An Idiomatic Framework for the Automated Synthesis of Topographical Information From Behavioural Specifications

Alexander Roger Deas

Ph.D. Thesis
University of Edinburgh
1985

Abstract

Between 1965 and 1985, microelectronic devices have grown in complexity from a few dozen devices per chip to VLSI processors comprising half a million transistors. Within the next decade wafer scale integration and sub-micron feature sizes are expected to make it possible to put many millions of gates on a composite chip. A number of observers have pointed out that the complexity of devices using these developments in fabrication technology is beyond that which can be managed with todays design methods and tools. It is clear that substantial advances in Computer-Aided Design (CAD) techniques are needed.

The most radical approach to managing the complexity of VLSI and ULSI is to aim for complete automation of the design process, to create a tool which produces mask geometry, performance estimates and test data from a high level description of the required chip. Unfortunately, up to now CAD tools for producing mask geometry have been limited to structural descriptions for design entry and specification, except in some specialised application areas. Systems that require a structural description do not address the problem of managing design complexity. This thesis describes methods for overcoming the problems associated with the increasing design complexity, by presenting a framework for taking a description at the behavioural level and synthesising topographical information from it. In particular, this thesis presents methods for mapping behavioural functions into structural functions and then computing a floorplan for the resulting connectivity graph. It describes methods for compiling the floorplan into the geometry of the mask set needed to manufacture the chips.

Declaration

I declare that I wrote this thesis and that the ideas in it are my own. No material from this thesis has been presented for any other degree or professional qualification unless specifically cited in the body of the text.

All papers that I have written whilst registered as a PhD student are listed below.

- "Silicon Compilation: A VLSI Complexity Management Strategy", IEEE Euromicro '84, Copenhagen 1984.
- "An Expert System for VLSI Floorplanning", Expert Systems and Silicon Compilation for VLSI Design Workshop, Edinburgh 1984.
- "Second Generation Silicon Compilers: Tools for Chip Architects", ITC Intelligent Signal Processing Workshop, Edinburgh 1985.
- With I. Nixon, "Chromatic Idioms for Automated VLSI Floorplanning", VLSI '85, Tokyo 1985.

The following reports have also been published.

- "UNIT and LEGO, Database Languages for the U2 Silicon Compiler", Report CSR-180-84, University of Edinburgh, 1984.
- With I. Nixon, "Chromatic Idioms for Automated VLSI Floorplanning", Report CSR-182-85, University of Edinburgh, 1985. This report contains the same material as the paper presented at VLSI '85.
- "Extremal Connectivity Methods for Partitioning VLSI Designs", Report CSR-183-85, University of Edinburgh, 1985.

Contents

1. Introduction 1

Complexity and design effort 3

Controlling Complexity 3

Implications of Rising Complexity in VLSI 4

The issues 6

Other motivations 7

CAD tools 7

Step One: Physical layout 7

Step Two: Symbolic layout 8

Step Three: Gatearrays 8

Step Four: Silicon Assemblers 9

Step Four (sideways): Module Generators 10

Step 5: Silicon Compilers 10

The Obstacles 11

Design Capture 11

Function Assignment 11

Library Design 12

Floorplanning 12

Routing 12

Silicon Assembly 13

User Interaction 13

The verification loop 14

What this thesis does 16

Experiments 16

How this thesis is organised 16

2. Degrees of Planarity 18

Notation 18

Definitions 19

Operations on Graphs 21

Degrees of Planarity 22

Properties of $\varphi(G)$ 27

Trivial Properties 27

Chromatic Properties 28

Implications of $\varphi(G) \geq \chi(G)$ 30

Time Complexity of $\varphi(G)$ 30

Local Planarity 31

Subtractive methods for finding the local planarity 33

Additive methods for finding local planarity 33

External predicates 34

Contraction methods for finding local planarity 34

Properties of $\varphi'(v_i)$ 35

Time complexity of $\varphi'(v_i)$ 35

Polynomial Approximations to $\varphi'(v_i)$ 36

Direct approximations to $\varphi'(v_i)$ 38

Indirect approximations to $\varphi'(v_i)$ 38

Time Complexity of the MC function 39

Error distribution of the MC function 40

An Improved Heuristic 41

Summary 41

3. Idiomatic Floorplanning 44

The quality of a floorplan 44

Background 45

An outline of the method 47

Basic Concepts 48

Method 52

Clustering 52

Classification 54

Placement 54

Implementation classes of floorplan 56

Summary 59

4. Hardware Description Languages **60**

A model of HDLs 60

The Walker model: Independence between domains of description and levels of abstraction 62

The semantics recovery tarpit 63

Implications for automated synthesis 64

Formulating Specifications 64

Behaviour, not structure 66

Requirements of a design capture language 67

Existing design capture languages 68

Technology Capture 69

Summary 70

5. The UNIT Language **71**

Design Capture 71

Program Structure 71

Refinement 73

Program Components 73

Preliminaries 73

Lexical Conventions 76

Reserved Words 76
Special Symbols 76
Identifiers 77
Comments 77
String Literals 77
Integer Literals 77
Compiler Options 77
Scope rules 79
Dressing 79
Lib Statement 79
EndFile Statement 80
Blocks 80
Header 80
Ports 80
Symbolic Constants 82
Instances 82
Connectivity 85
A Simple program 85

6. The LEGO Language **87**

Influence 87
Program Structure 87
Preliminaries 89
Lexical Conventions 89
Reserved Words 89
Special Symbols 89
Identifiers 89
Comments 89

String Literals 90
Integer Literals 90
Expressions 91
Conditionals 91
Compiler Options 93
Scope rules 94
Syntax 94
Symbolic Constants 94
Technology Statement 95
Design Rule Generator 95
Families 96
Family Headers 97
Cells 97
Cell Ports 98
Cell Body 102
History Statement 103
Report Statement 103
TransferEqn Statement 104
Power Statement 104
Size Statement 105
Geometry Statement 105
Dynamic Cells 105
Cell Examples 107
Species 108
Species Ports 109
CellPort Statement 110
Fmax Statement 110
Species Body 110

Parameters Statement 111
Composition 112
Transformations 114
Conditional Composition 115
More Examples 116
Composition Rules 116
Composition Checks 116
Cell Flexibility 119
Dynamic Species Definition 120
Species Selection 120
Reserved Words Revisited 122
Summary 122

7. A LEGO example **124**
Definition 124
A Counter Family 124
Port Conventions 125
Loadable Counter 125
Circuit Design 126
Validation 126
Port Positions 127
LEGO Program 128

8. The UNIT Language and Silicon Compilation **131**
Parsing 131
Knowledge Bases 131
Telescopic Languages 132
Compilation 133
Phase One 133

Design Validation 135
The VALID UNIT Language 136
Cell Definition 137
Design Analysis 138
Floorplanning 139
Location Statement 141
Posn Statement 142
Size Statement 143
Floorplan Editor 143
Code Swelling 143
Phase Two 144
Existing datastructures 145
Corner-Stitching 147
Silicon Assembly 149
Placement 149
Perturb and Slice 149
Power Routing 150
Global Routing 151
Skirting 154
Signal Routing 154
Routing Failure 158
Cell Creation 158
From LEGO to UNIT 158
Pad Placement 158
Spawn 159
Summary 159

9. Very high level optimisations **161**

Very high level specifications 162

Defining the datapath 163

Intrainstruction mergers 164

Recovering microcode and clock information 165

How useful is very high level optimisation? 166

Summary 166

10. Conclusion **168**

Areas for further research 169

A. List of Symbols **170**

Acknowledgements

I am grateful to both my supervisors, Hamish Dewar and Sidney Michaelson. I thank Hamish Dewar of the University of Edinburgh (now of CLAN Systems Ltd., Bush Estate, Mid Lothian.), for his suggestions whilst the ideas presented here were still embryonic. I thank Sidney Michaelson for his time, the long discussions we had and for granting me the freedom to explore and take responsibility, without which I would have found progress slower and any success more elusive.

Thanks to David Rees for the support from the VLSI group at Edinburgh, and for creating the environment in which the ideas presented here have matured. Thanks to Gordon Hughes for maintaining the VLSI tools and organising fabrication.

A special thanks belong to all those who have helped implement the things reported here, in particular, to Ian Nixon for producing the I.F. floorplanner and his feedback on the original chromatic idiom ideas. Thanks to Ron Morton for his work on power routing, Les Stretch for his work on planar routers, John Thomlinson for implementing Burstein's switchbox routing algorithm, Tom Waring for developing a much better switchbox router, Phil James for his work on tech files and DRG, Charles Kinnear for his ideas on a slicing algorithm and Kin-Yu Chung for his O(n) bloating algorithm.

Writing big parsers and lexical analysers is dull work, especially when the language definitions had to be changed often. So I am grateful to George McCaskill for producing the APG parser generator. The earliest and the latest parsers for both LEGO and UNIT were written by hand using recursive descent, but those in between were produced by APG.

I am greatly indebted to members of the Instrumentation and Applied Physics Division at AERE Harwell. From joining Harwell in 1976, I have been fortunate to receive the sound advice and council from Peter Fergus, without which I would probably never have entered University for an Honours degree, let alone for a PhD. To Peter, I am very grateful. Also to Alan Penn who organised everything at the Harwell end and supported what must have seemed the riskiest of ventures. Roger Watkins, as well as wearing the Division's educational offer hat for a while, has also been involved more intimately with our fledgling attempts to produce chips. There are many others. Brian Pierce, Colin Desborough, Norman

Whitehead and Alan Lewis all played important parts in making the work behind this thesis succeed. To these I am grateful.

Chapter 1

Introduction

During the two decades since 1965, microelectronic devices have grown in complexity from a few dozen transistors per chip to VLSI scale processors comprising half a million transistors. Sub-micron lithography on wafer scale devices is expected to allow the progress in fabrication technology to continue for another decade, so by the mid-1990s it may be possible to produce microelectronic devices comprising several billion logic elements.

The actual number of geometric elements on a chip limited by the fabrication technology of the day has been doubling every 18 months. An article by Waldschmidt lists five factors which contribute to this progress [Waldschmidt 82]. Namely:

1. Decreases in minimum feature sizes, from 25 microns in 1966 to around 0.5 micron available on the most advanced fabrication lines in use today.

2. Increases in chip area, from $10mm^2$ in the 1960's to the $100mm^2$ common today. Various research projects are now exploring the possibility of using the entire area of a 6 inch silicon wafer, a usable area of over $14{,}000mm^2$.

3. Circuit cleverness which allows the same functionality to expressed in fewer circuit elements. A well known example is the reduction in the number of transistors in a memory cell, from 6 transistors in 1972 to 1 transistor in 1977. Allen gives examples of where this has happened to other logical functions, including adders and exclusive OR gates [Allen 82].

4. Invention of new device technologies.

5. Better layout techniques, such as the gate matrices for random logic and the waffle-iron layout for MOS power transistors.

The spectacular nature of the growth in device complexity is evident from the two graphs on the next page. The reduction in feature sizes over time is charted in figure 1-1 alongside the aggregate rise in complexity in figure 1-2.

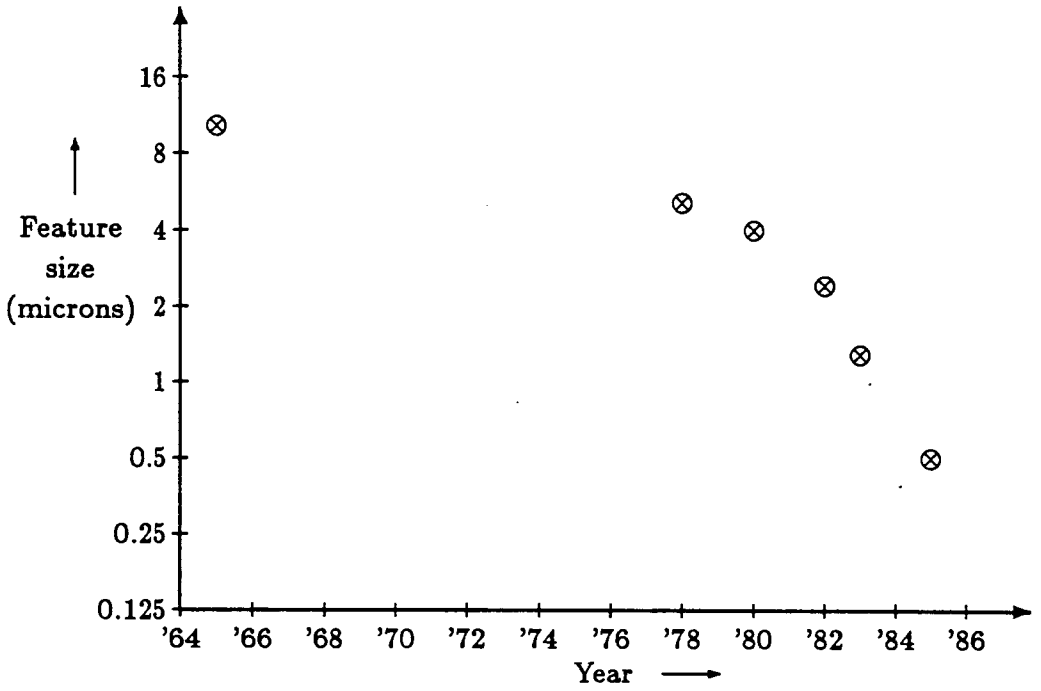


Figure 1-1: Smallest feature size on a production line

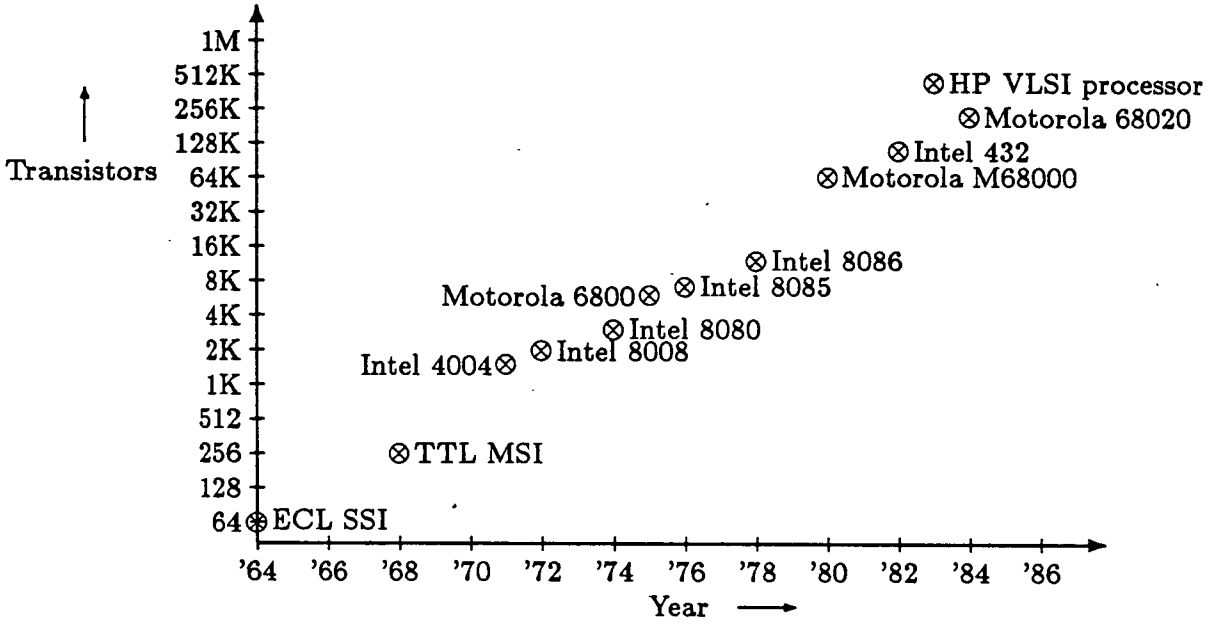


Figure 1-2: Greatest number of transistors on a production microprocessor

Gordon Moore, David McGreivy and Iann Barron have pointed out that the rapid progress in fabrication technology is outstripping the ability of engineers to manage design complexity [Moore 75], [Moore 79], [McGreivy 82], [Barron 83]. These observations, if correct, would imply that a large proportion of the billions of dollars spent each year on improving fabrication technololgy will be wasted unless there is a fundamental change in the way chips are designed. To understand the reasons behind these forecasts, it is necessary to consider the link between chip complexity and the effort, or time, needed to design a chip.

Complexity and design effort

The relationship between complexity and design effort for programming tasks has been the topic of numerous studies. Most studies conclude that design effort increases as a power of the code size. That is:

$$\text{Design effort} = Cn^{exponent}$$

The constant of proportionality, C, is the effort needed to write one line of code. The factor n is the total number of lines needed.

There are numerous factors controlling the size of C, including how regular the code is, the experience of the programmer and the difficulty implicit to the application being coded. Over the extremes, C may vary by a factor of a few hundred to one.

In the context of this thesis, we shall be concerned with large problems, where n is in the hundreds of thousands or millions. Under these circumstances the exponent of n becomes the dominating component in the effort equation.

In his classic study "The Mythical Man-Month" (*sf.* [Brooks 75]) Brooks cites Nanus [Nanus 64] and Weinwurm [Weinwurm 65] in support of his own experience that the value of the exponent is 1.5. That is:

$$\text{Design effort} = Cn^{1.5}$$

Controlling Complexity

The majority of modern programming languages provide facilities for controlling size of n, the code complexity. For example, hierarchical languages allow a large program to be written as a tree of modules and scope is used to restrict the number of free variables.

Unfortunately, of the methods for controlling complexity available to a programmer only a few can be applied to VLSI design. Whereas a program is usually executed sequentially, a microcircuit operates on data in parallel. A microcircuit does not have the clean interface of a program procedure to collect together data, instead it is monolithic with the special problems that implies in producing test patterns. Moreover, the problems associated with circuit timing, analogue voltage levels, fan out, current density and thermal gradients add extra dimensions to the task of implementing an algorithm in silicon.

Implications of Rising Complexity in VLSI

For the reasons that have just been outlined, it would be optimistic to view the design of a VLSI chip as a programming exercise. But for the moment, it is useful to do so. By applying the programmer's complexity-effort equation it is possible to get a best-case idea of how long it would take to develop a new chip.

Chips with regular structure will have a low value for C. A memory chip is highly regular in its layout and it is true that a memory chip can be designed with perhaps one hundredth the effort needed to design an irregular structure like a microprocessor.

What about the value of n? Whatever the implementation technology, n is the number of operations the designer performs, whether he be a programmer or a VLSI engineer. For a programming task, n is the number of statements or lines of code in a module. In the context of a VLSI chip designed using the tools currently available, n is the number of transistors or logic gates on a chip divided by some regularity factor. The regularity factor, the reciprocal of C, is the number of symbol instances in a design divided by the number of symbols. The regularity factor is usually less than 10 even in the most regular of architectures such as RISC machines [Patterson 81]. Furthermore, the regularity factor grows only very slowly as n increases. Therefore, to a first order approximation, the regularity factor can be ignored and n treated as a linear function of the number of transistors in a design.

We have already noted that the number of transistors that can be put on a chip is growing exponentially in calendar time. Whenever the complexity doubles, a designer must handle twice as many logic elements. Hence the amount of effort required to produce a chip at the limit of fabrication technology is growing in time as an exponential function raised to a power. It is hardly necessary to examine figures 1-3 and 1-4 to realise that this result has serious implications.

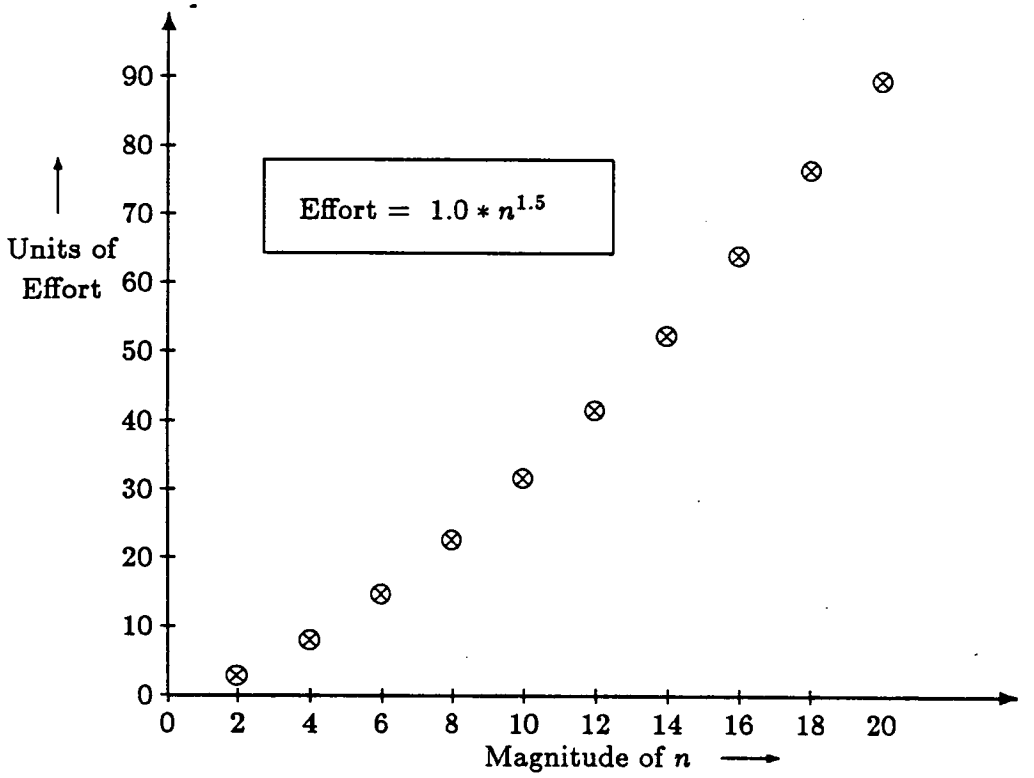


Figure 1-3: Increase in effort as a function of complexity

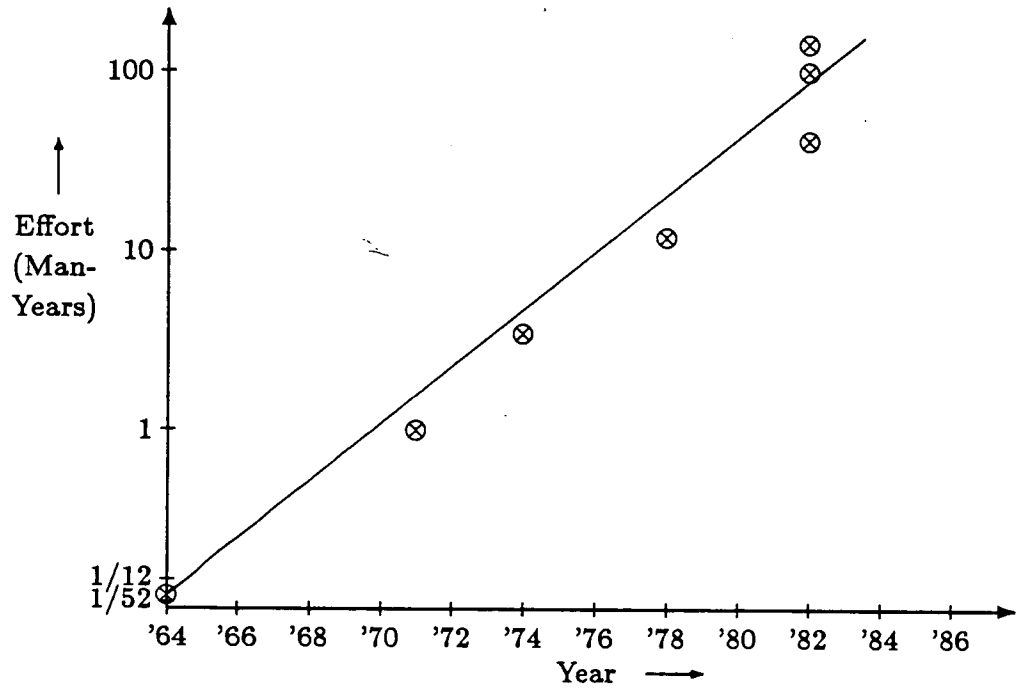


Figure 1-4: Effort to design chip at the limit of technology

For example, a half million transistor VLSI processor recently took one hundred man years of effort to design [Burkhart 83], [Canepa 83]. From this, we might expect the design of a three million transistor chip to consume roughly 1600 man years of effort. If progress were to continue at the present rate, then a chip making full use of the fabrication capability early in the next century would demand so many man years of effort to design that if the project period was of fixed duration, say ten years, then it would have to enlist support from more designers than there are presently people in the world. Alternatively, if the size of the design team was limited to ten designers, it would be necessary to start designing the chip long before the universe was created – between 4 and 5 billion years ago by current reckoning.

Recall that we have been treating VLSI design as a programming exercise, even though we noted that VLSI design is far harder. Therefore, the reality is worse than is suggested by the aforegoing argument.

