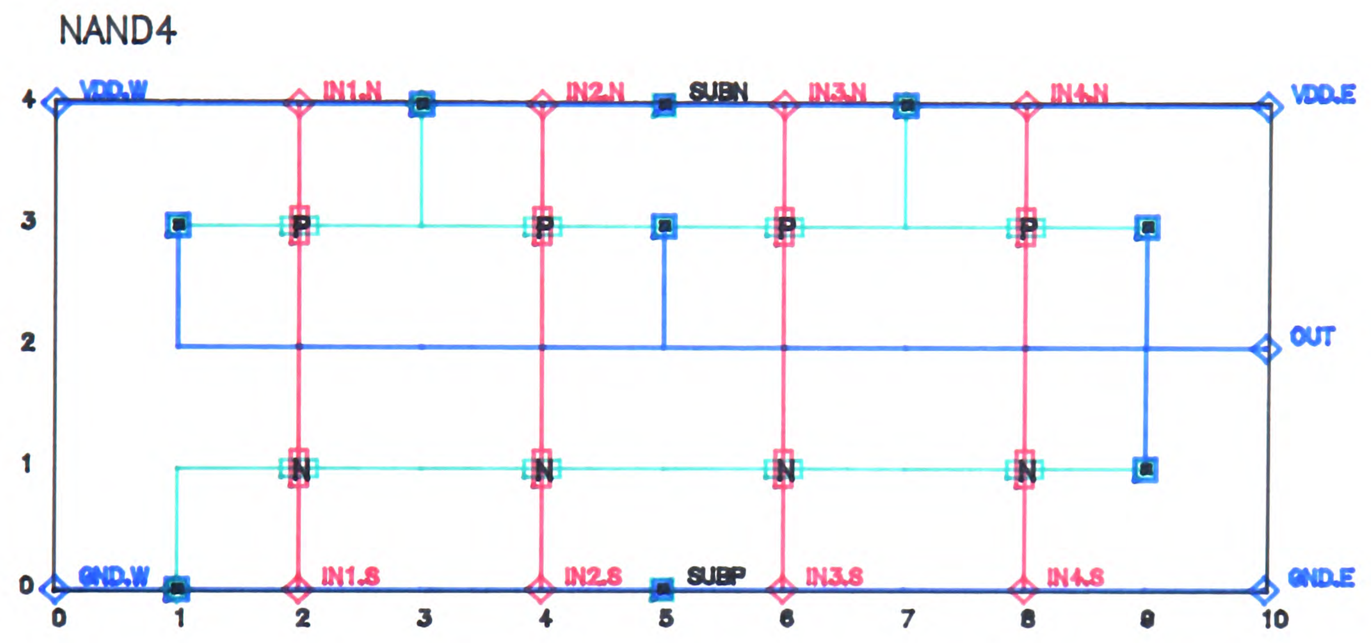


Figure 5–14: 4–input NAND Gate
in Generalised CMOS

N = can exist only in N–well

P = can exist only in P–well

U = can exist in either well

```
4   U —U —U —N —U —N —U —N —U —U —U
    |  |  |  |  |  |  |  |  |  |  |
3   U —N —N —N —N —N —N —N —N —N —U
    |  |  |  |  |  |  |  |  |  |  |
2   U —U —U —U —U —U —U —U —U —U —U
    |  |  |  |  |  |  |  |  |  |  |
1   U —P —P —P —P —P —P —P —P —P —U
    |  |  |  |  |  |  |  |  |  |  |
0   U —P —U —U —U —P —U —U —U —U —U

    0  1  2  3  4  5  6  7  8  9  10
```

Figure 5–15: Classification of Grid Points in 4–input NAND Gate

appropriate class as possible, but also so as to be only as large as is necessary to contain those devices, e.g. in figure 5-16, arrangement (c) would be preferred.

To generate rectangles of the desired type, the following algorithm is used, in this case for a P-well.

(1) Select a class-p grid point not yet included in a P-well. Put a rectangle just around that point.

(2) Increase the size of the rectangle by one grid square in each direction, (east, north, west, south) in turn, provided such an increase would not bring a class-n device into the rectangle. Continue increasing the rectangle, one grid square at a time, until it cannot be increased in any direction at all.

(3) Decrease the rectangle by one grid square in each direction in turn, provided such a decrease would not take a class-p device out of the rectangle. Continue decreasing until the rectangle cannot be decreased in any direction.

(4) Place a P-well "null device" at each class-u point in the rectangle, changing these to class-p points.

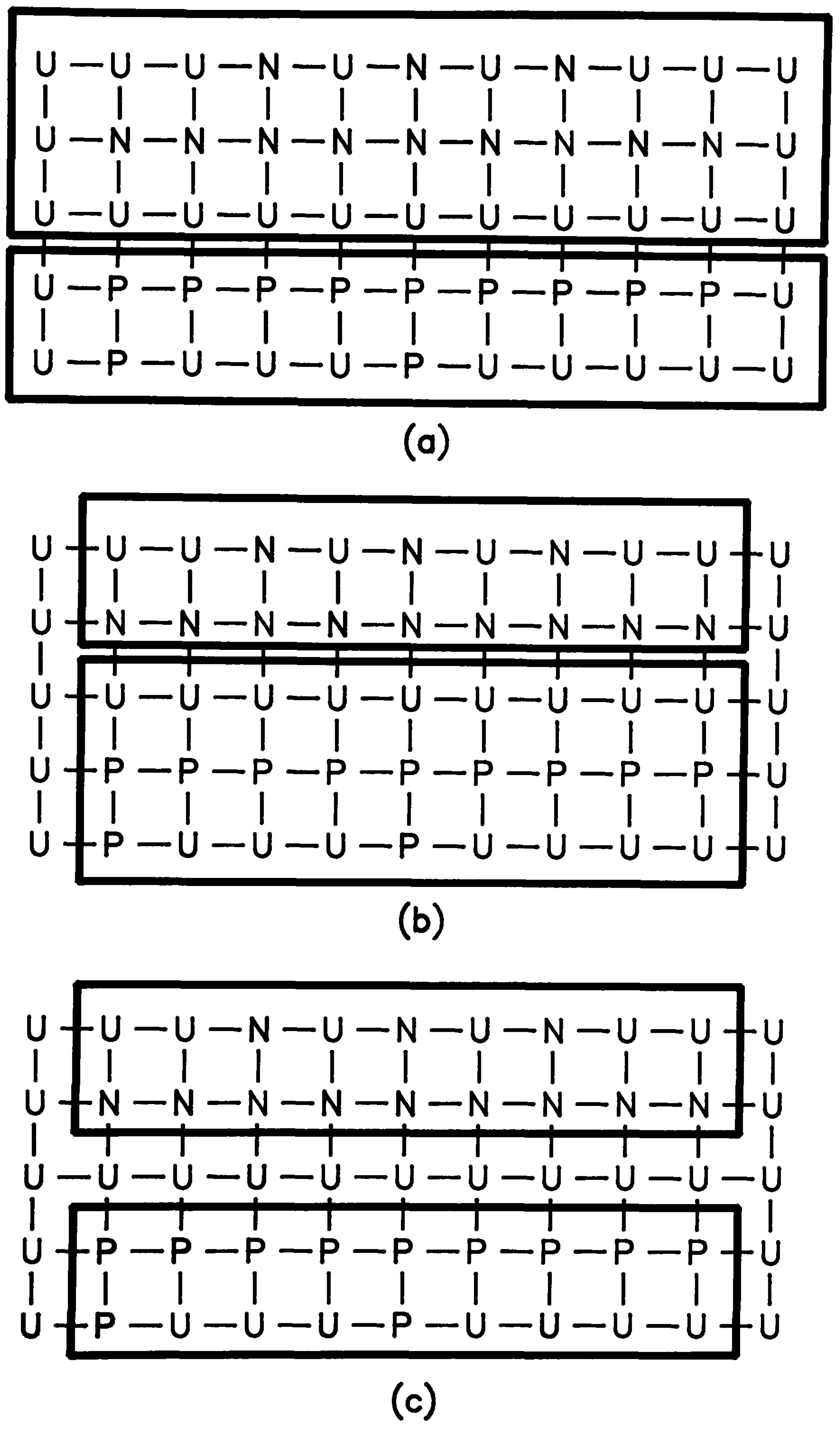

U — U — U — N — U — N — U — N — U — U — U
U — N — N — N — N — N — N — N — N — N — U
U — U — U — U — U — U — U — U — U — U — U
U — P — P — P — P — P — P — P — P — P — U
U — P — U — U — U — P — U — U — U — U — U
(a)
U — U — U — N — U — N — U — N — U — U — U
U — N — N — N — N — N — N — N — N — N — U
U — U — U — U — U — U — U — U — U — U — U
U — P — P — P — P — P — P — P — P — P — U
U — P — U — U — U — P — U — U — U — U — U
(b)
U — U — U — N — U — N — U — N — U — U — U
U — N — N — N — N — N — N — N — N — N — U
U — U — U — U — U — U — U — U — U — U — U
U — P — P — P — P — P — P — P — P — P — U
U — P — U — U — U — P — U — U — U — U — U
(c)

Figure 5–16: 3 Possible Well Area Groupings for CMOS NAND Gate (Arrangement (c) Preferred)

The "null devices" mentioned in (4) above are used in the same way as null devices for wires, to define a minimum size area of well geometry. In a twin-well process, where both N-well and P-well rectangles are grown, placing null devices in wells also prevents the same class-u point being in both a P-well and an N-well rectangle.

Well rectangles are grown before grid line spacings are determined. After grid line spacings have been determined, well rectangles can be added to the mask level representation.

Well mask layers are included in device templates, even if there is no physical mask corresponding to them, such as a P-well mask layer in an N-well process. In this case, this nominal layer represents "anti-well", or in other words, the distance which devices must be placed away from the edges of the other well layer. Since all class-p device templates contain a P-well mask layer, and all class-n devices contain an N-well layer, and all class-u devices contain neither, this provides a convenient method of determining the class of a device from its template. Including such well information in templates also ensures correct spacing between devices, without the need to consider the presence of the large well areas.

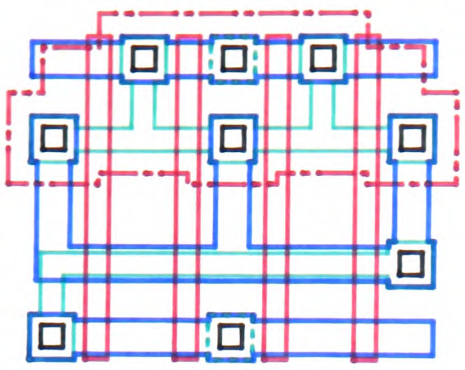
Substrate contacts are explicitly specified by the user in circuit descriptions, and in processes where these are not needed, they can be omitted by the compactor. The automatic insertion of substrate contacts would remove some of the predictability of the compaction process.

Figure 5-17 shows examples of N-well, P-well, twin-well and SOI mask level circuits corresponding to the single generalised CMOS circuit of figure 5-14.

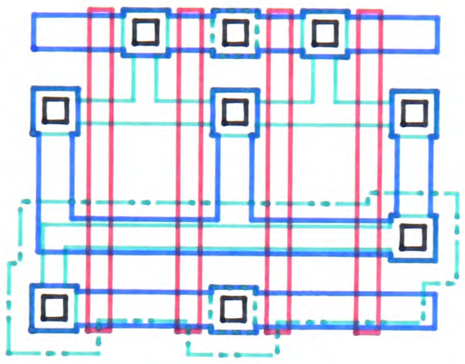
5.7: Sticks Extraction

The usefulness of sticks circuits as a method of design representation has already been emphasised, and in particular, the relatively straight-forward manner in which virtual grid sticks circuits can be translated to mask level representations has been discussed. An equally valid translation, which is potentially just as useful but has not been an area of particular research interest, is the reverse translation from mask level to sticks, here called sticks extraction.

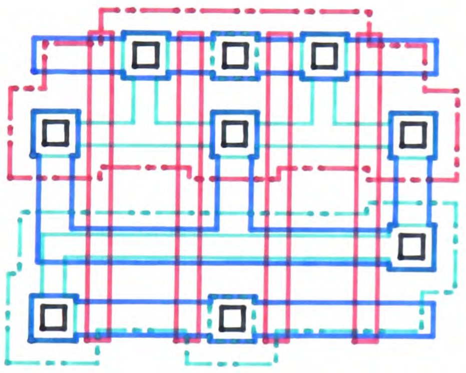
Sticks extraction reduces a mask level circuit to a topological arrangement of devices interconnected by wires. In other words, it removes design rule dependent features of the circuit such as widths and separations. If the sticks circuit is subsequently converted back to



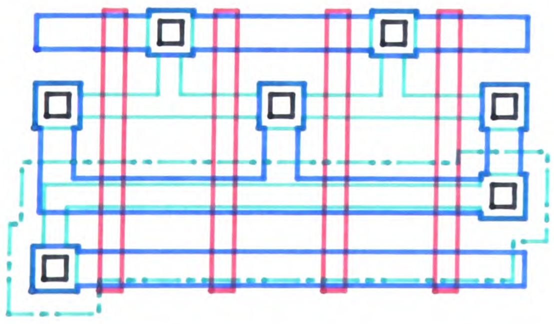
(a) N–well CMOS



(b) P–well CMOS



(c) Twin–well CMOS



(d) SOI CMOS

Figure 5–17: Generalised CMOS NAND Gate Implemented in Four Different Technologies

mask level, but with a different set of design rules, then the original circuit has effectively been automatically translated from one technology to another. The advantages of such translation are especially evident in CMOS circuits, where translation is possible not only between different sets of design rules, but also between different classes of CMOS technologies.

There is already a large investment which has been made in designing circuits at the mask level. If the lifetime of such circuits can be extended by allowing them to be automatically translated into newer technologies as these become available, then designers will be able to make better use of this investment. It is worth noting that this approach does not offer a simple way to take advantage of new processes offering more layers of interconnect.

The problem of sticks extraction has not been examined in great detail, but an algorithm which would seem to work for simple Manhattan geometry circuits is presented as a first excursion into this area.

Sticks extraction amounts to identifying devices and the wires which interconnect them within a circuit. As such, it is quite similar to the process of circuit extraction, but it must also retain the topology of a circuit. Sticks extraction could also be thought of as converting from a traditional mask level representation to a quasi-virtual grid representation. As an example, consider the inverter of figure 5-18, which is implemented in a twin-well CMOS process.

Devices can be located by finding particular combinations of intersecting (i.e. overlapping) mask layers. In a twin-well CMOS process, the combinations identifying the various devices are as follows.

NTYPE: diffusion, poly, pwell

PTYPE: diffusion, poly, nwell

PM: poly, metal, contact

NM: diffusion, pwell, metal, contact

DM: diffusion, nwell, metal, contact

PSUB: substrate, pwell, metal, contact

NSUB: substrate, nwell, metal, contact

Ports can be identified by points where geometry reaches a user-defined cell boundary. Figure 5-19 shows the devices and ports in the CMOS inverter under consideration.

Once identified, these sets of overlapping layers, which form the "core" of devices (i.e. they do not form the entire device templates), can be removed from the mask level representation. This should leave a number of disjoint areas on the different mask layers corresponding

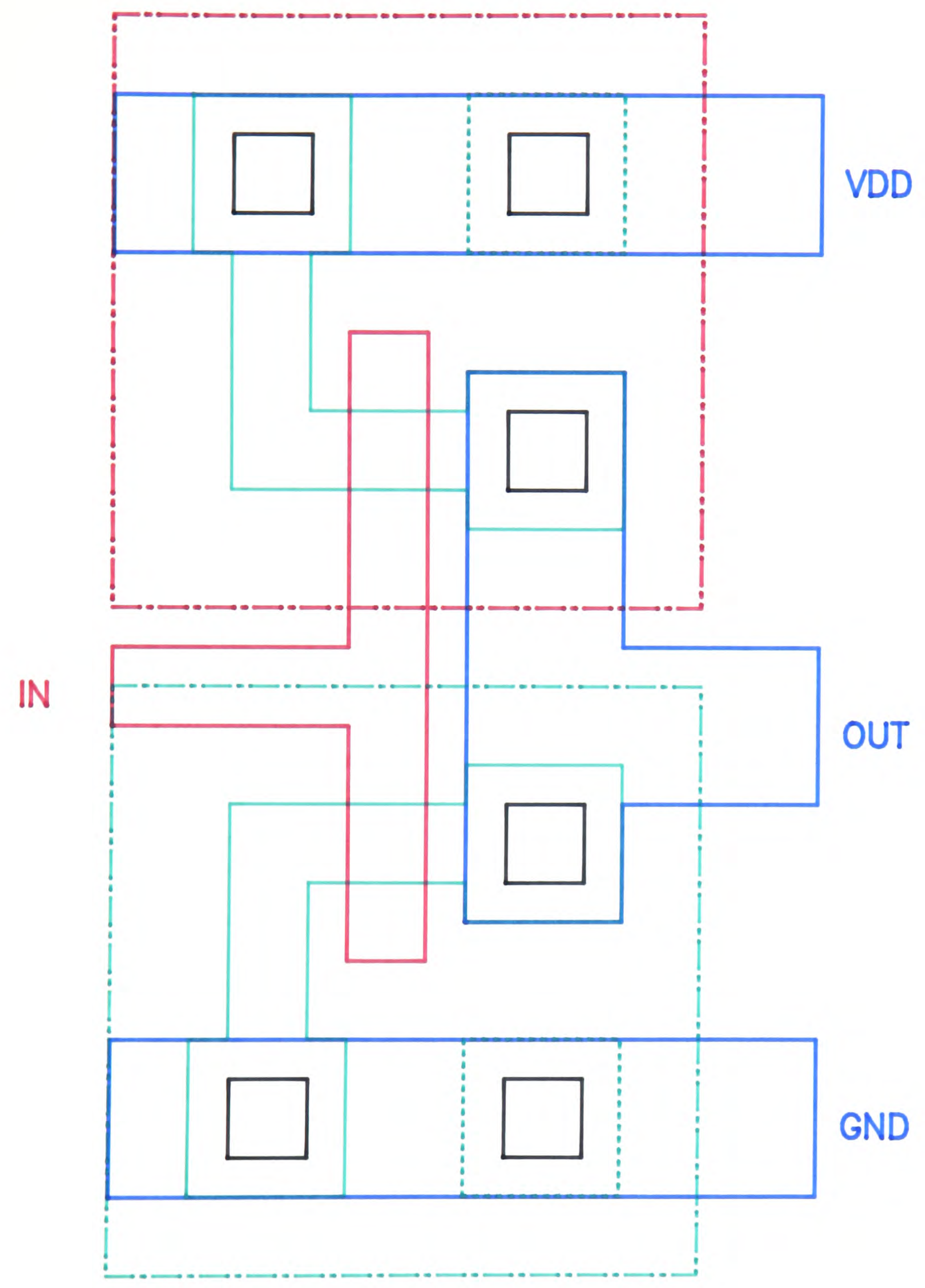


Figure 5–18: Twin–well CMOS Inverter

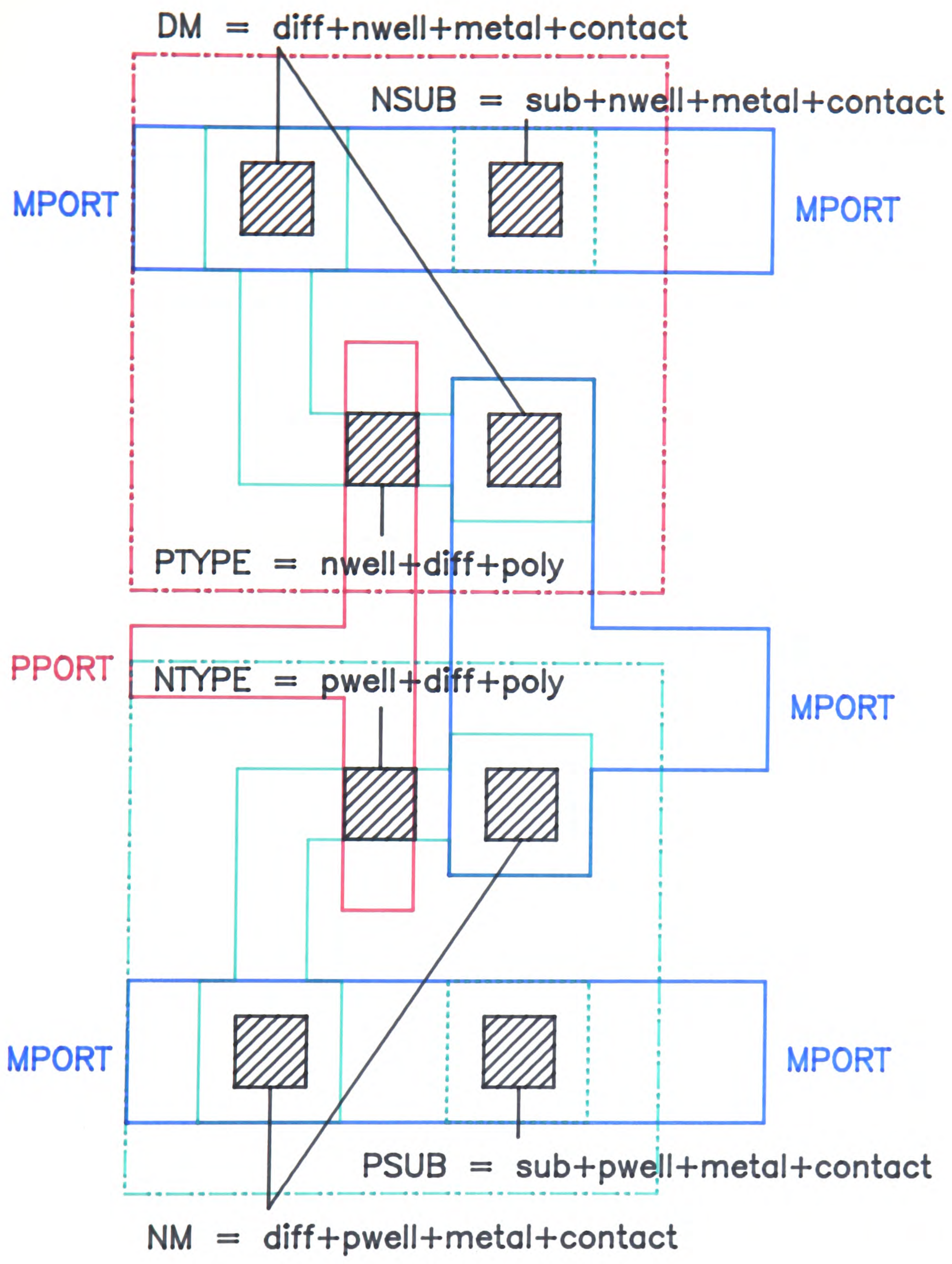

DM = diff+nwell+metal+contact
NSUB = sub+nwell+metal+contact
MPORT
MPORT
PTYPE = nwell+diff+poly
PPORT
NTYPE = pwell+diff+poly
MPORT
MPORT
MPORT
PSUB = sub+pwell+metal+contact
NM = diff+pwell+metal+contact

Figure 5–19: Devices Located By Overlapping Layers
Ports Located at Edges of Cell

to wires in a sticks circuit. In the example process, these layers are as follows.