The problems of design complexity become manifest in the form of extended development lead times, layout errors, microcode bugs, and timing failures. Of the commercial chips comprising 50K or more transistors, almost all display several of these problems.

The evidence suggests that we should look with suspicion at the specifications of some large chips being developed at the moment, we must look for bugs, we must expect failures. Furthermore, unless the complexity hurdle is overcome then it seems unlikely there be many chips with over a million transistors unless either the chip can be implemented as an extremely regular structure, or the primitives operations used in chip design become far more powerful. Only a tiny proportion of chip designs can be arranged as a regular structure, so the emphasis must be on producing more powerful CAD tools.

The issues

We have reasoned that in order to design a bug-free VLSI chip, the functional power of the design primitives must be greatly increased. The design process must become more abstract. It is asserted that for this to be possible, a mechanised method must be applied to generate low level information.

A radical interpretation of these requirements is to aim for the complete automation of the design process by creating a tool capable of interacting with engineers on their own level to capture a specification and of translating it into mask geometry: The ultimate VLSI CAD tool.

Other motivations

There are many incentives for producing this ultimate tool. The more important ones are catalogued below.

1. A big reduction in design effort would permit an engineer to explore different architectures. A study by Obrebska indicates that the difference between control micro-architectures in terms of silicon area and temporal performance, grows exponentially with linear increases in complexity [Obrebska 81] [Obrebska 82]. So if a tool exists which allows a chip to be designed and re-arranged very quickly then producing a chip in each of several architectural styles for the purposes of comparison could yield large gains in performance. Designers would be able to proceed by experiment rather than by intuition and guesswork.

2. A reduction in the lead time for a new design would allow a company to launch a product ahead of a competitor. History gives witness that, in microelectronics at least, the first company to launch a product into a new market tends to maintain the dominant position as the market matures.

3. Products could be upgraded more quickly.

4. Lower design costs would allow a greater degree of integration, leading eventually to custom VLSI usurping PCB technology as the primary implementation medium for electronic systems.

We have stated some reasons for wanting to produce this ultimate CAD tool. The question is how do we go about it?

CAD tools

The majority of design tools in use today have been developed during the last decade. Driven by demand, tools have evolved very quickly.

It is instructive to look at how the development of the contemporary computer aided VLSI design environment has happened.

Step One: Physical layout

The earliest silicon chips were designed by cutting shapes out of mylar film and then pasting the shapes onto celluloid. The earliest CAD tools emulated

this manual process. Tools dealt with physical mask layers without making any attempt to capture information about the significance of the geometry being laid out.

Most physical layout systems allow macros to be defined. Macros enable heavily used symbols to be parameterised. In practice, this was usually limited to defining transistors and contacts.

Step Two: Symbolic layout

During the latter half of the 1970's, mask macros were replaced by symbols. For example, one symbol might be used to represent a transistor and another a buried contact. By 1980, symbolic layout systems were available in many shapes and sizes, including the embedded programming languages known as LAPs [Locanthi 78], specialised VLSI design languages such as ALI [Lipton 83] and SCALE [Marshall 84] and systems for manipulating the style of topological layout known as STICKS [Williams 78]. However, the basic operation in all these symbolic layout tools is still very low level: transistors, contacts and wires. Engineers are very good at laying out small numbers of transistors in a leaf cell, but it is the interconnection of lots of leaf cells that leads to the complexity problem.

Step Three: Gatearrays

One level of abstraction higher than the symbolic transistor, is the logic gate.

CAD tools have been developed for translating designs expressed as a network of logic gates into a regular layout style known as a *gatearray.*

A gatearray consists of rows of gate primitives interdigitated by wiring channels. The majority of gatearray systems work by personalising the metal layers used for routing these channels though some, such as the Chipsmith [Gray 82] system, produce geometry for all the mask layers needed to manufacture the chip and so can rearrange the rows in the gatearray to ensure that the wiring channels are wide enough for the connections they have to carry.

Gatearray tools were the first to offer any built-in design verification software. Almost all gatearray layout tools on the market at the moment have associated with them a switch level simulator and some, such as the UK5000 system [SERC 84], can generate their own test patterns.

There are several problems with using gatearrays to implement chips on a VLSI scale. Firstly, a designer must describe a whole chip as a network of

primitive logic gates. This can hardly be said to tackle the complexity problem: gates are too low level. Secondly, gatearrays usually occupy about four times the silicon area of an equivalent custom layout. This leads to poor yields. Thirdly, gatearray layout tools use large amounts of computing time for the placement and routing procedures, which limits the size of design they can handle.

Step Four: Silicon Assemblers

The three categories of tool considered so far do not automate the custom design process: layout editors merely provide a convenient means for representing mask shapes, and gatearrays are only semi-custom.

The first CAD tools to automate the custom design process did not appear until around 1980. It was not until then that auto-place and auto-route techniques had progressed sufficiently enough to make a reality the *silicon assembler*, (known in some quarters as *chip assemblers*). Assemblers take a set of function blocks along with their logical interconnect, then place the blocks and route the power and signal nets. Type information associated with the ports on the periphery of each module ensure conformity with electrical and layout design rules.

The earliest attempt at producing a silicon assembler is described by Sato [Sato 79], but this was followed by some work on semi-automatic silicon assembler, reported in 1980 by Mudge [Mudge 80],

The most noteworthy silicon assemblers reported so far include those of Syed and Gamal [Syed 82], Szepieniec [Szepieniec 82], Hassett [Hassett 82] and Rivest [Rivest 82], but others are being reported all the time [Kuzmicz 83]. The past year in particular has seen a spate of announcements in the industry's trade magazines heralding new silicon assemblers. By now there must be at least twenty or thirty in existence.

Let there be no doubt: Silicon assemblers can produce very efficient layouts. But the input to a silicon assembler is at a structural level and a lot of work is involved in specifying each of the function blocks. The blocks must be configured to allow them to be connected to their neighbours easily, implying the need to have fore-knowledge of the final placement. The placement algorithms tend to be complex, hence several iterations are usually needed to configure the blocks. In short, silicon assemblers are too rigid: They cannot manipulate and reorganise the function blocks and must be seen therefore as a bridge between gatearray and full custom layout rather than as a synthesis tool competing with custom design.

Step Four (sideways): Module Generators

Programs for translating state tables or equations into Programmed Logic Arrays (PLAs) have been around for over 15 years. Recently, the power of PLA generators has been increased by incorporating automatic folding and partitioning, as well as allowing flexible port assignment. These developments are described by Hamilton [Hamilton 83] and Cole [Cole 84].

At the same time as the more versatile PLA generators started to become available, programs were developed for producing other complex blocks from functional descriptions, including Weinberger array generators [Weinberger 67], [Machar 83], datapath generators [Shrobe 82], [Schoellkopf 82], [Schoellkopf 83], [Siskind 82], [Camposano 84], microcomputer generators [Buric83], the so-called "topologisers" for synthesising topographics information from constrained circuit descriptions [Matheson 83], counter generators [Hughes 81] and so on.

In this thesis, these programs are referred to collectively as *module generators*, for that is what they do, although other people have called them "silicon assemblers", or more wishfully still, "silicon compilers". The term module generator will be used to refer to programs that can generate mask geometry for a functional module from a functional or behavioural specification but which lack the capability to produce artwork for an entire chip complete with pads and control logic.

Step 5: Silicon Compilers

In 1979, Johannsen postulated a design environment in which a designer can sit down, take some building blocks and experiment with many different designs before deciding on a specific architecture or implementation, an environment where designs can be created, simulated and translated into mask layout with very little effort [Johannsen 79].

Johannsen calls a tool with the attributes he describes a *silicon compiler.* Unfortunately this term has seen much abuse. So many keen salesmen misuse it that a silicon compiler is rapidly becoming identified in the popular eye with parameterised cell generators and silicon assembly. In this thesis, Johannsen's original definition will be adhered to and the corruption ignored.

Various silicon compilers have been developed. One of the most powerful is known as F.I.R.S.T. (*Fast Implementations for Real-time Signal Transforms*), by Bergmann, Denyer and Renshaw [Bergmann 83]. The F.I.R.S.T. compiler can produce commercial digital filters, correlators and other signal processors

from a description of the control and signal flow, but it is limited to bit-serial systems laid out according to a predetermined floorplan.

Most of the other silicon compilers developed since F.I.R.S.T. have similar restrictions on the type of design they can handle efficiently.

The Obstacles

No-one has so far produced a general purpose silicon compiler, one without any limitations on what can be implemented. Various obstacles have got in the way. The most awkward and pervasive of these are identified in [Deas 83].

This thesis presents a framework for writing silicon compilers. Obviously, for such a framework to be viable it must solve *all* of the problems that have been identified previously and coordinate the various solutions into a workable whole.

The remainder of this introductory chapter describes how this thesis is organised to reach that objective. By way of a plan of what is to come, the following sections will introduce each problem area in turn and state what contribution this thesis makes to their solution.

Design Capture

There has been a great deal of debate reported in the literature about the ideal form for capturing design specifications from an engineer. Groups with a computer science background assert that engineers wish to define a chip in a computer programming language such as C or Ada, whereas groups with an engineering background assert that a behavioural specification of the data and control flow as exemplified by an architectural diagram is more appropriate and useful.

This issue will be tackled later on, but which ever camp one happens to be in, the problem of designing a language to capture the specification from the designer remains.

This thesis presents a language for capturing specifications compatible with the design entry systems in common use.

Function Assignment

Assume the silicon compiler has got a specification in the form of a graph of interconnected functions, known as a *dataflow* graph. The first step in turning the dataflow graph into a chip is to decide how to implement each function. For

example, if one has a state machine, is it going to end up as a microcomputer, a PLA, a Weinberger array, a timing generator, a decision table or as a gatearray?

This thesis addresses the function assignment problem and presents a method for assigning function operators to module generators held in a library.

Library Design

A silicon compiler must have access to a database for information on module generators.

Some module generators are very flexible, for example some PLA generators can produce a folded PLA from a state table annotated with information on the desired position of input and output ports [Hughes 81]. Other module generators are highly constrained, as in the case of a fixed leaf cell. The problem of describing module generators in a way which both captures their flexibility and records their constraints is a difficult one.

This thesis tackles the library design problem and presents a language for specifying library parts.

Floorplanning

The process of defining the relative position and the shape of each node in the dataflow graph is known as *floorplanning.* Floorplanning must be efficient for a chip to be viable because it is the floorplanning decisions that determine the pin order, the wireability, the aspect ratio and the package area of a chip.

Existing floorplanners operate efficiently on specific classes of designs but fail to deal adequately with the general case.

This thesis addresses the floorlanning problem by presenting a novel method for VLSI floorplanning, using a connectivity metric to isolate clusters which are then assigned to one of a number of specialised floorplanners to be layed out. The complete floorplan is assembled from a hierarchy of floorplanned fragments.

Routing

Whilst numerous algorithms for routing wires across a channel have been published, there is a dearth of good algorithms for the (much harder) problems of power routing and switch-box routing.

In the power routing problem, all the ground and power needs of the function blocks on a chip must be met by producing a pair of non-overlapping tappered

routes on a single layer – the metal layer. Whilst there are several oblique references in the literature to ways these power routes can be produced, no published algorithm exists.

The ability to route the power rails automatically is an essential part of a silicon compiler. So, as part of the project reported in this thesis, the power routing problem was tackled by Ron Morton in the research phase of his MSc [Morton 85]. It would be inappropriate to describe Morton's methods here but one or two results do need to be mentioned in order to prove that automatic power routing algorithms can work within the database framework that is presented.

A similar situation exists for the switch-box routing problem, where ports one all four sides of a wiring space must be routed in no more than two layers. Again as part of the project reported in this thesis, Tom Waring developed an algorithm for switch-box routing superior to any of the methods described in the literature [Waring 85]. Waring did this work as part of an MSc research project and this thesis shall not preempt the Waring's own publication of his work, but because the existence of such an algorithm underlies the viability of a silicon compiler, one or two pertainent results will be mentioned along with a reference to where the details can be found.

Silicon Assembly

The output of a floorplanner can be laid out by a silicon assembler, but this requires support from the design database used by the compiler. This thesis addresses the problem of maintaining compatibility between compiler and assembler by using a homogeneous database for both activities.

User Interaction

Space on a silicon chip is an expensive resource. A linear increase in the size of a silicon die is accompanied by an exponential drop in the yield. For this reason, engineers are under pressure to make a chip die as small as possible.

No silicon compiler can produce a layout in the minimum area, just as no program compiler can produce machine code that uses memory optimally. But the trouble with silicon is that it is a two dimensional medium and as a result, it is easy to criticise a layout: Even someone who has never designed a chip can see "obvious" improvements in a layout that has 30% white space. At least with an optimising FORTRAN compiler a first year undergraduate cannot look at the machine code and say this or that has been implemented badly.

Even a perfect silicon compiler might not use all the space on a silicon die. It might be more efficient to leave a space rather than contort wires into it or site active circuitry over it. White space, unlike space covered with logic, makes scarcely any difference to the yield. But with a practical compiler, the areas of white space are larger than they could be. Unfortunately, when an engineer spots some white space on a chip he associates the space with inefficient compilation, which is not necessarily true. No matter how small the area of space, the sight of it seems to trigger an irresistable urge to alter things. There are two approaches to handling this problem:

1. Deny the user any interaction. This is quite difficult because the final mask geometry can be hacked using a text editor. If one succeeds in preventing hacks, users may become so frustrated that the compiler would fall into desuetude.

2. Provide a beautiful scheme for interacting with the compiler at the different levels of design abstraction. A user could call an interactive floorplan editor from within the compiler if he is unhappy with the floorplan, a more determined user might want to alter the way a logical interconnect is decomposed into sprees by the global router, and a user who is either incredibly brilliant or just pig-ignorant could even use a cell editor to alter a wiring channel or a leaf cell.

This thesis presents a framework for doing things the second way.

The verification loop

There must be a verification loop to allow users of a silicon compiler to confirm that the mask geometry generated by the compiler does in fact implement what was *meant* by the specification. To do this, it is necessary to confirm that:

1. The behaviour captured as a specification performs the function intended.

2. The library modules are correct. This is an important area to verify because modules are updated frequently and no matter how thorough the test procedure, it is not possible to test a module in every conceivable situation.

3. The compiler is correct. A silicon compiler is a gigantic program using highly sophisticated heuristics. Whilst it is unreasonable to expect such a tool ever to be completely debugged, it is also unreasonable to expect

users to keep on fabricating faulty chips every time a new bug becomes manifest. A fabrication cycle costs more than 10,000 pounds and takes ten weeks, so it is important to detected any bugs *before* chips are fabricated.

The verification problem can be tackled either formally or informally.

Formal verification methods

Formal verification involves proving the equivalence between a high level specification and a low level program under certain assertions.

A study by Basili on the effects of software complexity on program development concluded that the majority of errors in a program are caused by faulty specifications [Basili 84]. One implication of Basili's result is that formal verification methods, besides being extremely difficult to implement, would fail to detect the most serious errors. For this reason we shall not consider formal verification methods in the VLSI context.

Informal verification methods

Informal verification methods, namely simulation, can be used to verify behaviour.

Whilst circuit level simulators such as SPICE [Nagel 75] can simulate circuits comprising at most a few hundred transistors, switch level simulators can simulate the behaviour of an entire chip in a reasonable amount of time. Switch level simulators are event triggered, so the simulation time grows at a rate which is at worst linear with the number of connections. Commercially available switch level simulation engines can process hundreds of millions of events per second. As chips become more complex, the simulation engines become more powerful, especially as switch level simulators can be structured to make full use of the resources offered by parallel machine architectures. For these reasons, it is reasonable to expect to simulate an entire chip at the switch level right up to the ultimate physical limits of fabrication technology. Therefore, switch level simulation is a good candidate for verifying a complete chip.

The methods presented in this thesis tackle the verification problem by using a technique which allows a switch level description of the mask geometry for an entire chip to be extracted quickly from a design database.

What this thesis does

In summary, this thesis fires a broadside at the problems involved in silicon compilation. It coordinates solutions to each of the main problems that have been identified into a framework upon which a complete compiler can be developed.

With the exception of the optimisations mentioned in Chapter 9 and a few of the library features described in Chapter 6, everything you will read about has been implemented.

Experiments

To develop the ideas presented in this thesis, three silicon compilers have been written – or rather, there have been three complete re-writes of the same compiler. These compilers are known as U2.1, U2.2 and U2.3. The U2.3 compiler is the most recent and all implementation specific comments made in this thesis refer to this version of the compiler.

Frequent reference will be made to the U2 compilers, without apology. Although this introduces a fair amount of implementation detail into a couple of the chapters, it is probably the least awkward way of "This is how I have tackled this problem". You may wish to tackle the problem a different way, but at least if one way of doing a job is presented you will have a head start in developing your better method.

How this thesis is organised

Chapters 2 and 3 deal with floorplanning. Chapter 2 presents some theoretical concepts by way of a foundation for Chapter 3.

Chapters 4 through to 8 deal with the languages for capturing design and library information and describe the database techniques that sew everything together. Chapter 4 state the requirements a design capture language and a library language must meet. Chapter 5 describes the UNIT language, used for design capture. Chapter 6 describes the LEGO language, used for capturing cell libraries. Chapter 7 gives an example of how a new cell family can be described in LEGO. Chapter 8 describes how a UNIT program is compiled into a sequence of lower level, more detailed, dialects of the UNIT language and how these support the database needs of a silicon compiler.

Chapter 9 looks at how the framework developed in this thesis can be extended to a functional level by optimising dataflow graphs.

The conclusion in Chapter 10 summarises the main contributions made by this thesis and indicates areas for further research.

Chapter 2

Degrees of Planarity

For more than half a century, mathematicians have been devising predicates to test for whether a graph can be drawn on a plane without any of the edges crossing. A graph thus drawn is said to be embedded in the plane and graphs that are embeddable are said to be planar.

However, the predicates provide no information on how easily the embedding is to find, nor do they indicate how near an aplanar graph is to being planar. Reasoning intuitively, it is easier to embed a graph consisting of a single vertex than it is to embed a maximally interconnected set of four vertices. Similarly, a maximally interconnected graph with five vertices is nearer to being planar than a maximally interconnected graph with a million vertices. Whilst these things may be common-sense, there is no mathematical support for them.

The ability to distinguish between graphs according to the degree of interconnectedness, or the degree of planarity, is fundamental to the floorplanning method described later on. So in this chapter a planarity metric is developed and a function is derived from it for finding how much each vertex in a graph contributes to the planarity of the whole.

Notation

The branch of mathematics that deals with the discrete properties of graphs is known as *Extremal Graph Theory*. Extremal graph theory is yet in its infancy so whilst most of the terms and notation are generally accepted, there are differences between authors.

To avoid ambiguity, our conventions are defined in the next section and a list of symbols is given in Appendix A. In general, this thesis conforms with the notation used by Bollobas [Bollobas 78].

Ideas I introduce are given as *Propositions* and immediately proved. Other people's ideas are introduced as *Theorems*, along with a reference to the original paper in which they are proved.

Some theorems are so well known they are referred to without any introduction. In doing this, the assumption is made that the reader is familiar with graph theory up to the level taught for an honours degree in a numerate discipline.

Definitions

A graph $G = (V, E)$ consists of two disjoint sets: the set of vertices V and the set of edges E. The symbols $v_1, v_2, v_3, \ldots$ are used to represent the vertices and the symbols $e_1, e_2, e_3, \ldots$ to represent the edges. An edge connecting vertex v_i to v_j can be denoted by $e_{i,j} = (v_i, v_j)$.

The *order* of a graph G is the number of vertices in its vertex set and is denoted by $|G|$. An *arbitrary* graph of order n is denoted by G^n.

If two vertices are connected by a common edge then they are said to be *adjacent*. The vertices associated with an edge e are said to be the *end vertices* of e. An edge is *incident on* its end vertices.

A *path* in a graph G is an alternating sequence of distinct vertices and edges, starting and ending with a vertex. A *trail* is a similar sequence that starts and ends with an edge. A *circuit* is a path that starts and ends with the same vertex. A *Hamiltonian circuit* is a circuit containing all the vertices of G. The number of edges in a path is called the *length* of the path.

We shall be concerned only with *simple graphs*. A simple graph $G = (V, E)$ is one in which:

1. G is adirectional. That is, for any edge $(v_i, v_j) \in E$, $(v_i, v_j) \Leftrightarrow (v_j, v_i)$.

2. For any pair of vertices $v_i \in V$ and $v_j \in V$, there exists a path from v_i to v_j. An adirectional graph that fails to meet this restriction is said to be *disjoint*.

3. There are no self-connected vertices.

The number of edges incident on the vertex v_i is called the *degree* of the vertex and denoted by $d(v_i)$. The minimum degree of any vertex in graph G is denoted by $\delta(G)$ and the maximum degree is denoted by $\Delta(G)$. When $\delta(G) = \Delta(G) = |G|$, G is said to be a *complete graph*. It is common practice to denoted a complete graph of order n by the symbol K^n. The complete graphs K^3 to K^5 are illustrated in figure 2-1.

A simple graph $G = (V, E)$ is said to be *bipartite* if V can be partitioned into two subsets V^a and V^b such that all edges in V are incident on one vertex in

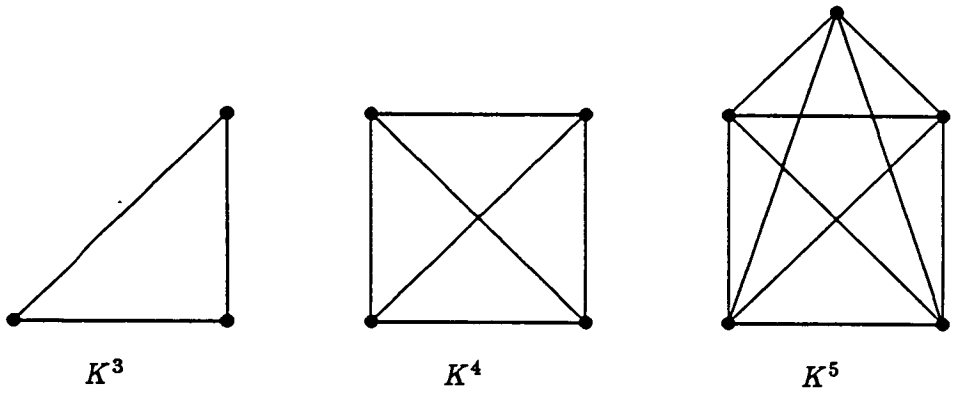


Figure 2-1: Some complete graphs

V^a and one vertex in V^b. If the number of vertices in V^a is p and the number of vertices in V^b is q, then G is denoted by $G^{p,q}$. A logical extension of bipartite graphs is to partition a graph into r subsets and a graph thus partitioned is said to be *r-partite.*

A bipartite graph $G = (V^a \cup V^b, E)$ is said to be *complete* if for every two vertices $v_a \in V^a$ and $v_b \in V^b$ there exists an edge (v_a, v_b) in E. A complete bipartite graph with k vertices in each vertex set is denoted by $K^{k,k}$. The complete bipartite graphs $K^{2,2}$ to $K^{4,4}$ are illustrated in figure 2-2.

The widespread use of the K notation for complete graphs is in deference to Kuratowski, and in particular, the two graphs K^5 and $K^{3,3}$ are known as the *Kuratowski Graphs.*

A *subgraph* is obtained from a parent graph $G = (V, E)$ by removing vertices from V and those edges incident on them from E.

If a subgraph is replaced by a single vertex such that all edges originally incident on a vertex in the subgraph become incident on the substitute vertex, then the replaced subgraph is called a *cluster* and the process of making the replacement is called *clustering.* Note that clustering introduces hierarchy to a graph. The letter p is reserved to denoted the order of a cluster.

A colour can be assigned to each vertex in a graph $G = (V, E)$ to produce a *colouring* of G. A colouring of G such that no vertex in V has the same colour as an adjacent vertex is called a *proper colouring* of G. The minimum number of colours needed to produce a proper colouring of G is called the *chromatic number* of G and is denoted by $\chi(G)$. If the edges of G are assigned colours instead of the vertices such that no two edges incident on the same vertex bear the same

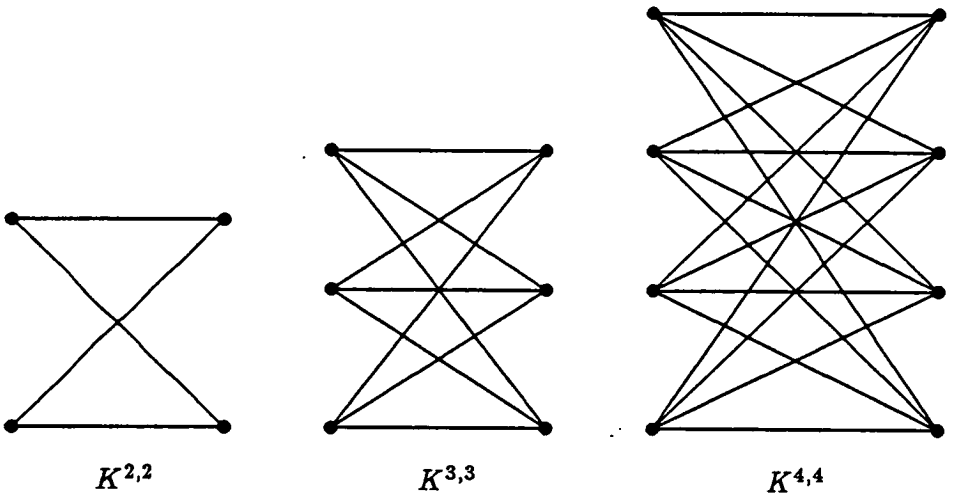


Figure 2-2: Some complete bipartite graphs

colour, then that assignment is a *proper edge colouring* of G. The minimum number of colours needed to produce a proper edge colouring of G is called the *edge chromatic number*, denoted by $\chi'(G)$. Note that $\chi'(G)$ is not related in any way whatsoever to $\chi(G)$.

Operations on Graphs

The following operations may be performed on a graph.

- **Union of two graphs**
  The union of two graphs $G = (V, E)$ and $H = (V', E')$ is formed by the union of their vertex and edge sets, that is, $G \cup H = (V \cup V', E \cup E')$.

- **Intersection of two graphs**
  The intersection of two graphs $G = (V, E)$ and $H = (V', E')$ is formed by the intersection of their vertex and edge sets, that is, $G \cap H = (V \cap V', E \cap E')$.

- **Identification of two vertices**
  If the graph $G = (V, E)$ contains the two vertices $v_i \in V$ and $v_j \in V$, then these may be *identified* by replacing v_i and v_j by a new vertex such that all edges originally incident on v_i or v_j become incident on the new vertex and removing v_i and v_j from V.

- **Contraction of an edge**
  If the graph $G = (V, E)$ contains the edge $e = (v_i, v_j)$, e can be *contracted* by identifying v_i and v_j. The contraction of an edge e removes it from E.

- **Contraction of a graph**
  A graph G is *contractible* to H if G can be transformed into the graph H by a series of edge contractions on G. The contraction of G to H is denoted by $G \rhd H$.

- **Subcontraction of a graph**
  A graph G is *subcontractible* to H if a subgraph of G can be contracted to H, denoted by $G \succ H$.

Whilst it is possible to perform other operations on a graph, those listed are sufficient for our needs.

Degrees of Planarity

The most basic requirement of a planarity metric is that it must distinguish between graphs that are planar and graphs that are not.

In 1930, Kuratowski proved that a graph is planar if and only if it cannot be contracted to K^5 or $K^{3,3}$. This result is known as *Kuratowski's Theorem.* Harary and Tutte have proved that the converse to Kuratowski's theorem is also true.

Kuratowski's theorem is the only strong characterisation of planar graphs we have, so any planarity metric must critically depend upon the syzygy $(K^5, K^{3,3})$. Supposing there is a series of such pairs, then the densest possible series must be:

$$\begin{pmatrix} K^0, K^{?,?} \end{pmatrix}$$
$$\begin{pmatrix} K^1, K^{?,?} \end{pmatrix}$$
$$\begin{pmatrix} K^2, K^{?,?} \end{pmatrix}$$
$$\begin{pmatrix} K^3, K^{?,?} \end{pmatrix}$$
$$\begin{pmatrix} K^4, K^{?,?} \end{pmatrix}$$
$$(K^5, K^{3,3})$$
$$\begin{pmatrix} K^6, K^{?,?} \end{pmatrix}$$
$$\begin{pmatrix} K^7, K^{?,?} \end{pmatrix}$$
$$\ldots$$

To produce a scale of planarity, it is necessary to find the bipartite partner for each complete graph in the series.

We could simply match bipartite graphs against the attributes one might associate with each point on the emerging planarity scale. For example, if the origin of the scale is based on the contraction of a graph to K^0, then the planarity would be zero for the null graph and only the null graph, the planarity would be 1 for a single vertex graph, 2 for acyclic graphs, at least 3 when cycles exist, 4 is the maximum for planar graphs, 5 for the simplest aplanar graphs and so on.

Alas, life is not so simple. Bipartite graphs can be contracted to complete graphs. If the complete graph that can be contracted from the bipartite graph is larger than the complete graph the bipartite graph is partnered to in the series, then the series become chaotic. The only way to avoid this mess is to ensure that the bipartite graphs cannot be contracted to their complete partners. So the problem changes to one of finding the largest complete graph contractible from a bipartite graph. We shall set about solving this problem by comparing the number of vertices and the number of edges in each of the two graph types.

By inspection, it is apparent that a complete bipartite graph $K^{m,m}$ has $2m$ vertices and m^2 edges. Each vertex v is of degree m, that is, $d(v) = m \; \forall \; v$.

Also by inspection, it can be seen that a complete graph K^k has k vertices,

$$(k-1) + (k-2) + (k-3) + \cdots + (k - [k-2]) + 1 \; = \; \sum_{i=1}^{k-1} i$$

edges and that each vertex is of degree $k - 1$.

Unfortunately, if we enumerate the number of edges and the number of vertices for each type of graph, there is no obvious correlation. To illustrate this point, the numbers of vertices and edges for the complete graphs K^1 to K^{18} are enumerated in the table overleaf.

Complete Graph	Number of Edges	Number of Vertices
K^1	0	1
K^2	1	2
K^3	3	3
K^4	6	4
K^5	10	5
K^6	15	6
K^7	21	7
K^8	28	8
K^9	36	9
K^{10}	45	10
K^{11}	55	11
K^{12}	66	12
K^{13}	78	13
K^{14}	91	14
K^{15}	105	15
K^{16}	120	16
K^{17}	136	17
K^{18}	153	18

And for bipartite graphs:

Bipartite Graph	Number of Edges	Number of Vertices
$K^{0,0}$	0	0
$K^{1,1}$	1	2
$K^{2,2}$	4	4
$K^{3,3}$	9	6
$K^{4,4}$	16	8
$K^{5,5}$	25	10
$K^{6,6}$	36	12
$K^{7,7}$	49	14
$K^{8,8}$	64	16
$K^{9,9}$	81	18
$K^{10,10}$	100	20
$K^{11,11}$	121	22
$K^{12,12}$	144	24

There is no obvious pairing so we must resort to some O-Grade algebra to find the smallest complete bipartite graph $K^{m,m}$ contractible to a given complete graph K^k.

Let us start by recapitulating what we already know about each type of graph.

Graph type	Num. edges	Num. vertices	Degree of each vertex
$K^{m,m}$	m^2	$2m$	m
K^k	$\sum_{i=1}^{k-1} i$	k	$k-1$

The sum expression for the number of edges in a complete graph is awkward to deal with, so let us start by converting it into a simpler form.

Commencing with:

$$\sum_{i=1}^{k-1} i = 1 + 2 + 3 + \cdots + (k-1)$$

Reversing the R.H.S. gives:

$$\sum_{i=1}^{k-1} i = (k-1) + (k-2) + (k-3) + \cdots + 1$$

Summing both of the above gives:

$$2 \sum_{i=1}^{k-1} i = k + k + k + \cdots + k$$

Rewriting R.H.S. gives:

$$2 \sum_{i=1}^{k-1} i = k\,(k-1)$$

Dividing both sides by 2 and rewriting gives:

$$\sum_{i=1}^{k-1} i = \frac{k^2 - k}{2}$$

A bipartite graph $K^{m,m}$ has m^2 edges, but as we have just shown, a complete graph has only $\frac{k^2-k}{2}$. This means that exactly $m^2 - \frac{k^2-k}{2}$ edges must be contracted.

A contraction removes one vertex from $V^a \cup V^b$. A bipartite graph has $a+b$ vertices, whereas K^k has only k vertices, so exactly $a+b-k$ vertices must be identified if $K^{a,b} \rhd K^k$. In the case of a complete bipartite graph $K^{m,m}$, this means that only $2m-k$ contractions may occur.

Thus we have two sets of equations: one for the number of vertices that must be removed, the other for the number of edges. These two equations are distinct – they cannot be equated with each other because the contraction of a single edge may cause any number of edges to be deleted. We need therefore some third equation to link everything together.

The third equation is obtained by considering how to minimise the number of edge contractions. Clearly, for a bipartite graph that has been subject to an

abitrary number of earlier edge contractions, further contractions cause the minimum number of edge deletions when it involves an interset edge. The incestuous contraction of edges within a vertex set will always cause $|G'|$ edges to be lost: G' is the graph obtained by performing an arbitary number of contractions in a bipartite graph. Therefore, if $K^{m,m}$ can be contracted to K^k, then it must be possible to do this by making $m-1$ interset contractions.

Our third equation says that $m-1$ edge contractions occur. Our first equation says that $2m-k$ contractions occur. So if we equate these together, we should get a relationship between m and k.

$$2m - k = \quad m - 1$$

Separating m and k gives:

$$k = \quad m + 1$$

It is possible to produce a similar expression based on the edge relation, but the vertex relation has been chosen instead because it avoids the complications introduced by having to set the sum factor $\frac{k^2-k}{2}$ factor equal to a the number of edge contractions, another arithmetic series for which the sum to m is $\frac{m^2-m}{2}$. The two quadratic sum terms in the edge relation act in the same way as the linear $m-1$ term does in the vertex relation.

The relationship between k and m is the key to finding the $K^{m,m}$ values in the planarity series: It means that a complete bipartite $K^{k-1,k-1}$ can be contracted to $K^k \; \forall \; k > 1$. Therefore, for the series $m = k - 1$.

Now that we know what bipartite graphs can be contracted to, we are able to match up the bipartite and complete graphs in the planarity series. It appears that there is simply a infinite series of bipartite-complete syzygies of the form $\left\{K^{k-2,k-2}, K^k\right\}$. But here there is a catch.

If $K^{1,1}$ is paired up with K^3 an ambiguity would exist because $K^{1,1} \equiv K^2$. Another problem at the start of the series is that we want to distinguish between acyclic graphs and cyclic graphs but K^3 is cyclic whereas $K^{1,1}$ is acyclic.

We can get over these problems by starting the bipartite series at $K^{3,3}$, Kuratowski's partner to K^5.

Hence, the series runs as follows:

$$K^0$$
$$K^1$$
$$K^2$$
$$K^3$$
$$K^4$$
$$\{K^5, K^{3,3}\}$$
$$\{K^6, K^{4,4}\}$$
$$\{K^7, K^{5,5}\}$$
$$\{K^8, K^{6,6}\}$$
$$\{K^9, K^{7,7}\}$$
$$\{K^{10}, K^{8,8}\}$$

This series points to the natural definition of the *degree of planarity* as being the contraction of a graph to a maximal $\left(K^k, K^{k-2,k-2}\right)$ pair. That is:

> **Definition 1:** The planarity of a graph G, denoted by $\varphi(G)$, shall be defined as:
>
> $$max\left\{ max\left(k : G \rhd K^k\right), max\left(k : G \rhd K^{k-2,k-2}, k \geq 3\right) \right\}$$

Notice that contraction to bipartite graphs smaller than $K^{3,3}$ is barred.

Figure 2-3 enumerates the critical $\varphi(G)$ graphs for $\varphi(G)$=1 to 6. It should be no suprise to see that the critical $\varphi(G)$ graphs are in fact $\left(K^k, K^{k-2,k-2}\right)$ pairs.

Properties of $\varphi(G)$

The definition of $\varphi(G)$ has some interesting properties. These properties can be divided up into two groups: properties that are evident from the construction of $\varphi(G)$ and properties that are more complex and so need to be proved. Among the complex properties is a relationship between $\varphi(G)$ and $\chi(G)$.

Trivial Properties

It can be seen from the way $\varphi(G)$ is defined that $\varphi(G) < 5$ if and only if G is planar. Note also that $\varphi(G) = |G|$ if G is a complete graph.

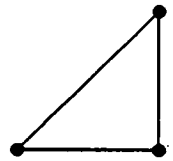
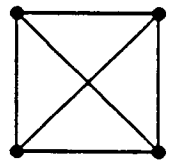
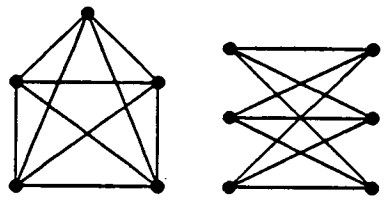
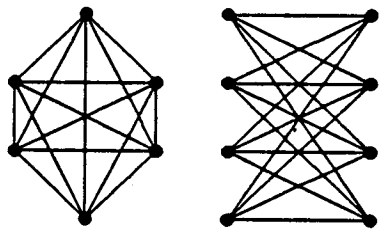
Planarity	Critical graphs
1	K^1
2	K^2
3	K^3 
4	K^4 
5	K^5 and $K^{3,3}$ 
6	K^6 and $K^{4,4}$ 

Figure 2-3: Least connected graphs for degrees of planarity 1 to 6.

Chromatic Properties

To investigate the chromatic properties of $\varphi(G)$ the following theorems are needed.

Theorem: 1 *(By Heawood [Heawood 90])*
If a graph G can be drawn on an orientable surface of genus $\gamma > 0$ then,

$$\chi(G) \leq H(\gamma) = \left\lfloor \frac{1}{2}\left(7 + \sqrt{1 + 48\gamma}\right)\right\rfloor$$

Theorem: 2 *Extension of Heawood's theorem by Dirac [Dirac 52], [Dirac 57] Suppose a graph G is drawn on an orientable surface of genus $\gamma > 0$ and $\chi(G) = k$, then $G \succ K^k$.*

With this equipment it is possible to prove the following proposition.

Proposition 1: For all simple graphs G, $\varphi(G) \geq \chi(G)$.

Proof: The function $\varphi(G)$ must be obtained either by contracting G to $K^{k-2,k-2}$ or by contracting G to K^k. Consider each case in turn.

Case 1: $G \rhd K^{k-2,k-2}$
Whenever $\varphi(G)$ depends on the contraction of G to a bipartite graph, then $\varphi(G)$ must be greater than the order of the biggest complete graph contractable from G. So if Case 2 of this proof is true, then Case 1 must also be true.

Case 2: $G \rhd K^k$
Using Dirac's extension to Heawood's theorem, if $\chi(G) = j$ then G is subcontractible to K^j. Therefore G must be contractible to K^j, or arranged another way, $G \rhd K^{\chi(G)}$. But if $\varphi(G)$ depends on the contraction of G to the largest complete subgraph in G, then $\varphi(G) \geq |K^{\chi(G)}|$, and so $\varphi(G) \geq \chi(G)$. □

Although the proof of $\varphi(G) \geq \chi(G)$ uses Heawood's and Dirac's theorems, the proof would become trivially simple if a certain conjecture by Hadwinger is ever proven in the affirmative [Hadwinger 43], [Hadwinger 58].

Hadwinger's conjecture goes as follows:

$$\text{If } \chi(G) = k \text{ then } G \rhd K^k.$$

This conjecture is essentially the inverse of Dirac's theorem. The conjecture is easily proven true for $k = 1, 2, 3$, and Dirac has proved it true for $k = 4$ [Dirac 57b]. Pages 251-253 of [Bollobas 78] prove that if the 4 colour conjecture (4CC) can be proved in the affirmative, then Hadwinger's conjecture is true for $k = 5$. As Appel, Haken and Koch [Appel 76], [Appel 76b] have since proved the 4CC, now the 4 colouring theorem (4CT), then Hadwinger's conjecture is true for $k = 5$. However, as much as the truth in Hadwinger's conjecture is

intuitively obvious, it remains unproven beyond $k = 5$. Due to the absence of a formal proof, no direct use of it will be made but it is mentioned here because it is far simpler than Heawood's theorem, so the reader may prefer the conjecture for informal use.

Implications of $\varphi(G) \geq \chi(G)$

Note from the proof of $\varphi(G) \geq \chi(G)$, that $\varphi(G)$ can only equal $\chi(G)$ when $\varphi(G)$ depends upon $G \triangleright K^k$. In the case where $\varphi(G)$ depends on $G \triangleright K^{k-2,k-2}$ then G may have an arbitrarily high degree of planarity whilst having an arbitrarily low chromatic number, provided $\chi(G) > 1$. For example, the complete bipartite graph

$$G' = \lim_{n \to \infty} K^{\frac{n}{2},\frac{n}{2}}$$

is 2 colourable, yet $\varphi(G') \to \infty$. To illustrate and reinforce this point, the graph G' is drawn in figure 2-4 for $n = 16$.

Time Complexity of $\varphi(G)$

To derive the upper bounds on the time needed to compute $\varphi(G)$, consider the graph G under the following proposition:

Proposition 2: The function $\varphi(G)$ is NP-Complete in time.

Proof: The value of $\varphi(G)$ must be obtained either by contracting G to $K^{k-2,k-2}$ or by contracting G to K^k.

Case 1: $G \triangleright K^{k-2,k-2}$ is NP-Complete
Proved by Statman, (cited by Garey [Garey 79]).

Case 2: $G \triangleright K^k$ is NP-Complete
Karp has proved that partitioning a graph into cliques is NP-Complete – a clique is the biggest complete graph in G [Karp 72]. So the case of $G \triangleright K^k$ is proven.

As $\varphi(G)$ can be computed by a series of two NP-Complete operations, it must be either in P or NP-Complete. Trivially, it cannot be in P because declining either one of the two contractions that need to be computed would result in a method for computing a known NP-Complete function in polynomial time. Therefore, $\varphi(G)$ must be NP-Complete. □

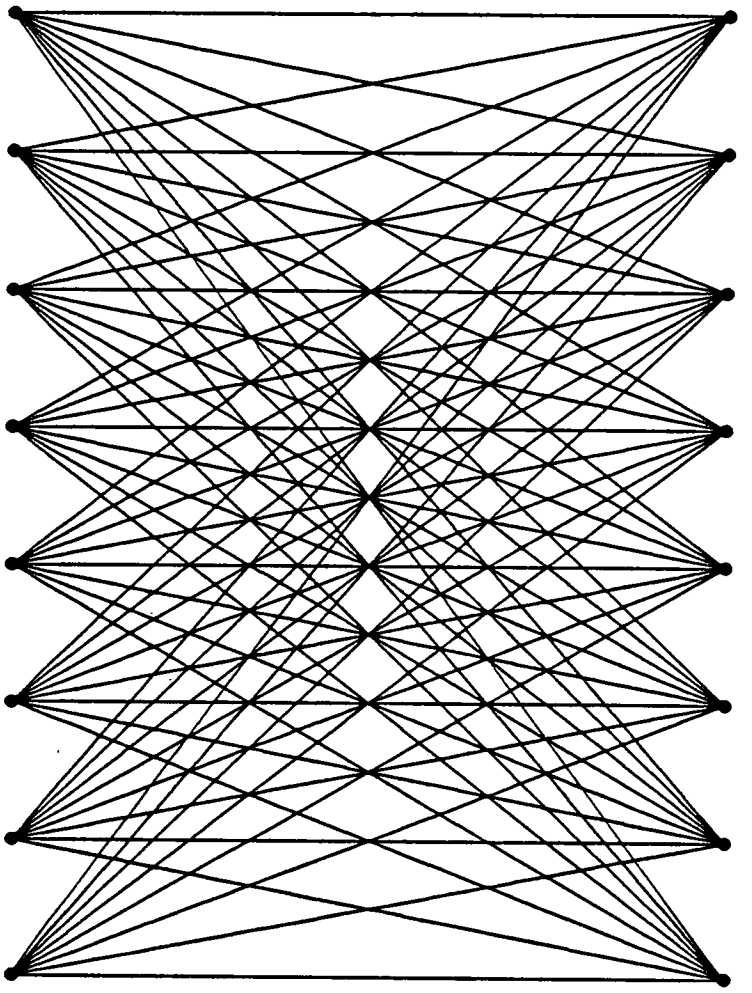


Figure 2-4: $K^{8,8}$ is highly aplanar, $\varphi(K^{8,8}) = 6$, yet 2 colourable.

Local Planarity

It is often useful to know how each vertex in a graph G effects the planarity of G. For example, a graph G in which $\varphi(G) = 2$ except for the addition of a subgraph K^7 might be clustered by replacing all the vertices that form K^7 by a single vertex, thereby allowing G to be embedded. For this clustering to be possible, it is necessary to isolate those vertices in the aplanar subgraph K^7.

To find the local planarity of a vertex $v_i \in V$ in a graph $G = (V, E)$, a subgraph must be built from v_i and its neighbours in such a way that the subgraph tells us how important v_i is to the global interconnect of G.

The concept of local planarity must involve the neighbours intimately, so a more exact definition of the word "neighbour" is needed than is afforded by its

popular usage. In fact, "neighbour" carries with it so many implicit meanings that we should use the idea of a *connection set* instead. The connection set of a vertex is defined as follows:

> **Definition 2:** For a graph $G = (V, E)$, the mth degree *connection set* of the vertex $v_i \in V$, denoted by $\mathbf{CS(v_i, m)}$, is that subset of V that can be reached by a non-looping path of length m precisely. This path is allowed to backtrack only to the start vertex and then only as the last step in the path.

By definition $CS(v_i, 0) = \{v_i\}$ and $\{v_i\} \cap CS(v_i, 1) = \emptyset$, where $\emptyset$ is the empty set. The degree m must be ordinal. The operators of set algebra apply to CS sets.

> **Proposition 3:** The function $CS(v_i, m)$ is NP-Complete in time.

Proof: (Due to Brebner [Brebner 84])
Consider a graph $G = (V, E)$. Trivially, the worst-case problem of computing $CS(v_i, m)$, where $v_i \in V$, is when $m = |G|$, which is identical to the problem of finding a Hamiltonian circuit in G. The latter problem has been proven NP-Complete by Karp [Karp 72]. Therefore, if the function $CS(v_i, |G|)$ is both the worst case and is functionally isomorphic to a known NP-Complete function, then $CS(v_i, m)$ must be NP-Complete for arbitrary values of m. □

Things are not quite as grim as might be inferred from the above proof. Indeed, if the size of m is known in advance, the members of $CS(v_i, m)$ can be found in polynomial time. This fact is not obvious, so its formal proposal and proof is given below.

> **Proposition 4:** When m is fixed and known, then the function $CS(v_i, m)$ can be computed in polynomial time.

Proof: By observation, $CS(v_i, 0) = \{v_i\}$, and $CS(v_i, 1)$ is simply the vertices adjacent to v_i. Nixon gives an $O(|G|^2)$ algorithm for computing $CS(v_i, 2)$ [Nixon 84]. Therefore, by virtue of the existence of polynomial methods for finding $CS(v_i, m)$ for $m = 0, 1, 2$, $CS(v_i, m)$ must be in P for the contraint on m proposed: The case where $m = |G|$ must be excluded, otherwise this proposition would revert to the NP-Complete state considered earlier. □

Subtractive methods for finding the local planarity

One approach to computing the local planarity might be to remove each vertex in turn from the vertex set V and recalculate the planarity of the remaining graph. If the planarity of the remaining graph is less than that of the original then it might be said that the vertex subtracted must be causing the original graph to be whatever planarity it happens to be.

However, the change in the planarity of a graph upon subtracting a vertex is quite different from the change caused by adding a vertex. In fact, unless a vertex forms a critical link between two subgraphs each of planarity lower than the original, then the subtraction of a vertex does not effect the planarity of the subgraphs at all.

To illustrate, consider the graph G formed by connecting an extra vertex v_i to one of the vertices in K^5. The planarity of K^5 is 5. The addition of v_i to K^5 is the same in planarity terms as the addition of v_i to a single vertex graph. A single vertex graph has a planarity of 1 before adding v_i and 2 afterwards, so the local planarity of v_i attached to the point graph must be 2. Therefore, the local planarity of v_i in G must also be 2. However, if the local planarity of v_i were to be calculated by some recursive subtraction process then the local planarity of v_i could be five! Clearly, the subtractive method is unusable.

Additive methods for finding local planarity

Earlier it was said that:

> "To find the local planarity of a vertex $v_i \in V$ in a graph $G = (V, E)$, a subgraph must be built from v_i and its neighbours in such a way that the subgraph tells us how important v_i is to the global interconnect of G."

This statement a subgraph should be created on the vertices $\{v_i, CS(v_i, 1)\}$ by adding edges to $CS(v_i, 1)$ to simulate the background planarity of the original graph. Clearly, edges must be added to $CS(v_i, 1)$ but not to v_i.

One way of adding edges to a graph to simulate the background connectivity is to add an edge if the transitive closure between two vertices is less than some number. But the blind addition of edges to $CS(v_i, 1)$ according to whether there is a transitive closure between any two vertices in $CS(v_i, 1)$ is no better than the blind subtraction method that has been dismissed already. In fact, blind addition is even worse because potentially it could mean trying to compute the value of the local planarity of a vertex from a complete subgraph of an order greater than the global planarity of the unadulterated graph, $\varphi(G)$.