PWIRE: polysilicon

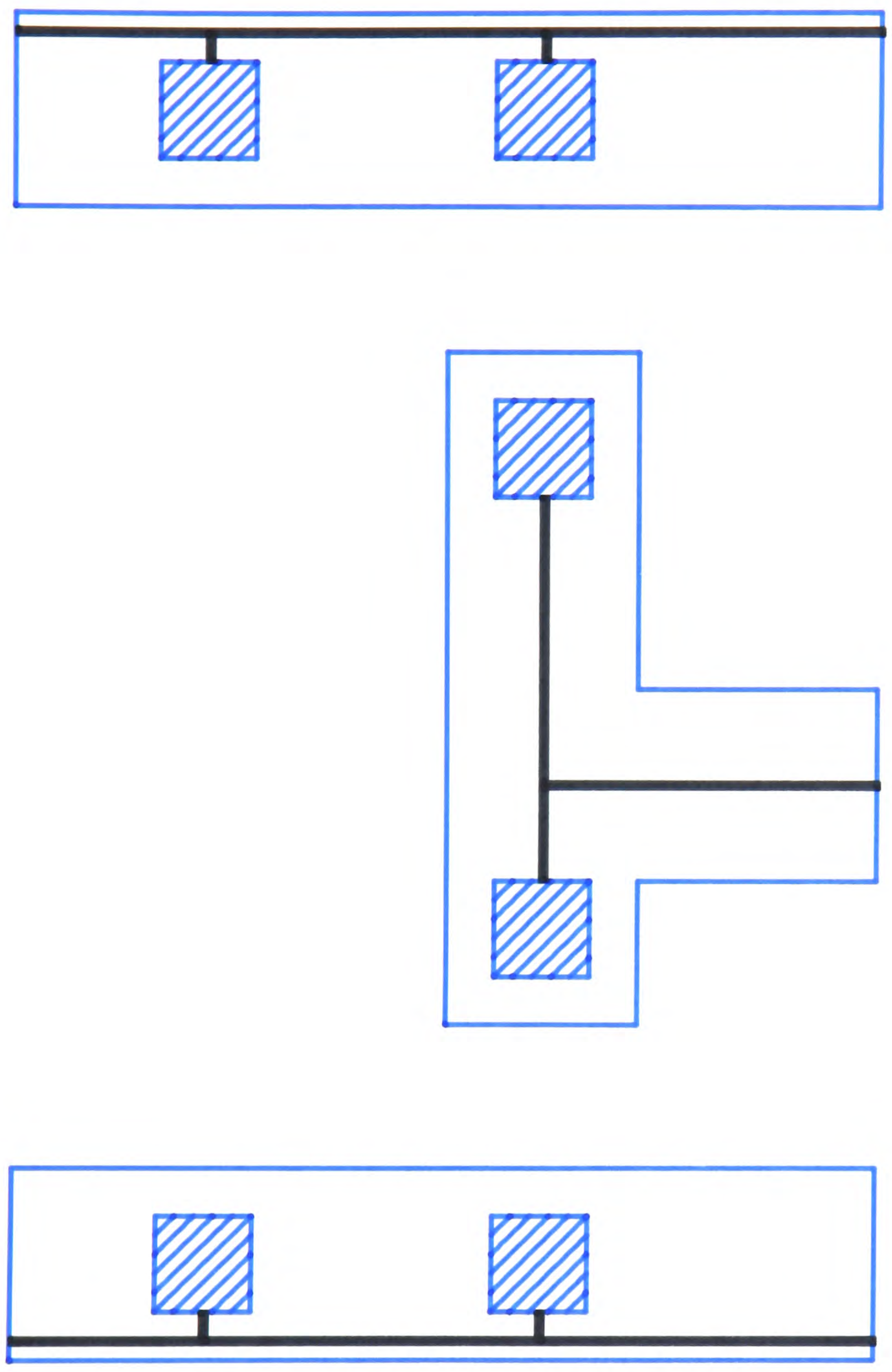
NWIRE: diffusion and pwell

DWIRE: diffusion and nwell

MWIRE: metal

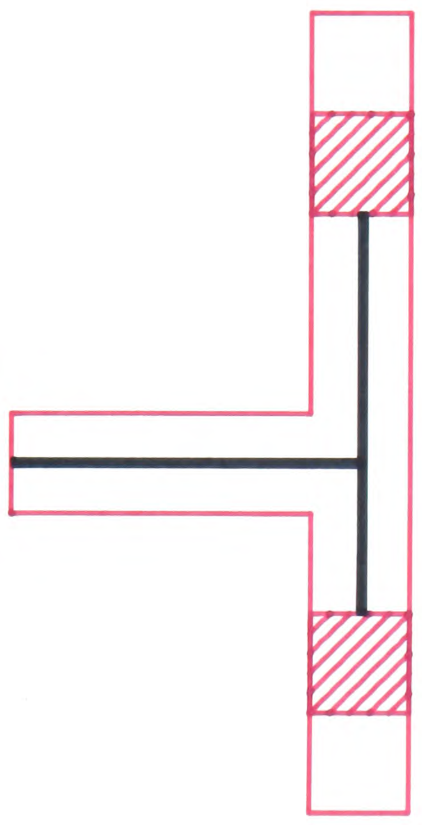
Each separate area on each mask layer will be connected to a number of points where devices have been removed from the circuit. Each area can then be reduced to a number of connected line segments which both connect all such points, and also lie wholly within the particular mask layer area. These line segments correspond to wires in the sticks circuit. Figure 5-20 shows a possible set of wires for the CMOS inverter.

Finally, in order to represent the sticks circuit as a virtual grid, it is necessary to identify a set of "lines of action" which are sufficient to allow all devices and wires to be placed on an integer coordinate plane. The lines of action for the inverter are shown in figure 5-21, and the resulting virtual grid is shown in figure 5-22. Figure 5-23 shows a SOI implementation of the same inverter circuit, illustrating how circuits can be translated automatically from one class of CMOS technology to another.



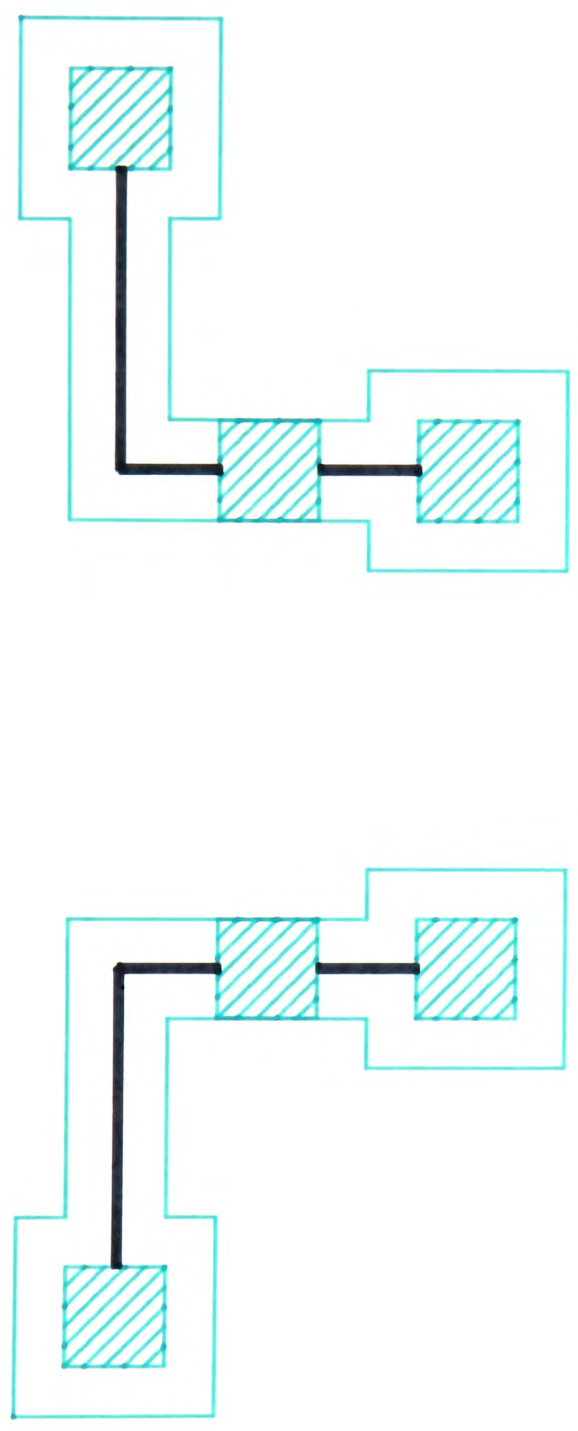
(a) Wires on Metal Layer

.../



(b) Wires on Polysilicon Layer

.../



(c) Wires on Diffusion Layers

Figure 5–20: Finding Wires Joining Ports and Devices

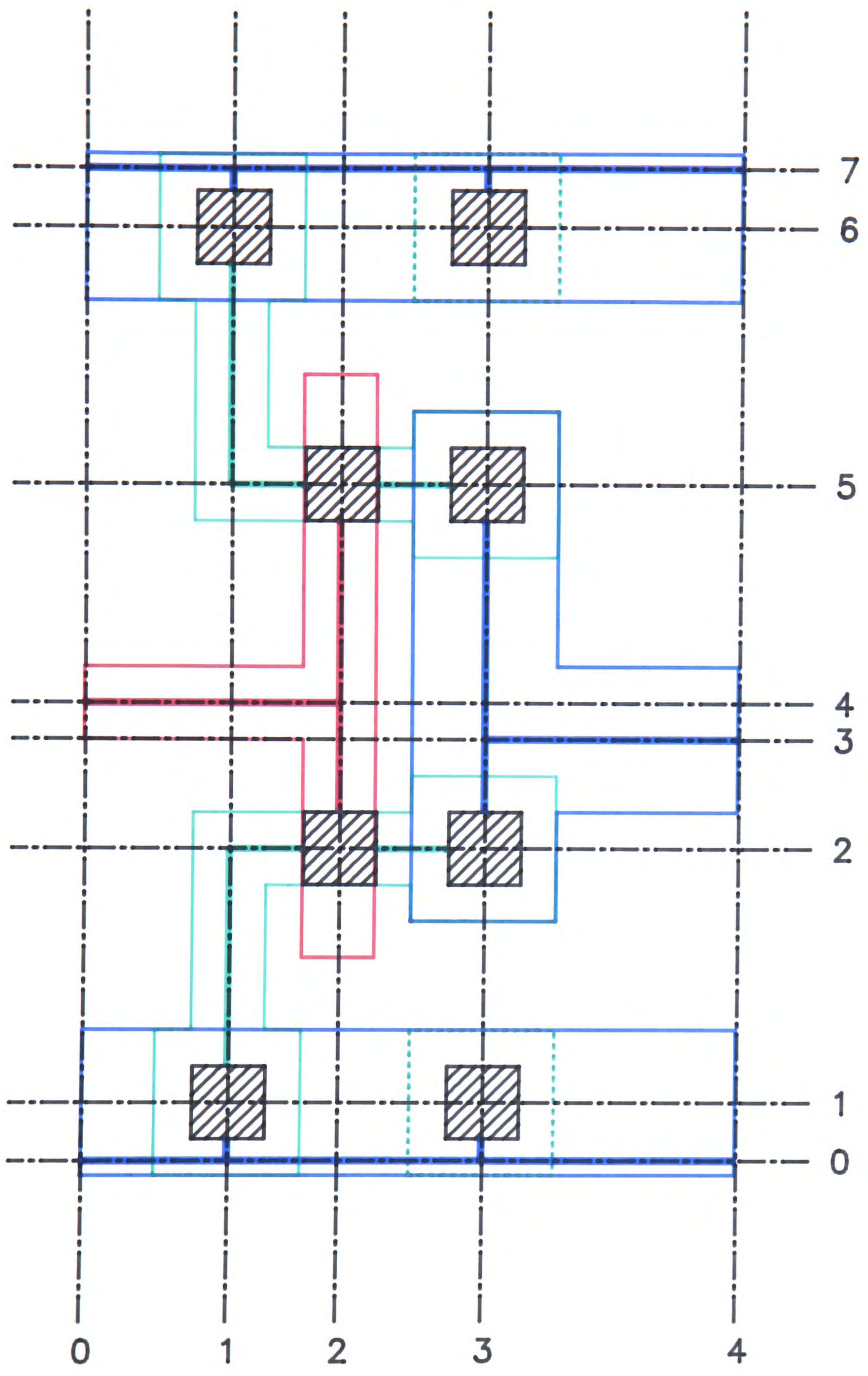


Figure 5–21: Lines of Action in CMOS Inverter

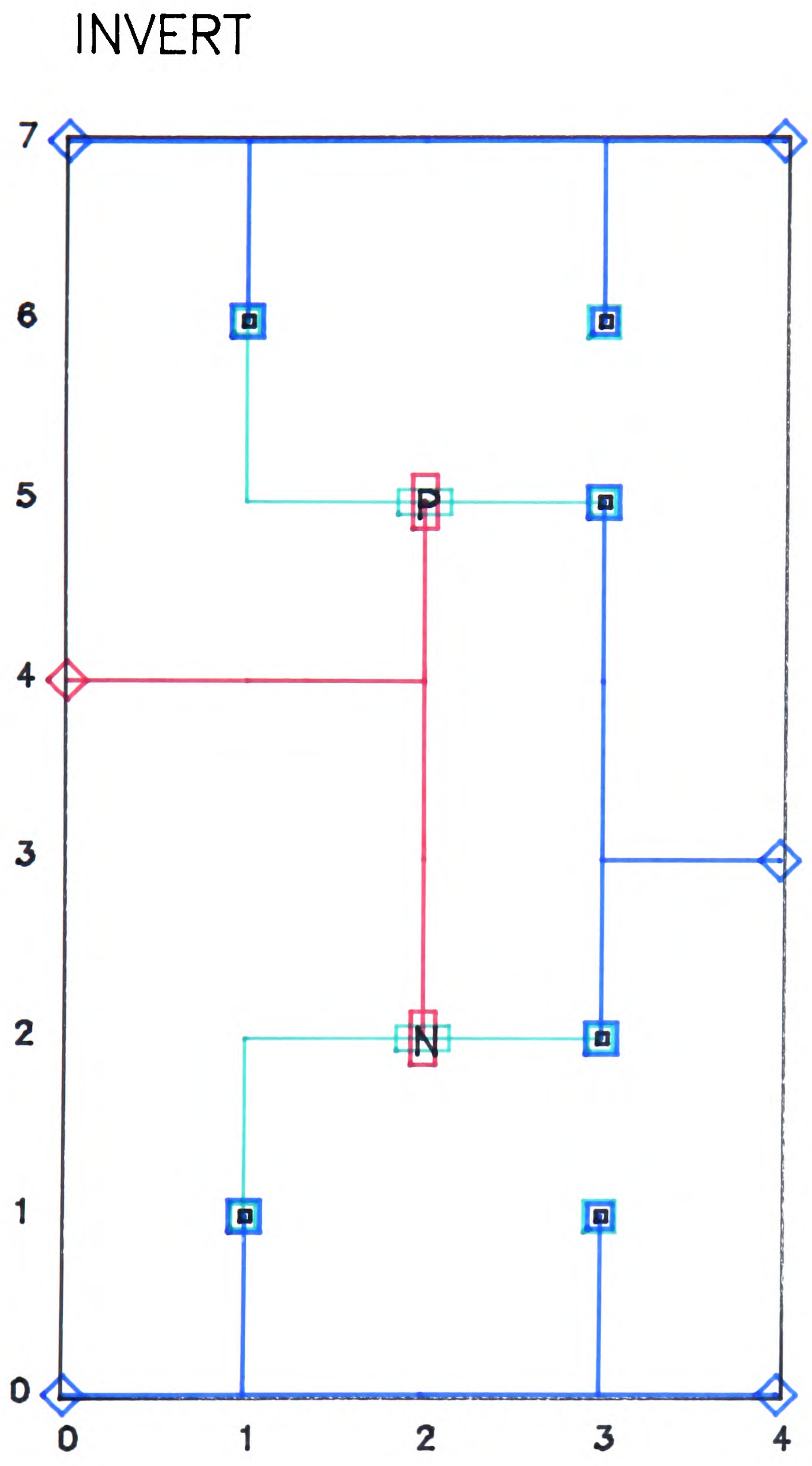


Figure 5—22: Virtual Grid Representation of CMOS Inverter

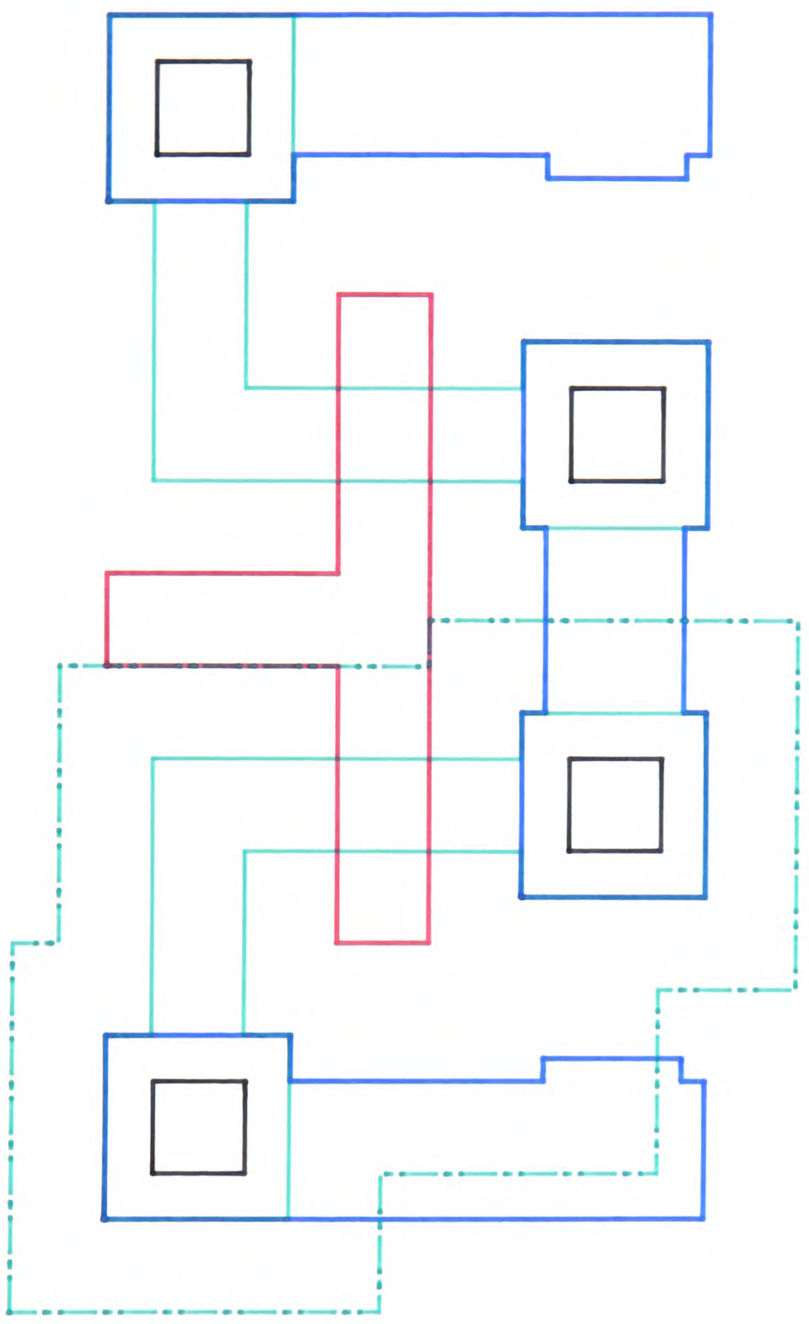


Figure 5-23: Inverter Converted to SOI

Since existing mask level circuits may use constructs which are specific to one technology (such as island to island contacts) or devices such as bent transistors, it may be necessary to do some manipulation on these circuits at the sticks level before recompaction. For example, an island to island contact could be converted to a ptype diffusion to metal contact joined by a metal wire to an ntype diffusion to metal contact.

No software has been produced to implement the sticks extraction algorithm presented here, rather the area of sticks extraction is presented as an interesting idea which has emerged during the course of the main work on the thesis.

5.8: Summary

A sticks compactor has been developed, which while being quite simple, has explored some interesting and novel ideas.

Firstly, the quasi-virtual grid has been introduced as a convenient method of describing mask level circuits, which retains structural information and is inherently stretchable.

It has been shown that predictable, controllable conversion from sticks level to mask level is achieved by adding the concepts of device templates and coordinate mappings to a virtual grid to give a quasi-virtual grid circuit.

Device templates plus a list of layer to layer separations have been shown to be an elegant method of describing simple sets of design rules.

A new CMOS design style called generalised CMOS has been introduced, and a new algorithm for adding CMOS wells in a technology dependent manner has been introduced to support this design style.

Finally, the notion of sticks extraction has been introduced as an aid to the automated conversion of mask level circuits from one technology to another.

6: A PROTOTYPE IMPLEMENTATION OF THE VIRGIL SYSTEM

6.1: System Overview

An experimental implementation of a system for the capture, storage and implementation of MOS idioms has been developed as part of this thesis. Figure 6-1 shows an overview of the structure of this prototype system.

All programs for the VIRGIL system have been written in the general purpose high level programming language IMP [Robertson 83], and have been implemented on a DEC VAX 11/780 running the VMS operating system.

It is intended to describe the function of each component of the software system in the following sections, and also to comment on some issues which have arisen during the production of this relatively large suite of programs.

6.2: A Cell Librarian

The VIRGIL system describes idioms in terms of composition cells and leaf cells. In order that composition cell definitions can access the appropriate leaf cell definitions, it is useful to hold all cell definitions in a single database. A simple database

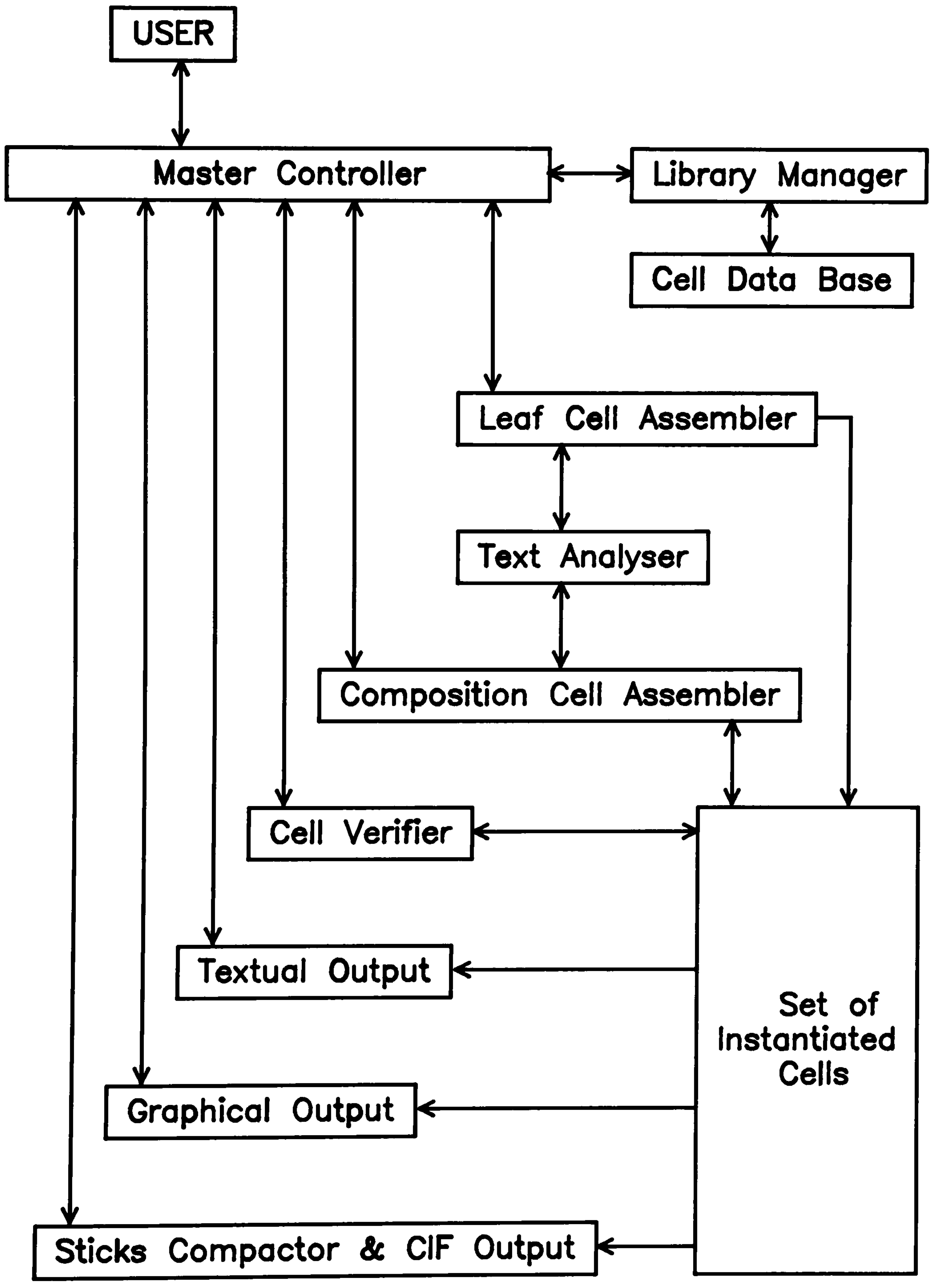


Figure 6-1: Overview of Software Environment

manager has been written to handle such storage of cell definitions.