External predicates

It is not necessary to exhaust all possible connectivity schemes to find a suitable method for computing the local planarity. The two methods that have been considered, namely blind addition and blind subtraction, add or subtract edges from a graph according to some external predicate. Clearly, using an external and quite unrelated predicate to compute the planarity function is a minefield: Chromatic and planarity theorems are notoriously difficult to prove and so even if we were to find a predicate that gave the correct results on a series of benchmarks, we would probably still be unable to prove that the method works for all graphs.

Up to now we have considered methods for computing the local planarity of a vertex by adding or subtracting edges according to some arbitrary predicate. These methods have had to be dismissed because the predicates they use do not behave in the same way as the global planarity function $\varphi(G)$. So why not use the global planarity function itself to find the local planarity?

Contraction methods for finding local planarity

One such introactive method for finding the local planarity that can be guaranteed to be a true measure of how individual vertices contribute to the global planarity of a graph involves contracting a graph onto a subgraph S consisting of a vertex v_i and those vertices adjacent to v_i, namely $\{v_i \cup CS(v_i, 1)\}$, in such a way as to maximise $\varphi(S)$. Only by *contracting* G onto $\{v_i \cup CS(v_i, 1)\}$ can we be sure that the planarity of the resulting subgraph accurately represents the contribution of v_i to $\varphi(G)$.

Using the method of contracting G, we may define *local planarity* of a vertex formally as follows:

Definition 3: The *local planarity* of a vertex $v_i \in V$ in the graph $G = (V, E)$, denoted by $\varphi'(v_i)$, is defined as the maximum value of $\varphi(G)$ that can be produced by iteratively contracting edges incident on a vertex in the set $\{ v_i \cup CS(v_i, 1) \}$.

Properties of $\varphi'(v_i)$

To illustrate how $\varphi'(v_i)$ behaves, consider some examples.

For the first example, consider a graph G'' comprising a square array of vertices such that $\Delta(G'') = 4, \delta(G'') = 2$ and $\varphi(G'') = 3$. An array is highly regular, so we would expect each vertex to contribute the same amount to the global planarity $\varphi(G'')$. If for each vertex we contract G'' onto the subgraph formed by $V = \{v_i \cup CS(v_i, 1)\}$ and measure the planarity, we find that it is equal to 3 for every such subgraph in G''. Hence all the vertices in G'' have a local planarity of 3.

Figure 2-5 demonstrates how $\varphi'(v_i)$ behaves by giving the φ' values of several other graphs for each vertex alongside the $\varphi(G)$ and $\chi(G)$ values for comparison. It can be seen from the examples that $\varphi'(v_i)$ allows the components of a graph with similar planarity to be identified.

Time complexity of $\varphi'(v_i)$

To derive the upper bounds on the time needed to compute $\varphi'(v_i)$, consider a vertex $v_i \in V$ in the graph $G = (V, E)$ under the following proposition:

Proposition 5: The function $\varphi'(v_i)$ is NP-Complete in time.

Proof: The function $\varphi'(v_i)$ is defined as the biggest value of $\varphi(G')$ that can be obtained by contracting G onto a subgraph G'. Thus to prove that $\varphi'(v_i)$ is NP-Complete, it is necessary to prove four things:

1. That the upper time bound of the function $\varphi(G)$ is no worse than NP-Complete.

2. That $\varphi'(G)$ is not in P, the set of polynomial time functions.

3. That the vertices of G' can be found in polynomial time.

4. That the contraction of a graph to a maximal subgraph, for which there exists a polynomial satisfyability predicate, is NP-Complete.

Consider these problems one at a time.

Part 1:
The upper time bound on the function $\varphi(G)$ has been shown to be NP-Complete already. *QED*

Part 2:
From the definition of $\varphi'(G)$, its computation necessarily involves computing $\varphi(G)$ – an NP-Complete function. *QED*

Part 3:
The vertices of G' are simply the union of v_i, which is given, and those vertices adjacent to v_i, which can be found in linear time. Therefore, if the upper bound time complexity of the function $\varphi'(G)$ is worse than linear, then the problem of finding G' is not the reason for it. *QED*

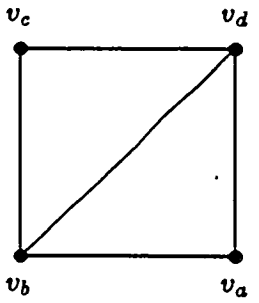
Part 4:
We mentioned earlier that finding the largest bipartite subgraph by contraction from some parent graph is NP-Complete (Proved by Statman). It is easy to write a polynomial time predicate to test whether a particular graph is bipartite. Yet the bipartite contraction problem is NP-Complete, even though the test for a bipartite graph is has a polynomial time bound. So all maximal contractions must be NP-Complete, unless the function being maximised is itself worse than NP-Complete. *QED*

Once parts 1 through 3 have been proved true, the proof of the initial proposition is a direct corollary of part 4. □

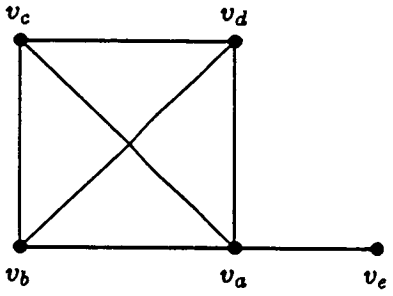
As a practical note, finding the value of $\varphi'(v_i)$ when v_i is a vertex in the graph G^n, can take an amount of time proportional to C^{2n}, where C is a constant.

Polynomial Approximations to $\varphi'(v_i)$

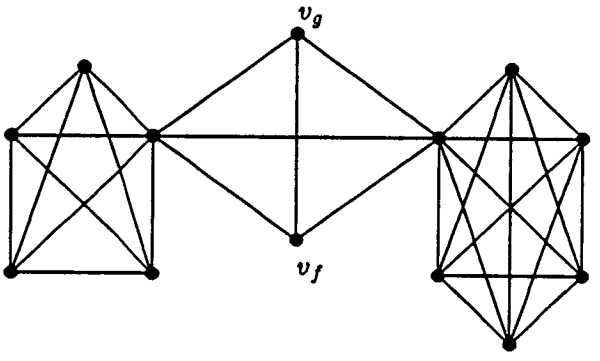
A computer program for calculating the value of $\varphi(G)$ could require an amount of time that tends towards the exponential of the size of the problem.



$\varphi(G) = 3$
$\varphi'(v) = 3 \; \forall \; v$
$\chi(G) = 3$



$\varphi(G) = 4$
$\varphi'(v) = 4$ for $v \in \{v_a, v_b, v_c, v_d\}$
$\varphi'(v_e) = 2$
$\chi(G) = 4$



$\varphi(G) = 6$
$\chi(G) = 6$
$\varphi'(v) = 4 \; \forall \; v \in \{v_f, v_g\}$
$\varphi'(v) = 5 \; \forall \; v \in K^5$ subgraph
$\varphi'(v) = 6 \; \forall \; v \in K^6$ subgraph

Figure 2-5: $\varphi'(v_i)$,$\varphi(G)$ and $\chi(G)$ values for several graphs

The time complexity of $\varphi'(v_i)$ is even worse. To compute $\varphi'(v_i)$ for every vertex in a graph G, would take an amount of time that tends to

$$|G| * 2^{2*|G|}$$

So if the computer program can assess a possible contraction in 10^{-6} seconds, a speed beyond that available from the fastest machines in existence today, then to compute $\varphi'(v_i)$ for each vertex in a graph with 100 vertices might take $100 * 2^{2*100}$ seconds: A *very* long time.

In the context of VLSI, we may need to calculate the local planarity of graphs with tens of thousands of vertices. Obviously, a a way of overcoming the time complexity problems must be found before going any further.

Direct approximations to $\varphi'(v_i)$

By a *direct* approximation, I mean a procedure for solving the problem that has been defined. Later on we shall look at *indirect* methods that instead of solving the problem that is defined, make an approximation by mapping it into a different problem that can be solved more easily.

Many algorithms have been published for finding approximate solutions to NP-Complete problems using direct methods. Some of these achieve results very close to the optimal solution. Probably, the best examples of this are the graph colouring heuristics: eight different colouring algorithms compared by Brelaz were all within a few percent of $\chi(G)$ [Brelaz 79].

Researchers who have experimented with direct approximations of $\chi(G)$ have found that their errors tend to be largest when colouring sparse graphs [Wood 69]. We have proved that a close relationship exists between the global planarity function $\varphi(G)$ and the graph colouring function $\chi(G)$, so we should expect that a direct heuristic for computing the local planarity function φ' would also be least accurate on sparse graphs.

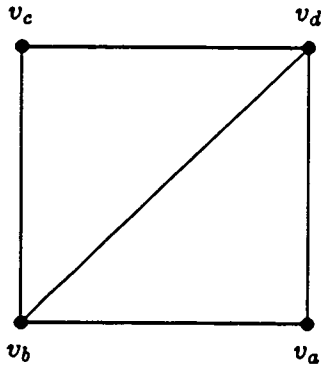
Unfortunately, VLSI connectivity graphs are often very sparse. So using direct methods to find φ' could invoke serious problems. The evidence is clear: Using direct heuristic techniques is likely to result in a produce an unreliable planarity function.

Indirect approximations to $\varphi'(v_i)$

Rather than attempt to produce a direct approximation to φ', we could try to find a polynomial time function that although quite different from $\varphi'(v_i)$, would behave similarly enough to φ' for the job in hand, namely, VLSI floorplanning.

The local planarity function φ' returns information on how tightly interconnected is the subgraph around a particular vertex. Our needs can be restated as wanting to know about how strongly the connectivity set is connected within itself, or in other words, how many vertices in the connectivity set are mutually connected.

An infinite number of functions that return information on the mutual connectivity of a subgraph can be constructed. Out of the functions that have been tried, the one that seems to work best is defined as follows.



$$CS(v_a,1) = \{v_b, v_d\}$$
$$CS(v_b,2) = \{v_a, v_c, v_d\}$$
$$CS(v_d,2) = \{v_a, v_b, v_c\}$$
$$CS(v_d,1) = \{v_a, v_c\}$$
$$MC(v_a,1) = 2$$
$$MC(v_a,2) = 3$$
$$MC(v_b,2) = 3$$

Figure 2-6: Example graph and MC values

Definition 4: The mth degree *Mutual Connectivity* of a vertex v_i, denoted by $MC(v_i,m)$, is the maximum number of vertices in the set $CS(v_i,1) \cup \{v_i\} \cup \{CS(v_j,1) \cup \{v_i\}\}$, where $v_j \in CS(v_i,m)$. The degree m must be ordinal.

To illustrate how the MC function works, figure 2-6 gives the CS sets and MC values for an example graph.

The MC function has a number of interesting properties. $MC(v_i,0) = d(v_i)$, $MC(v_i,1)$ is a similarity index and $MC(v_i,2)$ is an approximate measure of the local planarity.

An algorithm for computing $CS(v_i,m)$ is given in [Nixon 84].

Once $CS(v_i,m)$ and $CS(v_i,1)$ have been found, $MC(v_i,m)$ can be computed using simple set operations.

Time Complexity of the MC function

It has already been shown that $CS(v_i,m)$ is NP-Complete. As $MC(v_i,m)$ is no more than the union of several CS sets, it too must be NP-Complete. But if $MC(v_i,m)$ is NP-Complete, why use the MC function instead of the local planarity function φ'?

It was proved earlier that $CS(v_i,m)$ can be computed in polynomial time if m is fixed and $m < |G|$. This means that if for small values of m the time complexity is of the MC function is a low order polynomial function of n – the number of vertices.

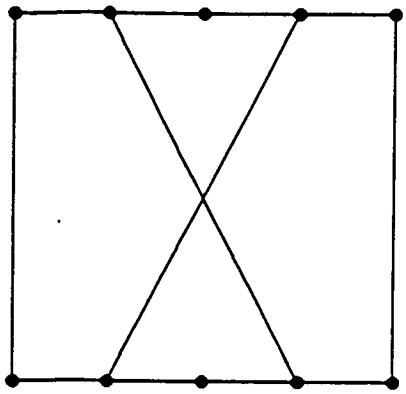


Figure 2-7: Example of a graph where $MC(v,2) \neq \varphi'(v)$ for all v.

Later on we will be concerned with $MC(v_i,2)$, for which the time complexity is of the form $O(p.l)$, where l is the number of vertices adjacent to v_i. For a practical VLSI design, l is usually a small number between 2 and 20, though for the theoretical worst case of a complete graph, l is equal to $p-1$. This result for the worst case behaviour of $MC(v_i,2)$ agrees with polynomial complexity result of Nixon [Nixon 84].

In short, whilst the MC function is NP-Complete in the general case, it is possible to compute $MC(v_i,2)$ in virtually linear time and in $O(p^2)$ time in the worst case. The $MC(v_i,2)$ values are sufficient for our purposes, so the underlying NP time complexity of $MC(v_i,m)$ causes no trouble.

Error distribution of the MC function

For the MC function, whenever $MC(v_i,2) \neq \varphi'(v_i)$ the MC function must be considered to be in error. Errors must occur: Unless the MC function possesses some special magic, it cannot work both in polynomial time and be isomorphic with φ'.

Given that there will be some graphs with vertices for which $MC(v_i,2) \neq \varphi'(v_i)$, it is important to find out what proportion of simple graphs contain these *error vertices*, the magnitude of the errors and, hopefully, characterise the graphs in which the errors occur. Only when all these things have been done, assuming the MC function has been designed properly, can it be used in place of φ', provided the system that uses the MC function is carefully designed so as to be insensitive to the error vertices.

The value of $MC(v_i,2)$ must be computed by searching for vertices reachable

by a path of length 2 precisely, starting with each of the vertices adjacent to v_i. This is the same as looking for paths of length 3 from v_i, or circuits of length less than 5. Therefore, the MC function may give a completely inaccurate measure of the planarity for graphs in which either there are either no circuits or in which the shortest circuit is of length 5 or more. Let us call this set of graphs, *error graphs.*

Figure 2-7 gives an example of an error graph. It was built by adding vertices to K^4.

An Improved Heuristic

The set of MC error graphs is unbounded, hence infinite, and the set includes all very sparse graphs. As VLSI connectivity graphs tend to be sparse, this means that the basic MC function is virtually useless for VLSI. However, an important observation is that the MC function is always correct for very dense graphs, including complete graphs and graphs that are almost complete. If the MC function is used to test for the maximum degree of planarity of a contracted subgraph then the function returns values close to the true local planarity. So, if a graph could be condensed into some maximally connected form before applying the MC function, then we would get a far better heuristic. Therefore, the problem now becomes one of finding the maximal planar contraction, that is:

$$max(MC(v_i, 2) : G \rhd (CS(v_i) \cup \{v_i\}, E'))$$

There are a great many heuristics for contracting a graph subject to a cost function: In this case the cost function is $MC(v_i, 2)$. One such algorithm is given in figure 2-8. We shall refer to the estimates of local planarity produced by this particular algorithm as the *Planarity Estimate* (PE) of a vertex. From this point on, the PE will be used whenever a practical method for measuring the local planarity is needed. The PE function can be consider a improved version of the MC function.

Summary

In this Chapter we have:

1. Introduced the concept of *degrees* of planarity.

2. Developed the function $\varphi(G)$ for measuring the degree of planarity of a graph.

3. Proved a relationship between the planarity function $\varphi(G)$ and the vertex colouring function $\chi(G)$.

4. Developed the local planarity function $\varphi'(v_i)$ for measuring the contribution a vertex makes to the planarity of its parent graph.

5. Developed a practical method for computing the local planarity function $\varphi'(v_i)$.

In the next Chapter, these mathematical tools are applied to the problem of forming clusters in graph in such a way that each cluster can be floorplanned easily.

```
integerfunction planarity estimate(vertex: v; graph: G)
    { Returns the local planarity of v – a vertex in the graph G }

    listfunction condense(list of vertices: vlist)
        var list of vertices: list of contenders
            vertex: vi
        list of contenders := empty list
        vi := vlist
        while vi ≠ empty list cycle
            append CS(vi, 1) to list of contenders
            vi := next vertex in list
        repeat
        vi := list of contenders
        while vi ≠ empty list cycle
            if vi is in vlist then
                remove vi from list of contenders
            fi
            vi := next vertex in list
        repeat
        vi := vlist
        while vi ≠ empty list cycle
            contract all vertices in list of contenders adjacent to vi onto vi
            vi := next vertex in list
        repeat
    end

    var list of vertices: list of start vertices

    { contract G to a dense subgraph }
    list of start vertices := CS(v, 1)
    append v onto list of start vertices
    while there exists a vertex in the list of start vertices
        that is connected to a vertex not in the list cycle
        sort the vertices in the list of start vertices
            by ascending degree
        condense(start vertices)
    repeat
    { now that G is contracted, find MC value }
    result := MC(v, 2)
end { of planarity estimate function }
```

Figure 2-8: Heuristic for estimating the local planarity of a vertex

Chapter 3

Idiomatic Floorplanning

This chapter presents a method of determining the relative position and aspect ratio of functional blocks in a microcircuit in such a way that the total area and communication costs are minimised. This area planning process is known as *floorplanning*, after the age old problem of determining the shape and position of rooms within a building so that the rooms are sufficient for the utility they accommodate whilst at the same time minimising the corridor space, the communication costs.

The quality of a floorplan

The quality of a floorplan is the single most important factor affecting the technical viability of a design. If the floorplan uses silicon inefficiently, the yield will be low. If the aspect ratio deviates too far from 1:1 then the chips cannot be sawn from the wafer and mounted reliably. If a new chip is intended to be a functional replacement for another, then it may be important that a particular pad position is adopted. The *quality* of a floorplan is a composite goodness factor formed by the amalgamation of all these others. If the individual quality factors are identified, then one floorplan can be considered to be *better* than another and so it becomes possible to judge the comparative attributes of floorplans.

It is common practice to make qualitative judgements of a floorplan according to the following criteria:

1. **Wire Length**
   A layout comprises function blocks and wires. The area taken by the function blocks is fixed by the application, but it is the way the floorplanner lays out these function blocks that controls how much area is taken up by wiring. As well as increasing the chip area, excess wiring degrades the temporal performance. Thus, the total wire length and the ratio of wiring area to function area are probably the most important criteria by which the quality of a floorplan can be measured.

2. **Total Area**
   It should go without saying that there must be as little as possible white space between function blocks.

3. **Aspect Ratio**
   The aspect ratio of a chip should be close to 1:1. The floorplan becomes worthless if the aspect ratio increases much beyond 2:1.

4. **Pad Order**
   A cost might be associated with moving pads away from their predetermined position, perhaps to maintain plug compatability with some earlier chip. Also, for chips with many pads it is important to place pads evenly around the periphery of the chip to keep the bonding wires as short as possible. Long bonding wires reduce the reliability of a chip because when the chip is stressed, perhaps by vibration, the bonding wires move and may touch each other or, under high levels of stress, the wires may break through metal fatigue or the bond could fail.

5. **Specialised Costs**
   The costs listed above apply to all chips, but a layout technician may trade off costs of a more individual nature, to capture things like, "the amount of metal used for wires must be maximised", or even more specialised, "function block ABC must be as close as possible to PQR and PQR must be close to both XYZ and pad 2 and all of that lot must be as far away from signals A, B and C as possible because ABC, PQR and XYZ carry some low level analogue lines that might couple with the digital signals A, B and C which sink a lot of current".

A good floorplan is one which finds a reasonable trade off between these cost functions.

Mead and Conway observe that for chips of an LSI or VLSI scale, the wiring costs dominate [Mead 80]. So the methods presented in this thesis concentrate on solving the wireability part of the problems.

Background

There are three methods for automatic floorplanning currently in vogue:

- Box packaging
- Target Architectures

- Graphical Methods

The box packaging technique exemplified by Preas [Preas 78] and Szepieniec [Szepieniec 80], tries to pack the box outlines of function blocks into the smallest space. The resulting placement is then perturbed and shuffled to take into account the connectivity between the blocks. These box packaging methods tend to produce low quality floorplans because they are directed primarily by the outlines of the function blocks rather than by their connectivity.

The target architecture approach is used in the first generation silicon compilers, such as Bristle Blocks [Johannsen 79] and FIRST [Bergmann 83]. The method employs a predetermined architectural style to lay out all designs. Consequently, these floorplanners can be optimised by incorporating expert knowledge that may not be applicable to other floorplanning situations, but this restricts their application by making them highly specialised.

The graphical methods come in many different forms, the best known being planar graph dualisation, as typified by Heller [Heller 79], [Heller 82], [Heller 82b]. Graphical methods can be considered to be the opposite of the box packing methods in that whereas the box methods are directed first by the box outlines and second by the connectivity, the graphical methods try first to minimise the interconnect length then look at the outlines afterwards. Consequently, the planar dualisation methods produce floorplans of the highest quality, but as their name suggests, they are limited to planar connectivity graphs. This restriction has serious implications in VLSI because many architectural idioms can produce aplanar connectivity graphs. For example, cell transparency is an extremely important layout technique that is used widely to lay out aplanar graphs such as datapaths and cell arrays. The planar restriction of the graphical methods is compounded by their exponential time complexity, which greatly limits the size of designs which can be handled [Brebner 83], [Buchanan 83], [Nixon 85].

The method presented here, that of idiomatic floorplanning, is based on two precepts:

1. There exist a finite number of architectural patterns, or idioms, which can be laid out by specialised floorplanners that have been optimised on an individual basis.

2. That these architectural idioms are recognisably distinct.

The first of these precepts is based on the observation that whilst the performance of the floorplanners described previously is inadequate for the purposes

of floorplanning the great mass of general purpose chips, they are effective at laying out specific design families. The second precept implies that it is possible to decompose a general graph into clusters that correspond to the classes of designs that can be laid out. Given a set of specialised floorplanners, the problem becomes one of assigning clusters to floorplanners and then assembling the complete floorplan from a hierarchical set of floorplanned fragments.

These precepts do have a sound theoretical basis. Leiserson has proved that if there exists a good separator theorem for a class of graphs, then this is a sufficient condition for there to be a good VLSI layout of that class [Leiserson 82]. Unfortunately, Leiserson also proved that a perfect separator theorem is NP-Complete. It is because of this result that a lot of time was spent considering how to build a good heuristic for the planarity function developed in the last chapter: we intend to use the local planarity function $\varphi'(v_i)$ as a graph separator function.

An outline of the method

The idiomatic approach to floorplanning can be divided into three phases:

1. Clustering
2. Classification
3. Placement

The three phases are interdependent. The problems of clustering are interwoven with those of classification and the classification system must reflect the capabilities of the specialised floorplanners that perform the placement.

The clustering phase is a preprocessing stage intended to simplify the job of classification by introducing additional levels of hierarchy to an already structured design.

Classification is the process of identifying which class a particular cluster belongs to. It is assumed that disjoint clusters are separated during clustering.

Placement is the mechanical process of laying out the clusters. This stage is made up of a number of individual "expert" floorplanners, each dealing with a particular class.

Basic Concepts

Given a general graph, the first problem is to extract clusters which match the capabilities of the available specialist floorplanners. To do this, it is necessary to understand why a designer chooses to use architectural idiom 1 to lay out a problem instead of idiom 2 or idiom 3.

Take an example. Imagine that only five architectural idioms are known:

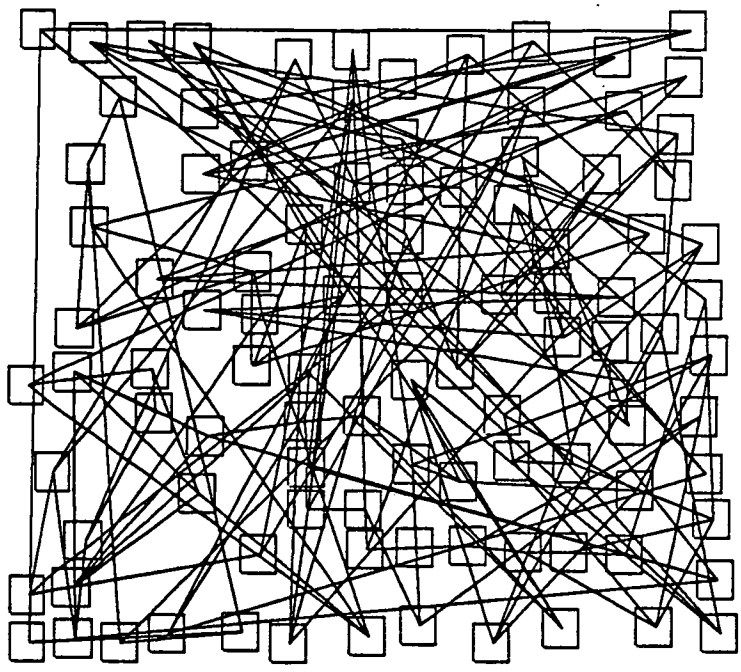
1. Arrays
2. H-trees
3. Transparent cells that produce datapath-like layouts
4. Wiring channels for supercell-like layouts
5. Planar remnants

Choose examples of graphs that are suited to laying out in each of the idioms listed. What makes the graphs different? What makes the graphs less suitable for being laid out using one of the other idioms?