The basic operation which the database manager performs during idiom instantiation is, given a cell name, to return the definition of that cell. The database manager is therefore more accurately described as a librarian. An interactive interface to the librarian also allows cell definitions to be added, deleted and updated.

6.3: Text Analysis

Text analysis is the name given to the lexical analysis and syntax analysis phases of the VIRGIL language compiler. Since selection and repetition are textual operations, these are performed as part of lexical analysis. Any IF or FOR construct is fully expanded, and resulting lines of text are passed one by one to the syntax analyser.

The syntax analyser uses the recursive descent method of parsing [Davie 81], and is driven from an external list of production rules.

Cell header statements must be analysed before lexical analysis of the body of cell definition commences, so that values of parameters, which may be used subsequently in IF and FOR constructs, are known.

6.4: Internal Representation of a Virtual Grid

Virtual grid representations of instantiated cells are stored internally as a graph, with grid points represented as graph nodes, and the grid lines between grid points represented as bidirectional graph edges. Each graph representing a virtual grid has an accompanying header record which lists the cell name, the ports available for interconnection, and the values of any cell parameters.

Once a leaf cell or an intermediate cell has been composed, a copy of the internal representation of that cell is held in a set of instantiated cells. In this way, if a cell is used more than once in a complete idiom definition, this internal representation need only be generated once from the textual definition. Different values for cell parameters will result in different cell instantiations, hence the need to store parameter values in the cell header. The location of ports within a cell instantiation is stored in the header to assist in port hiding and renaming operations.

6.5: Analysing Leaf Cell Definitions

The header statement in a leaf cell definition includes the virtual grid coordinate bounds for that cell, and so after this statement has been analysed, it is possible to set up the graph used to hold the internal representation of the virtual grid.

Subsequent statements in the fully expanded leaf cell definition consist of the names of structural components and their positions. These components can be added to the graph-base representation independently of each other, and so each statement in the cell definition can be dealt with as it is produced by the text analyser.

Some errors can be dealt with during the analysis of individual statements, such as coordinates outside the stated bounds, and wire paths which are not orthogonal. Errors concerned with invalid circuit constructs are detected later, when the entire virtual grid circuit is checked in the grid verification phase. This grid verification consists of checking the circuit constructs at each grid point in accordance with the rules presented in section 4.7. Detection of such errors is a powerful aid in circuit debugging.

6.6: Analysing Composition Cell Definitions

Each statement in the body of a composition cell definition defines the way in which previously instantiated cells are composed together to form a new intermediate cell. Within a statement, each composition can be considered as a separate operation, namely that of joining the specified cell instantiation to the partly completed intermediate cell to produce a more complete intermediate cell.

Each composition operation is performed by merging grid points and grid lines on the abutting edges of the cells being composed, ensuring that ports which are joined together match in both type and name. Cells can be stretched so that matching ports are brought into alignment, as was described in section 4.5. If a pair of unmatched ports is encountered, then the composition fails and instantiation of the idiom is halted.

In VIRGIL, composition is a simple but surprisingly powerful operation. The fact that cells are automatically stretched during composition to ensure correct connection of ports means that the designer is relieved from the need to "pitch match" adjoining cells, as is required in non-stretchable design styles. Most especially, changes to the pitch of one cell do not require adjoining cells to also be redesigned.

6.7: Outputting the Virtual Grid

Once a cell has been analysed, and an internal representation produced, this internal representation can be output in either graphical or textual form. Both forms are output by scanning the grid in order of increasing Y coordinate, and within the same Y, in order of increasing X coordinate. Devices at nodes and wires on edges are output as they are encountered.

Graphically, wires are output as coloured lines, and devices as symbolic shapes (icons). Use of a largely device independent graphics package [Hughes 81] allows graphical output on a number of hard copy and screen oriented graphics devices.

6.8: Conversion from Virtual Grid to CIF 2.0

Conversion from the internal representation of a virtual grid to a mask level representation in the language CIF 2.0 [Mead 80] is achieved by first producing a quasi-virtual grid of the type described in chapter 5.

As device templates are created, they are also output to a CIF file as CIF "symbols". Each device in the virtual grid, including null devices, is then converted to CIF by calls to these symbols at the appropriate physical positions, as determined by the quasi-virtual to physical coordinate mapping. Wires are constructed as rectangles of the appropriate width and stretching between physical grid line coordinates.

The overlap of wires and devices gives the complete mask level circuit.

6.9: User Interface

To instantiate an idiom, the user must supply the name of the cell defining the idiom, the database in which this cell definition is to be found, the parameter values which specify the particular instantiation of the idiom, and the form in which the instantiation is to be output.

The output format may be either mask level textual representation (CIF 2.0), virtual grid textual representation, or virtual grid graphical representation. The format or formats to be output are specified by the user as VAX/VMS command language qualifiers [VAX 78].

Other information regarding the particulars of the idiom are specified interactively in response to prompts issued by the idiom instantiation program. The following is typical of a terminal dialogue to produce an idiom instantiation. Text output by the program is underlined.

text enclosed in braces is annotation and not part of the dialogue.

```
$ vgdo/plot/compact  {output plotted and in cif}
DATABASE?: cells      {library name is "cells"}
CELL?: shiftreg       {name of cell definition}
BITS?: 4              {parameter values}
WORDS?: 3
$
```

For some structures such as PLA's and ROM's, large arrays of parameters would be needed to specify the exact idiom instantiation, and it would be tiresome and error prone to enter these interactively. The system therefore allows such lists of parameters to be input from an external file, which in turn may have been generated by some other program (e.g. the PLA parameter generator described in chapter 7). A typical terminal dialogue is such a case might be:

```
  $ vgdo/compact
  DATABASE?: pla
  CELL?: pla
  INPUTS?: @data.pla
           {take further input from file "data.pla"}
  $
```

In the current implementation, lines of text as output from the text analyser, and also information regarding various stages of translation to mask level are displayed on the terminal so that the user can have some feedback as to the progress of the current program run.

6.10: Some Implementation Issues

Particular problems arise when programming "in the large", i.e. when composing and debugging large computer programs. Such problems have mostly to do with being able to control the complexity of such large programs, and these can best be solved by imposing some sort of hierarchy. The methods used to overcome these problems in this particular implementation of a large system are worthy of brief comment.

The VIRGIL system consists of a single executable program, which in turn is composed of a large number (about 20) of separately compiled modules, held in a so-called object library. Each module is written in the programming language IMP, which has only moderate support for separately compiled modules. This support consists of so called include files, which allow arbitrary text from another file to be included in the source code of a program, and also the facility to declare routines and variables which are externally visible from separately compiled modules other than the one in which they are declared.

In the VIRGIL system, each module includes one or more externally visible routines (and in a few cases externally visible variables), through which all "communication" to and from the module is made. All other internal structure of the module is hidden from other modules, giving an easily managable method of preventing interference between modules.

The source code for each module *includes* a global declaration file. This file consists of a list of global constant and data-type declarations, plus the specifications of all externally available routines in all modules. A copy of this file appears in appendix A. Use of this single global file for such declarations ensures that all modules maintain a consistent view of the system as a whole. Most especially, all modules use exactly the same data type and constant definitions, and also specifications for externally available routines can all be checked with the actual definitions of the routines.

Partitioning of the complete VIRGIL software (about 10,000 lines of code), into a number of relatively simple modules (about 200 to 1000 lines of code each) allowed each module to be written, tested and added to the object library relatively independently of other modules.

Newer programming languages, notably ADA [ADA 79], specifically provide language features to support modular development of programs, and so a discussion of these issues might seem out of place in a thesis such as this. The motivation for including this discussion is to indicate that with care and discipline most of the advantages of modular programming are available with currently available languages, and furthermore that the author has found these techniques to be most useful, and recommends such a structured, modular approach to others undertaking a project of similar size.

7: EXAMPLE IDIOMS

7.1: Shift Register

A simple example of an idiom - a generalised shift register - has already been presented in section 4.8. Figure 7-1 shows a mask level representation of the 2 bits wide by 4 clock phases long shift register instantiation discussed in section 4.8. A mask level representation of a different instantiation of the same idiom, this time 9 bits wide by 3 clock phases long, is shown in figure 7-2.

7.2: Programmable Logic Array

7.2.1 Introduction

PLA's (Programmable Logic Arrays) are a widely known idiom for implementing irregular combinatorial functions using a regular structure. For those unfamiliar with PLA's, a good introduction appears in section 3.10 of Mead and Conway [Mead 80].

Since one aim of this thesis is to investigate the capture of known idioms, it was decided to attempt the capture, as exactly as possible, of an existing PLA idiom. Since the PLA is quite a complicated idiom in terms of the number of different leaf cells needed to

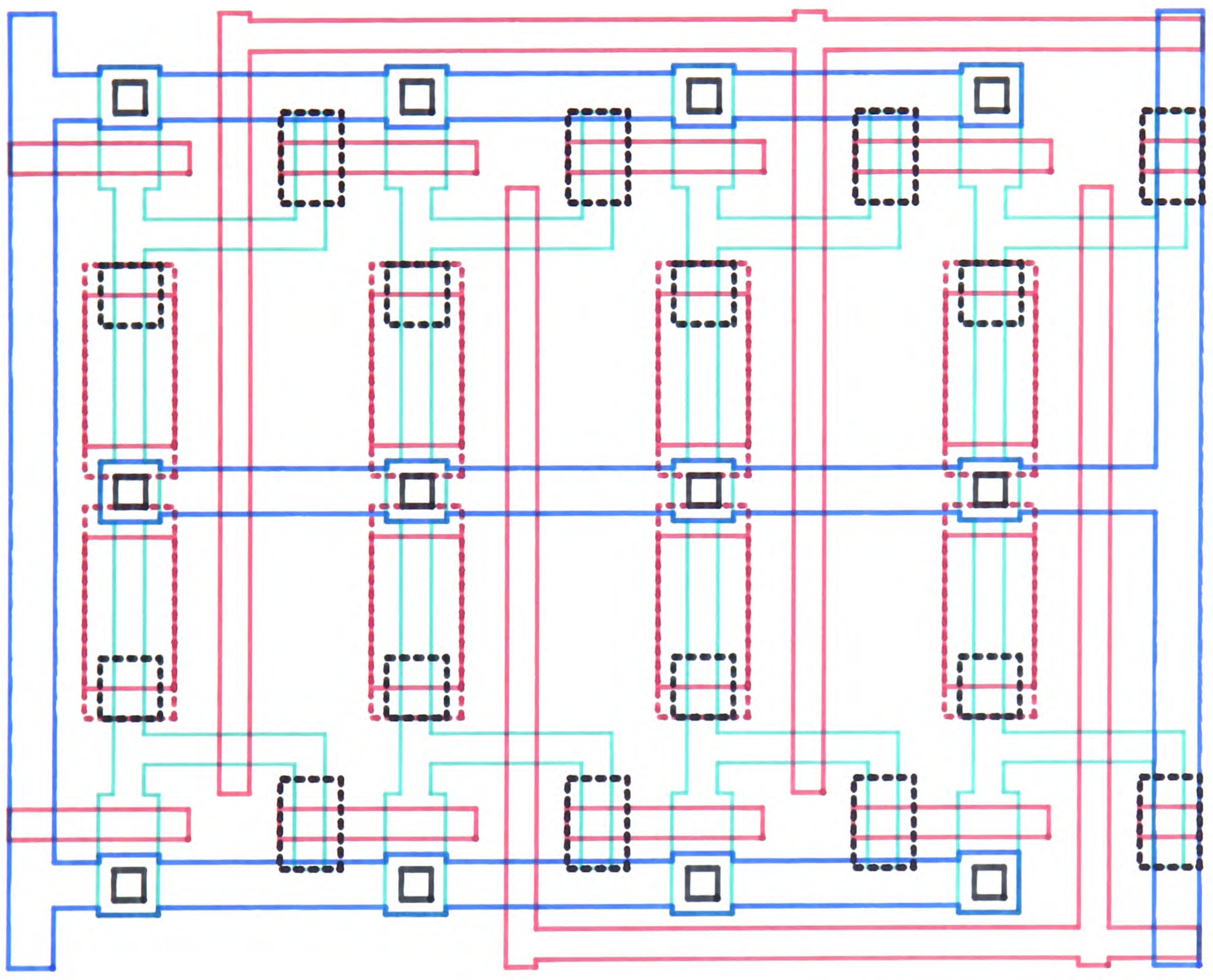


Figure 7–1: 2 X 4 Shift Register Instantiation

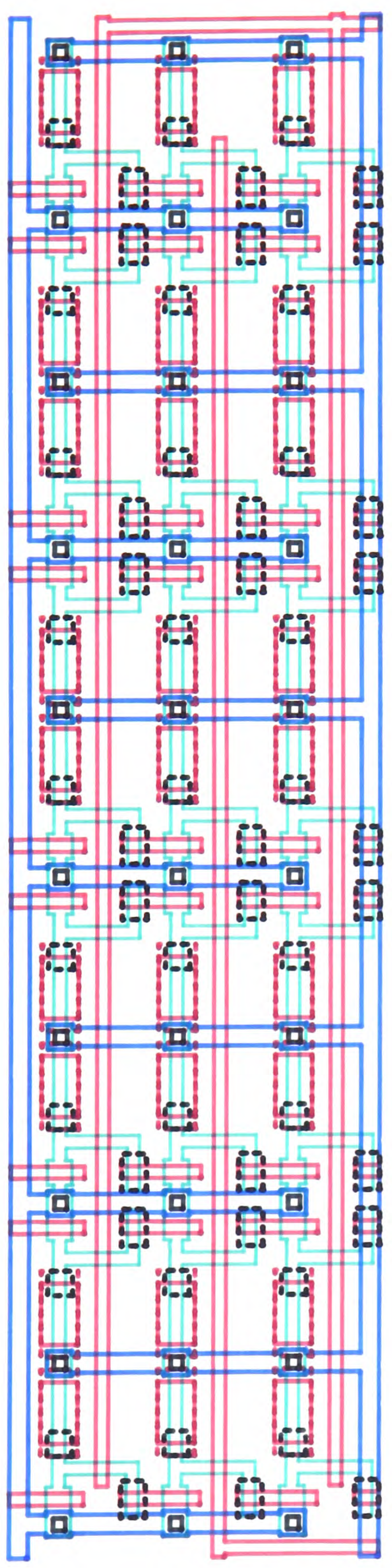


Figure 7–2: 9 X 3 Shift Register Instantiation

describe it, and also because of the high degree of parameterisation, the study of this idiom provides good insight into the use of VIRGIL as a means of idiom capture.

"The VLSI Designer's Library" [Newkirk 83] is a publication which gives details of a large collection of leaf cell descriptions (at mask level), and also some information on how these can be composed together to form larger structures. As such it represents one of the few attempts to make a collection of useful idioms publicly available.

Included in this collection is a set of cells for the construction of various types of PLA's. It has been decided to implement a simple clocked input, clocked output PLA generator using the cells presented in this collection. The capture of this one idiom will be done at both mask level (i.e. a "traditional" PLA generator), and also using the VIRGIL system, so that the two methods can be directly compared.

7.2.2: Capture at the Mask Level

"The VLSI Designer's Library" describes a collection of leaf cells for composing PLA's, giving a graphical representation of each cell, the CIF 2.0 representation of each cell, the sizes of the cells, and in a few cases some information about how certain cells abut to other cells. A picture of a completed PLA is the only information which details the way the individual cells are arranged to produce the complete circuit. The reader is left to deduce the exact nature of the composition, and to capture this composition using whatever design system is available. Again this highlights the lack of any widespread, standard language for the description of parameterisable designs.

The leaf cells presented in "The VLSI Designer's Library" are easily captured simply by copying the CIF descriptions of the various cells. Graphical representations of the various leaf cells, including some wiring cells not included in the book, are shown in appendix B.

The composition of these cells has been described using the embedded, mask level IC design language ILAP [Hughes 83]. With a few exceptions, cells are composed by simple abutment. Although ILAP does not directly support the abutment of cells as a primitive operation, it can be implemented by a sequence of operations such as:-

```
                .

                .

     Move to position (X,Y)

          Place Cell A

Move to position (X',Y), where X' = X + width of A

          Place Cell B

                .

                .
```

Such composition merely places cells next to each other, and does nothing to ensure that correct interconnections between the two cells being abutted have been made.

The user of a PLA generator needs some method of specifying the various parameters which define a particular PLA instantiation. Within the Computer Science department at Edinburgh University the design tools include several programs for use in the design and specification of PLA's [Hughes 83]. To specify PLA parameterisation, i.e. the number of inputs, outputs and minterms, and also the programming of the AND and OR planes of the PLA, a form of table is used. This tabular form is also used for the specification of parameters for the PLA generator discussed here. A full description of this form appears in section 3.1 of "VLSI Design Tools" [Hughes 83], but the basics are that entries in the table for the AND plane may take the values:

X = Minterm does not use input term

1 = Minterm uses input term

0 = Minterm uses complement of input term

and for the OR plane:

1 = Output uses minterm

0 = Output does not use minterm

Figure 7-3 gives an example of such a PLA parameter table (for the traffic light controller described in section 3.11 of Mead and Conway, after appropriate logic minimisation), and figure 7-4 shows the PLA corresponding to these parameters, as generated by the mask level PLA generator.

7.2.3: Capture Using VIRGIL

The mask level generation of PLA's is achieved by composing cells using a series of vertical and horizontal abutments. Since this type of composition by abutment is supported directly in VIRGIL, all that is necessary to capture the PLA idiom is to convert each mask level leaf cell to a VIRGIL leaf cell, and then describe the manner of their composition in a VIRGIL composition cell.

The programming of mask level PLA AND plane and OR plane cells is achieved by overlaying the leaf cell "PlaCell" with another cell, such as "PlaProgLeft" (see appendix B). In the VIRGIL leaf cells, this

```
IN   C,TL,TS,Y0,Y1
     X  X  X  0  X    0  0  0   0   0   1   0
     X  X  X  0  1    1  0  0   0   1   0   0
     X  X  1  0  1    0  1  1   0   0   0   0
     0  X  X  1  1    0  1  1   1   0   0   0
     X  1  X  1  1    0  1  1   1   0   0   0
     X  X  1  1  0    0  0  1   1   0   0   1
     X  X  0  1  0    0  1  0   1   0   0   1
     1  1  X  0  0    1  0  1   0   0   0   0
     1  0  X  1  1    1  1  0   1   0   0   0
OUT                  Y1,Y0,ST,HL0,HL1,FL0,FL1
```

Figure 7–3: PLA Parameter Table

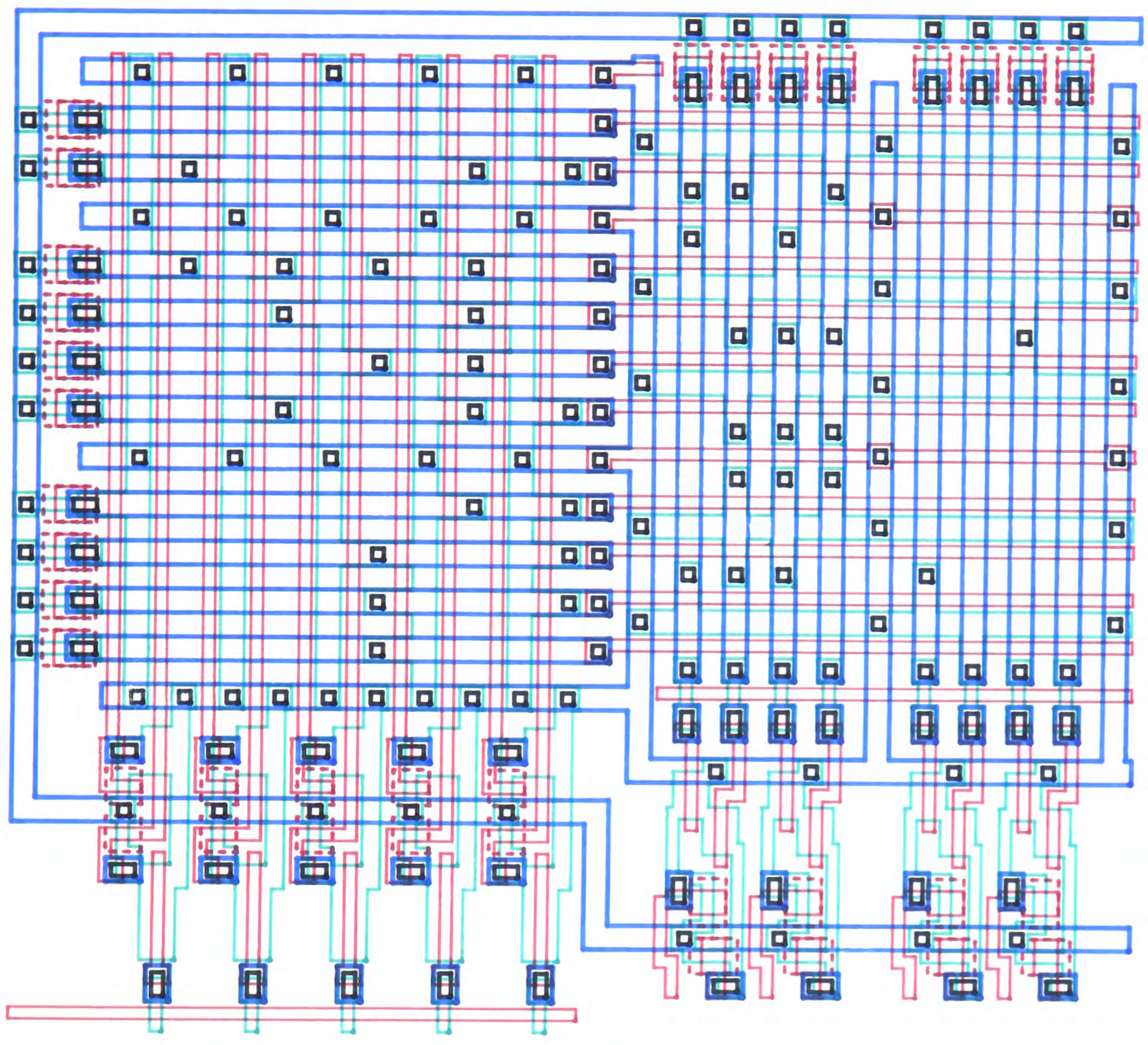


Figure 7-4: PLA Generated at Mask Level

programmability can be more directly expressed by making "PlaCell" a parameterisable leaf cell. Another feature of programming a PLA is that adjacent programming cells can share a single contact between the drain of the programming transistor and the minterm line. This is achieved in VIRGIL by placing these contacts on the edges of cells, so that if contacts appear on the abutting edges of two cells then they will be merged.

The textual descriptions of the VIRGIL PLA leaf cells, and of the PLA composition cell, plus graphical representations of some leaf cell instantiations, and an example of a complete idiom instantiation are all given in appendix C.

Since VIRGIL PLA parameters are expressed as a set of integers for the number of inputs, outputs and minterms, plus boolean arrays representing the programming of the AND and OR planes, a program has been written to convert from the table form of parameter specification described earlier to the list of integer and boolean values needed for the idiom instantiation program.