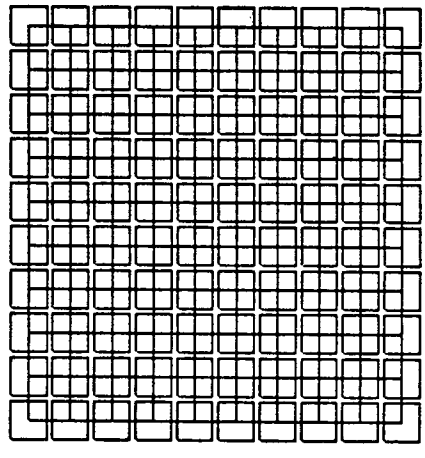
At first there may seem to be no similarity between the graphs at all. Even something as simple as an array may not look anything remotely like an array when it is drawn. Figure 3-1 shows a graph alongside its layout produced by an idiomatic floorplanner. Once laid out the graph is obviously an array, but before layout, it could be anything.

One experiences difficulty in recognising the array before it was floorplanned because the information normally associated with an array, namely regular connections, had been deliberately scrambled. Once the regularity information has been hidden, all that is left is the connectivity information.

This array example also highlights the fact that people perceive problems in a quite different way to machines. Unless the regularity information is made explicit, all an automated floorplanner has to go on is the connectivity. The scambled array demonstrated that whilst connectivity information is a necessary prequisite to floorplanning, it is scarcely sufficient: extra information is needed. It is necessary to identify what this extra information is, and either find a way of presenting it explicitly to a floorlanner, or find a method of synthesising it automatically from the connectivity.



Before Floorplanning



After Floorplanning

Figure 3-1: An array: before and after floorplanning

If the most important single attribute of a good floorplan is the way connectivity is handled, then a good place to start might be to characterise the various specialised floorplanners according to how well they handle different degrees of interconnectedness. The planarity metric developed in the previous chapter provides the tool with which to do this. One such classification is tabulated below.

Floor Idiom		Planarity	Regularity
Arrays	Low order	3 or 4	Very High
	High order	5 or more	Very High
H-trees		2 or 3	High
Datapath-like layout		4 or more	High
Supercell-like layout		5 or more	Very Low
Planar dualisation		4 or less	Low

Figure 3-2: Characterising floorplan idioms

Notice that the floorplanning idioms can be uniquely identified by the planarity and the regularity values. The regularity is a measure of the distribution of the local planarity values and the orders of the vertices in the connectivity graph.

The use of local planarity to model layout idioms is best understood by considering the case where the degree of planarity is equal to the chromatic number. The well known principle in cartography called the 4CT states that it is possible to colour a map using no more than 4 colours so that no neighbouring countries have the same colour. If one allows aplanar maps then more colours are needed, this is Heawood's Theorem mentioned in the last chapter. Given a colouring, it is possible to measure the rate of change of the colouring across the graph as well as the absolute number of colours that are used. The number of colours around a vertex is a measure of the local planarity and the rate of change of the local planarity becomes a measure of the regularity of a graph.

One possible classification system for graphs that have been clustered into groups of like planarity and regularity is given in figure 3-3.

The full classification system will be described later on, but notice from the figure that it is a two phase process. The outermost filter separates regular graphs from irregular graphs and planar graphs from aplanar. The innermost filter uses a few implementation specific rules to determine exactly which floorplanner should be chosen.

```
proc classify(cluster c)
    if c is regular then
        if c is planar then
            if all vertices in c are of degree 2 and
                    H-tree test is positive then
                floorplan fragment as H-tree
            elseif array test is positive
                floorplan fragment as array of order Δ(c)
            else
                there is an idiom missing
                give fragment to irregular planar floorplanner
            fi
        else { c is aplanar }
            if array test is positive then
                floorplan fragment as array of order Δ(c)
            elseif transparent cell test is positive
                floorplan fragment as datapath-like layout
            else
                there is an idiom missing
                give fragment to irregular aplanar floorplanner
            fi
        fi
    else { c is irregular }
        if c is aplanar then
            if channel test is positive bf then
                floorplan fragment using wiring channels
            else { c is aplanar remnant }
                floorplan fragment as datapath-like layout
            fi
        else { c is planar }
            if c is in a special case then
                apply specialised tests
            else { c is planar remnant }
                floorplan using planar dual floorplanner
            fi
        fi
    end { of procedure }
```

Figure 3-3: A miniature classification system

Method

An outline of the idiomatic floorplanning method has been given. The remaining sections of this chapter build on this foundation by presenting a detailed description of each of the three stages involved: clustering, classification and placement.

Clustering

As mentioned earlier, clustering is a pre-processing stage intended to improve classification. Clustering introduces additional structure to a hierarchical design so that the classification system can give a simple class label to each cluster. In a very well structured design, clustering is redundant. In a worst-case flat design, clustering attempts to build a design hierarchy that reflects the needs of the classification system.

The clustering system requires three stages to be applied in turn. The stages are:

1. Fragmentation clustering
2. Planarity Clustering
3. Functional Clustering

Fragmentation clustering is simply a way of handling very large unstructured designs. Clusters with more than a given number of vertices, say 1000 vertices, are broken into fragments based on their connectivity. These fragmentation methods are limited in a practical sense because of the complexity of the design process they are trying to imitate, that of design segmentation. Only the most unstructured designs are fragmentation clustered, for structured designs fragmentation clustering is redundant.

Planarity clustering is based on the PE function and is designed to mirror the classification system: it produces clusters that can be recognised easily. This involves two parts. The first part clusters according to the planarity, the second according to the regularity. The end product is a tree of clusters labelled with information about whether or not they are regular and-or planar. A planarity directed clustering algorithm is given in figure 3-4. An algorithm for doing the regularity clustering is much the same but looks at the order of a vertex in addition to the $PE(v_i, 2)$ values.

```
proc planarity cluster(

      integer: n { order of the graph, |G| };
      integer array: v(1..n) { array of vertices } )

      var integer: planarity, i

      planarity := 0
      for i := 1 to n cycle
            v(i) := PE(i,2)
            planarity := max(planarity,v(i))
      repeat
      if planarity > 2 then
      for i := planarity downto 2 cycle
            if there exists a vertex i with PE(i ,2) := i then
                  create surrogate vertex
                  move i under surrogate
                  bring under surrogate all vertices in
                        CS(i ,1) with PE(i ,2) := i
                  if new cluster contains only one vertex then
                        bring under surrogate all vertices in
                              CS(i,2) with PE(i,2) := i
                  fi
                  if new cluster still contains only one vertex then
                        bring under surrogate all vertices in
                              CS(i,1) with PE(i,2) > 4
                  fi
                  if new cluster still contains only one vertex then
                        bring under surrogate all vertices in
                              CS(i,2) with PE(i,2) > 4
                  fi
                  if new cluster is disjoint then
                        bring common members of CS(i,1) under surrogate
                  fi
                  close surrogate
            fi
      repeat
end { of clustering procedure }
```

Figure 3-4: An algorithm for planarity clustering

Functional clustering is a method of introducing implementation-dependent features into the clustering process. Functional clustering relies on an *ad. hoc.* collection of rules to accommodate any further clustering action necessary to produce clusters that particular specialised floorplanners can lay out.

Classification

During the development of an idiomatic floorplanner, it became apparent that there is a distinction in classification between what can be recognised as a distinct type of graph and what can be laid out by the implemented "expert" floorplanners. This led to the idea of a two level classification system. The first level, closely allied to clustering, recognises certain general types of graph. These general types are referred to as *planarity classes.* The second level, that of *implementation classes*, tests for a more selective range of graph types that closely match those that can be laid out by the specialised floorplanners that have been implemented.

The structure of the two level classification system is shown in figure 3-5.

Placement

One of the basic precepts on which idiomatic floorplanning is based, is the need for specialist floorplanners. It is at the placement stage of the floorplanning process that these "expert" floorplanners are called into use.

Of the three processes involved in idiomatic floorplanning, placement is the most mechanistic and poses the fewest theoretical problems. One of the attractions of the idiomatic approach is that the individual floorplanners can be optimised to lay out their particular class well. However one problem which does arise is the need for some of the floorplanners to cope with clusters which have not been optimally classified, that is clusters which would best be laid out using an idiom that has not been implemented. This problem requires some of the expert floorplanners to be robust enough to cope with such clusters.

It was envisaged from the outset that a graph dualisation process such as that of Heller's [Heller 82] would be used as a catch-all for planar graphs. The aplanar graphs use a datapath generator to catch aplanar remnants.

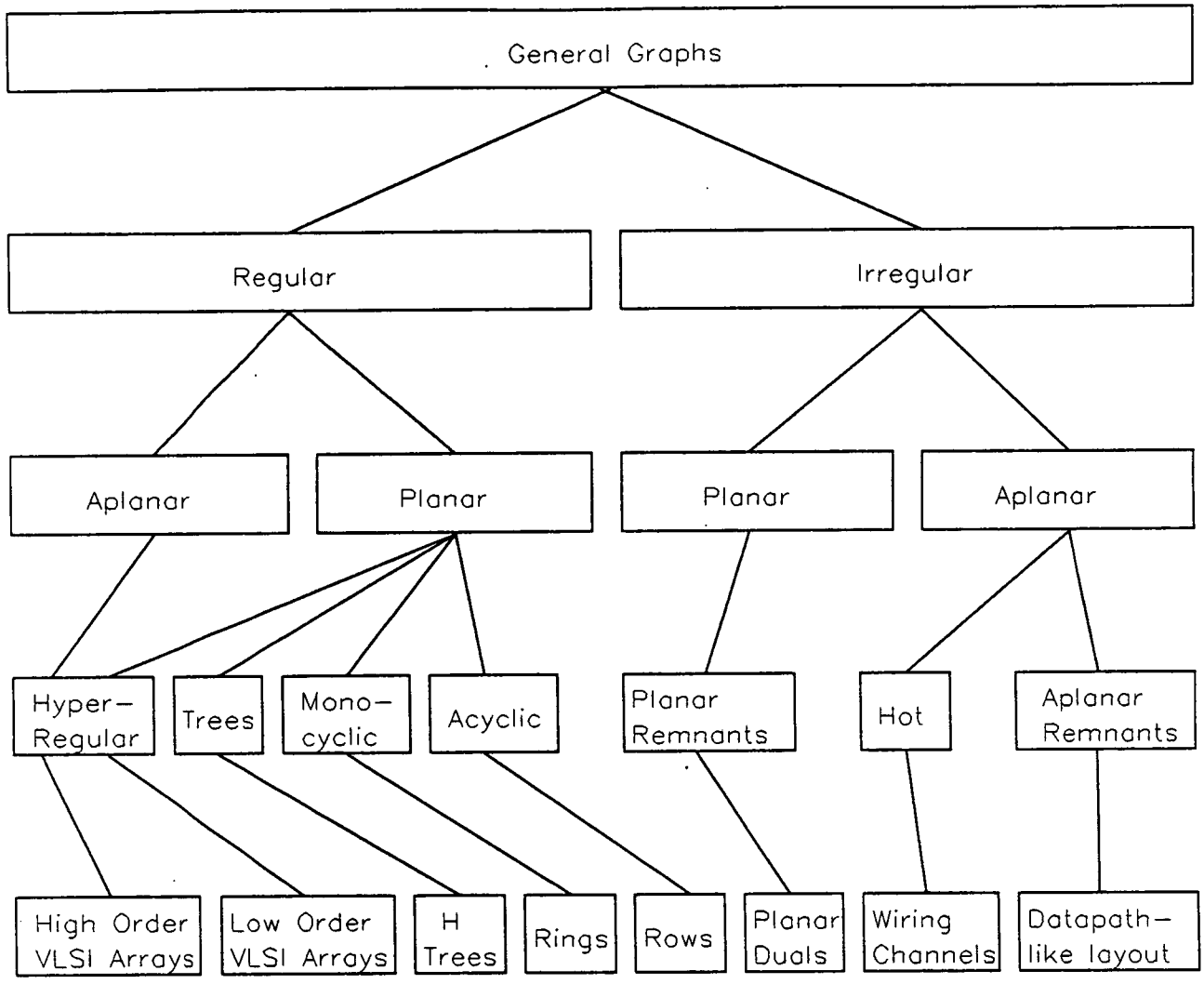


Figure 3-5: Structure of two level classification system

Implementation classes of floorplan

This section describes some implementation classes and shows how they relate to the planarity classes considered already.

The implementation classes identified so far include:

1. **Hyper-regular graphs**
   These are implemented as VLSI arrays. Hyper-regular graphs are in two planarity classes: that of regular planar, or regular aplanar. The implementation class test consists of recursively removing the strongest connected vertices in a cluster and then applying PE rules to verify that the remaining graph belongs to a class of rings, either as a simple ring or a complex ring.

   A ring is a graph G which contains a hamiltonian circuit, but in which there are no circuits of length between 3 and $|G|$. The value 3 is the degree of planarity of the vertices in the corners of the array.

   By way of a practical aside, the class of arrays has proved the hardest to classify. It is easy to identify a 1024 by 1024 matrix of cells as an array, but what about a similar matrix with 6% of the cells missing? What if 21% of the cells are missing, is the sparse matrix still an array? What if all 21% of the missing cells all come from the center of the array?

2. **Trees**
   These are planar graphs that can be implemented efficiently as VLSI H-Trees. The implementation test for trees comes after that for arrays so it only has to verify that the cluster is acyclic with $\Delta(G) = 3$.

3. **Regular Monocyclic Planar Graphs**
   These are implemented as VLSI rings. A simple ring is a cluster in which $\Delta(G) = \delta(G) = 2$.

4. **Regular Biplanar Graphs**
   It is possible to enumerate all biplanar graph types (that is, graphs for which $\varphi(G) = 2$). When this enumeration is carried out, it becomes apparent that a biplanar graph can be contracted either to a single star or a row. A star is a graph in which $\Delta(G) = |G| - 1, \delta(G) = 2$ and there are $|G| - 1$ vertices of order 2. A row is a graph in which $\Delta(G) = 2$ and $\delta(G) = 1$.

All graphs is this group can be laid out easily on a grid, using a folded linear grid in the case of rows and a compacted matrix in the case of single stars.

5. **Irregular, Hot Aplanar Graphs**
   These are implemented using wiring channels. The "hot" term refers to the distribution of local planarity values, which is very broad, with many vertices with low values though with a fair scattering of high value vertices. The vertices with high values of local planarity are *hot-spots* of highly interconnected sub-clusters.

   In practice, identification of this class using connectivity based tests must be reinforced by functional tests.

   One possible connectivity test is to count the number of single stars indirectly connected to another star, that is, where a vertex of degree $\Delta(G)$ is a star centre and contains in its $CS(v_i, 2)$ at least one other star centre.

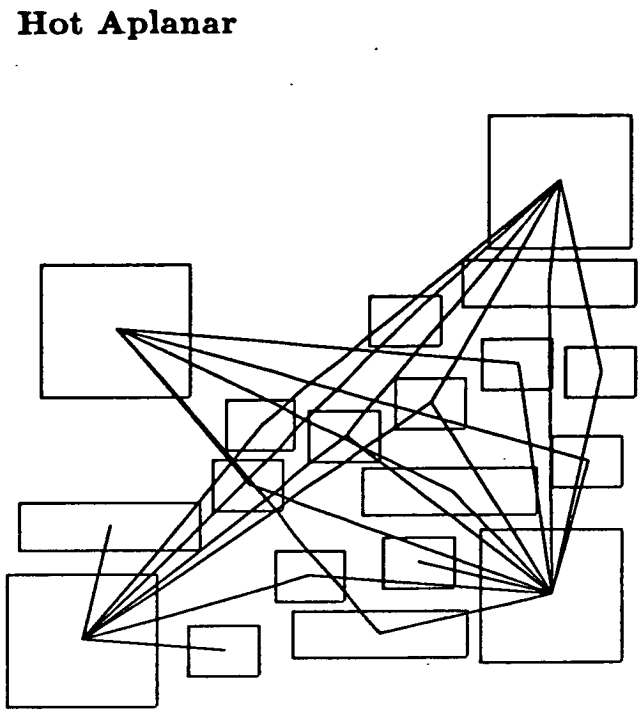
   The functional tests are system dependent and so will not be described.

6. **Aplanar Remnants**
   These are implemented as transparent cells using a datapath layout. This acts as a catch all, so it need not be positively tested for: all aplanar graphs that are not anything else are aplanar remnants.

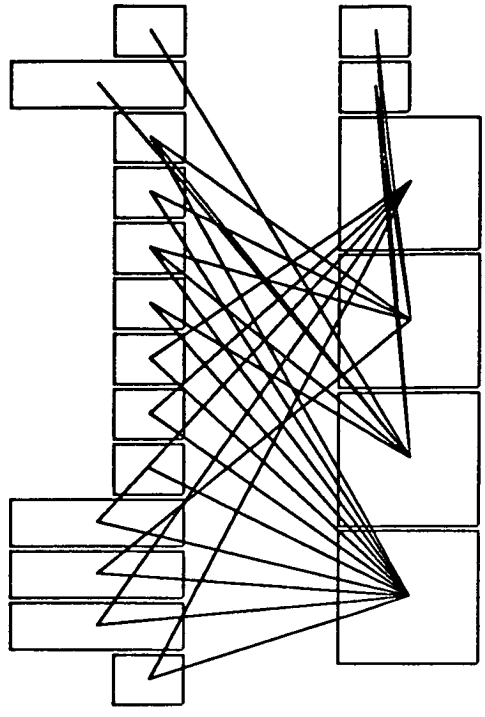
7. **Planar Remnants**
   Planar Remnants are implemented using a planar dualisation method. Planar remnants act as the catch all for planar clusters and so again, they need not be positively tested for.

Figure 3-6 contains some examples of classification and layout for the classes of Aplanar Remnants, and Irregular, Hot Aplanar graphs.

Hot Aplanar

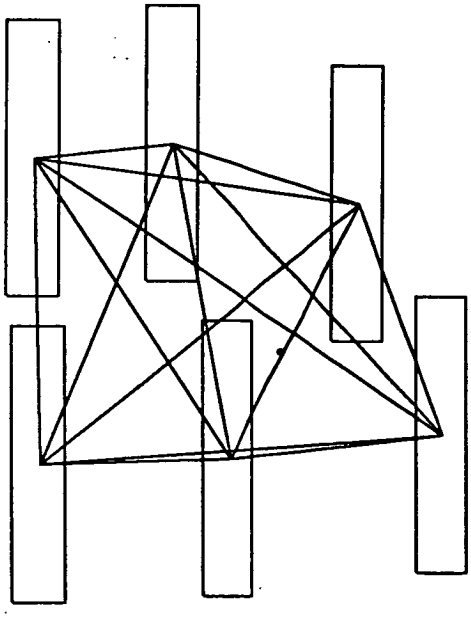


Before

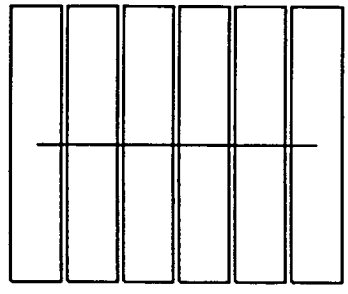


After

Datapath



Before



After

Figure 3-6: Examples of two implementation classes

Summary

Existing floorplanning algorithms can produce highly efficient layouts for restricted families of design types, but they fail to produce adequate layouts for the generalised floorplanning problem. The approach to generalised floorplanning that is presented here uses a simple function based on the connectedness of vertices within a cluster to develop rules for assigning the cluster into classes. The cluster can then be floorplanned using a floorplanner optimised for that class. The strength of this new approach lies in its ability to apply many different floorplanning techniques, each one being applied only to those situations where it is well suited.

The importance of using the PE function is stressed. Without the function, the idiomatic approach would involve many thousands of rules and the resulting system would behave unpredictably. The PE function makes the idiomatic approach viable and the classification that is used makes the system effective.

Chapter 4

Hardware Description Languages

Existing Hardware Description Languages (HDLs) do not satisfy adequately the demands made upon them by silicon compilers. In fact, it seems to be these inadequacies that cause most of the problems identified in the introduction to this thesis. To solve the problems, it is necessary to develop a suitable HDL. To do this, we must first identify *what* demands a silicon compiler makes on a HDL and then decide *how* to address those requirements in the design of a new language.

A model of HDLs

Dealing with the problem of *what* to describe, the job would be much easier if there was some formal model with which the various features and characteristics of a language could be discussed in a precise context.

Numerous attempts at producing such a model have been made. All of the three models we shall go on to consider make a distinction between *domains of description*, such as a behavioural domain or a geometric domain, and the *levels of abstraction*. This distinction is a useful one because it separates the specification of what is being described from the functionally orthogonal concept of how a language handles degrees of detail.

The earliest attempt at modelling HDLs relevant to this discussion is undoubtedly Gajski and Kuhn's Y chart [Gajski 83]. The Y Chart has three arms, the first arm represents functional information, the second structural information and the third geometrical (physical) information. When the arms are arranged axially, increases in the amount of detail in a design, the levels of abstraction, are considered to run along each arm towards the discontinuity where the arms meet.

Napp and Parker refine Gajski's work by adding a fourth domain, causing the three others to be reorganised, and adding a hierarchical aspect to allow the stepwise refinement of a design through various intermediate descriptive domains to be expressed, all the way from some high level starting point to a manufactured

object [Napp 84]. The four domains in the resulting tree structure are labelled the *data flow behaviour subspace*, the *structural subspace*, the *physical subspace* and the *timing and control subspace.*

Walker and Thomas set out to combine the best features of both Gajski's and Napp's models into their own Directed Acyclic Graph (DAG) view of hardware descriptions [Walker 85]. Walker and Thomas label their three domains the *behavioural*, the *structural* and the *physical.*

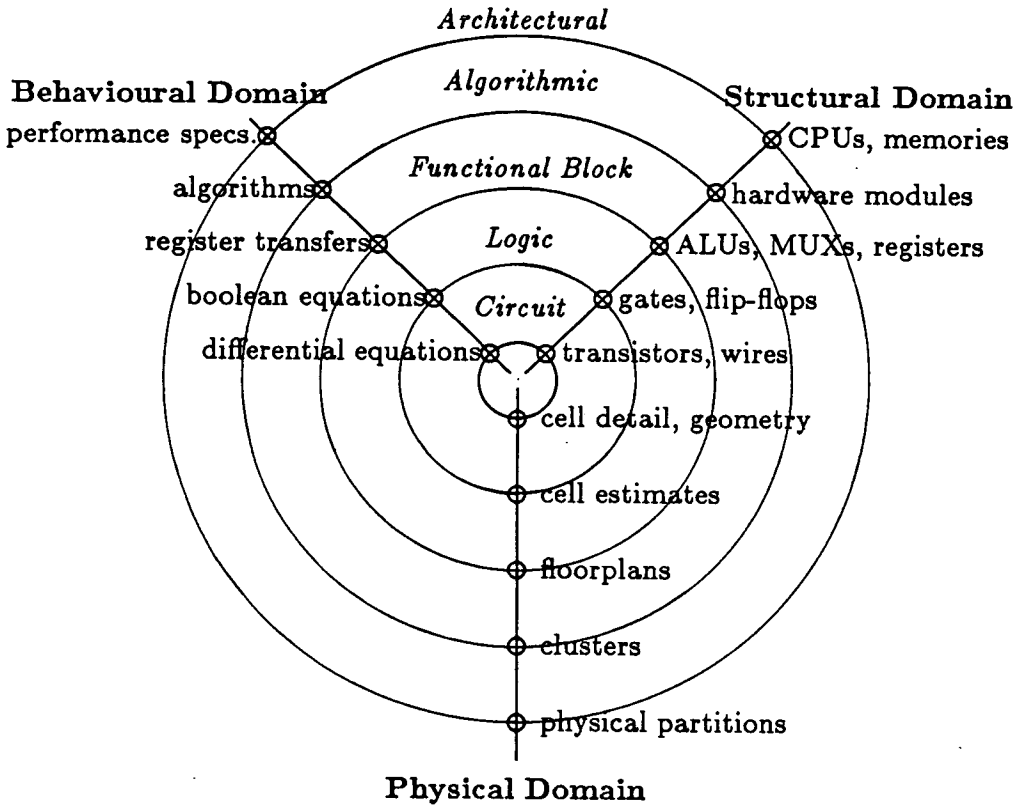


Figure 4-1: An axial view of Walker's subspaces

The Walker and Thomas model is the most useful for our purposes. Each domain in their DAG may be decomposed into a more detailed composite domain without implying any isomorphic decomposition anywhere else. This ability to represent independent decomposition provides a framework for hierarchical synthesis and allows the model to be extended to incorporate other domains of a specification, such as the functional, the temporal, the topological, the topographical or the geometrical domains.