This program was used to process the PLA specification in figure 7-3. The resulting list of parameters were then presented to the idiom instantiation program, and after conversion to mask level, the circuit shown in figure 7-5 was produced. A minimum metal wire width of 4

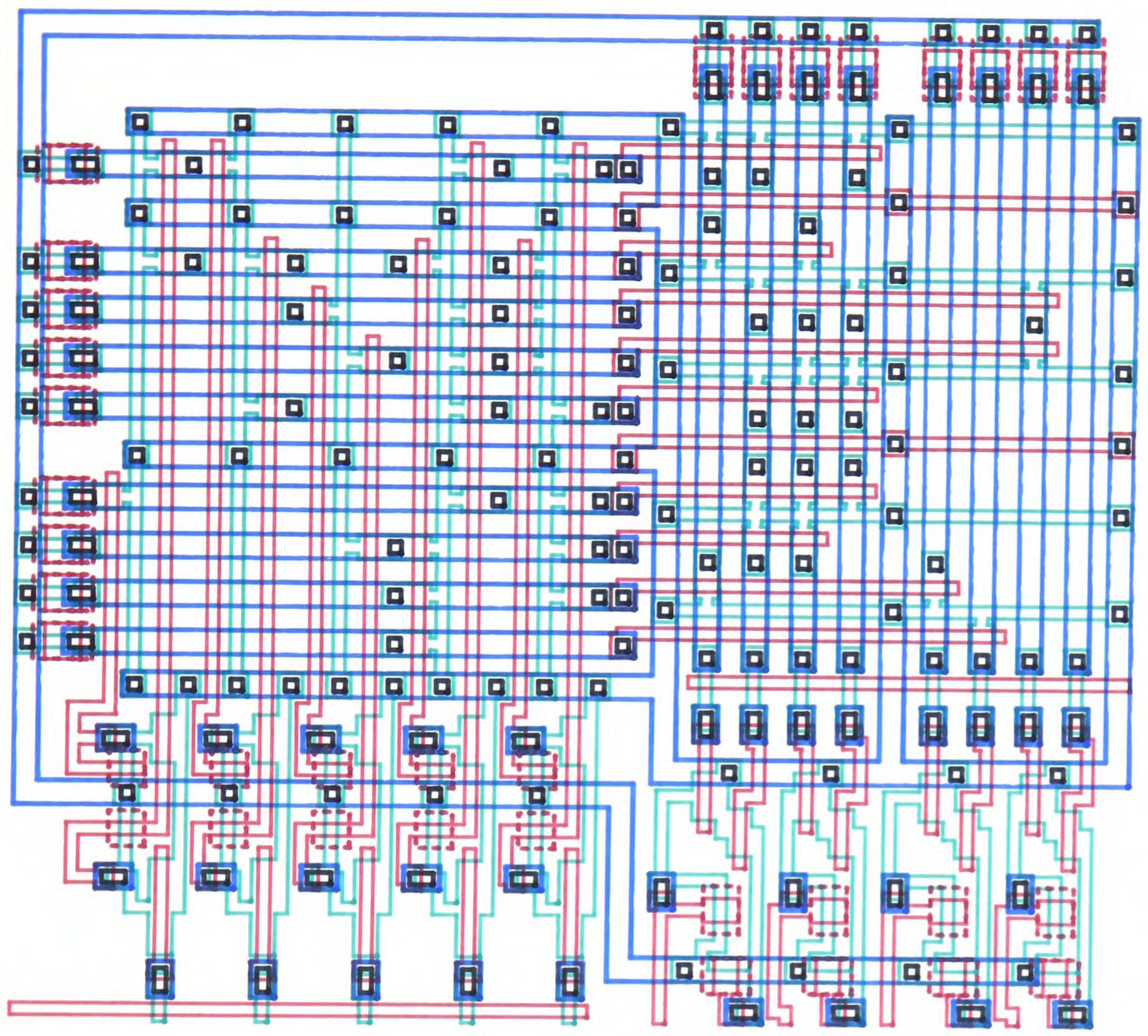


Figure 7–5: PLA Generated by VIRGIL

lambda was used during compaction to more closely model the mask level PLA generator, which uses 4 lambda wide metal throughout.

7.2.4: Comparison and Discussion

Perhaps the most striking differences between the mask level PLA and the VIRGIL PLA are due to the effect of automatic wire trimming. Wires are reduced in length (at the virtual grid level) to be only as long as is necessary. In the example of the PLA used, only 9 minterms are used, but since minterm rows are added in pairs a blank minterm line appears along the top of the PLA. The wire trimming algorithm has detected that this wire is connected at only one end to another object, and so has removed it completely, including the unused pullup at the left hand side. Minterm lines across the OR plane and input lines up through the AND plane have also been shortened.

Overall, the sizes of the two PLA's are almost identical, 186 by 167 lambda for the VIRGIL PLA, 188 by 170 lambda for the mask level PLA. A more sophisticated compaction algorithm could probably decrease the size of the VIRGIL PLA still further.

The relatively good compaction of the virtual grid representation of the VIRGIL PLA was achieved by taking advantage of the controllablity (ability to change final layout by changing virtual grid layout) and the predictability (knowing the effect that changes will be likely to have) of the sticks compactor. The cells were originally laid out on the virtual grid by capturing the precise topology of the corresponding mask level circuit. This was then compacted, and areas in which the circuit would obviously benefit from small changes were then adjusted. Only the design of the relatively complicated input and output buffer driver cells had to be iterated to achieve a good overall layout.

Unlike with the mask level PLA generator, the VIRGIL system can check the validity of all cell interconnections as cells are being composed. The design rule free nature of sticks design ensures that no geometric design rule violations will appear in the final circuit, and also that this single PLA idiom description will be valid for a wide range of different NMOS butting contact processes, not just those which conform to Mead and Conway lambda based rules. The description of the way in which cells have been composed to give a complete idiom is neat and concise. Finally, all these advantages have been gained without any overall increase in circuit area.

7.3: Parallel Multiplier

7.3.1 A Hardware Algorithm

In this example, a single idiom - a parallel multiplier - has been designed which performs a complete LSI function, so being suitable for hardware implementation without the need for any extra circuitry.

Multiplication of binary numbers can be achieved by a successive shift and add algorithm, much the same as is used in manual decimal number multiplication. Figure 7-6 shows how two four bit numbers could be multiplied by a series of shifts and additions. The algorithm represented by figure 7-6 maps very neatly into hardware. Figure 7-7 shows external connections to a gated full adder circuit which performs the binary addition:

(S,A.B,C) -> (S',C') where S' = sum, C' = carry

Such cells can be arranged in an array as shown in figure 7-8 so as to perform binary multiplication of the unsigned binary integers A and B. Rearranging the array slightly gives the square array of figure 7-9. Diagonal connections can be replaced by routing these connections through adjoining cells, giving the array shown in figure 7-10. This array is particularly attractive for VLSI

$$\begin{array}{cccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & X & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & & B_0.A_3 & B_0.A_2 & B_0.A_1 & B_0.A_0 \\
 & & & B_1.A_3 & B_1.A_2 & B_1.A_1 & B_1.A_0 & \\
 & & B_2.A_3 & B_2.A_2 & B_2.A_1 & B_2.A_0 & & \\
+ & B_3.A_3 & B_3.A_2 & B_3.A_1 & B_3.A_0 & & & \\
\hline
S_7 & S_6 & S_5 & S_4 & S_3 & S_2 & S_1 & S_0
\end{array}$$

Figure 7–6: Multiplication by Shift and Add

C' = carry bit of (A.B + S + C)

S' = sum bit of (A.B + S + C)

A,B connected through

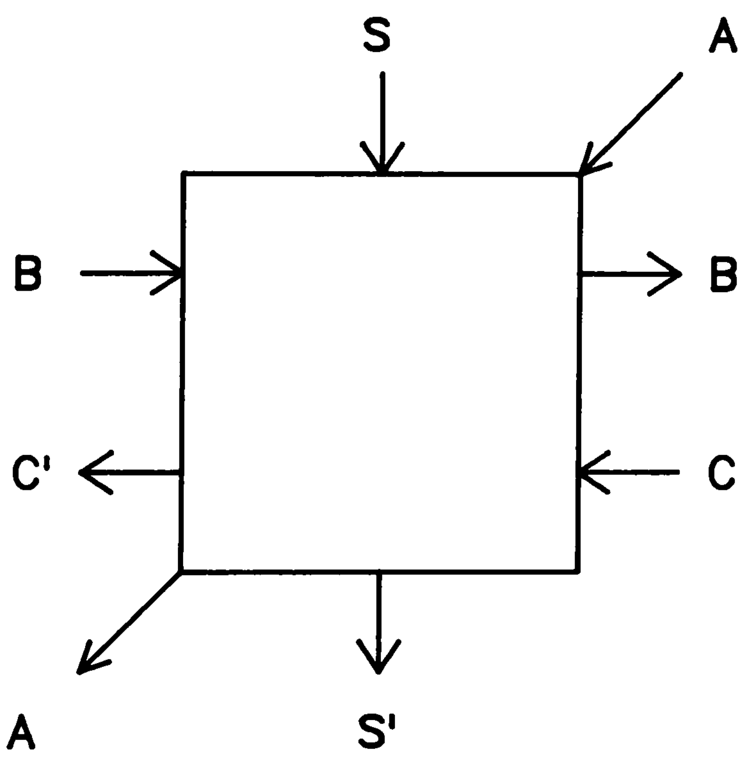


Figure 7–7: Gated Full Adder Cell

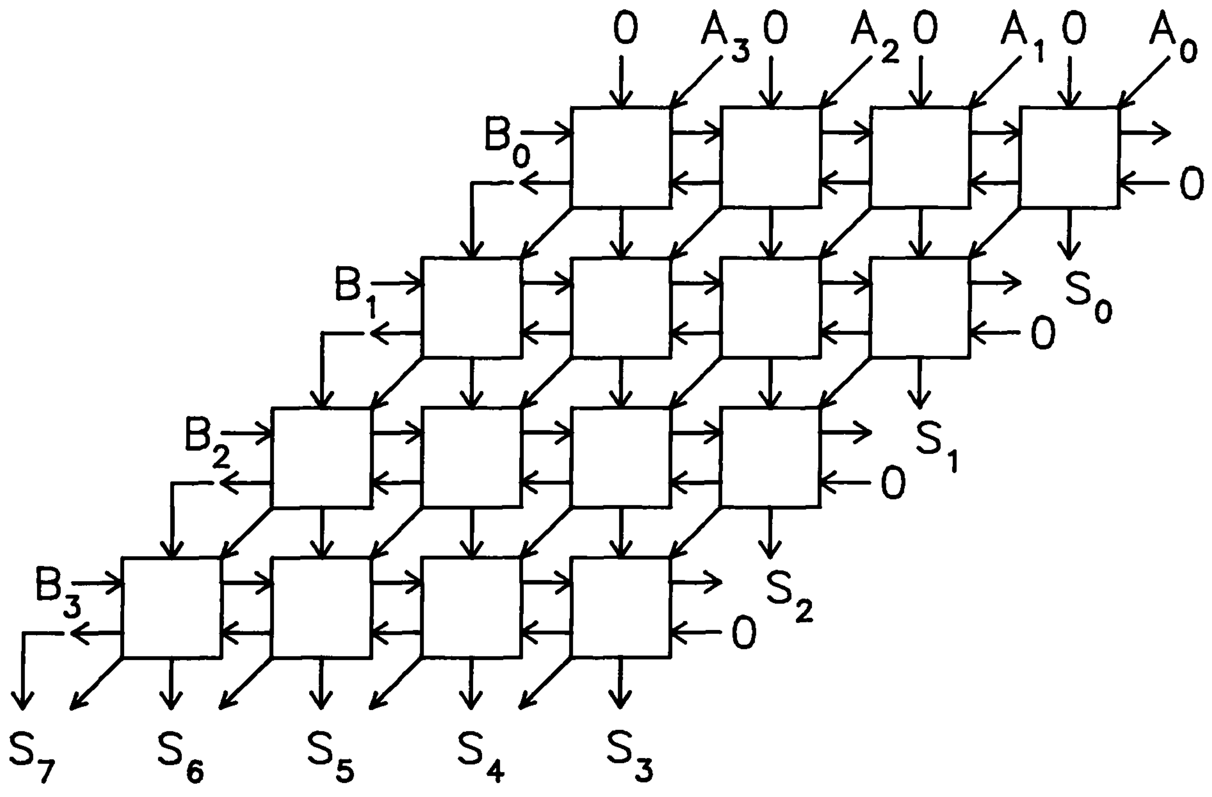


Figure 7–8: Array of Cells to Perform Multiplication

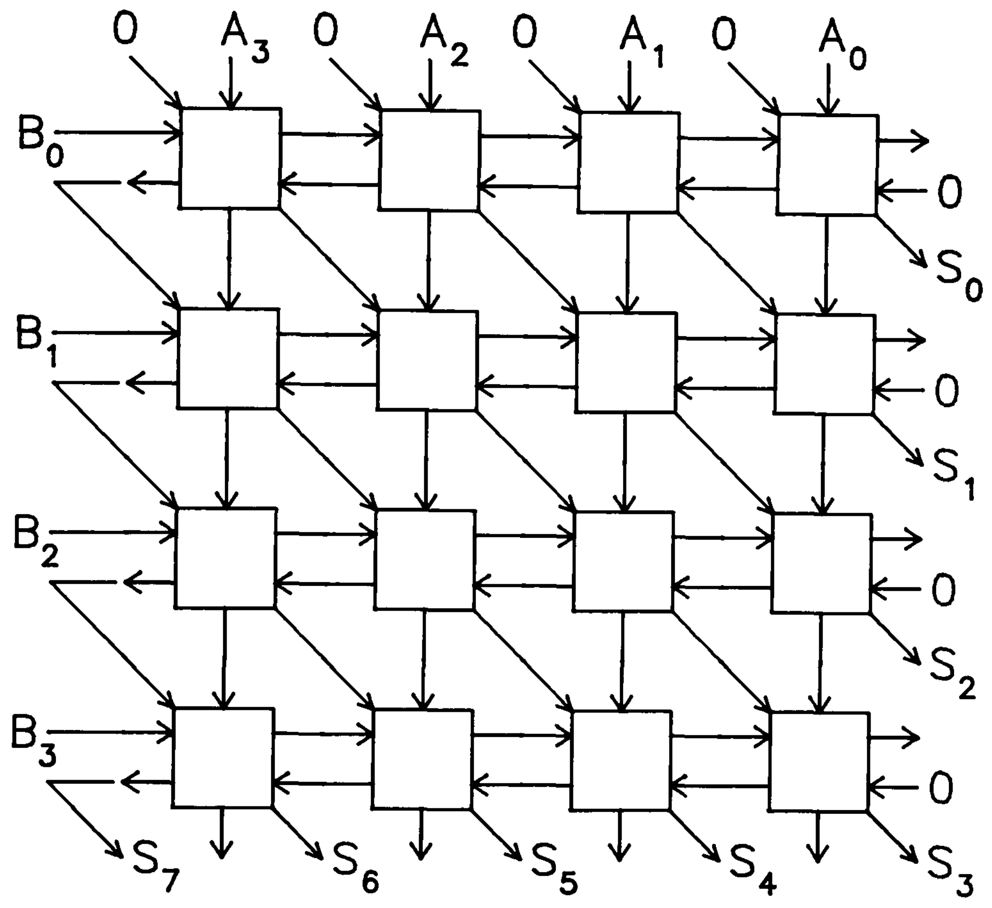


Figure 7—9: Array Rearranged to be Rectangular

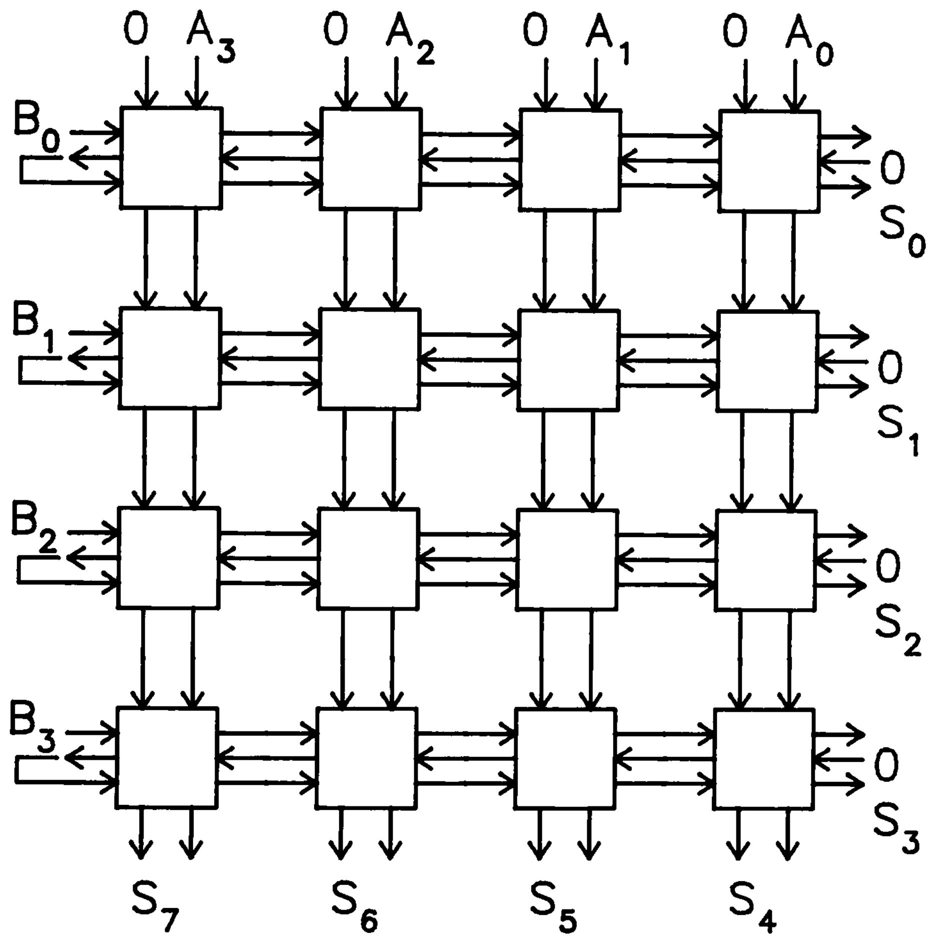


Figure 7–10: Array with only Orthogonal Connections

implementation because every cell communicates only with its four nearest neighbours, and so no global wiring is needed. 'A' inputs come in from the top of the array, 'B' inputs come in from the left side, and outputs come out from the right side and from the bottom of the array.

The basic cell from which the array is composed is shown in figure 7-11. The outputs Scarry, A and B are all connected directly to the corresponding inputs. The other outputs are given by:

$S' = S * (A.B) * C$ (* = exclusive OR)

$C' = S.(A.B) + S.C + (A.B).C$

The adder inputs S and (A.B) can be recoded to give signals P (propogate) and G (generate) to reduce ripple carry propogation time [Mead 80]. The resulting equations are then:

$P = (A.B) * S$

$G = (A.B).S$

$S' = P * S$

$C' = P.C + G$

7.3.2: A Multiplier Circuit

For simplicity, it was decided to implement the adder circuitry using only logic gates, rather than any more conceptually complex structures such as pass transistor chains. The resulting NMOS gates then correspond to the

$C' =$ carry bit of $(A.B + S + C)$

$S' =$ sum bit of $(A.B + S + C)$

A, B, S_{carry} connected through

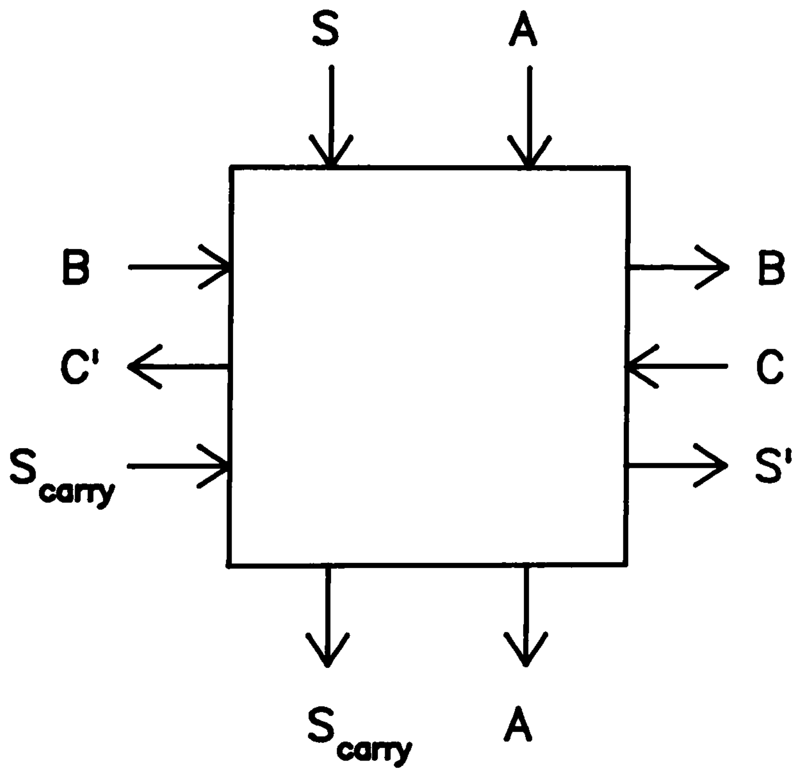


Figure 7–11: Basic Multiplier Array Cell

following equations:

$$\begin{aligned}
\text{Gbar} &= \overline{\text{A.B.C}} \\
\text{Pbar} &= \overline{\text{((A.B)+S).Gbar)}} \\
\text{P} &= \overline{\text{Pbar}} \\
\text{Cbar} &= \overline{\text{C}} \\
\text{C'} &= \overline{\text{Gbar.(Pbar+Cbar)}} \\
\text{S'} &= \overline{\text{(Cbar+P).(C+Pbar)}}
\end{aligned}$$

These gates have then been implemented in NMOS as a virtual grid circuit, described by a VIRGIL leaf cell definition.

When these cells are arranged to form a multiplier, the 'S' inputs for the top row of cells, and the 'C' inputs for the right side cells are unused and can be tied to ground. For cells on the left side of the array, the C' output and the Scarry input need to be connected together. Rather than add these connections externally, the VIRGIL leaf cell definition can be made parameterisable to conditionally include them internally in the appropriate cells.

As in the shift register idiom described earlier, common power and ground lines can be added up the left and right sides of the complete array. In this case these lines are conditionally included in the single multiplier leaf cell definition.

The manner in which an array of these cells is composed to give a complete multiplier is then very straightforward - individual cells are built up into rows, and these rows are composed to give the complete idiom.

The VIRGIL leaf cell and composition cell definitions for this multiplier are given in appendix D, along with a selection of graphical representations of leaf cell instantiations.

7.3.3: A Multiplier Chip

A 4 by 4 bit multiplier has been instantiated and converted to mask level. With the aid of an automatic pad placement program included in the Edinburgh University Computer Science department VLSI design tools [Hughes 83], pads have been added to this multiplier circuit to give a complete multiplier chip. The layout for this chip is shown in figure 7-12.

The chip has been fabricated using a 6 micron NMOS process. Total circuit size is about 2 mm square. The fabricated devices have been tested, and some working parts have been obtained. With zero volts substrate bias, worst case multiplication time is about 400 ns.