Another feature of the Walker and Thomas model is that the spaces between each domain are identified with particular levels of design abstraction. This can

	Behavioural Domain	Structural Domain	Physical Domain
Architectural level	performance specs	CPUs memories switches controllers buses	physical partitions
Algorithmic level	algorithms (datastructure manipulations)	hardware blocks datastructures	clusters
Functional block level	operations register transfers state sequencing	ALUs blocks datastructures	clusters
Logic block level	boolean equations	gates flip-flops PLAs	cell estimates
Circuit level	differential equations	transistors capacitors resistors	cell estimates cell details

Figure 4-2: A tabular view of Walker's subspaces

be seen most easily from a Y graph representation of the Walker DAG, shown in figure 4-1. The DAG can also be translated into a table, as shown in figure 4-2.

The Walker model: Independence between domains of description and levels of abstraction

In designing their model, Walker and Thomas make a key observation: Levels of abstraction have nothing whatsoever to do with the domains of description. It means that the domain a specification happens to be in has little bearing on its succinctness. For example, it is possible to describe a system at the physical level in some language more succinctly and more clearly than at the functional level in some other language. Also, it is possible to describe a system at the same level of detail in each of several different domains. Walker and Thomas illustrates this point by describing a specification at the Function Block level of abstraction in each of the three primary domains, as follows:

1. **Behavioural Domain**
   The behavioural domain specifies what a circuit does (the function) and the manner in which it does it (how fast, in what order *etc.*).

   The behaviour of a system at the Functional Block Level of abstraction might be represented as a set of registers and abstract function blocks, along with a timed dataflow graph.

2. **Structural Domain**
   The structural domain specifies the logical organisation of a system.

   The structure of a system at the Function Block Level might still use abstract function blocks, but all control would be explicit and if one system was controlled by PLAs and another by microcoded ROM, then their structural specifications would differ.

3. **Physical Domain**
   The physical domain specifies how a structure is to be implemented.

   If two systems are described at the Functional Block level of abstraction, their physical descriptions would differ if one was implemented in ECL and the other in TTL, or even if they are both in TTL, their descriptions would differ if they are partitioned onto circuit cards differently.

The semantics recovery tarpit

The work of Walker and Thomas goes against much of the research on silicon compilation done before 1985, when their model was first published. Until then, several groups had been trying to compile silicon structures from computer programs written in systems programming languages. Other groups had proposed HDLs that could describe only the structural and physical aspects of a specification. The fact that neither of these approaches are likely to produce a useful silicon compiler is apparent from the Walker model.

For example, Kahr's SILI compiler [Kahrs 85] compiles 'C' programs into silicon structures [Kernighan 78]. With the Walker model one can see clearly that in order for C programs to be compiled into *efficient* silicon structures, it is necessary to synthesise information at a higher level of abstraction than a program. We know that this type of level recovery is very difficult: Witness that it is very hard to translate language A to language B by producing object code from A and then recovering the semantics to produce B.

The proponents of low level HDLs, it seems, avoided the problems of semantics recovery but were not equipped to tackle the problems of automated design synthesis.

Implications for automated synthesis

The Walker model for representing design information, if accurate, implies that:

1. HDLs for silicon compilers must allow a specification to be described at a very high level of abstraction and keep describing the specification as it is translated from one level of abstraction to another, with arbitrarily fine granularity, until the specification ends up wholly in the physical domain with sufficient detail to spawn manufacturing information.

2. At any one level of abstraction, a specification may exist in several domains simultaneously.

Formulating Specifications

We set out on the quest of developing:

> "... a tool that interacts with engineers on their own level to capture a specification that is translated into mask geometry automatically: The ultimate VLSI CAD tool."

This begs the question: At what level do engineers interact with each other? What is an engineer's "own level"?

This question was tackled in [Deas 83], and the arguments presented there are reproduced below. The original argument conveys the ideas on HDLs in a succinct and readable form, so there is no need to recast them. The main argument is quoted below:

Beginning of quote (with minor changes):

"The ideal interface to a silicon compiler is at the highest level, that is, at the same level as a client would use to specify a system to an engineer. Observations and experience of how this is done in industry indicates that the following dialogue is typical:

Social introduction ...

Client: "We're working on a system which I reckon might be better off as a chip. Can I explain what it does and perhaps you could give me your comments?

Engineer: "O.K. Here's some chalk ... First, could you tell me a little about what your system does as a whole so I can get a feel for what is going on.

Client: "O.K. We want to send out ultrasound pulses into fixture welds on steel pipes, interpret the echoes and draw a picture of the weld on a monitor.

I'll draw you a sort of block diagram of what we've got.

We have four transducers which ... and we end up with ... we calculate the values of X, Y and Z using this equation ... matrix transformations ... and we end up with ... a picture.

Engineer: "O.K. I think I understand what's going on. Let's talk about the bit you want to implement as a chip. Can you draw me a block diagram of the ...

Client: "Let's see. If we start with the middle bit ...

Engineer: "Hang on, let's keep it simple. First of all, what lines do you have coming in? ... and what does that bit do?

... ...

The engineer solicites a conceptual block diagram of the proposed chip, providing feedback to the client about performance issues as he progresses. The engineer's experience is used by the client to evaluate different ideas and experiment with what is possible, trading off performance in one part of the chip against that in others and balancing extra features against cost *etc.*

Finally:

Client "Oh yes, I almost forgot – we only need 10 chips at the moment but we must have them by the New Year. Let's see, this is October 29th, so that leaves a good three months ... is that O.K.?

End of direct quote

A number of features in this protocol can be identified.

1. The protocol begins with an abstract description of the whole system in the context of its working environment.

2. The protocol involves producing a block diagram containing several levels of abstraction. Notice that an esoteric requirement will be specified in a great deal of detail, other requirements will be more vague. For example, one block might have to run exceptionally fast. Then the speed will be specified (a performance requirement). Other blocks are state machines, and to specify the performance of a state machine might require a transfer equation or a state table. Certainly, the contents of ROMS – it might be the set of ASCII codes, or a microcoded instruction set – would have to be specified. In the Walker model of HDLs, these things, the performance information and the state table, are at two opposite ends of the abstraction spectrum.

3. In producing the block diagram, both client and engineer find it useful to start with the input-output lines and walk through the innards in a series of hierarchical passes.

4. Client uses feedback from the engineer to assess the implications of the various design constraints and to evaluate alternative architectures.

Behaviour, not structure

The *ibid.* argument for a block based approach to specifying VLSI parts is rather lengthy; we have just selected some of the salient points. But it is clear that engineers do not formulate their needs using some highly theoretical model. They deal with block diagrams where the blocks perform functions recognised by client and engineer.

Whatever comes out of the discussion between our engineer and his client is certainly a behavioural description and in the context of Walker's DAG, the description is at the functional block level of abstraction.

It is important to recognise that the client does not give the VLSI engineer any structural information. But outwardly, a structural description might be identical in its syntax to a behavioural description. To avoid any confusion between the two, the differences in semantics will be highlighted by an example.

Say a block in the client's model is a multiplier. A VLSI engineer might find that the multiplier only multiplies numbers one up or one down from the previous number it multiplied. So the engineer decides to implement the multiplier as a counter and an adder, to make the chip smaller. Furthermore, the engineer might decide to split the counter and adder in two, site each part on opposite sides of a chip and run the carry signals across the chip. Perhaps, it is more efficient to do this than to carry the input to this composite pseudo-multiplier across a chip. The engineer has the freedom to make this sort of decision by virtue that the information given to him by the client is purely behavioural. It is not structural. If it been structural, then the engineer would have been obliged to implement the multiplier as a general purpose multiplier, no matter how inefficient that may have been.

Requirements of a design capture language

Using the Walker model, it is possible to identify a number of requirements the HDLs used by a silicon compiler must meet in addition to those mentioned earlier.

1. It must capture *behavioural* information from the user.

2. Whilst the user may throw out information at many levels of abstraction, the bulk of it is at the *functional block level.* The language should be designed therefore to interact with the user primarily at this level, but be sufficiently flexible to allow relevant information at any level to be stored.

3. During the compilation the amount of detail in the specification will increase enormously. The language must store this information in an easily accessible form.

Thinking ahead to the way in which a silicon compiler might be used causes further demands to be made on a HDL. For example, it was mentioned in the introductory chapter that a silicon compiler must:

- Support interaction. We decided that a user must be allowed to change absolutely anything. This means allowing people to interrupt an automated process, modifying things in a controlled way and then continue.

- Represent partial designs. Even if one can design a chip in a minute, the hackers will take longer and even hackers need tea-breaks.

- Eventually, describe physical layout. To do this, the HDL must be able to represent every detail anyone could conceivably have a use for: from what a block is to the type, position, orientation and connectivity of a buried contact. This information must be stored in such a way that geometrical *and* topological information can be extracted quickly and easily. The topographical requirement comes from our need to support switch level simulators.

- The HDL must be extensible. We certainly will not cover all the requirements for a HDL, so we must design the HDL in such a way that extensions can be added painlessly.

So far we have identified roughly what it is that must be captured by a HDL. We have also identified a suitable model to represent the various transforms that have to be applied to map the initial specification into the domains it needs to be in for an implementation of that specification to be manufactured. We can now start thinking about what form that language will take. Rather than replicate work, if a suitable HDL exists then we should use it and not set about creating another. But does a suitable HDL exist?

Existing design capture languages

Of the very many HDLs described in the literature, very few can describe design information in all three descriptive domains: Most support structural information only, or physical information, or more limited still, just geometrical information. There are two outstanding exceptions to this generalisation: TIDAL and EDIF (Electronic Design Interchange Format) [EDIF 84]. Only these two seem to address the whole spectrum of design description.

TIDAL was the first language that could describe every aspect of a design, from conception to how the chip is mounted in a carrier. It is superseded by EDIF.

EDIF is the work of the EDIF Steering Committee, a body representing the coordinated efforts of seven large companies active in the commercial CAD/CAE

business: DAISY Systems Corp., Mentor Graphics Corp., Motorola Inc., National Semiconductor Corp., Tektronix Inc. and Texas Instruments Inc. EDIF is rapidly becoming an industry standard format for design information. It introduces the possibility of Open Systems, allowing products from different vendors to be integrated into one, and as a medium for communicating design information from one company to another. But EDIF is not intended as a design language: It is a *format* for the transmission of design information. The introduction to the EDIF specification, states that:

> "Essentially, EDIF describes a single, large data structure. *It is neither a programming language nor a data-base system per se.*"

Although EDIF was not designed with silicon compilers in mind, it could be used for that purpose. But EDIF is not a pleasant language to work in: It was designed to the churned out from some pre-existing database rather than to be written by hand.

Whilst the EDIF syntax is quite simple – it looks like LISP – the language is very large indeed, and growing rapidly as the many EDIF Technical Committees extend it. The extensions are made by adding keywords to the EDIF specification. Already there are many keywords that a user might include in a specification but might be meaningless to a silicon compiler. For example, to support gatearray layout systems EDIF allows wire delays to be specified, but whilst this can be done, the specification will probably be ignored by a silicon compiler. If a user can describe many different aspects of a system that are ignored by a silicon compiler then the compiler is failing to implement the full specification. The view is taken that a user should not be left in any doubt as to what has been specified and what has not. This view rules out using EDIF.

These comments are not criticisms of EDIF; they simply reiterate the statement made by the EDIF committee: EDIF is a format. It is not a programming language, nor is it a design language. EDIF was designed for transmitting CAD information between CAD systems. For that purpose, it is well designed. For the purposes of providing an interface to a silicon compiler, it is ill-suited.

If EDIF does not fit our requirements then a new language supporting automated design tools must be designed.

Technology Capture

There are three different things a compiler's HDL must capture:

1. The specification of a design and the transformations applied to it.

2. The library of information used by the compiler to carry out those transformations.

3. A set of design rules for a particular process technology.

Up to now we have only considered the first of these three.

The information associated with the each information category is almost completely co-orthogonal, so it is sensible to use three different languages, one for each category.

The language that handles the first category of design information, called UNIT, is described in the next chapter, Chapter 5. Chapter 6 describes a language called LEGO for representing the library information, called LEGO. But the third category gets just the scarcest of mentions in Chapter 8: It is discussed in the literature and Phil James attempts to deal with it in his MSc thesis [James 85]. Technology and process information is far removed from the subject of this thesis and the problems of designing tech files are by no means unique to silicon compilers so we shall avoid dealing with them here.

Summary

The framework for silicon compilation presented in this thesis uses two languages. One language, called UNIT, is used for capturing the design specification and the other language, called LEGO describes the target technology, it indexes a set of design rules and describes a group of module generators and leaf cells that conform to those rules.

In this Chapter the characteristics each of is languages must exhibit has been estabilished and a rationalé governing the design of them was expounded. The next Chapter presents a detailed description of the UNIT language, and the chapter after that presents LEGO. UNIT must support interaction, allow that interaction to be recorded, dumped and recaptured. To do this, the original UNIT program supplied by a user is translated into lower level dialects. Chapter 8 is devoted to describing each dialect and illustrating how these support the different components of a silicon compiler.

Chapter 5

The UNIT Language

UNIT is not one language, but seven consistent languages covering different levels of design abstraction, from behaviour to topography. This chapter introduces the highest level of the language group, used to capture chip specifications. The lower level sister languages will be described in later on.

This chapter begins with a short discussion on design styles and uses a hierarchical design example to illustrate the structure of a UNIT program. This is followed by the lexical details and the syntax of the language.

Design Capture

The UNIT language can be used in conjunction with engineering workstations in common use, for example the DAISY^{TM} Workstation, by entering an architectural schematic on a workstation and then compiling the design capture file into UNIT. On the DAISY, the UNIT program would be extracted from the :DIF and TREE.DLNK files. The DAISY information can then be back-annotated by the compiler spawning an IMAGE.SOM file.

The alternative to using an engineering workstation for design entry is to compose the UNIT program using a text editor, which is probably faster but demands that a user learn a new language (UNIT).

Program Structure

The UNIT language describes design hierarchies at the behavioural block level. The behavioural block level of abstraction is that used in an architectural block diagram: the architecture is defined but the manner in which each of the blocks are implemented is not.

In a design hierarchy, each successive level of the hierarchy contains more detail than its parent. The hierarchy starts at a very abstract level and, if the chip were to be designed by hand rather than by a compiler, then the lowest levels would describe mask geometry, but in UNIT, the leaf level consists of functional

blocks such as adders, state tables and transfer equations. The compiler can produce mask geometry for these functions automatically.

We shall proceed by using a real-life example to demonstrate the features of the UNIT language.

There are some drawbacks to using a real example. Choosing an existing chip will certainly prevent the example from becoming too contrived, but it means that a lot of irrelevant detail is introduced very early on. In a completely new design, details do not arrive all at once, but rather, they are added as the design develops.

Say, for the purposes of an example, that a functional equivalent to the Motorola 68000 microprocessor is needed. The purpose of this chapter is to describe the UNIT, it is not to describe someone's microprocessor. To avoid getting bogged down in a turgid account of how the M68000 works, it will be necessary to grossly simplify the microarchitecture. For the real story on the 68000, Marchal's analysis of the machine is recommended [Marchal 84].

The input and output signals on the 68000 are organised into the functional groups shown in figure 5-1.

In UNIT we are concerned with only the functional aspects of the chip, so the input and output signals seen on the pinout would be translated into the following UNIT block.

```
Chip M68000CPU(  InvertingInput  Dtack(1), Bgack(1),
                                 IPL(0:2), Berr(1),
                                 VPA(1), BR(1);
                          Phi1   CLK(1);
                InvertingOutput  VMA(1), AS(1), UDS(1),
                                 LDS(1), BG(1);
                        Output   Addr(1:23), FC(0:2),
                                 RW(1), E(1);
                         Bidir   Data(0:15);
                 InvertingBidir  Halt(1), Reset(1))
    :
  {Lots of things inside here}
    :
EndChip M68000CPU
```

The header should not include power pads and it need not include the clock.

A UNIT program is a series of nested blocks, with the outermost representing the entire chip. A block is declared by a header consisting of the reserved word `Module`, the name of the block and a list of ports. The outermost block is treated specially, and so is distinguished from normal modules by being declared with the reserved word `Chip`.

For symmetry, whenever there is a statement that opens an environment, such as a `Chip` or a `Module` header, then there is a matching terminator. In the M68000CPU example, the statement `EndChip M68000CPU` terminates the `Chip` statement.

Refinement

By making one step of refinement to the top level UNIT program, we arrive at the program given in figure 5-3. This program corresponds with the diagram shown in figure 5-2. Many of the details have been omitted in order to emphasise program structure and to skip over the syntax. The precise syntax will be dealt with later on. Try to correlate the functional objects in the architectural diagram with the constructs in the UNIT program.

Program Components

Figure 5-3 illustrates all of the five parts that make up a UNIT block, namely:

1. A header,
2. Symbolic constants,
3. Nested modules,
4. Instances,
5. Connectivity.

Look at the program in figure 5-3, in particular at the instance declarations. Notice that some of the instances refer to nested modules, whilst others (*e.g.* `ParameterisationLogic`), refer to function synthesisers contained in a technology dependent library.

Preliminaries

The statements used in the 68000 example are described in detail later on, after we have covered the lexical preliminaries.

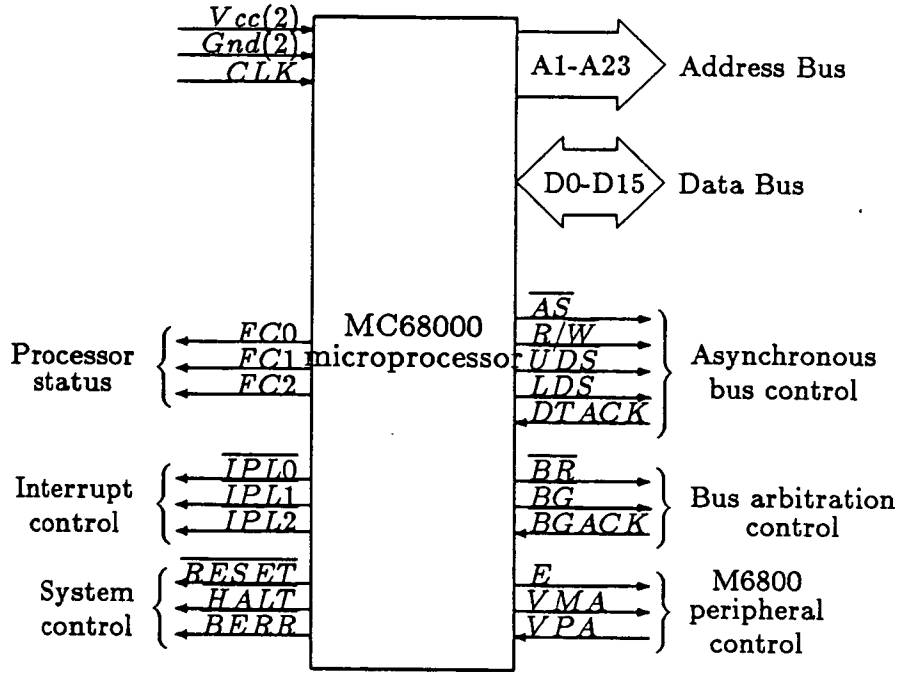


Figure 5-1: External interface to MC68000

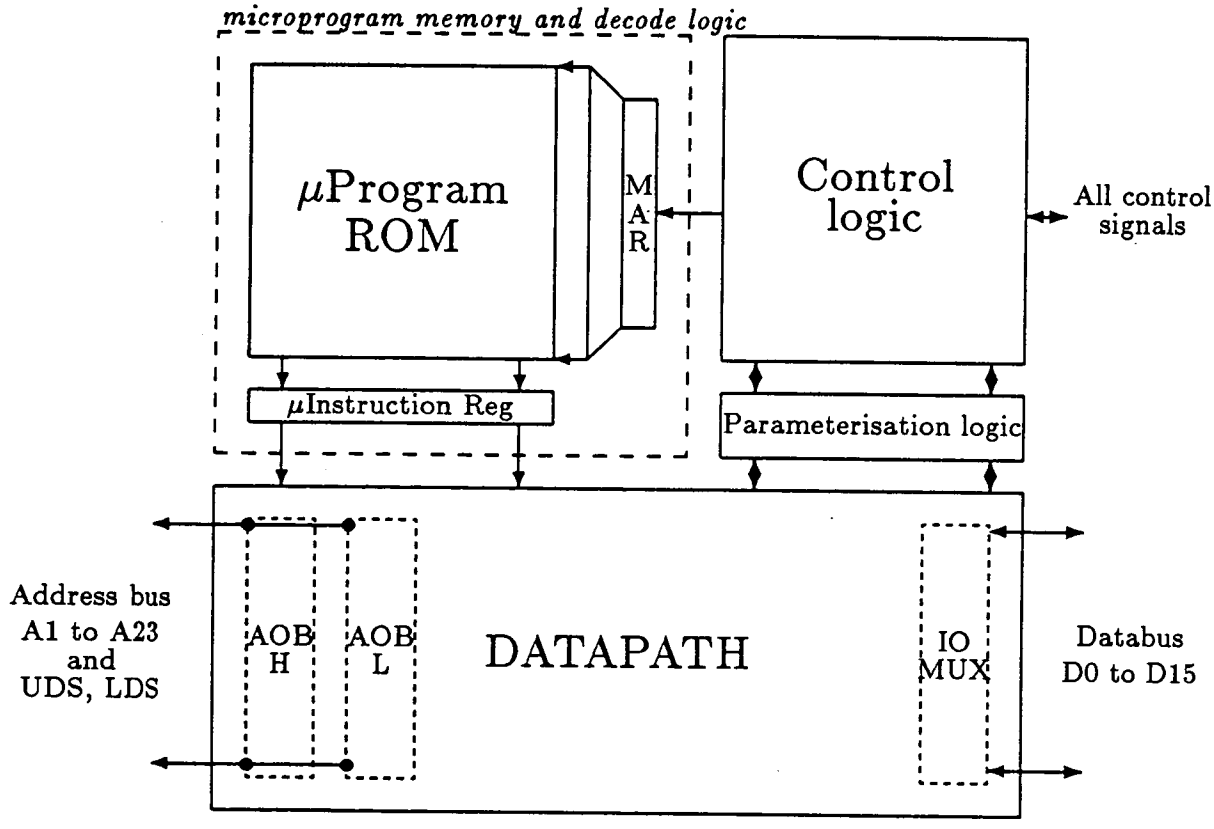


Figure 5-2: Architecture of the imitation 68000

```
Lib(nmos2)

Chip M68000CPU(  InvertingInput  Dtack(1), Bgack(1),
                                 IPL(0:2), Berr(1),
                                    :
                                 {Rest of header}
                                    :
                 InvertingBidir  Halt(1), Reset(1))

      {Symbolic Constants}
      Constant word("0:31"), bit("0:0")

      {Nested Modules}
      Module ControlLogic( {Port List} )
         EndModule ControlLogic
      Module MicroInstrMemory( {Port List} )
         EndModule MicroInstrMemory
      Module DataPath( {Port List} )
         EndModule DataPath

      Instance  ControlLogic       Ctrl
      Instance  MicroInstrMemory   MInstrMem
      Instance  Logic              ParameterisationLogic(
                                   Statetable({A HUGE State table}))
      Instance  Datapath           maths

      {Connectivity}
         M68000CPU_FC(0:2) => Ctrl_FC
         M68000CPU_Reset(1) => Ctrl_Reset
               :
         {Datapath connections}
         MInstrMem_CtrlPts(0:196) => Datapath_ctrlpts
         ParameterisationLogic_Inhib(0:3) => Datapath_XBusInhibit
         Datapath_addressbus(1:23) => M68000CPU_addressbus
               :
         {many more connectivity declarations}
   EndChip M68000CPU
EndFile.
```

Figure 5-3: UNIT Program For Single Chip CPU Level B

`Bidir`	`Chip`	`Class`
`Constant`	`Dir`	`EndChip`
`EndFile`	`EndModule`	`Fmax`
`Gnd`	`Input`	`Instance`
`Lib`	`Location`	`Module`
`Output`	`Ports`	`Posn`
`Phi1`	`Phi2`	`Range`
`Size`	`Vdd`	`Wordlength`

Figure 5-4: UNIT Reserved Words

Lexical Conventions

The UNIT language uses six classes of lexical token: reserved words, special symbols, identifiers, integer literals, string literals and white space formed from spaces, tabs, newlines and comments. White space is ignored, except where it occurs within a string literal or as serving to delimit other tokens.

Source files are deemed to be in the ASCII character set.

UNIT is not case sensitive: UPPER CASE and lower case letters may be used interchangeably to improve legibility. Nonprinting characters are illegal.

Reserved Words

A list of reserved words is given in figure 5-4. Throughout this document, reserved words are printed in `plain font` with the first letter in UPPER CASE to distinguish them from running text.

Special Symbols

Special symbols serve to delimit other tokens, as punctuation and to introduce context to the grammar.

All printing symbols that are not letters, digits or white space are treated as special symbols. Not all special symbols are legal, for example, the grammar does not include the up-arrow "^".

Identifiers

Identifiers can be any string of letters and digits up to 31 characters long, the first character being a letter. None of the reserved words may be used as identifiers.

Comments

Comments are a type of white space made up of printable characters enclosed within curly braces. Comments can be of any length, including zero, and run over any number of lines.

String Literals

String literals are a series of printable characters enclosed within double quotes "". To include a double quote in the string, put two double quotes next to each other: *e.g.* The string """""" is 1 character long. The empty or *null* string is written "". Strings can be of any length and run over any number of lines.

Integer Literals

An integer literal is a number in the range 0 to $2^{31} - 1$. Integers are assumed to be in decimal unless prefixed by a radix (either 2, 8, 10 or 16). For example:

```
2_1111    8_17    10_15    15    16_F
```

Negative (-ve) integers are never sensible in the UNIT language, for this reason, it is not possible to declare them.

Compiler Options

Compiler options are used to control error messages, source listings and maintenance diagnostics *etc.*

Aside from any implementation manuals (that may or may not exist!), information on what options are available on any particular compiler can be obtained in abbreviated form when calling U2, by replacing the command line parameters by two question marks.

Compiler options are usually selected when calling the compiler, but they can be changed at any point in a UNIT program. An `Option` statement looks like:

```
%AnOptionName(NewOptionValue)
%Diagnostics(On)
%Fmax(1)
%Listing(Off)
%Report(Chip fabricated in July 1984, ref EU4567)
%Site()
%Warnings(Off)
```

The percent sign % is a special symbol because it is a printing character that is not a letter, a digit or white space. The percent introduces a context dependent option identifier known to the compiler. The option identifier does not clash with other identifiers.

Note that some options take things that look like strings in their argument. They are not. Strings must be enclosed in double quotes, whereas options can take anything in their argument and will interpret double quotes literally. For example, the option

```
%Report("How about "" this then")
```

will type `""How about "" this then""` on the screen.

Whilst the number of options may vary from machine to machine and compiler to compiler, all UNIT parsers are deemed to provide the following six:

Diagnostics takes values (ON-OFF). When diagnostics are ON, maintenance diagnostics are printed on the user's terminal and in the listing file. The default is OFF.

Fmax takes an integer value that sets the maximum clock speed for all ports. For example, if Fmax is set to 5 (MHz), and a port named LOAD is declared, then the compiler will ensure that the LOAD port will function correctly when toggled at 5MHz, that is, everything LOAD is connected to will operate at 5Mhz. The default Fmax is set by the library file. For NMOS2, the default is 1MHz.

Listing takes values (ON-OFF). When listing is OFF, no messages are sent to the listing file. The default is ON.

Report takes a string up to 64 characters in length. The string is sent to your terminal and to the listing file immediately it is

parsed. The default message is the null string. The effect of a report statement with a null string is to interrupt the compilation, send a blank line to the terminal and wait for a continue signal from the user.

Site prints various site details on the user's terminal and in the listing file. The site details include, at least, the version number of the compiler that is being used, the site name, the date and the name of the person responsible for maintenance at that site. For example:

```
!-----------------------------------!
!         U2.3(.0) at EUCSD         !
!  UNIT Source Compiled on 01/04/85 !
!  Maintenance: Alex Deas  Ext 9999 !
!-----------------------------------!
```

Warnings takes values (ON-OFF). When warnings are OFF, no warnings are given, (but messages about full errors will still be posted). The default is ON.

Scope rules

The grammar implies Algol scope rules for declared constants, but adopts strict block limited scoping for everything else. That is, instances may only connect with or reference those items declared at the same level of hierarchy within a parent module.

Dressing

A UNIT program is a tree of modules, except for two statements: `Lib` and `Endfile`. Consider each statement in turn.

Lib Statement

The `Lib` statement appears at the very start of a UNIT program; it looks like:

```
Lib(nmos2)
```

This indexes a technology dependent library file (here it is called nmos2), containing a LEGO program. A LEGO program says what is possible in a particular fabrication technology: what the design rules are and what function synthesisers are available. The LEGO language is described in the next chapter.

`Lib` must be the first statement in a UNIT program.

EndFile Statement

The statement `EndFile.` is placed at the end of a UNIT program, to tell the compiler that it has reached the end of the file. It must be followed by a fullstop.

Blocks

The single chip CPU example given earlier illustrates all of the five parts that make up a block: a header, constants, nested modules, instances and connectivity. The constants, nested modules and the instances are optional. Headers and connectivity are not: a block is invalid without them. Note that if a program part is used it must be given in the order listed. Consider each of the five parts in detail.

Header

Modules open with a header formed by the reserved word `Module`, followed by the name of the module and a list of input and output ports. An example of a header with its matching terminator is given below.

```
Module  asubmodule( {Port List} )
   {body of asubmodule}
EndModule asubmodule
```

It has been mentioned already that the opening `Chip` is merely a variant of `Module` allowed to use real pads rather than just the simple port types `Input`, `Output` *etc.*

Ports

The port list is an integral part of the header, it looks like:

```
( Input operanda(0:11), operandb(0:11), Output y(0:0) )
```

The components of the list are described below.

Direction All ports have a direction, either `Input`, `Output` or `Bidir`, unless the port is on the outer-most module of a design. The `Chip` module is treated specially in that it can have any direction known to the "Pad" family in the constraint library. All libraries should contain the following pads: `Input`, `Output`, `Bidir`, `InvertingInput`, `InvertingOutput`, `InvertingBidir`, `AnalogueIn`, `AnalogueOut`, `HighDriveOut`, `Phi1` and `Phi2`. The more esoteric pads have multiple fields that change the name of the port by adding various suffixes. For example, `Bidir` has an ENable wire, an INput wire and an OUTput wire, so a port declared as `Bidir porta(0:3)` would be transformed into `PortA.En`, `PortA.In` and `PortA.Out`. Connections to and from multiwire pads must use the suffixed port name.

Port Name The port name is an identifier. The *full* name of a port is given by the name of the module, the port name, a pad suffix if there is one and either a range or a width, in the form:

<part name> _ *<port name>* [. *< pad suffix>*]
(*<port range>*)

Range The range determines what values the signal carried by the port is allowed to take. For example, the range (4:8) implies that the signal carried by a particular port can take any value from 2^4 to 2^8 inclusive. The range can be replaced by the port width if the lower bound of the range is zero. For instance, the range (0:12) can be replaced by the width (13), because 13 bits are required to represent a signal over the range 2^0 to 2^{12}.

An Example Header

The following example illustrates what a header might look like for a hypothetical 8 bit multiplier:

```
Module multiplier(Input Operanda(0:7), Operandb(0:7),
                  Output Product(0:15))
```

```
   {lots of things inside}
EndModule multiplier
```

Or, alternatively:

```
Module multiplier(Input Operanda(8), Operand(8),
                  Output Product(16))
   {lots of things inside}
EndModule multiplier
```

The two programs given above are identical, except that the first example specifies the range of a port, whereas the second example specifies the width.

Symbolic Constants

To simplify the design of general purpose modules, the UNIT language provides facilities for declaring symbolic constants. The following constants are especially useful:

```
Constant bit ("0:0"),
         byte("0:7"),
         word("0:15")
```

The following constants are predeclared:

```
Constant Yes("1"), No ("0")
```

Constants may be nested to any depth, although care must be taken to ensure that they do not recurse. Recursion may not be trapped by the parser, and the only indication that it has happened may be a few seconds silence followed by a heap overflow message. On the prototype U2.1 compiler this otherwise fatal event could be corrected by the selecting the screen editor (the cursor will be one token after the first place where the offending constant identifier is used), and making the correction.

Instances

After the header, constants and any nested modules, come the instance declarations. The word *instance* here refers to a behavioural instance, (as opposed to a geometric instance). The compiler may translate two instances of the same

module into two entirely different pieces of geometry, so long as each instance has the same functionality.

An instance declaration looks like:

```
Instance  function instancename
```

The function must be either a nested module or a function synthesiser known to the constraint library. Function synthesisers are specified by a generic name such as *adder*, *logic* or *counter*. Some function synthesisers require parameters to fully determine the functionality of an instance. Logic blocks, for example, require a state table and as another example, a delay operator might require a delay length. These function specific parameters are supplied between parentheses after the instance name. The names of the parameters are given in the documentation that comes with a constraint library. The parameters themselves are divided into string parameters and integer parameters, enabling the compiler to flag an error if a parameter is missing or if too many are given.

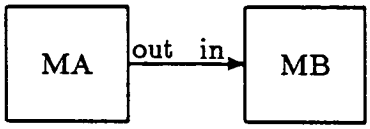
The example given below illustrates the "logic" function with two parameters, one of which is a state table, the other is a flag.

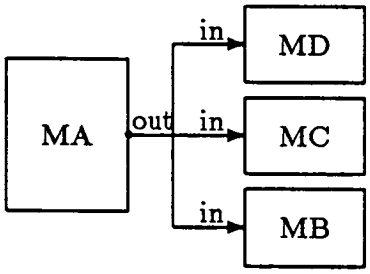
```
Instance logic  apla(statetable(
      IN  ireset,slowcl,s2,s1,s0
            0     x     1  0  0 0  0  1   0     0
            0     x     0  0  x 1  0  0   0     0
            0     x     0  x  1 0  1  0   0     0
            0     0     0  x  x 0  0  0   1     0
            0     1     0  x  x 0  0  0   0     1
            0     x     0  1  0 0  0  1   0     0
      OUT                        s0,s1,s2,firea,fireb),
      SMFlag(Dynamic))
```

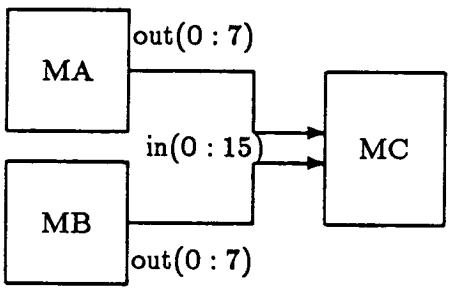
Multiple instances of a function can be declared using a list of instance names. For example:

```
Instance counter  counter1, counter2, counter3
```

A function synthesiser named `Undefined` allows blocks to be declared by their area, that is, as a block whose area is known (roughly) but whose precise functionality is not. This provision allows UNIT programs to be written top-down.

Configuration	Declaration
Point to Point	
	`MA_out => MB_in`
Bifurcated Input	

	`MA_out => MB_in`
	`=> MC_in`
	`=> MD_in`

Bifurcated Output	
	`MA_out(0:7) => MC_in` `MB_out(0:7) => MC_in(8:15)`
Bidirectional Bus	

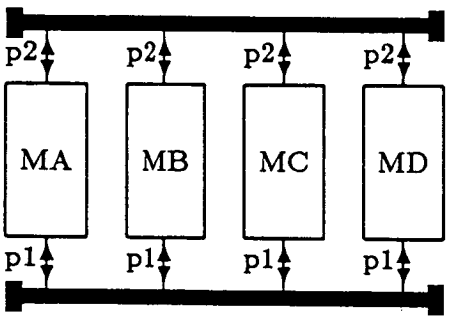
	`MA_p1 => MB_p1 => MC_p1 => MD_p1`
	`MB_p1 => MC_p1 => MD_p1 => MA_p1`
	`MC_p1 => MD_p1 => MA_p1 => MB_p1`
	`MD_p1 => MA_p1 => MB_p1 => MC_p1`
	`MA_p2 => MB_p2 => MC_p2 => MD_p2`
	`MB_p2 => MC_p2 => MD_p2 => MA_p2`
	`MC_p2 => MD_p2 => MA_p2 => MB_p2`
	`MD_p2 => MA_p2 => MB_p2 => MC_p2`

Figure 5-5: Connectivity Declarations

Connectivity

The final part of a UNIT block declares the connections between instances.

A connection between two ports is declared by giving the full name of the source port followed by the flow symbol ("=>"), followed by the full name of the sink port. Wires and buses that drive multiple sinks use the flow operator to indicate where the bus bifurcates. That means that all the sinks driven by an output port must be declared by a single compound flow statement. Examples are given in figure 5-5.

The default connections can be used to simplify net descriptions. If the port range is omitted, it is assumed that you want the same range as was used for the last connection and if there is no last connection, then only bit 0 is connected. In UNIT, only signal connections are specified: power is routed by the compiler. Also, you need not declare buffers nor should you ground (or pull up) unused inputs *etc.* The compiler does these things automatically.

Implicit Nets

In UNIT, only signal ports and their connections are specified. A UNIT program should not declare:

- Power nets. The compiler adds power ports to instances automatically.
- System clock nets. All chips are given one clock pad and this is routed to all clock ports that have not been connected explicitly. The clock ports are identified by their names `Phi1` and `Phi2`. Function modules that use a two phase clock have a phase splitter placed beside them automatically.
- Nets strapped to Gnd or Vdd. The compiler does it automatically. If the compiler objects to strapping a line, then something is wrong, either with the design or in the way it is described.

A Simple program

So far, no complete example of a UNIT program has been given; examples of individual statements have been given, but even the CPU chip example was not a complete program because the nested modules had no connectivity. Let us rectify this situation now.

Consider a very simple UNIT program:

```
Lib(nmos2.lib)

   Chip  tinychip(Input a(0:15), Output b(0:15))
         tinychip_a(0:15) => tinychip_b
   EndChip tinychip

Endfile.
```

This ditty specifies a chip with 16 input pads driving 16 output pads. The `Lib` statement tells the compiler to get information on the target technology from a file named `nmos2.lib`. The remainder of the program comprises a block (called `tinychip`), a declaration of two ports named `a` and `b`, and a 16 bit wide connection from port `a` to port `b`. The compiler adds a Vdd and a Gnd pad automatically to power the output pads: power pads are not declared.

Chapter 6

The LEGO Language

This chapter describes the LEGO language. A LEGO program, indexed from the UNIT language by the `Lib` statement, tells the silicon compiler what is possible in a particular fabrication process by giving the compiler a set of design rules and a collection of functional primitives which the compiler can call upon to generate modules.

This chapter starts by describing the structure of a LEGO program and then uses this to introduce the lexical details and the syntax of the language.

Influence

There are many similarities between the LEGO and UNIT languages and so some repetition of the previous chapter is inevitable. It was decided that rather than make omissions, to print duplicate sections for the sake of completeness.

Program Structure

A LEGO program comprises two parts:

1. A `Technology` statement that indexes a geometric design rule file.
2. A Series of *Family* declarations, each of which contain two parts:
   (a) Descriptions of leaf cells common to a group of module generators
   (b) Descriptions of module generators, called *Species*

Figure 6-1 contains a skeleton of a LEGO program. Notice from this figure that the `Technology` statement is the first statement in a LEGO program and is followed by a series of family declarations. The families group together functionally similar module generators with a common set of leaf cells.

```
Constant stdword("16")  {A symbolic constant}

Technology( {filename} )

Family  counters( ...
    Cell  slice( ...
       EndCell slice
    Cell  base( ...
       EndCell base
       :
    Species  std( ...
       EndSpecies  std
    Species  updown( ...
       EndSpecies  updown
    Species  fastcarry( ...
       EndSpecies  fastcarry
Endfamily  counters

Family  adders( ...
    Cell  slice( ...
       EndCell slice
    Cell  lookahead( ...
       EndCell lookahead
       :
    Species  full( ...
       EndSpecies  full
Endfamily  adders

Family  logic( ...
    Cell  pla( ...
       EndCell pla
    Cell  wbarray( ...
       EndCell wbarray
       :
    Species  wbarray( ...
       EndSpecies  wbarray
Endfamily  logic
EndFile.
```

Figure 6-1: Skeleton of a LEGO Program

Preliminaries

The statements in the skeleton program examples will be described in detail, but to do this, we must first dispense with the lexical details.

Lexical Conventions

The LEGO language uses six classes of lexical token: reserved words, special symbols, identifiers, integer literals, string literals and white space formed from spaces, tabs, newlines and comments. White space is ignored, except where it occurs within a string literal or as serving to delimit other tokens.

Source files are deemed to be in the ASCII character set.

LEGO is not case sensitive: UPPER CASE and lower case letters may be used interchangeably to improve legibility. Nonprinting characters are illegal.

Reserved Words

A list of reserved words is given in figure 6-2. Throughout this document, reserved words are printed in `plain font` with the first letter in UPPER CASE to distinguish them from running text.

Special Symbols

Special symbols serve to delimit other tokens, as punctuation and to introduce context to the grammar.

All printing symbols that are not letters, digits or white space are treated as special symbols. Not all special symbols are legal, for example, the grammar does not include an up-arrow "^".

Identifiers

Identifiers can be any string of letters and digits up to 31 characters long, the first character being a letter. None of the reserved words may be used as identifiers.

Comments

Comments are a type of white space made up of printable characters enclosed within curly braces. Comments can be of any length, including zero, and run over any number of lines.

`Assign`	`Add`	`And`
`Bidir`	`Bit`	`Both`
`Cell`	`Compose`	`Constant`
`Date`	`Demand`	`Div`
`Endcell`	`Endfamily`	`Endfile`
`Endspecies`	`Exp`	`Exor`
`External`	`Family`	`Fetch`
`Flag`	`Fmax`	`Gnd`
`Geometry`	`Highdriveout`	`History`
`Input`	`Integer`	`Internal`
`Interrogate`	`Inv`	`Inx`
`Iny`	`Location`	`Log`
`Lshift`	`Mandatory`	`Max`
`Mod`	`Mult`	`Nand`
`Neg`	`Nor`	`Optional`
`Or`	`Output`	`Parameters`
`Phi1`	`Phi2`	`Power`
`Report`	`Rshift`	`Sub`
`Size`	`Source`	`Species`
`Strap`	`String`	`Swop`
`Technology`	`TransferEqn`	`Type`
`Validated`	`Vdd`	`Wordlength`

Figure 6-2: LEGO Reserved Words

String Literals

String literals are a series of printable characters enclosed within double quotes `""`. To include a double quote in the string, put two double quotes next to each other: *e.g.* the string `""""""` is 1 character long. The empty or *null* string is written `""`. Strings can be of any length and run over any number of lines.

Integer Literals

An integer literal is a number in the range 0 to $2^{31} - 1$. Integers are assumed to be in decimal unless prefixed by a radix (either 2, 8, 10 or 16). For example:

```
2_1111    8_17    10_15    15    16_F
```

Negative (-ve) integers should never be needed, for this reason they cannot be declared explicitly but they can be formed using an arithmetic expression. Negative integers must be in the range 0 to -2^{31}: All arithmetic expressions must return results in the range -2^{31} to $2^{31}-1$.

Expressions

Any integer literal can be replaced by a reverse polish expression. Expressions can use literals, symbolic constants or integer functions as operands and anything in figure 6-3 can be used as a operator.

By way of an example, the following expressions all return the result 16:

```
16   (16)   (2_100,EXP)   (2,3,*,4,5,*,+,10,-).
```

Conditionals

LEGO makes no distinction between conditional and arithmetic expressions: all arithmetic expressions can be used as conditionals and all conditionals are arithmetic expressions. A condition is taken to be FALSE if the result from evaluating an expression is zero, otherwise the condition is TRUE.

Some of the arithmetic operators in figure 6-3 are label as *Conditional operators.* Whilst they can be used in any arithmetic expression to yield a binary (1 or 0) result, they are provided to simplify the construction of genuine conditionals. Genuine conditionals are used in only one statement (`Compose`). The `Compose` statement is introduced later on in this chapter.

Monadic Operators (Operand A)	
INV	Pop A, Push 2's complement negation of A
NEG	Pop A, Push 1's complement negation of A
LOG	Pop A, Push bottom $\log 2A$
EXP	Pop A, Push 2^A
FETCH	Pop A, fetch value of A, Push value
Diadic Operators (Operands A and B)	
AND	Pop B, Pop A, Push bitwise logical AND of A and B
OR	Pop B, Pop A, Push bitwise logical OR of A and B
NOR	Pop B, Pop A, Push bitwise logical NOR A and B
NAND	Pop B, Pop A, Push bitwise logical NAND of A and B
EXOR	Pop B, Pop A, Push bitwise logical exlusive or of A and B
ADD	Pop B, Pop A, Push A + B
SUB	Pop B, Pop A, Push A - B
DIV	Pop B, Pop A, Push bottom A Div B
MULT	Pop B, Pop A, Push A * B
LSHIFT	Pop B, Pop A, Push A shifted left B times
MOD	Pop B, Pop A, Push A Mod B
MAX	Pop B, Pop A, Push minimum of A and B, Push maximum of A and B
RSHIFT	Pop B, Pop A, Push A shifted right B times
SWOP	Swops two topmost elements of stack
ASSIGN	Pop A, Pop B, Assign A to name B
Conditional Operators (Operands A and B)	
=	Push 1 if $A = B$, else push 0
>	Push 1 if $A > B$, else push 0
<	Push 1 if $A < B$, else push 0
>=	Push 1 if $A \geq B$, else push 0
<=	Push 1 if $A \leq B$, else push 0
|	Push 1 if either A OR B is not 0
&	Push 1 if both A AND B is not 0, else push 0
~	Push 1 if $A = 0$, else push 1
Function Operators	
Wordlength	Push wordlength of part
<parameter name>	Push value of a integer parameter
a,b,c...z	Push loop counter value

Figure 6-3: Arithmetic Operators

Compiler Options

Compiler options are used to control error messages, source listings and maintenance diagnostics *etc.* Information on what options are available on any particular compiler can be obtained in abbreviated form when calling U2, by replacing the command line parameters by two question marks.

Compiler options are usually selected when calling the compiler, but they can be changed at any point in a LEGO program. An `Option` statement looks like:

```
%AnOptionName(NewOptionValue)
%Diagnostics(On)
%Listing(Off)
%Report(The NMOS1 library has not been tested.)
%Report(You are advised to use NMOS2 instead.)
%Site()
%Warnings(Off)
```

Note that some options take things that look like strings in their argument. They are not. Strings must be enclosed in double quotes, whereas options can take anything in their argument and will interpret double quotes literally. For example, the option

```
%Report("How about "" this then")
```

will type "`"How about "" this then"`" on the screen.

Whilst the number of options may vary from machine to machine and compiler to compiler, all LEGO parsers are deemed to provide the following five:

Diagnostics takes values (ON-OFF). When diagnostics are ON, maintenance diagnostics are printed on the user's terminal and in the listing file. The default is OFF.

Listing takes values (ON-OFF). When listing is OFF, no messages are sent to the listing file. The default is ON.

Report takes a string up to 64 characters in length. The string is sent to the user's terminal and to the listing file immediately it is parsed. The default message is the null string. The effect of a report statement with a null string is to interrupt the

compilation, send a blank line to the terminal and wait for a continue signal from the user.

Site prints various site details on the user's terminal and in the listing file. The site details include, at least, the version number of the compiler that is being used, the date, the site name and the name of the person responsible for maintenance at that site. For example:

```
!-----------------------------------!
!         U2.3(.0) at EUCSD         !
!  LEGO Source Compiled on 01/04/85 !
!  Maintenance: Alex Deas  Ext 9999 !
!-----------------------------------!
```

Warnings takes values (ON-OFF). When warnings are OFF, no warnings are given, (but messages about full errors will still be posted). The default is ON.

Scope rules

The LEGO grammar applies Algol scope rules everywhere.

Syntax

We have looked at the structure of a LEGO program and at the lexical conventions. Let us now turn our attention to the statements that make up the syntax of the language.

Symbolic Constants

Symbolic constants can be used to clarify the meaning of otherwise awkward coding and to simplify the design of general purpose modules. Symbolic constants must be declared at the very start of a LEGO program (that is, before any other statements), and they may be nested to any depth, although care must be taken to ensure that they do not recurse.

Symbolic constants look like:

```
Constant identifier("string substitution")
Constant metal("4M")
```

```
Constant maxsize("28"),
         divisor("wordlength,16,div")
```

The following constants are predeclared:

```
Constant Yes("1"), No ("0")
```

Technology Statement

The Technology statement indexes a file containing geometric design rules, using the form:

```
Technology(afilename)
```

The design rules should be in a form dumped by DRG (Design Rule Generator), a self-contained utility for interactively formulating and editting information about fabrication processes. It is appropriate to give a few words of explanation here about what DRG does (and what it sometimes gets wrong).

Design Rule Generator

The rules produced by DRG fall into four groups:

1. Source rules (rules about where the rules came from), including who entered the rules, the date they were entered and from whom they were obtained.

2. Global Rules which state things like the maximum power consumption per unit area, the maximum operating voltage, the yield fall-off for increasing area and the number of masks.

3. Mask Rules which state for each mask layer, what the layer does, what it is called (its name), what colour to draw it, the minimum linewidth, minimum separations, the sheet resistance and the interlayer capacitance.