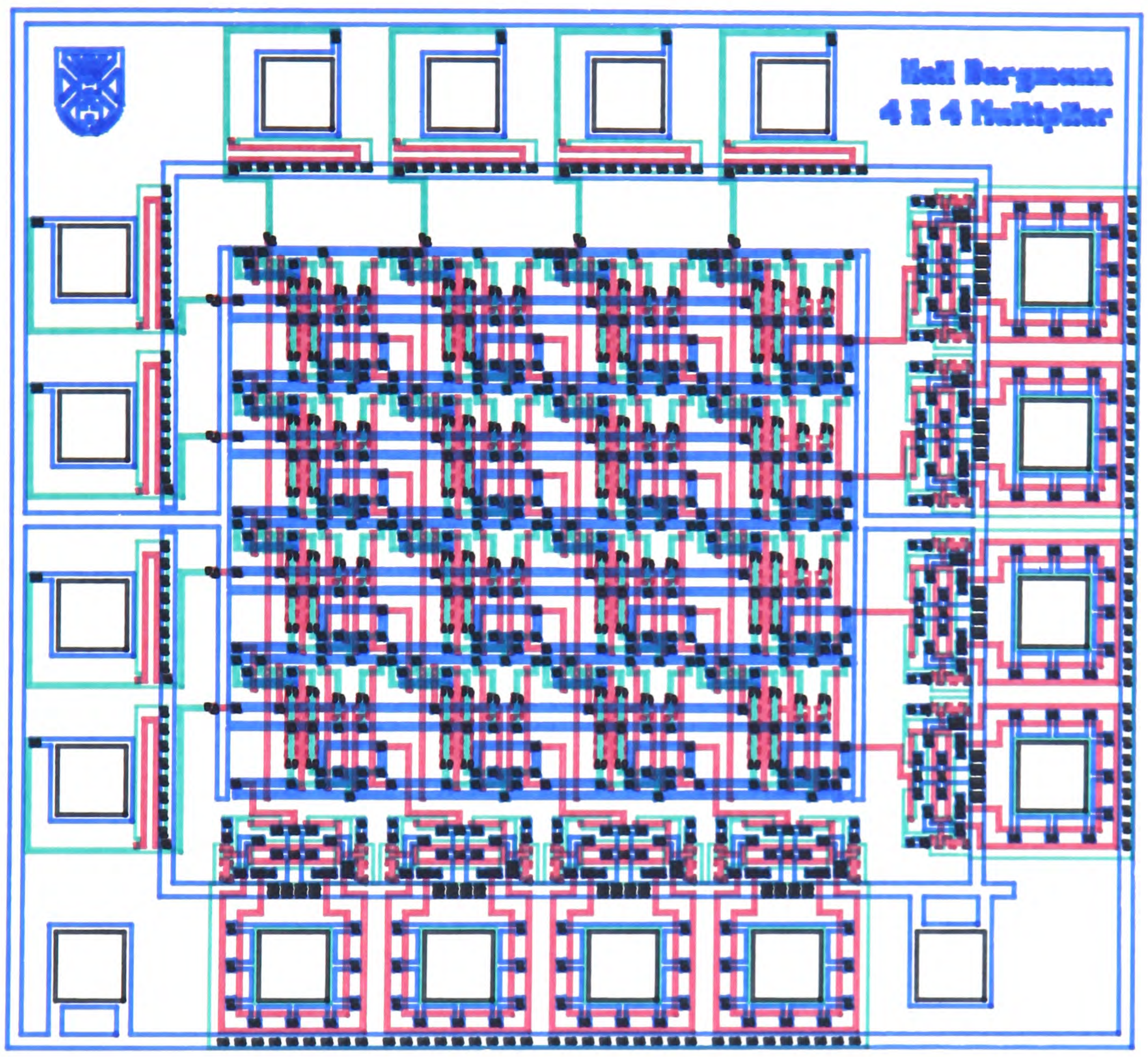


Figure 7–12: Multiplier Chip

7.3.4: Discussion

The total time for the complete design of this multiplier chip, including design of the original algorithm, specification of the VIRGIL idiom, instantiation of the idiom and addition of bonding pads was about 2 person-days.

Although the multiplier is a simple circuit, design at the virtual grid level was still considerably easier than it would have been at the mask level. Cell design was aided by the circuit validity checks incorporated in the VIRGIL system, and also the design rule free nature of virtual grid design. Composition was aided by both the concise manner in which it could be described in the VIRGIL language, and the automatic checking and matching of ports during cell abutment.

Circuit area for the multiplier could have been reduced by further iterations of the cell layout at the virtual grid level, and also by designing special cells for the top and left side cells, where grounded inputs mean some of the internal circuitry is not needed. However, the size of the multiplier chip is already limited by the area required for the pads, and this extra effort would not decrease overall chip size, and so this effort was not considered necessary.

8: CONCLUSIONS AND LIMITATIONS

8.1: Conclusions

The most fundamental issue facing IC designers is still the overwhelming complexity of VLSI circuits. The ideas presented in this thesis are directly concerned with the control of this complexity. The design of circuits containing many thousands of devices is a daunting task. By having an easily accessible collection of useful circuit structures, the designer will be encouraged to use such structures as basic circuit building blocks, reducing the overall complexity of the design task. An experimental system for the capture, storage and instantiation of such circuit "idioms" has been presented and discussed in this thesis.

Unlike most other IC design systems, the emphasis in the VIRGIL system is on the capture of generalised, parameterisable circuit structures rather than specific circuits. A purely textual description for the capture of circuit idioms has been developed as part of the VIRGIL system.

The novel implementation of selection and repetition as textual operations which are performed in an initial "macro expansion" phase of idiom instantiation gives a very simple yet very powerful and flexible method of describing the parameterisable features of an idiom.

The VIRGIL system provides a structural description of circuits and idioms at the sticks level using a so-called virtual grid. The inclusion of structural as well as physical design information allows a large number of checks to be made about the validity of various circuit constructs, reducing the possibility of undetected design errors. Design at the sticks level gives freedom from geometric design rules, which also reduces the possibility of design errors being introduced. The virtual grid has been shown to be an elegant method of sticks level circuit description (as has also been noted by other researchers), and furthermore has been shown to be an elegant method for the capture of idioms at the same level.

The VIRGIL system also includes special support for the hierarchical composition of circuits and idioms. All composition is done using simple horizontal and vertical abutment. The notions of port hiding and port renaming are used to support the enforced matching of ports on the abutting edges of cells which are being composed, further reducing the chances of design errors being introduced.

In summary, the VIRGIL language allows the rapid and accurate capture of design idioms at the cell level. Idiom descriptions are quickly debugged because of the design rule free nature of sticks level design, and because of the considerable number of circuit validity checks which are applied during idiom instantiation.

A style of CMOS design called "generalised CMOS" has been introduced which allows circuits and idioms to be described which are valid in a large number of different CMOS technologies. The notion of "sticks extraction" has been introduced which would allow the automatic conversion of existing mask level circuits into other circuits using related, but different, fabrication technologies.

A sticks compaction algorithm which is simple, controllable and predictable has been presented. This algorithm employs the novel notion of a quasi-virtual grid, and incorporates some novel ideas about the independent translation of wires and devices from sticks to mask level representations.

Examples of idioms which have been captured using the VIRGIL system have been presented in this thesis, including an example of a complete IC which was subsequently fabricated, tested and found to function correctly. The advantages of VIRGIL as a method of idiom capture have been illustrated by investigating the capture of a single idiom (a PLA) using both VIRGIL and more traditional methods.

While the main work of this thesis has been on the capture of idioms at the cell level, idioms exist at all levels of the design hierarchy. The FIRST silicon compiler has been presented as an example of a captured idiom at the highest level of the design hierarchy.

8.2: Limitations & Areas for Further Research

With the exception of the investigation of the FIRST silicon complier, the work on the capture of idioms has concentrated on idioms at the cell level. The VIRGIL system would also provide a useful starting point in the development of a system for the capture of idioms at higher levels of the design hierarchy. Such idioms might be hierarchically composed from smaller idioms, although more complicated composition operations involving automatic routing would be likely to be required.

In the VIRGIL system, parameterisation is expressed in terms of physical attributes, rather than in terms of functional attributes. In the case of the PLA example, specially written programs were needed to convert from logic equations into VIRGIL physical parameters. A solution to this problem would require investigation into more general methods of capturing idiom parameters at a functional level.

In a system where parameters were entered functionally, an expert system approach might be feasible for finding and choosing between suitable idioms for implementing that function. With such a system, VIRGIL descriptions of idioms might form the knowledge base of known circuit constructs. Each idiom description would need to include additional information to allow the expert system to understand the function of that idiom, and to allow the expert system to choose between various idioms for a particular application. The VIRGIL system is only an initial step towards such a goal.

The validity checks which are applied during VIRGIL idiom instantiation provide a powerful means of detecting circuit errors. However, such checks are only able to assist in the checking of the correctness of individual idiom instantiations. It is not possible, with the VIRGIL system, to show that all possible instantiations of a given idiom would be valid circuits. Such a problem is considerably more difficult. This limitation is not considered a serious drawback since any erroneous instantiation will be discovered if it is invoked by a user.

A more significant limitation of the validity checks is that although they provide a good deal of confidence that all the circuit constructs used are valid, they do not provide any real assurance that the combination of these constructs performs the desired function. Such automatic circuit verification is, like proving programs correct, a difficult problem, and research in this area is still in its infancy. The problem of proving a generalised idiom correct, rather than just a circuit, would be an even more difficult problem.

Perhaps the most important limitation of the VIRGIL system is that it exists only as a stand-alone system. To be a really useful production tool, rather than just an experimental tool, it needs to part of a complete, integrated design system. Such a design system, based on the capture of designs at the virtual grid level, could provide an integrated design environment with facilities such as graphical entry of circuits, simulation of circuits, help with global placement and routing, and secure, consistent handling of parts of the IC design description as they are entered and composed together.

Despite these limitations, the development of the VIRGIL system remains a valuable experimental exercise which successfully examines the usefulness of circuit design at the virtual grid level with subsequent conversion to mask level, and, more importantly, the capture of idioms at the virtual grid level by the use of a purely textual language.

APPENDIX A: VIRGIL "INCLUDE" FILE

In the IMP programming language, the contents of a text file, called an "*include*" file, may be included, verbatim, within the source text of a program.

In the VIRGIL system, all separately compiled modules "*include*" the following file, which contains a list of global data type declarations, some global variables and constants, and specifications of globally accessible routines as described in section 6.10. As such, it gives some idea of the various components in the VIRGIL system, and it is for this reason that a copy has been included here.

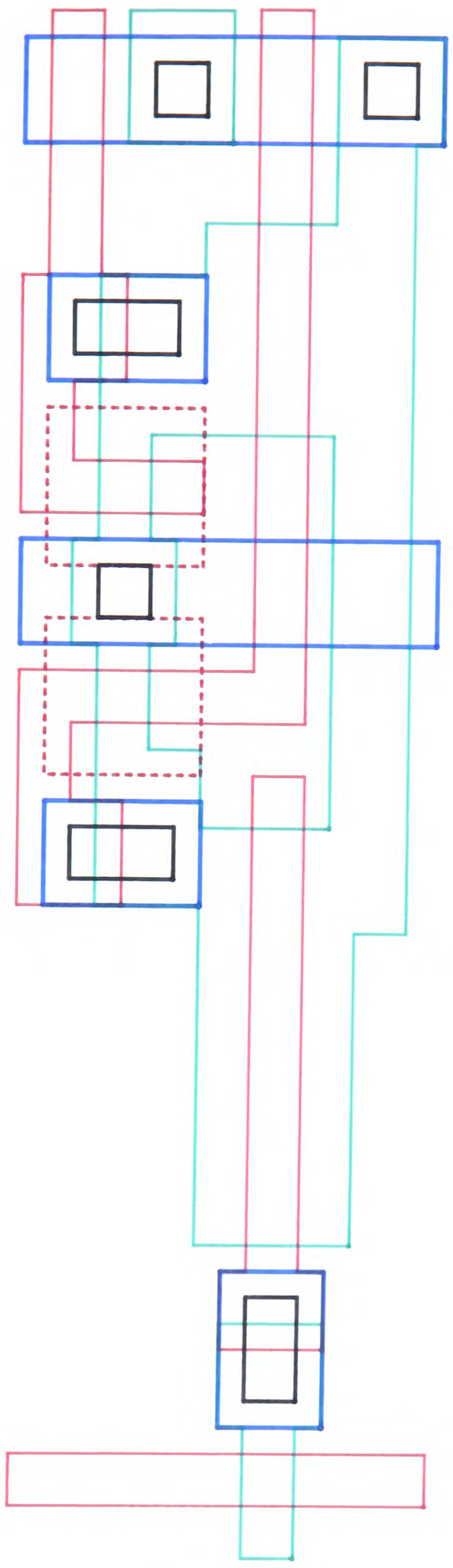
```
! VIRGIL specs included
end of list

!---------------------------------------------------!
!                                                   !
!     INCLUDE FILE FOR VIRGIL PROGRAMS              !
!                Version 1.0.0                      !
!               Author : Neil Bergmann              !
!                                                   !
!---------------------------------------------------!


{Global Constants}
const integer true=1, false=0
const integer llx=1, lly=2,urx=3,ury=4
const integer mink = 128,   { lowest keyword token }
                minp = 256,   { lowest phrase token  }
                maxp = 400    { highest phrase token }

const integer error signal=12

const integer undefined = 0,    {Data type values}
                reserved  = 128,
                variable  = 256,
                constant  = 257,
                statement = 258,
                integer = 1,
                boolean = 2,
                relational = 3,
                for variable = 4,
                parameter = 5,
                coord = 6,
                port = 7,
                contact = 8,
                device = 9,
                array = 10,
                nowell = 11,
                pwell = 12,
                nwell = 13

{ Values for operator types                   }
{ Priority is encoded into 2nd byte     }
{ General form is TYPE ! PRIORITY LEVEL << P'TY OFFSET }
{ Decreasing level of precedence }

const integer po = 8, pmask = 255
const integer umask = 1 << 16
const integer c
  uminus   = 1 ! 8 << po ! umask,
  divide   = 2 ! 7 << po,
  multiply = 3 ! 7 << po,
  plus     = 4 ! 6 << po,
  minus    = 5 ! 6 << po,
  LE       = 6 ! 5 << po,
  LT       = 7 ! 5 << po,
  GT        = 8 ! 5 << po,
  EQ        = 9 ! 5 << po,
  NE       = 10 ! 5 << po,
  GE       = 11 ! 5 << po,
  not      = 12 ! 4 << po ! umask,
  and      = 13 ! 3 << po,
  or       = 14 ! 2 << po,
  maxop    = 14

{Virtual grid item values}

const integer pwire mask = 2_0001,    {Wires}
                nwire mask = 2_0010,
                dwire mask = 2_0100,
                mwire mask = 2_1000

const integer buried mask = 2_0111,          {contacts}
                butting mask = 2_1111,
                burieda mask = buried mask,
                buriedb mask = 2_10000 ! buried mask,
                bor shift = 6,
                bor mask = 2_111111,
                pm mask = pwire mask ! mwire mask,
                dm mask = dwire mask ! mwire mask,
                nm mask = nwire mask ! mwire mask,
                psub mask = 2_011000,
                nsub mask = 2_101000

const integer ptype = 1000,                 {devices}
                ntype = 1001,
                depletion = 1002,
                load = 1003

{Record Format Specifications}
{Symbol Table Entry}
record format HASHF( string (*) name sname,
                          integer type,
                         (integer value or
                        record (*) name rname))

{Single Lexical Item}
record format LEXF( integer type,
                         (record (hashf) name data c
                          or integer value))

{Complete Lexical Entity}
record format LEX ARRAY( c
                    record (lexf) array name array,
                    integer first,last,max)

{Elements of the Syntax Tree}

record format spec ELTF
record format ELT NAME (record (eltf) name name)
record format ELTF (record (elt name) next,
                        integer item)
const record (eltf) name nil elt == 0

record format spec ALTF
record format ALT NAME (record (altf) name name)
record format ALTF (record (alt name) next,
                        record (elt name) items)
const record (altf) name nil alt == 0

{Record to hold syntax trees for complete grammar}

record format PHRASEF (record (alt name) def,
                          string(15) ident)
record format REDUCEF (c
          record (phrasef) array phrases(minp:maxp),
          integer currp)
const record (reducef) name nil reduce == 0

{Analysis Record Nodes}

record format spec SYNTAXF
record format SYNTAX NAME (c
                  record (syntaxf) name name)
record format SYNTAXF ( record (lexf) item,
                     record (syntax name) next,def)
const record (syntaxf) name nil syntax == 0

{Array Bounds Info}
record format BOUNDF ( integer low, high, type)

{Array Header}
const integer max bounds = 6
record format ARRAY HEAD ( integer base type,
          integer number of bounds,
          integer size,
          record (boundf) array bound(1:max bounds),
          integer array name array name,
          record (*) name rname)

const record (array head) name nil array head == 0

{Expression Evaluation Records}

record format EXPRF(integer type,value)
record format spec STACKF
record format STACK NAME (record (stackf) name name)
record format STACKF (record (exprf) item,
                     record (stackf) name next)
const record (stackf) name nil stackf == 0

{Virtual Grid Item}
record format DEVICEF ( integer l,w,sor,dtype)
const record (devicef) name nil device == 0

record format spec ITEMF
record format ITEM NAME( record (itemf) name name)
record format ITEMF( integer type,
                     record (item name) next,
                     string (64) item name,
                     record (*) name symbol,
                     (record (devicef) name dname c
                     or integer value))
const record (itemf) name nil item == 0

{Virtual Grid Nodes and Edges}

record format spec VGEDGE
record format VGNODE ( record (item name) item list,
                       record (vgedge) name n,s,e,w)
const record (vgnode) name nil vgnode == 0

record format VGEDGE ( integer wire,
                       record (vgnode) name n1,n2,
                       integer mwire width)
const record (vgedge) name nil vgedge == 0

{Port List declarations}
record format spec portf
record format port name(record (portf) name name)
record format portf(string(64) port name,
                    record (vgnode) name portp,
                    record (port name) next)
const record (portf) name nil port == 0

{Parameter Lists}
record format spec plistf
record format plist name(record (plistf) name name)
record format plistf(integer type,
                     (integer value or c
                     record (array head) name aname),
                     string (64) pname,
                      record (plist name) next)
const record (plistf) name nil plist == 0

{Virtual Grid Pointer}
record format gridf ( integer array bounds(llx:ury),
                      string (255) cell name,
                      record (plist name) plist,
                      record (port name) port list,
                      record (vgnode) name vgp )
const record (gridf) name nil grid == 0

{Virtual Grid Coordinate}
record format COORDF ( integer x,y)

{List of Ext'l Integers Identifying Grammatical Items}

external integer spec  true const,
                          false const,
                          unop phrase,
and name,
or name,
not name,
if name,
for name,
then name,
else name,
repeat name,
operand phrase,
op phrase,
expr phrase,
exprrest phrase,
for item phrase,
for rest phrase,
for list phrase,
fmlitem phrase,
type phrase,
simple phrase,
default phrase,
header phrase,
celltype phrase,
formals phrase,
fmlrest phrase,
integer name,
boolean name,
cell name,
actitem phrase,
call phrase,
actuals phrase,
actrest phrase,
gridsize phrase,
coord phrase,
label phrase,
griditem phrase,
nodeitem phrase,
position phrase,
path phrase,
pathrest phrase,
offset phrase,
port phrase,
contact phrase,
device phrase,
wire phrase,
mport name,
pport name,
dport name,
nport name,
buried name,
pm name,
dm name,
nm name,
butting name,
psub name,
nsub name,
ptype name,
                              ntype name,
                              depletion name,
                              load name,
                              pwire name,
                              dwire name,
                              nwire name,
                              mwire name,
                              end name,
                              rename phrase,
                              arrayelt phrase,
                              idxlist phrase,
                              idxrest phrase,
                              array phrase,
                              array name,
                              of name,
                              bounds phrase,
                              bdsrest phrase,
                              bdsitem phrase,
                              callst phrase,
                              complist phrase,
                              uprest phrase,
                              rtitem phrase,
                              rtrest phrase,
                              upitem phrase,
                              qual phrase,
                              renitem phrase,
                              rotation phrase,
                              reflect phrase,
                              inx name,
                              iny name

{External Routines Available in Utils.olb}

{ERROR Module Routines}
external routine spec FATAL ERROR (string(255) s)
external routine spec WARNING MESSAGE(string(255) s)
external routine spec SOURCE ERROR(string(255) s)
external string(255) fn spec ERROR TEXT

{HEAP Module Routines}
external string(*) map spec NEWS (string(255) s)

{HEAPA Module Routines}
external routine spec NEW ARRAY(c
                             record (array head) name ap)
external routine spec DISPOSE ARRAY(c
                             record (array head) name ap)

{HASH Module Routines}
external routine spec INITIALISE HASH
external record (hashf) map spec ENTER KEY(c
                                    string(255) s)
external routine spec RESET HASH
external routine spec USE LOCAL HASH
external routine spec USE GLOBAL HASH
external routine spec COPY GLOBAL TO LOCAL
external routine spec PRINT HASH

{DBMS Module Routines}
const integer t=80, f=20, n=40    {string sizes}

external routine spec INITIALISE DB(c
                         string(f) directory stem)

external string(f) fn spec FIND FILE(c
                         string(t) text)

{EXTRACT Module Routines}
external routine spec EXTRACT (c
          record (lex array) name complete,line )

{LEX Module Routines}
external routine spec LEXICAL ANALYSIS (c
                      record (lex array) name complete,
                      string (20) file)
external routine spec STRING ANALYSIS (c
                        record (lex array) name complete,
                        string(255) name s)

external routine spec PRINT LEX (c
                       record(lex array) name rec)

{REDUCE Module Routines}
external routine spec REDUCE(string(30) filename)
external routine spec PRINT SYNTAX
external routine spec SET GRAMMAR (c
                       record (reducef) name grammar)
external record (reducef) map spec GRAMMAR

! Syntax analysis declarations included


{FIND Module Routines}

external routine spec FIND NAMES (c
                      record (reducef) name grammar)

{COMPARE Module Routines}
external record (syntaxf) map spec ANALYSE (c
                            record (lex array) name line)
external record (syntaxf) mapspec SYNTAX ANALYSE (c
                             record (lex array) name line,
                             integer token)

external routine spec PRINT ANALYSIS (c
                              record (syntaxf) name elt,
                              integer level)

external routine spec DISPOSE ELT ( c
                            record(syntax name) name s)
```

```
external routine spec CHECK(c
                      record (syntaxf) name sp,
                      integer type)

{EXPR Module Routines}
external record (exprf) function spec EVALUATE ( c
                              record (syntaxf) name exp)
external routine spec PRINT EXP(record(exprf) e)

{LINE Module Routines}
external routine spec PROCESS LINE(c
                     record (lex array) name line)
external routine spec use process item
external routine spec use build

{MACRO Module Routines}
external routine spec PROCESS LEX (c
                      record (lex array) name from,
                      integer name from p)

{GRID Module Routines}

external routine spec SET GRID (c
                     record (gridf) name grid)
external record (gridf) map spec GRID
external routine spec CREATE GRID
external record (vgnode) map spec FIND NODE(c
                             record (gridf) name grid,
                             integer x,y)

{HEADER Module Routines}
external record (gridf) fn spec PROCESS HEADER (c
                              record (lex array) name lex)

{COORD Module Routines}
external record (coordf) fn spec POSITION(c
                              record (syntaxf) name p)

{ITEM Module Specs}
external routine spec PROCESS ITEM (c
                      record (syntaxf) name p,
                      record (gridf) name grid)

{VERIFY Module Routines}
external routine spec VERIFY GRID

{PLOT Module Routines}
external routine spec DRAW GRID(c
                     record (gridf) name grid)

{MATCH Module Routines}
external record (gridf) map spec CELL CALL (c
                     record (syntaxf) name callp,
                     integer ask)
```

```
external record (gridf) map spec COPY GRID(c
                      record (gridf) name oldgrid)
external routine spec DISPOSE GRID (c
                      record (gridf) name grid)

{Array Module Routines}
external routine spec ANALYSE ARRAY(c
                     record (syntaxf) name p)
external record (exprf) fn spec FIND ARRAY ELT (c
    record (syntaxf) name p)

{Formals Module Routines}
externalrecord(plist name) fnspec ANALYSE FORMALS(c
      record (syntaxf) name p,
      record (plist name) actuals,
      integer ask)

{Build Module Routines}
external routine spec build cell(c
                     record (syntaxf) name sp)
external record (gridf) map spec compose(c
                     record(syntaxf) name sp)

{Join Module Routines}

external routine spec join grids(c
                     record (gridf) name new gridp,
                                           gridp,
                      integer ontop)
external routine spec DESPIKE GRID(c
                     record (gridf) name gridp)

{READIN Module Routines}
external record (gridf) map spec READ CELL (c
                      string(255) file)


{FLESH Module Routines}
external routine spec COMPACT GRID(c
                     record (gridf) name grid)

{DUMP Module Routines}
external routine spec DUMP GRID(c
                     record (gridf) name grid)

end of file
```

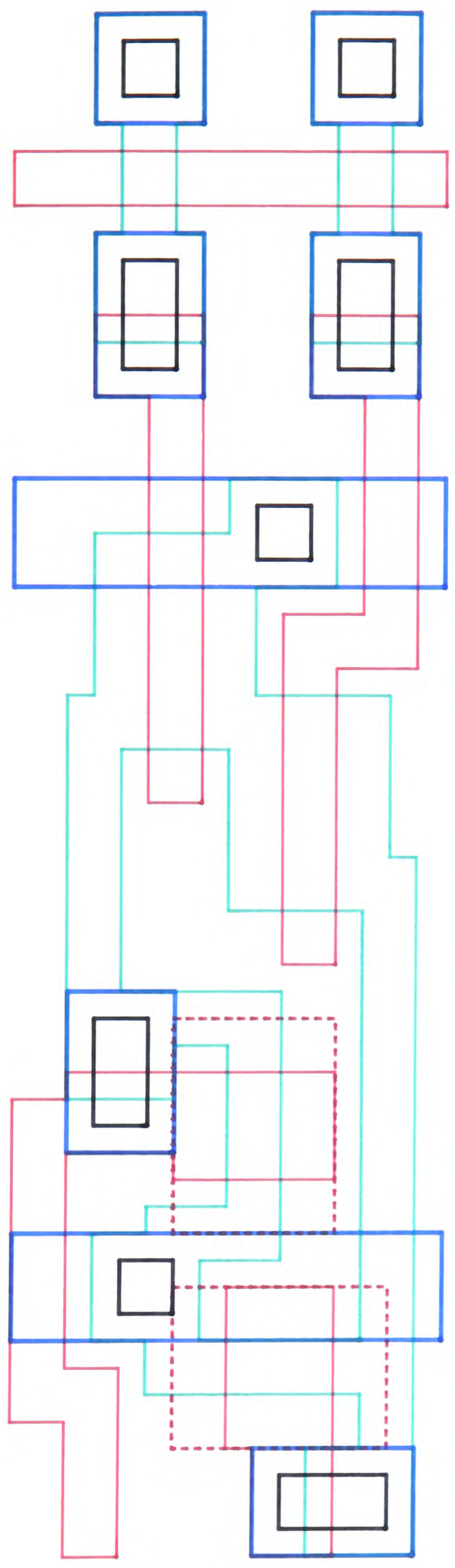
APPENDIX B: PLA MASK LEVEL LEAF CELLS

This appendix contains graphical representations of the various leaf cells which are used in the mask level PLA generator described in section 7.2.2.



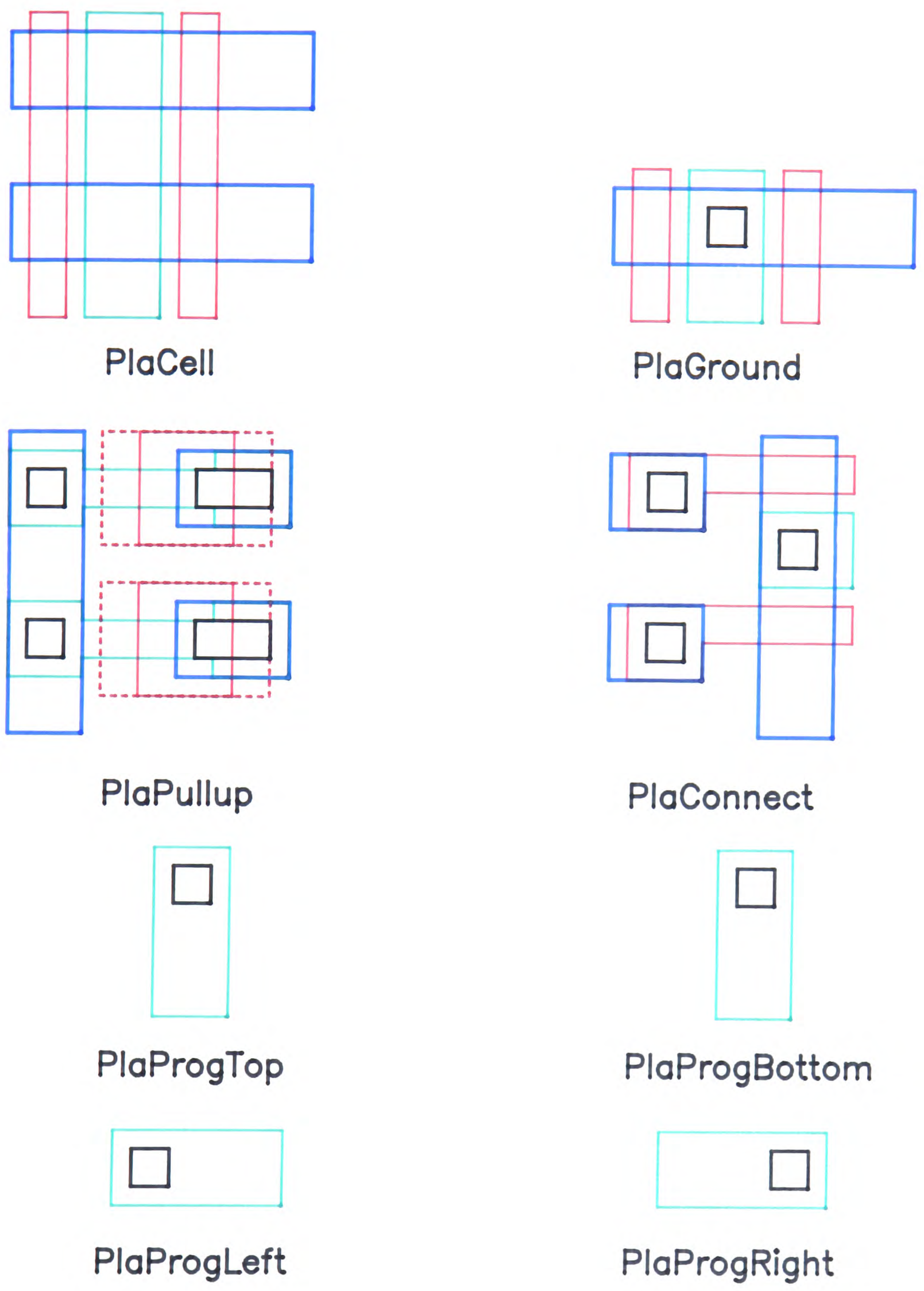
PlaClockIn

PLA Mask Level Leaf Cells ...../

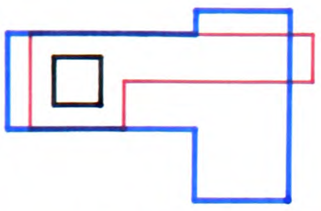


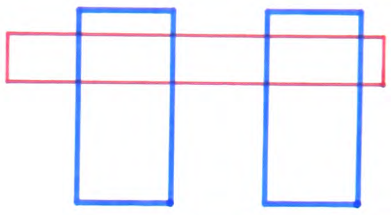
PlaClockOut

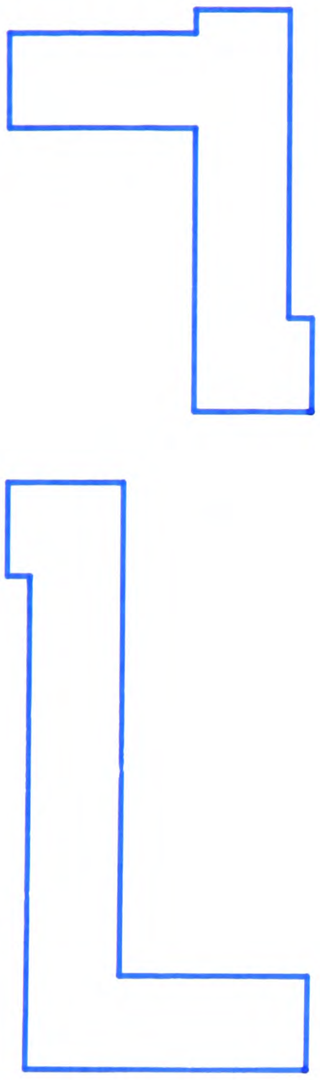
PLA Mask Level Leaf Cells .../

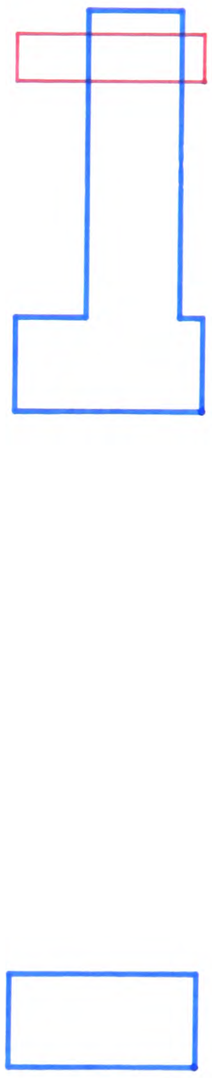


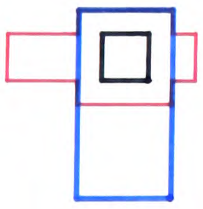
PLA Mask Level Leaf Cells .../

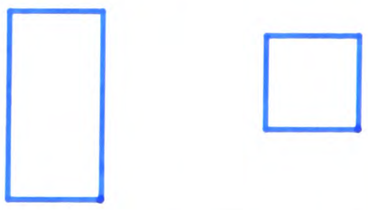

PlaConnectSpace


PlaOrSpace


PlaHoleWires


PlaOutSpace

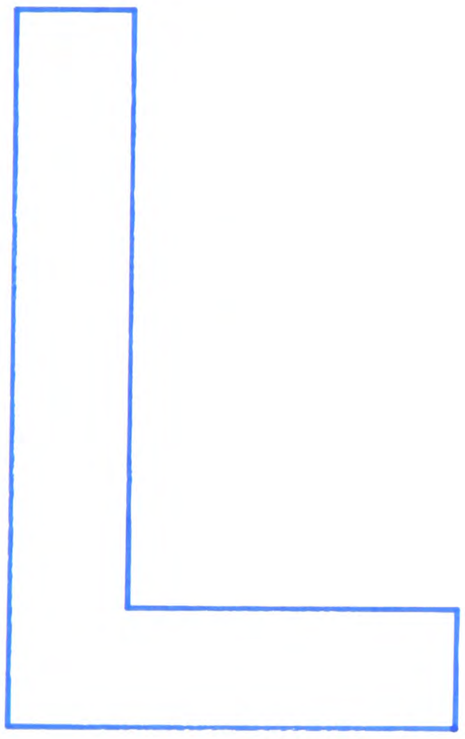

PlaGroundSpace

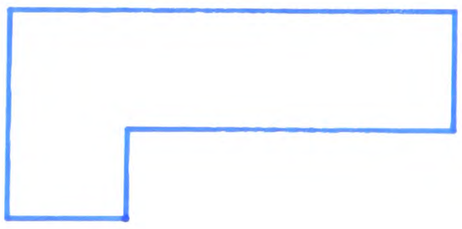

PlaPullupSpace

PLA Mask Level Leaf Cells .../

PlaVddTop

PlaVddHole


PlaVddSide

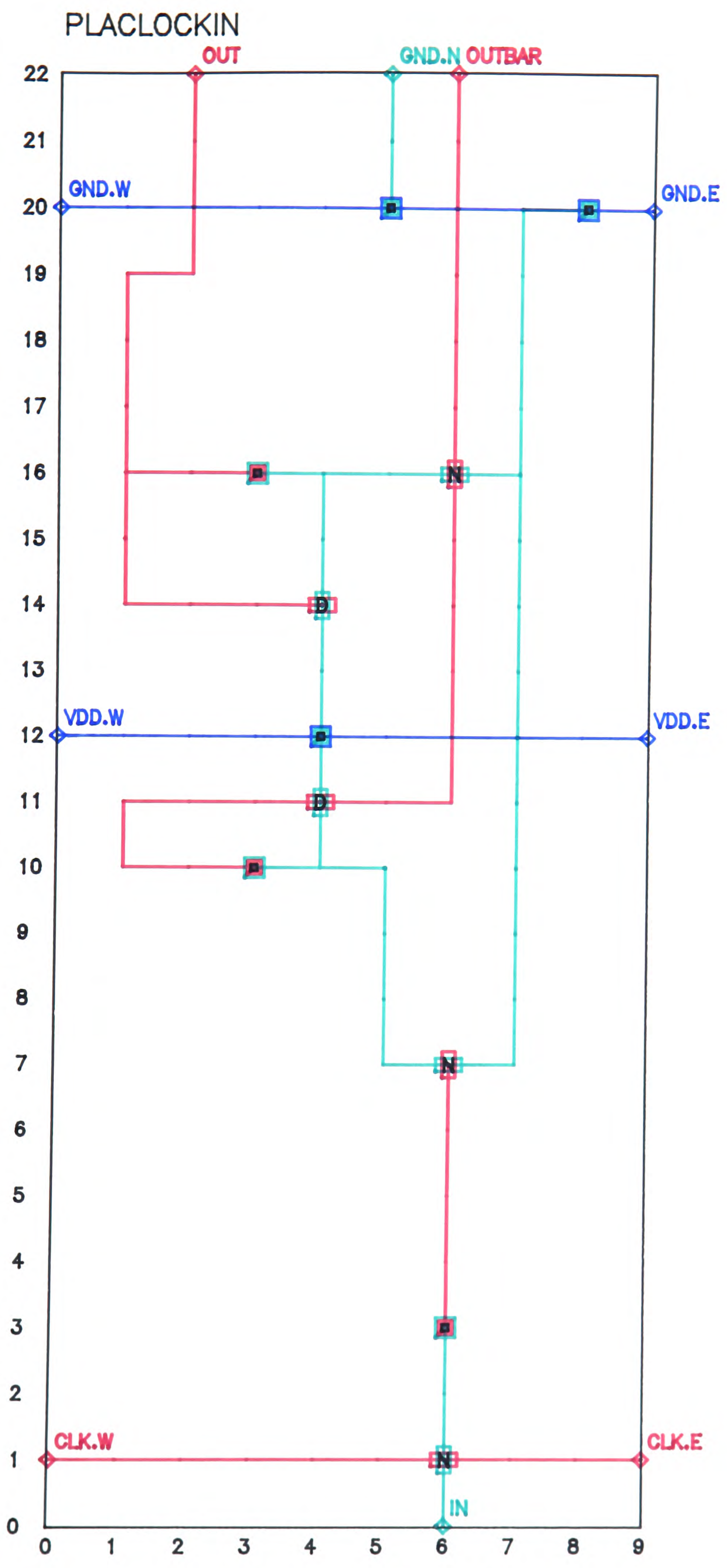

PlaVddCorner

PLA Mask Level Leaf Cells

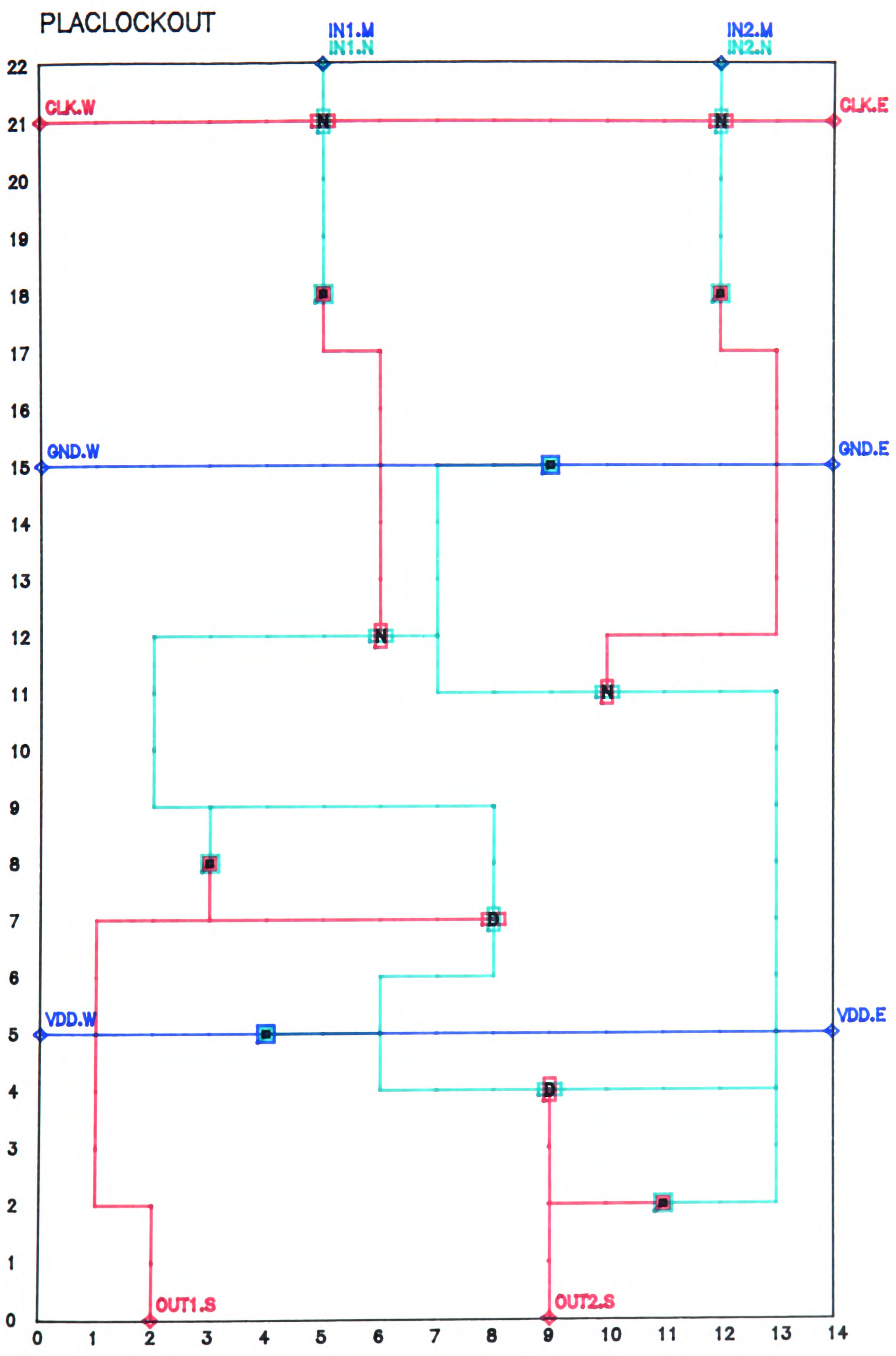
APPENDIX C: PLA VIRGIL CELL DEFINITIONS

This appendix contains the VIRGIL leaf cell and composition cell definitions which capture the PLA idiom described in section 7.2.3. Graphical representations of cell instantiations are also shown.

```
Leaf Cell PlaClockIn = (0,0,9,22)
  clk.e: pport @ (9,1)
  clk.w: pport @ (0,1)
  pwire @ clk.e -> clk.w

  vdd.e: mport @ (9,12)
  vdd.w: mport @ (0,12)
  mwire @ vdd.e -> vdd.w

  gnd.e: mport @ (9,20)
  gnd.w: mport @ (0,20)
  mwire @ gnd.e -> gnd.w
  gnd.n: nport @ (5,22)
  nm @ (5,20)
  nwire @ (5,20) -> gnd.n

  in: nport @ (6,0)
  out: pport @ (2,22)
  outbar: pport @ (6,22)

  {devices}
  ntype(w=8) @ (6,7)
  ntype(w=4) @ (6,16)
  depletion @ (4,11)
  depletion @ (4,14)
  ntype @ (6,1)

  nwire @ in -> (6,3)
  butting @ (6,3)
  pwire @ (6,3) -> (6,7)

  nm @ (8,20)
  nwire @ (8,20) -> (7,20) -> (7,7) -> (5,7)
  nwire @ (5,7) -> (5,10) -> (3,10)
  nwire @ (4,10) -> (4,16)
  nwire @ (3,16) -> (7,16)
  nm @ (4,12)
  butting @ (3,10)
  butting @ (3,16)

  pwire @ (3,10) -> (1,10) -> (1,11) -> (6,11) -> outbar
  pwire @ (3,16) -> (1,16) -> (1,14) -> (4,14)
  pwire @ (1,16) -> (1,19) -> (2,19) -> out
End
```

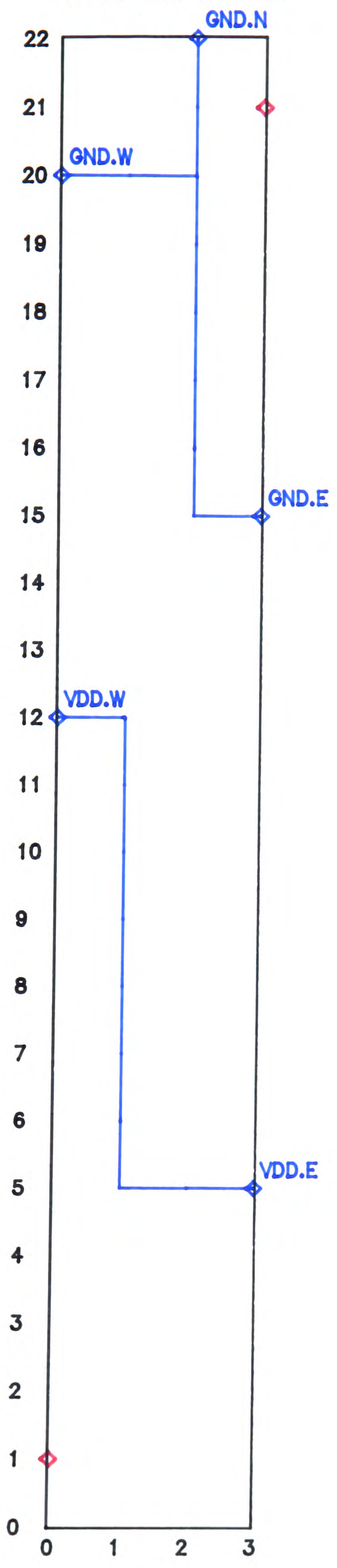

PLACLOCKIN
OUT
GND.N
OUTBAR
GND.W
GND.E
VDD.W
VDD.E
CLK.W
CLK.E
IN
N
D
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9

```
Leaf Cell PlaClockOut = (0,0,14,22)
  clk.e: pport @ (14,21)
  clk.w: pport @ (0,21)
  pwire @ clk.e -> clk.w

  vdd.e: mport @ (14,5)
  vdd.w: mport @ (0,5)
  mwire @ vdd.e -> vdd.w
  gnd.e: mport @ (14,15)
  gnd.w: mport @ (0,15)
  mwire @ gnd.e -> gnd.w

  in1.n: nport @ (5,22)
  in2.n: nport @ (12,22)
  in1.m: mport @ (5,22)
  in2.m: mport @ (12,22)
  out1.s: pport @ (2,0)
  out2.s: pport @ (9,0)

  {devices}
  depletion(l=2) @ (9,4)
  depletion(l=2) @ (8,7)
  ntype @ (5,21)
  ntype @ (12,21)
  ntype(w=4)@ (6,12)
  ntype(w=4) @ (10,11)

  nm @ (4,5)
  nm @ (9,15)
  butting @ (11,2)
  butting @ (3,8)
  butting @ (5,18)
  butting @ (12,18)

  pwire @ out1.s -> (2,2) -> (1,2) -> (1,7) -> (8,7)
  pwire @ (3,7) -> (3,8)
  pwire @ out2.s -> (9,4)
  pwire @ (9,2) -> (11,2)
  pwire @ (6,12) -> (6,17) -> (5,17) -> (5,18)
  pwire @ (10,11) -> (10,12) -> (13,12) -> (13,17)
  pwire @ (13,17) -> (12,17) -> (12,18)

  nwire @ (12,18) -> in2.n
  nwire @ (5,18) -> in1.n
  nm @ in1.n
  nm @ in2.n

  nwire @ (11,2) -> (13,2) -> (13,4) -> (6,4) -> (6,5)
  nwire @ (4,5) -> (6,5) -> (6,6) -> (8,6) -> (8,9)
  nwire @ (8,9) -> (3,9)
  nwire @ (3,8) -> (3,9) -> (2,9) -> (2,12) -> (7,12)
  nwire @ (9,15) -> (7,15) -> (7,11)
  nwire @ (7,11) -> (13,11) -> (13,4)
End
```

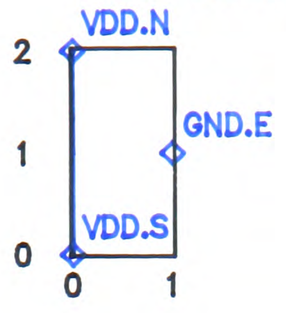

PLACLOCKOUT
IN1.M
IN1.N
IN2.M
IN2.N
CLK.W
CLK.E
GND.W
GND.E
VDD.W
VDD.E
OUT1.S
OUT2.S
N
N
N
N
D
D
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

```
Leaf Cell PlaHoleWires = (0,0,3,22)
  vdd.e: mport @ (3,5)
  vdd.w: mport @ (0,12)
  mwire @ vdd.w -> (1,12) -> (1,5) -> vdd.e
  gnd.e: mport @ (3,15)
  gnd.w: mport @ (0,20)
  gnd.n: mport @ (2,22)
  mwire @ gnd.w -> (2,20) -> (2,15) -> gnd.e
  mwire @ gnd.n -> (2,20)
  pport @ (3,21)
  pport @ (0,1)
End

Leaf Cell PlaPullupSpace = (0,0,1,2)
  vdd.s: mport @ (0,0)
  vdd.n: mport @ (0,2)
  mwire @ vdd.n -> vdd.s
  gnd.e: mport @ (1,1)
End

Leaf Cell PlaOutSpace = (0,0,2,4)
  gnd.e: mport @ (2,2)
  gnd.w: mport @ (0,2)
  gnd.n: mport @ (1,4)
  mwire @ gnd.e -> gnd.w
  mwire @ gnd.n -> (1,2)
  vdd.e: mport @ (2,1)
  vdd.w: mport @ (0,1)
  mwire @ vdd.e -> vdd.w
  clk.e: pport @ (2,3)
  clk.w: pport @ (0,3)
  pwire @ clk.e -> clk.w
End

Leaf Cell PlaVddSide = (0,0,1,3)
  vdd.e: mport @ (1,2)
  vdd.n: mport @ (0,3)
  mwire @ vdd.e -> (0,2) -> vdd.n
  clk.e: pport @ (1,1)
  clk.w: pport @ (0,1)
  pwire @ clk.e -> clk.w
End

Leaf Cell PlaVddCorner = (0,0,1,1)
  vdd.e: mport @ (1,1)
  vdd.s: mport @ (0,0)
  mwire @ vdd.e -> (0,1) -> vdd.s
End
```

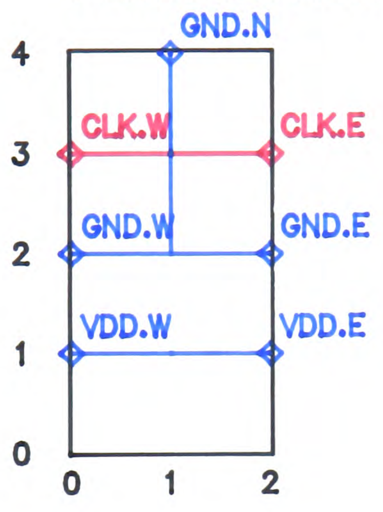
PLAHOLEWIRES



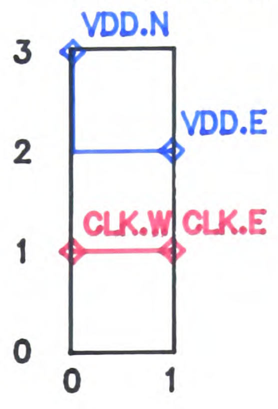
PLAPULLUPSPACE



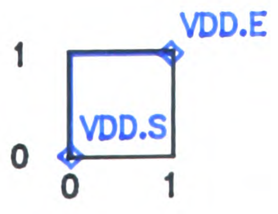
PLAOUTSPACE



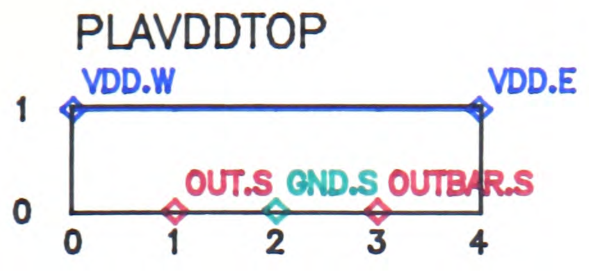
PLAVDDSIDE

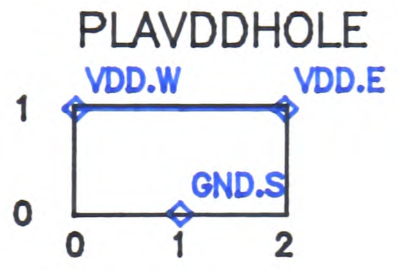


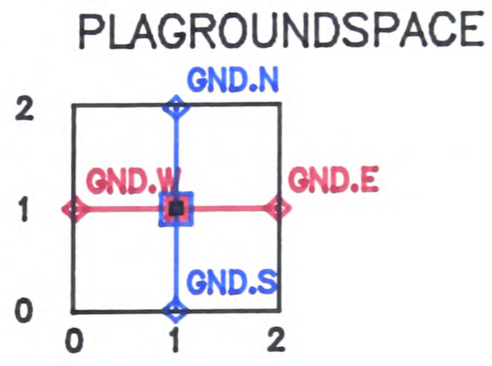
PLAVDDCORNER



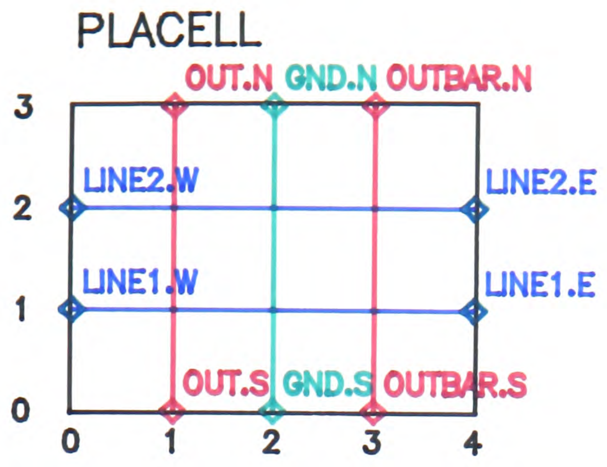
```
Leaf Cell PlaVddTop = (0,0,4,1)
  out.s: pport @ (1,0)
  outbar.s: pport @ (3,0)
  gnd.s: nport @ (2,0)

  vdd.e: mport @ (4,1)
  vdd.w: mport @ (0,1)
  mwire @ vdd.e -> vdd.w
End

Leaf Cell PlaVddHole = (0,0,2,1)
  vdd.e: mport @ (2,1)
  vdd.w: mport @ (0,1)
  mwire @ vdd.e -> vdd.w
  gnd.s: mport @ (1,0)
End

Leaf Cell PlaGroundSpace = (0,0,2,2)
  gnd.s: mport @ (1,0)
  gnd.n: mport @ (1,2)
  mwire @ gnd.s -> gnd.n
  gnd.e: pport @ (2,1)
  gnd.w: pport @ (0,1)
  pwire @ gnd.e -> gnd.w
  pm @ (1,1)
End
```


PLAVDDTOP
VDD.W
VDD.E
OUT.S
GND.S
OUTBAR.S
1
0
0
1
2
3
4

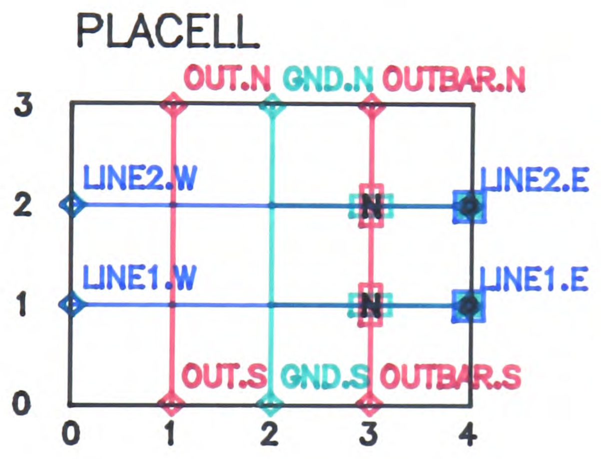

PLAVDDHOLE
VDD.W
VDD.E
GND.S
1
0
0
1
2


PLAGROUNDSPACE
GND.N
GND.W
GND.E
GND.S
2
1
0
0
1
2

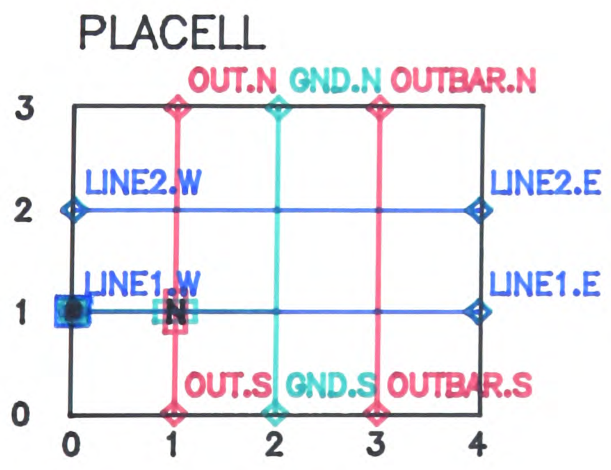
```
Leaf Cell PlaCell (ll:boolean=false,ur:boolean=false,
                   lr:boolean=false, ul:boolean=false,
                   left:boolean=false) = (0,0,4,3)
  out.s: pport @ (1,0)
  out.n: pport @ (1,3)
  pwire @ out.s -> out.n

  outbar.s: pport @ (3,0)
  outbar.n: pport @ (3,3)
  pwire @ outbar.s -> outbar.n

  gnd.s: nport @ (2,0)
  gnd.n: nport @ (2,3)
  nwire @ gnd.s -> gnd.n

  line1.e: mport @ (4,1)
  line1.w: mport @ (0,1)
  mwire @ line1.e -> line1.w

  line2.e: mport @ (4,2)
  line2.w: mport @ (0,2)
  mwire @ line2.e -> line2.w

  nport @ line1.e
  nport @ line1.w
  nport @ line2.e
  nport @ line2.w

  [if ll then
    ntype(w=2) @ (1,1)
    [if not left then nm @ (0,1)]
    nwire @ (0,1) -> (2,1) ]
  [if lr then
    ntype(w=2) @ (3,1)
    nm @ (4,1)
    nwire @ (4,1) -> (2,1) ]
  [if ul then
    ntype(w=2) @ (1,2)
    [if not left then nm @ (0,2)]
    nwire @ (0,2) -> (2,2) ]
  [if ur then
    ntype(w=2) @ (3,2)
    nm @ (4,2)
    nwire @ (4,2) -> (2,2) ]

End
```



ur = true
lr = true

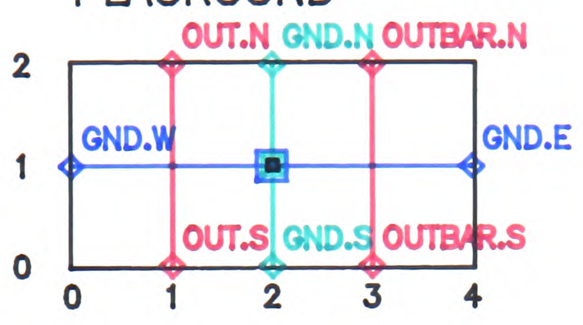


ll = true

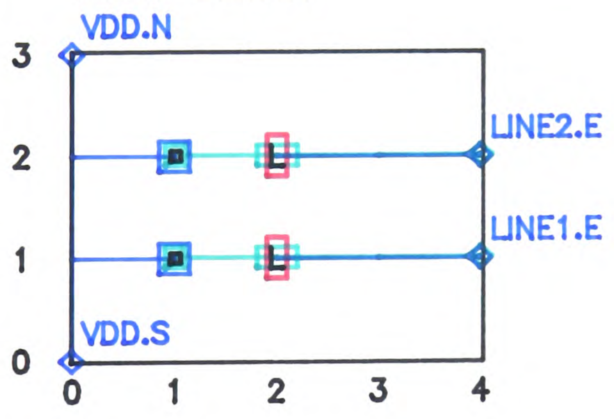


```
Leaf Cell PlaGround = (0,0,4,2)
  out.s: pport @ (1,0)
  out.n: pport @ (1,2)
  pwire @ out.s -> out.n

  outbar.s: pport @ (3,0)
  outbar.n: pport @ (3,2)
  pwire @ outbar.s -> outbar.n

  gnd.s: nport @ (2,0)
  gnd.n: nport @ (2,2)
  nwire @ gnd.s -> gnd.n

  gnd.e: mport @ (4,1)
  gnd.w: mport @ (0,1)
  mwire @ gnd.e -> gnd.w

  nm @ (2,1)
End

Leaf Cell PlaPullup = (0,0,4,3)
  vdd.s: mport @ (0,0)
  vdd.n: mport @ (0,3)
  mwire @ vdd.n -> vdd.s
  line1.e: mport @ (4,1)
  line2.e: mport @ (4,2)
  nport @ line1.e
  nport @ line2.e
  load(l=3) @ (2,1)
  load(l=3) @ (2,2)
  mwire @ (0,1) -> (1,1)
  mwire @ (0,2) -> (1,2)
  nm @ (1,1)
  nm @ (1,2)
  nwire @ (1,1) -> line1.e
  nwire @ (1,2) -> line2.e
  mwire @ (2,1) -> line1.e
  mwire @ (2,2) -> line2.e
End
```

PLAGROUND

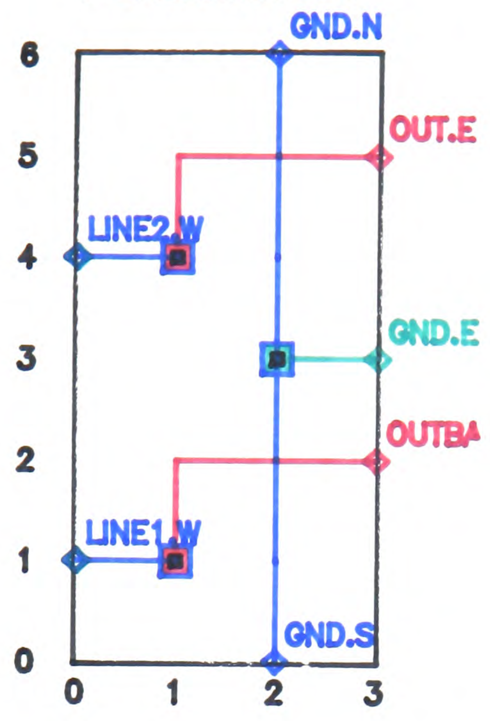


PLAPULLUP

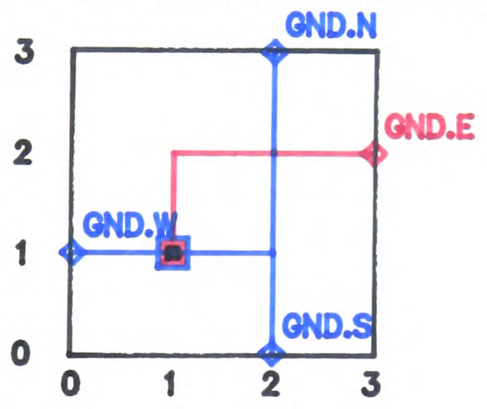


```
Leaf Cell PlaConnect = (0,0,3,6)
  gnd.s: mport @ (2,0)
  gnd.n: mport @ (2,6)
  mwire @ gnd.s -> gnd.n
  line1.w: mport @ (0,1)
  line2.w: mport @ (0,4)
  out.e: pport @ (3,5)
  outbar.e: pport @ (3,2)
  pm @ (1,1)
  mwire @ line1.w -> (1,1)
  pwire @ (1,1) -> (1,2) -> outbar.e
  pm @ (1,4)
  mwire @ line2.w -> (1,4)
  pwire @ (1,4) -> (1,5) -> out.e
  gnd.e: nport @ (3,3)
  nm @ (2,3)
  nwire @ (2,3) -> gnd.e
  nport @ line1.w
  nport @ line2.w
End

Leaf Cell PlaConnectSpace = (0,0,3,3)
  gnd.s: mport @ (2,0)
  gnd.n: mport @ (2,3)
  mwire @ gnd.s -> gnd.n
  gnd.e: pport @ (3,2)
  gnd.w: mport @ (0,1)
  mwire @ gnd.w -> (2,1)
  pm @ (1,1)
  pwire @ (1,1) -> (1,2) -> gnd.e
End

Leaf Cell PlaOrSpace = (0,0,3,2)
  line1.s: mport @ (1,0)
  line2.s: mport @ (2,0)
  line1.n: mport @ (1,2)
  line2.n: mport @ (2,2)
  nport @ line1.n
  nport @ line2.n
  nport @ line1.s
  nport @ line2.s
  mwire @ line1.s -> line1.n
  mwire @ line2.s -> line2.n
  gnd.e: pport @ (3,1)
  gnd.w: pport @ (0,1)
  pwire @ gnd.e -> gnd.w
End
```

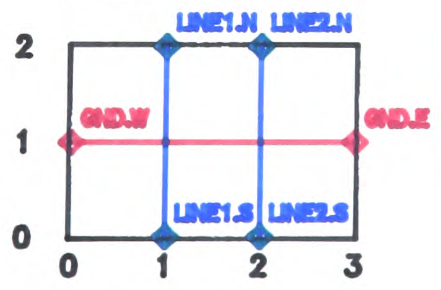
PLACONNECT



PLACONNECTSPACE

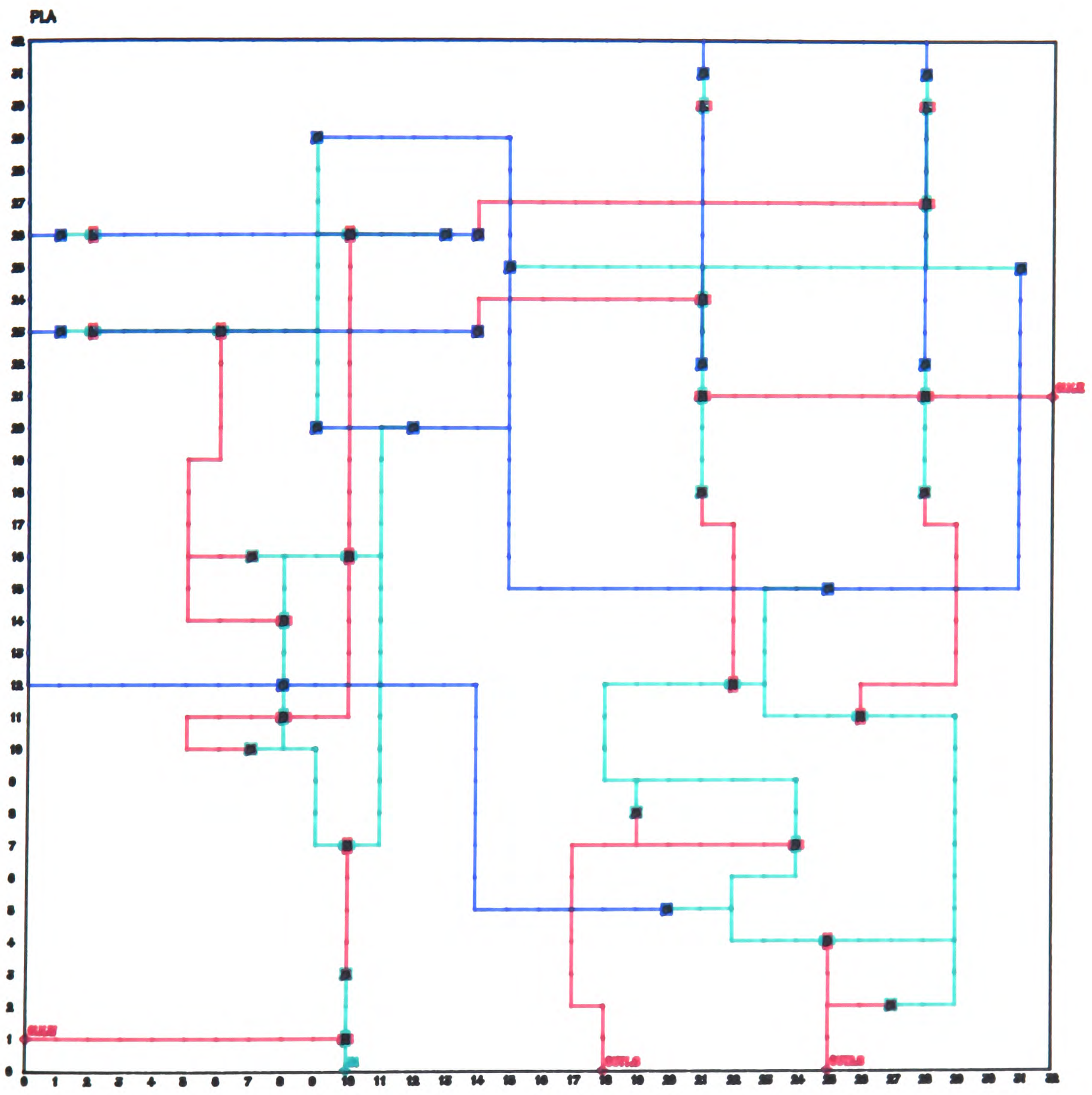


PLAORSPACE



```
Composition Cell PLA ( ni: integer = 2,
      no: integer = 2,
      nmin: integer = 2,
      anda: array(1:ni*2,1:nmin) of boolean = true,
      ora: array(1:no,1:nmin) of boolean= true)

[for in = 1..ni repeat
incol_in = ^^ PlaClockIn [if in=1 then /gnd.w] -
           [for min = 1..nmin/2 repeat -
             ^^ PlaCell (ll=anda(in*2-1,min*2-1), -
                         lr=anda(in*2,min*2-1), -
                         ul=anda(in*2-1,min*2), -
                         ur=anda(in*2,min*2), -
                         left=in=1) -
             [if min-min/2*2=0 or min=nmin/2  -
               then ^^ PlaGround]] -
           ^^ PlaVddTop
]

[for out = 1..(no+1)/2 repeat
outcol_out = ^^PlaClockOut /in1.m=line1.m/in2.m=line2.m -
             [for min = 1..nmin/2 repeat -
               ^^ PlaCell (lr=ora(out*2-1,min*2-1),
                           ur=ora(out*2,min*2-1),
                           ll=ora(out*2-1,min*2),
                           ul=ora(out*2,min*2), -
                           left=min=nmin/2) @ 3 -
              [if min-min/2*2=0 and min#nmin/2 -
                 then ^^ PlaOrSpace ]] -
             ^^ PlaPullup @ 3
]

leftcol = ^^ PlaVddSide -
          [for min = 1..nmin/2 repeat -
            ^^ PlaPullup -
           [if min-min/2*2=0 or min=nmin/2 -
              then ^^ PlaPullupSpace]]-
          ^^ PlaVddCorner

holecol = ^^ PlaHoleWires -
          [for min = 1..nmin/2 repeat -
            ^^ PlaConnect -
           [if min-min/2*2=0 or min=nmin/2 -
             then ^^ PlaConnectSpace]-
           [if min=nmin/2 then /gnd.e]] -
          ^^ PlaVddHole

spacecol = ^^ PlaOutSpace -
           [for min = 1..nmin/2 repeat -
             ^^ PlaGround @ 3 -
            [if min-min/2*2=0 and min#nmin/2 -
               then ^^ PlaGroundSpace ]]-
            ^^ PlaPullupSpace @ 3
```

```
rightcol = ^^ PlaOutSpace /gnd.e/vdd.e -
           [for min = 1..nmin/2 repeat -
             ^^PlaGround /out.n/outbar.n/gnd.n @ 3 -
             [if min-min/2*2=0 and min#nmin/2 then -
              ^^ PlaGroundSpace/gnd.e]] -
           ^^ PlaPullupSpace/vdd.n @ 3

pla = >> leftcol  -
      [for in=1..ni repeat >> incol_in] -
      >> holecol -
      [for out=1..(no+1)/2 repeat >> outcol_out -
        [if out-out/2*2=0 and out#(no+1)/2 -
           then >> spacecol]]-
      >> rightcol

End
```



Example PLA: ni=1, no=2, nmin=2

APPENDIX D: MULTIPLIER CELL DEFINITIONS

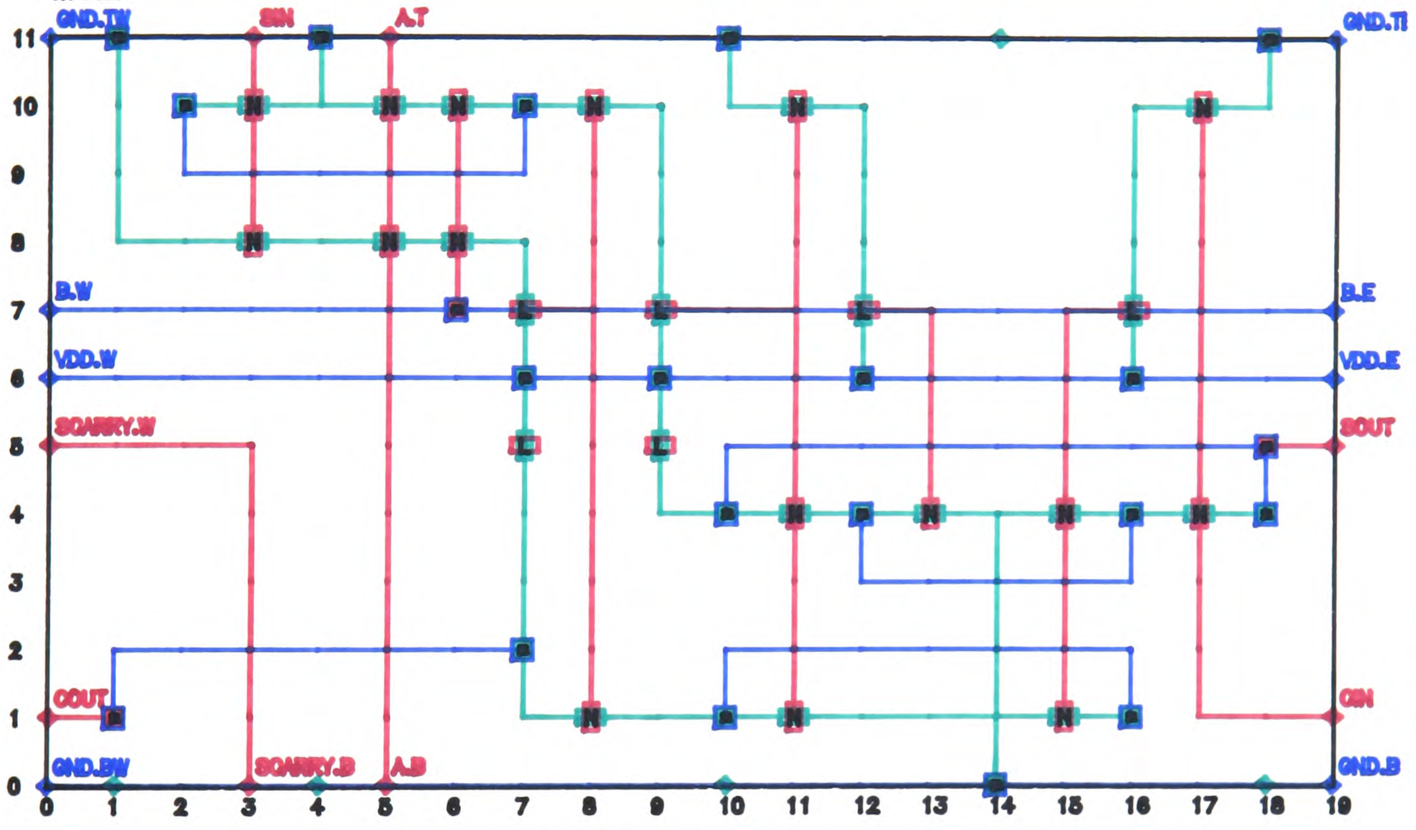
This appendix contains the VIRGIL leaf cell and composition cell definitions which capture the Multiplier idiom described in section 7.3. Graphical representations of some leaf cell instantiations are also shown.

```
Leaf Cell Mult (left:boolean=false,
                right:boolean=false,
                top:boolean=false) = (0,0,19,11)

vdd.w: mport @ (0,6)
mwire @ vdd.w -> (16,6)

gnd.be: mport @ (19,0)
mwire @ (1,0) -> gnd.be

gnd.te: mport @ (19,11)
mwire @ (1,11) -> gnd.te

[if right then
  mwire(w=2) @ gnd.be -> gnd.te
else
  vdd.e: mport @ (19,6)
  mwire @ (16,6) -> (19,6)
]

[if left then
  vdd.b: mport @ (0,0)
  vdd.t: mport @ (0,11)
  mwire(w=2) @ vdd.b -> vdd.t
  gnd.tw: mport @ (1,11)
  gnd.bw: mport @ (1,0)
else
  gnd.bw: mport @ (0,0)
  mwire @ (0,0) -> (1,0)
  gnd.tw: mport @ (0,11)
  mwire @ (0,11) -> (1,11)
]

a.t: pport @ (5,11)
a.b: pport @ (5,0)
pwire @ a.t -> a.b

[if not right then
  b.e: mport @ (19,7)
  mwire @ (6,7) -> b.e
]

[if left then
  b.w: pport @ (0,7)
  b.c: pm @ (1,7)
  pwire @ b.w -> b.c
  mwire @ b.c -> (6,7)
else
  b.w: mport @ (0,7)
  mwire @ (6,7) -> b.w
]

s,carry b: pport @ (3,0)
[if left then
  pwire @ (1,1) -> (3,1) -> s,carry b
else
  s_carry_w: pport @ (0,5)
  pwire @ s_carry_b -> (3,5) -> s_carry_w
  c_out: pport @ (0,1)
  pwire @ cout -> (1,1)
]

s_in: pport @ (3,11)
[if top then
  pm @ (2,11)
  pport @ (2,11)
  pwire @ (2,11) -> (3,11)
]

s_out: pport @ (19,5)

[if right then
  mwire @ (17,1) -> (17,0)
  pm @ (17,1)
else
  c_in: pport @ (19,1)
  pwire @ (19,1) -> (17,1)
]

pwire @ (17,1) -> (17,10)
pwire @ (15,1) -> (15,7) -> (16,7)
pwire @ (13,4) -> (13,7) -> (12,7)
pwire @ (11,1) -> (11,10)
pwire @ (8,1) -> (8,10)
pwire @ (6,7) -> (6,10)
pm @ (6,7)
pwire @ (3,8) -> s_in
pwire @ (7,7) -> (8,7)
pwire @ (9,7) -> (11,7)

nm @ (1,11)
nport @ (1,11)
nport @ (1,0)
nwire @ (1,11) -> (1,8) -> (7,8) -> (7,6)
nm @ (7,6)

nwire @ (2,10) -> (9,10) -> (9,6)
nm @ (9,6)
nwire @ (4,10) -> (4,11)
nm @ (4,11)
nport @ (4,11)
nport @ (4,0)

nport @ (10,11)
nport @ (10,0)
nport @ (18,11)
nport @ (18,0)
nm @ (10,11)
nm @ (18,11)
nwire @ (10,11) -> (10,10) -> (12,10) -> (12,6)
```

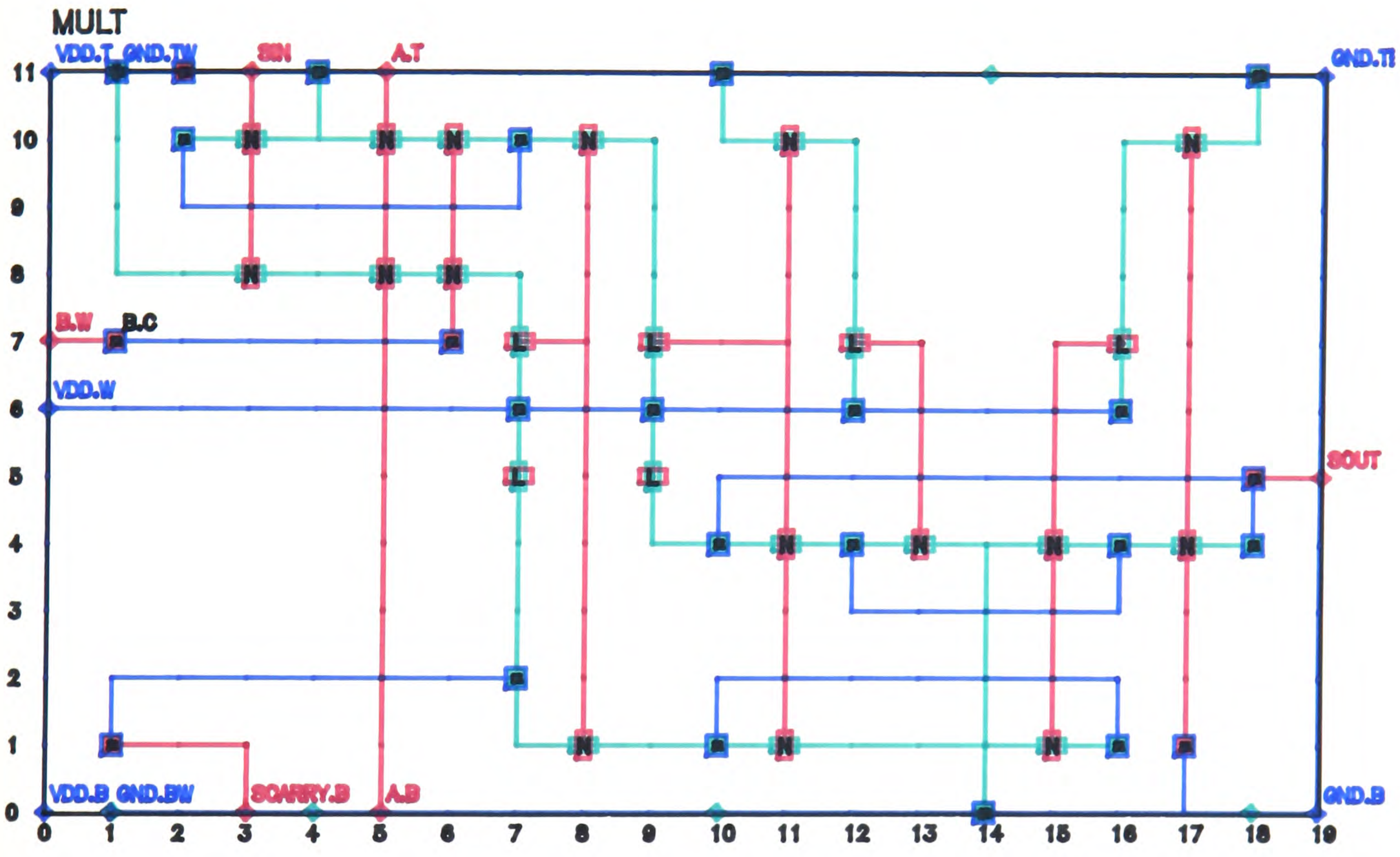
```
nm @ (12,6)
nwire @ (18,11) -> (18,10) -> (16,10) -> (16,6)
nm @ (16,6)

nwire @ (7,6) -> (7,1) -> (16,1)

nwire @ (9,6) -> (9,4) -> (18,4)
nm @ (14,0)
nport @ (14,0)
nport @ (14,11)
nwire @ (14,0) -> (14,4)

nm @ (2,10)
nm @ (7,10)
mwire @ (2,10) -> (2,9) -> (7,9) -> (7,10)


pm @ (1,1)
nm @ (7,2)
mwire @ (1,1) -> (1,2) -> (7,2)

nm @ (10,1)
nm @ (16,1)
mwire @ (10,1) -> (10,2) -> (16,2) -> (16,1)

nm @ (12,4)
nm @ (16,4)
mwire @ (12,4) -> (12,3) -> (16,3) -> (16,4)

nm @ (10,4)
nm @ (18,4)
mwire @ (10,4) -> (10,5) -> (18,5) -> (18,4)
pm @ (18,5)
pwire @ (18,5) -> s.out

ntype @ (3,10)
ntype(w=2) @ (5,10)
ntype(w=2) @ (6,10)
ntype @ (8,10)
ntype @ (11,10)
load(sor=1) @ (12,7)
load(sor=1) @ (16,7)
ntype @ (17,10)

ntype(w=2) @ (3,8)
ntype(w=2) @ (5,8)
ntype(w=2) @ (6,8)
load(sor=1,l=2) @ (7,7)
load(sor=1,l=2) @ (9,7)

load(l=2) @ (7,5)
ntype @ (11,4)
ntype @ (13,4)
ntype @ (15,4)
ntype @ (17,4)
```

```
load(l=2) @ (9,5)
ntype @ (8,1)
ntype @ (11,1)
ntype @ (15,1)

END


Composition Cell Multiplier(sizea:integer=4,
                            sizeb:integer=4)

[for i=1..sizeb repeat
  row_i = [for j=1..sizea repeat -
   >> mult(top=(i=sizeb),left=(j=1),right=(j=sizea))]
]

multiplier = [for i=1..sizeb repeat ^^ row_i]

End
```

MULT



left = false
right = false
top = false



left = true
right = true
top = true

REFERENCES

[ADA 79] "Preliminary ADA Reference Manual", ACM SIGPLAN Notices, Vol. 14, No. 6, Part A, June 1979.

[Alagic 78] S. Alagic, M. Arbib, "The Design of Well-Constructed and Correct Programs", Springer-Verlag, 1978.

[Bergmann 83] N. Bergmann, "A Case Study of the F.I.R.S.T. Silicon Compiler", Proceedings of 3rd Caltech Conference on VLSI, Computer Science Press (ed. Bryant), 1983. pp 379-394

[Boyer 83] D. Boyer, N. Weste, "Virtual Grid Compaction Using the Most Recent Layers Algorithm", Proceedings of the IEEE ICCAD Conference, 1983. pp 92-93

[Buchanan 80] I. Buchanan, "Modelling and Verification in Structured Integerated Circuit Design", Ph.D. Thesis, University of Edinburgh, 1980.

[Buchanan 82] I. Buchanan, "A Language for the Combined Physical and Structural Description of Leaf and Composition Cells", Proceedings of Microelectronics 82, The Institute of Engineers, Australia, 1982.

[Cardelli 81] L. Cardelli, "Sticks and Stones: An Applicative VLSI Design Language", University of Edinburgh, Dept. of Computer Science, Report CSR-85-81, 1981.

[Clary 80] D. Clary, R. Kirk, S.Sapiro, "SIDS - A Symbolic Interactive Design System", Proceedings of the 17th Design Automation Conference, June 1980. pp 292-295

[Davie 81] A.J.T. Davie, R. Morrison, "Recursive Descent Compiling", Wiley, 1981.

[Davis 82] T. Davis, J. Clark, "SILT: A VLSI Design Language", Computer Systems Laboratory, Stanford University, Technical Report No. 226, October, 1982.

[Deas 83] A.R. Deas, "The UNIT Silicon Compiler", University of Edinburgh, Dept. of Computer Science, Report CSR-145-83, 1983.

[Dijkstra 72] E.W. Dijkstra, "Notes on Structured Programming" in "Structured Programming" (ed. Dahl et al), Academic Press, 1972. pp 1-82

[DoD 78] U.S. Department of Defence STEELMAN Requirement for High Order Computer Programming Languages, June 1978.

[Dunlop 80] A. Dunlop, "SLIM - The Translation of Symbolic Layouts into Mask Data", Proceedings of the 17th Design Automation Conference, June 1980. pp 595-602

[Gibson 76] D. Gibson, S. Nance, "SLIC - Symbolic Layout of Integrated Circuits", Proceedings of the 13th Design Automation Conference, June 1976. pp 434-440

[Gordon 81] M. Gordon, "A Very Simple Model of Sequential Behaviour in Nmos", VLSI '81, Academic Press (ed. Gray), 1981.

[Gray 82] J. Gray, I.Buchanan, P. Robertson, "Designing Gate Arrays Using a Silicon Compiler", Proceedings of the 19th Design Automation Conference, June 1982.

[Hughes 81] J.G. Hughes, "The Edwin User's Guide", University of Edinburgh, Dept. of Computer Science, Report CSR-74-81, 1981.

[Hughes 83] J.G. Hughes, "VLSI Design Tools", University of Edinburgh, Dept. of Computer Science, Internal Report, 1983.

[Jensen 74] K. Jensen, N. Wirth, "Pascal: User Manual and Report", Lecture Notes in Computer Science, Vol. 18, Springer-Verlag, 1974.

[Johannsen 79] D. Johannsen, "Bristle Blocks", Proceedings of the 16th Design Automation Conference, June 1979. pp 310-313

[Knuth 74] D.E. Knuth, "Structured Programming with GOTO Statements", Computing Surveys, Vol. 6, 1974. pp 261-301

[Lengauer 84] T. Lengauer, K. Mehlhorn, "The HILL System: A Design Environment for the Hierarchical Specification, Compaction, and Simulation of Integrated Circuit Layouts", Proceedings of the 1984 MIT Conference on Advanced Research in VLSI, Artech House (Penfield ed.), Jan 1984. pp 139-149

[Locanthi 78] B. Locanthi, "LAP: A Simula Package for IC Design", Caltech Display File No. 1862, 1978.

[Lopez 80] A.D. Lopez, H.F. Law, "A Dense Gate Matrix Design Style for MOS LSI", Proceedings of ISSSC, Feb 1980.

[Lyon 81] R. Lyon, "A Bit Serial Architectural Methodology for Signal Processing", VLSI '81, Academic Press (ed. Gray), 1981.

[Marshall 84] R. Marshall, I. Buchanan, "SCALE: A Language for VLSI Design", University of Edinburgh, Dept. of Computer Science, Report CSR-158-84, 1984.

[Mead 80] C. Mead, L. Conway,
"Introduction to VLSI Systems",
Addison-Wesley, 1980.

[Mead 81] C. Mead,
"VLSI and Technological Innovation",
VLSI '81, Academic Press (ed. Gray), 1981.

[Milne 83a] G.J. Milne, "CIRCAL -
A Calculus for Circuit Description",
Integration, Vol. 1, Nos. 2 & 3, 1983.

[Milne 83b] G.J. Milne, "The Correctness of
a Simple Silicon Compiler",
Proceedings of the 6th International
Symposium on Computer Hardware Languages
and their Application, North-Holland,
May 1983.

[NATO 76] "Software Engineering, Concepts and
Techniques: Proceedings of the NATO
Conferences", (ed. Naur et al),
Petrocelli/Charter, 1976.

[Newkirk 83] J. Newkirk, R. Matthews,
"The VLSI Designer's Library",
Addison-Wesley, 1983.

[Noyce 77] R.N. Noyce, "Microelectronics", Scientific American, Vol. 237, No. 6, September 1977. pp 63-69

[Pettengill 83] R.C. Pettengill, G.A. Haynes, "The SLED/SDL Symbolic VLSI Design System", Proceedings of the IEEE ICCAD Conference, 1983. pp 10-11

[Rees 80] D.J. Rees, "Skimp MkII", University of Edinburgh, Dept. Of Computer Science, Report CSR-52-80, 1980.

[Rees 83] D.J. Rees, "High-Level Tools for the Composition of VLSI Designs", Research Proposal, University of Edinburgh, Dept. of Computer Science, 1983.

[Robertson 83] P. Robertson, "The IMP77 Language", University of Edinburgh, Dept. of Computer Science, Report CSR-19-77, 3rd Edition, 1983.

[Rosenberg 82] J. Rosenberg, N. Weste, "ABCD - A Better Circuit Description Language", Microelectronics Centre of North Carolina, Technical Report No. 82-01, 1982.

[Rosenberg 84] J. Rosenberg, "Chip Assembly Techniques for Custom IC Design in a Symbolic, Virtual-Grid Environment", Proceedings of the 1984 MIT Conference on Advanced Research in VLSI, Artech Hose (Penfield ed.), Jan 1984.

[Rowson 80] J.A. Rowson, Understanding Hierarchical Design, Ph.D. Thesis, Dept. of Computer Science, California Institute of Technology, 1980.

[Schlag 83] M. Schlag, Y.Z. Liao, C.K. Wong, "An Algorithm for Optimal Two-Dimensional Compaction of VLSI Layouts", Proceedings of the IEEE ICCAD Conference, 1983. pp 88-89

[Sequin 82] C.H. Sequin, "Generalized IC Layout", Report, Computer Sciences Division, Electrical Engineering and Computer Sciences, University of California, Berkeley. 1982.

[Siskind 82] Siskind, Southard, Crouch, "Generating Custom High-Performance VLSI Designs from Succinct Algorithmic Descriptions", Proceedings of MIT Conference on Advanced Research in VLSI, 1982.

[Smith 83] Lee Smith, Acorn Computers, Personal Communication.

[Sutherland 77] I.E. Sutherland, C.A. Mead, "Microelectronics and Computer Science", Scientific American, Vol. 237, No. 3, September 1977. pp 210-228

[VAX 78] VAX11 Software Handbook, Digital Equipment Corporation, 1978. pp 29-32

[Weste 81a] N. Weste, "Virtual Grid Symbolic Layout", Proceedings of the 18th Design Automation Conference, 1981. pp 225-233.

[Weste 81b] N. Weste, "MULGA - An Interactive Symbolic Layout System for the Design of Integerated Circuits", The Bell System Technical Journal, Vol. 60, No. 6, July-August 1981. pp 823-857.

[Werner 83] J. Werner, G. Robson, R. Harris, "Comparing the Computer-Aided Engineering Systems in Action", VLSI Design, Vol. IV, No. 7, November 1983.

[Williams 77] J. Williams, "Sticks - A New Approach to LSI Design", MSEE Thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., June 1977.

[Williams 78] J. Williams, "STICKS - A Graphical Compiler For High Level LSI Design", Proceedings 1978 NCC, May 1978. pp 289-295

[Wirth 71a] N.Wirth, "The Programming Language Pascal". Acta Informatica, Vol. 1, No. 1, 1971. pp 35-63

[Wirth 71b] N. Wirth, "Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14, No. 6, April 1971. pp 221-227

[Wirth 77a] N. Wirth, "Modula, A Language for Modular Multiprogramming", Software Practice and Experience, Vol. 7, 1977. pp 3-35

[Wirth 77b] N. Wirth, "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?", Communications of the ACM, Vol. 20, No. 11, November 1977. pp 822-823

[Wolf 83a] W. Wolf, J. Newkirk, R.Matthews, R. Dutton, "Dumbo, A Schematic-to-Layout Compiler", Proceedings of 3rd Caltech Conference on VLSI, Computer Science Press (ed. Bryant), 1983. pp 379-394

[Wolf 83b] W. Wolf, R. Matthews, J. Newkirk, R. Dutton, "Two-Dimensional Compaction Strategies", Proceedings of the IEEE ICCAD Conference, 1983. pp 90-91

[Yourdan 78] E. Yourdan, L. Constantine, "Structured Design", Yourdan Press, 1978.

[Zinszner 83] R. Zinszner, H. De Man, K. Croes, "Technology Independent Symbolic Layout Tools", Proceedings of the IEEE ICCAD Conference, 1983. pp 12-13