4. Log Rules that tell the compiler about interactions between mask layers the correspondence between physical mask layers and virtual *log* layers. The log rules tell the compiler how to produce active devices (transistors), contacts *etc.*, and what design rules to obey.

Design rules are often incomplete, either the silicon foundry is reluctant to issue a full set of design rules, or perhaps the design rules have been obtained by staring at chips under an electron-microscope. A silicon compiler obviously needs a complete set of design rules, so when confronted with an incomplete rule set, DRG guesses whatever is missing. DRG should know what values for a rule are reasonable and is guess conservatively. Where this *rule guessing* has been done, the silicon compiler will post the warning, "Technology file includes unoffical design rules". Whilst this warning need not cause direct concern, it could portend trouble in that there may be no foundry to fabricate your chips. Information on where the rules came from and which ones have been guessed can be obtained by running DRG on the technology file.

Families

Immediately after the Technology statement is a list of the functional primitives available to the compiler. The functional primitives create modules such as adders, counters and logic arrays upon demand.

In LEGO, modules that perform the same generic function are grouped into *families*, the members of which are known as *species*. Most families include a collection of leaf cells, available to all species in that family. The complete family declaration gives the compiler information on how big a proposed module will be, what it does (its behaviour), where the ports can go and even which species to use.

A skeleton of a family of counters is shown below:

```
Family  counter( ...
        Cell  dirslice( ...
           EndCell dirslice
        Cell  loadslice( ...
           EndCell loadslice
        Cell  countslice( ...
           EndCell countslice
        Cell  stdbase( ...
           EndCell stdbase
           :
        Species  std( ...
           EndSpecies  std
        Species  updown( ...
           EndSpecies  updown
```

```
        Species  loadable( ...
           EndSpecies  loadable
        Species  allsinging( ...
           EndSpecies  allsinging
Endfamily  counters
```

The counter family has within it a collection of leaf cells that are available to its four species. The species are: a standard up counting resettable counter, a loadable counter, an up-down counter and a loadable up-down counter.

Note that there are three parts to a family:

1. A header (and matching terminator),
2. `Cell` statements that describe leaf cells,
3. `Species` statements that describe how the leaf cells are composed.

Consider each part in turn.

Family Headers

Families are declared with a header formed by the reserved word `Family` and the name of that family. For consistency, the Family construct is terminated by the `EndFamily` statement.

The skeleton program in figure 6-1 contains three examples of family headers.

Cells

After the family header comes a list of cells. There is one `Cell` statement for each leaf cell used by the species which make up the family. The `Cell` statement states how big a cell is, where the ports go and how the geometry is created. For the time being, we shall consider just cells with fixed geometry as might be produced with the aid of a layout editor. This type of cell looks like:

```
Cell  slicea( {Port Information} )
   History( ... )
   Power( {Power Consumption in Microwatts} )
   TransferEqn( {SimulatorName}, {FileName} )
   Size( {x size},{y size} )
```

```
    Geometry( {FileName} )
EndCell  slicea
```

The `Cell` statement has two parts:

1. A Header consisting of the reserved word `Cell`, the name of the cell and a port list.

2. A body that says where the cell came from (who layed it out), how much power the cell dissipates, what it does (its behaviour), how big the cell is and which file contains the geometry.

Cell Ports

The ports on the periphery of a leaf cell are listed in the cell header. An example is shown below.

```
Cell  slicea(
        botgnd(Type(External,Gnd),Location(0 S 60 M)),
        topvdd(Type(External,Vdd),Location(0 N 60 M)),

        a(Type(External,Input),Bit(i),Strap(L),
            Demand(Wordlength),Location(25 W 4 M + 25 W 4 P)),

        b(Type(External,Input),Bit(i),Strap(L),
            Demand(Wordlength), Location(35 W 4 M + 35 W 4 P)),

        cin(Type(Both,Input),Bit(i),Demand(0),
              Location(3 S 2 D)),

        cout(Type(Internal,Output), Location(3 N 2 D)),

        sum(Type(External,Output),Bit(i),
            Demand(1), Location(30 E 3 M)))
   {Cell Body}
EndCell  slicea
```

Ports in the port list are separated from each other by commas. Each port comprises a name and a set of *personality statements*. The personality statements specify where on the cell boundary the port is, how wide it is, what layers

it is on, what type the port is (`Input`, `Output`, `Gnd` *etc.*), what to do if it is left unconnected in a specification and what bit position it occupies when the cell is composed in a module. There are five personality statements: `Bit`, `Demand`, `Location`, `Strap` and `Type`. All ports must have a `Location` and a `Type` statement, but `Bit`, `Demand` and `Strap` statements are optional because they are not relevant to all port types, including output ports, power ports and internal ports.

Duplicate personality statements are illegal.

The semantic content of the personality statements is not immediately apparent from their names, so a detailed description of each of the five statements is shown below.

Type The `Type` statement has two fields. The first is one of the reserved words `Internal`, `External` or `Both` to denote whether the port can be used for internal connections only, external connections only, or for both. The second field states what the port is for, either `Input`, `Output`, `Bidir`, `Gnd`, `HighDriveOut`, `Phi1`, `Phi2` or `Vdd`. All ports must be declared, including the power ports (`Gnd` and `Vdd`).

Location Port locations are expressed as an abscissa along a given edge (measured south to north and west to east), of a certain width on a particular layer. For example, a port 30 lambda units along the west edge of a cell that is in 4 lambda wide polysilicon is declared by:

```
Location(30 W 4 P)
```

The delimiting spaces between fields can be omitted. If this is done, the statement given above would look like:

```
Location(30W4P)
```

Ports that appear in more than one position are declared by a sequence of port positions separated by plus signs ("+"). For example, if the port at 30W4P was on both polysilicon and metal layers and the port ran through the cell to appear on both lateral edges, then it would be declared by:

```
Location(30 W 4 P + 30 W 4 M +
         30 E 4 P + 30 E 4 M)
```

In the example shown above, the net is assumed to be connected within the cell. If it were not and the compiler is expected to complete the net by connecting the two ports together, then it might be declared as:

```
Location(30 W 4 P + 30 W 4 M,
         30 E 4 P + 30 E 4 M)
```

This auto-connect facility is useful when declaring power and ground lines on a sliced module.

The layer letters are given in the design rule file indexed by the `Technology` statement. In this report it is assumed that the rule file assigns the letter `D` to the diffusion layer, `P` to polysilicon and `M` to metal. Other files may use further letters, such as `Q` for second layer polysilicon or `T` for second layer metal. There should be some documentation somewhere that says what is available – if there isn't, run DRG on the tech' file.

Strap

The `Strap` statement tells the compiler how to disable input ports that are left floating, by connecting the port to either Vdd or Gnd.

If a port can be strapped high (connected to `Vdd`) when it is left unused, then it must be declared as:

```
Strap(H)
```

LEGO understands high states (`H`), low states (`L`), unknown states (`X`) for when both highs and lows are significant and (`R`) states for when the compiler must route it to all other ports with the same name on the chip (used for clock wiring) and (`T`) for when it must be synthesised from something else (such as Phi2 from a Phi1 clock). In general, (`R`) and (`T`) straps are applied to Phi1 and Phi2 ports respectively. The compiler uses a simple set of rules to synthesise (`T`) straps.

An example of a (`X`) strap is where a port cannot be strapped to Gnd (`L`) or Vdd (`H`), because the leaf cell needs that input to function sensibly, then it is declared as:

```
Strap(X)
```

The `Strap` statement is valid only for ports of type `Input`, `Phi1` and `Phi2`.

Demand

The `Demand` statement tells the compiler what to do with any port that is left floating: has the user forgotten to connect the port up, or is the port insignificant and so if it is an input it can be strapped to Vdd or Gnd, or left floating if it is of any other type. In addition to helping detect simple semantic errors in specifications, the `Demand` statement helps the compiler to pick species from a family when a member of that family is used in a design.

Consider when a cell is composed. The individual ports on its periphery are bound into groups that form *words*. The `Demand` statement tells the compiler how many of the individual ports must be connected for the composite cell to work properly. If a word consists of n ports, where n is known as the wordlength, and at least one port from the word must be connected then the ports are declared as:

```
Demand(1)
```

If all the ports forming the word must be connected, then the ports are declared as:

```
Demand(Wordlength)
```

If the ports are so insignificant that they can be safely strapped high or low by the compiler, a least significant carry input say, then the ports are declared as:

```
Demand(0)
```

The `Demand` statement is not valid for `Gnd`, `Phi1`, `Phi2` or `Vdd` ports.

Bit

In general, modules are formed using *many* leaf cells. For example, a common way of making an adder is to stack n slices vertically on top of each other, where again n is known as the wordlength. It is necessary to state which slice is what bit of the adder: is the most significant bit at the top, at the bottom, in the middle, or somewhere else? The `Bit` statement is used to convey this information.

When there is only one port with a particular name on the periphery of a module, it is labelled `Bit(0)`. The clock input of a counter is a good example of this type of port: no counter should have more than one clock input.

Usage of the Bit statement is intimately related to the way in which a cell is composed. The composition algebra used in LEGO makes available the loop variables that are used to replicate a leaf cell.

The Composition statement in LEGO behaves like a series of nested *for* loops. The innermost loop uses the loop counter variable `a`, the next innermost uses `b` *etc.* There is a limit of 26 on the depth in which these *for* statements can be nested, for obvious reasons. In addition to their primary use in composing cells, these loop variables can be used in arithmetic expressions. For instance, in the bit-slice adder example given earlier the operanda ports might be indexed 0 to $n-1$. This could be done by saying:

```
Bit(a).
```

If the most significant bit was composed first, then the expression might look like:

```
Bit(Wordlength,a,-).
```

One consequence of allowing the bit position to be dependent on the cell composition is that leaf cells used in different configurations must be defined several times, each definition using a different name.

Note that ports of type `Gnd`, `Phi1`, `Phi2` or `Vdd` do not take `Bit` statements.

Cell Body

There are six statements in the cell body: `History`, `Report`, `TransferEqn`, `Power`, `Size` and `Geometry`. Only the `Report` statement is optional. Consider each statement in turn.

History Statement

It is important to maintain information on where cells come from, who designed them and when the cell was first used. Traditionally, this information has been incorporated in comment statements, with the result that documentation is *ad. hoc.* and generally inadequate. The `History` statement by itself cannot guarantee adequate documentation, but it can ensure that there is someone to turn to when a cell does not work.

Each leaf cell has one `History` statement, which must come straight after the cell header. A `History` statement looks like:

```
History(Date(01/04/85),Source(EUCSD:AD),
        Version(26),Validated(Yes))
```

Each of the personality statements `Date`, `Source`, `Version` and `Validated`, are explained in detail below.

Date should be in the format used in the example. The LEGO parser will check that a date is given and that the date is not in the future.

Source a string up to 31 characters long, stating who designed the leaf cell. The LEGO parser will check that the string is not null.

Version an integer.

Validated must be either YES or NO. A Cell should not be marked `Validated(YES)` unless it has been simulated. A warning is posted if an unvalidated cell is indexed by a UNIT program.

Report Statement

`Report` statements can be used to give the user a message at the time when a particular cell is used, for example:

```
Report(Cell "slicea" has not been tested)
```

Messages can be any string up to 64 characters long. Messages longer than this must be broken up; there is no limit on the number of `Report` statements allowed in each cell.

Note that `Report` statements are quite different from `%Report` options: `Report` statements send a message to the user's terminal when a cell is instanced whereas the `%Report` option sends the user a message when the option is parsed (when the LEGO is read into a library).

TransferEqn Statement

A Silicon compiler can generate input for simulators at a functional level, a switch level or at a circuit level. The latter two levels are generated from the topographical database but to produce functional level information requires support from the Constraint Library.

There are two stages in generating the functional level simulator files, the first produces macros which describe the behaviour of each of the cells, the second produces the interconnect net. Both macros and net generation are dependent on the simulator: most simulators have their own individual specification language. Obviously, for each simulator there must be a net generator and, within the cell definition, a macro. These macros are specified using the `TransferEqn` statement, by stating which simulator the macro is for and the name of a file where the macro can be found, in the form:

```
TransferEqn(Simulator, Filename)
```

Any number of `TransferEqn` statements may be given, one for each simulator known to the compiler. The compiler will allow null strings to be used in both fields of the statement.

Power Statement

The `Power` statement tells the compiler how many microwatts a cell dissipates (peak, assuming max Vdd). If a cell dissipates less than one microwatt, it should be declared as:

```
Power(0)
```

By way of safeguard, the technology file specifies an upper bound power consumption as a function of a cell's area, in addition to a chip limit. The limit is quite high: 10 Watts for the prototype library called NMOS2, but the limit is easy to alter and could be anything.

Size Statement

Size information is expressed as an x,y pair in lambda space[1]. If the `slicea` example given earlier were 120 lambda units wide by 34 units high, then the cell definition would look like:

```
Cell  slicea( {Port Information} )
   History( ... )
   TransferEqn("AUTO",".")
   Power( {Power Consumption in Millwatts} )
   Size(120,34)
   Geometry( {filename} )
EndCell  slicea
```

Geometry Statement

The `Geometry` statement indexes a file which contains the raw transistors, contacts and wires of a topological symbol produced by a layout editor. TED [Rees 83], STICKS [Dennison 84] and SCALE [Marshall 84] are the layout editors available locally, but any editor could be used so long as LEGO recognises the geometry format.

LEGO does not describe what the geometry is, only what it looks like from the outside, its ports and its size. For example, if the file `oddadderslice.top` contains the geometry of a cell called `oddslice`, then the cell definition would look like:

```
Cell oddslice( {Port Information} )
   {Rest of Cell Body}
   Geometry(Oddadderslice.top)
EndCell oddslice
```

Dynamic Cells

Up to now, we have considered how fixed leaf cells are described using `Cell` statements. To recapitulate, a cell looks like:

---

[1]One lambda unit is half the smallest linewidth available using the fabrication process described by the technology file. The lambda metric is one of convenience, more exacting applications will demand the use of a smaller unit of measure.

```
Cell cellname( {Port Information} )
   History(Date(01/04/85),Validated(No),
           Source(EUCSD:AD),Version(2))
   TransferEqn(Simulator,filename)
   Power( {Power in Microwatts} )
   Size( {an x,y pair} )
   Geometry( {a filename} )
EndCell cellname
```

We have discussed how size and port information is declared for a fixed leaf cell, but what about leaf cells produced by cell synthesisers? How could size estimates be generated for each of the following:

- RAM Generators?
- ROM Generators?
- Datapath Synthesisers?
- Folded PLA Synthesisers?
- Decision Table Synthesisers?
- Stick Cell Fleshers?

It is difficult to predict the size of a part produced by a cell synthesiser without access to a general-purpose programming language. The constructs and datastructures provided by a programming language are essential for converting equations, state tables and other complicated parameters, into personality matrices. Without the personality matrix it is impossible to predict accurately anything bearing on the topography of the end-cell. For these reasons, the reserved word `Interrogate` can be used to introduce a reference to an IMP [Robertson 80] routine for generating this information each time a generator is used. The `Interrogate` statement can be used anywhere that a literal expression is expected which would be evaluated at run time (when the part is used, as opposed to when the LEGO is parsed). For example, a PLA synthesiser is declared as follows:

```
Cell PLAexample(Interrogate(PortRoutineName))
   History(Date(01/04/85),Validated(No),
           Source(EUCSD:AD),Version(2))
```

```
    TransferEqn(DSim,Interrogate(SimRoutinename))
    Power(Interrogate(PowerRoutineName))
    Size(Interrogate(SizeRoutineName))
    Geometry(Interrogate(MainRoutineName))
EndCell PLAexample
```

The ability to call a routine in a general-purpose programming language would be of little value unless access is provided to the environment in which a synthesised module is used. Clearly, it should not be necessary to hunt through a compiler's internal datastructures to find environmental parameters, so in the prototype silicon compiler U2, a procedural interface is provided instead. The procedural interface allows cell synthesisers to explore the environment in which a cell is used. It provides a clean mechanism for looking at the personalising parameters (usually a state table, set of boolean equations or a transfer equation), finding out what the neighbouring modules are and how these are configured.

The U2 Silicon Compiler is implemented in a language that does not support mapping between strings and routines. This means that when an `Interrogate` statement is used, the SRMAP.imp module must be updated and recompiled. The recompiled module is linked to the rest of the compiler dynamically. Details, both of the procedural interface for perusing environments and of the updating, are given in the documentation at the head of the SRMAP.imp code module.

The LEGO language provides powerful composition constructs so the need to add new generators by indexing IMP routines should occur only rarely.

Cell Examples

Several examples of cell declarations have already been given. To reinforce these, two more are shown below.

```
Cell  passcell(
    passline(Type(Internal,Bidir),Demand(Wordlength),Bit(i),
             Location(6 S P 2 + 6 N P 2)),
    gate(Type(Both,Input),Strap(X),Demand(Wordlength),Bit(i),
         Location(8 W 2 D + 8 E 2 D)))
    History(Date(01/04/85),Version(1),
            Validated(No),Source(EUCSD:AD))
    TransferEqn("AUTO",".")
    Power(1)
    Size(21,21)
```

```
   Geometry(somewhere.top)
EndCell  passcell

Cell  sliceb(
   gndports(Type(External,Gnd),
            Location(30 E 4 M,
                130 E 4 M + 130 W 4 M,
                230 E 4 M + 230 W 4 M)),
   vddports(Type(External,Vdd),
            Location( 0 E 4 M,
                100 E 4 M + 100 W 4 M,
                200 E 4 M + 200 W 4 M)),
   datain(Type(External,Input),Strap(X),Demand(1),Bit(i),
          Location(68 W 4 M + 68 E 4 M)),
   load(Type(Both,Input),Bit(i),Strap(L),Demand(0),
        Location(35 S 2 P)),
   load(Type(Internal,Output),Bit(i),Strap(L),Demand(0),
        Location(35 N 2 P)),
   qout(Type(External,Output),Bit(i),
        Demand(1),Location(30 W 3 M)))
   History(Date(01/04/85),Version(1),
           Validated(Yes),Source(EUCSD:AD))
   TransferEqn("", "")
   Power(24)
   Size(56,234)
   Geometry(somewhereelse.top)
EndCell  sliceb
```

Species

Thus far, we have discussed the header that introduces a generic family of module generators and how the leaf cells which are used by that family are declared. Now we turn our attention to describing the individual module generators: the things that compose cells to form a module.

Module generators are declared by `Species` statements. There is one `Species` statement for each module generator. The species comes after the cell declarations in the body of a LEGO family. The `Species` statement states how the ports along the boundary of a module should be packaged (the ports themselves are

declared by the leaf cell declaration), what parameters are needed to personalise the module and how the module is composed from leaf cells.

A Species statement looks like:

```
Species  fulladder( {Port Unification Information} )
   Parameters( {Parameter Specification List} )
   Compose( {A Composition Phrase} )
EndSpecies  fulladder
```

The Species statement has two parts:

1. A Header consisting of the reserved word Species, the name of the species and a port list. The port list says how to unify the species ports with the cell ports that lie on the bounding edges of the module.

2. A body consisting of two statements, the first says what parameters should be given when selecting the module from a UNIT program, the second says how to compose leaf cells.

The port list will be described first, followed by a description of the two body statements: Parameters and Compose.

Species Ports

Ports are listed in the species header, using the comma as a separator. Each port has a name, a Cellport statement and an Fmax statement, so a species port list looks like:

```
Species fulladder(
   InA(CellPort(Adderslice_OperandA),
       Fmax(9)),
   InB(CellPort(Adderslice_OperandB),
       Fmax(9)),
   Cin(CellPort(Adderslice_Carryin),
       Fmax(9)),
   Cout(CellPort(Adderslice_Carryout),
       Fmax(9)),
   Out(CellPort(Adderslice_Out),
       Fmax(9)))
   {Species Body}
EndSpecies adder
```

CellPort Statement

The `CellPort` personality statement unifies a species port name with a cell port name. The cell port name is specified by a cell name followed by the name a cell port, using an underscore "_" to separate the two.

The `Bit` statement that forms part of the cell port declaration, defines the bitwise significance of every port on the leaf cells which make up a module. This information is used to reconstruct the bit range of the module. In effect, the species ports have a range declared implicitly by the `Bit` statements in the leaf cell declarations.

Fmax Statement

The `Fmax` statement declares an upper operating frequency for a port in MHz. The value of Fmax is computed dynamically so an arithmetic expression or an `Interrogate` statement can be used. For example, the adder species declared on the previous page must have had full look-a-head because for ripple carry adders, the maximum operating frequency is a function of the wordlength. A ripple carry adder with a 50nS delay per slice (hence, a 20Mhz operating limit), might be declared as:

```
Species ripplefulladder(
   InA(CellPort(Adderslice_OperandA),
      Fmax(20,Wordlength,/)),
   InB(CellPort(Adderslice_OperandB),
      Fmax(20,Wordlength,/)),
   Cin(CellPort(Adderslice_Carryin),
      Fmax(20,Wordlength,/)),
   Cout(CellPort(Adderslice_Carryout),
      Fmax(20,Wordlength,/)),
   Out(CellPort(Adderslice_Out),
      Fmax(20,Wordlength,/)))
   {Species Body}
EndSpecies rippleadder
```

Species Body

The species body consists of two statements, `Parameters`, `Compose`. The `Parameters` statement is optional, `Compose` is not.

Parameters Statement

The `Parameters` statement specifies what parameters are needed by a species when it is instanced in a UNIT program. For instance, our ubiquitous PLA synthesiser examples requires a state table. This table parameter might be declared as:

```
Species pla( {Port Unification Information} )
   Parameters(table(String, Mandatory))
   Compose( {Composition Phrase} )
EndSpecies pla
```

A Parameter declaration consists of the reserved word `Parameters` followed by a list of parameter names enclosed within parentheses. Each parameter name has two flags associated with it. The first flag, called the `pattern` flag, gives a pattern against which parameter values are matched. The second flag, the `action` flag, tells the compiler what action to take if the parameter value is missing when the species is called in a UNIT program. Consider each flag in more detail.

Pattern Flag must be either `Integer` or `String`. Integer parameters are ordinal but string parameters can be anything at all.

Action Flag must be either `Mandatory` or `Optional`. This flag tells the compiler what to do if an instance is called either with no parameter list, or with a parameter list that omits the parameter being declared. If action `Mandatory` is indicated then the UNIT program is either in error, or it must refer to another species whose parameters it does match. If action is `Optional`, then the species can create a module without it, that is, the parameter value is partially redundant.

Three more examples of parameter declarations are shown below.

```
Parameters(RegisterEqns(String,Mandatory))

Parameters(boolequations(String,Mandatory),
           drive(Integer,Optional))

Parameters(address(Integer,Optional))
```

Composition

The `Compose` statement is responsible for selecting leaf cells and composing them to form an oblong space completely covered with cells without any overlaps. A `Compose` statement looks like:

```
Compose( {composition phrase} )
```

The composition phrase takes the form:

<cell> <side> <iterations>

The LAP equivalent to a composition phrase is shown in figure 6-4.

The LAP code uses the integer variable *i* for the loop counter. In a LEGO compiler, the variables `a`, `b`, `c` through to `z` are used to denote the loop counter for the innermost loop progressing outwards. The values of these loop variables are available to the LEGO programmer in the form of function operators, thereby allowing the loop variables to be tested and the results of the test to be used to select cells as a function of their bitwise location. Consider an example:

```
Compose(((cell N 4) N 5) N 6)
```

In a LAP language, this `Compose` statement would be written as follows:

```
proc nest(integer: iterationa,
                   iterationb,
                   iterationc;
          N,S,E,W: direction;
          symbol: cell)
     var  integer: a, b, c
     for c := 0,1,iterationc-1 cycle
          for b := 0,1,iterationb-1 cycle
               for a := 0,1,iterationa-1 cycle
                    :
               repeat
          repeat
     repeat
end { of nest }
```

```
proc compose(integer: iterations;
             N,S,E,W: direction;
             symbol: cell)
    var  integer: last edge,i
    symbol(cell_name)
        { In LEGO, the origin of a cell is }
        { in the bottom left hand corner. }
        last edge := 0
        for i := 0,1,iterations-1 cycle
            if direction = N then
                instance(cell,0,last edge)
                last edge := last edge + height(cell)
            elseif direction = S
                instance(cell,0,last edge)
                last edge := last edge - height(cell)
            elseif direction = E
                instance(cell,last edge,0)
                last edge := last edge + width(cell)
            elseif direction = W
                instance(cell,last edge,0)
                last edge := last edge - width(cell)
            fi
        repeat
    endsymbol
end { of procedure }
```

Figure 6-4: LAP version of a simple composition phrase

The following composition statement generates a vertical column of 8 `slice` cells.

```
Compose(slice N 8)
```

Evaluation of the composition phrase results in a composite leaf cell to allow composition phrases to be nested. For example, an (8 x 8) array of `systolic` cells would be:

```
Compose((systolic E 8) N 8)
```

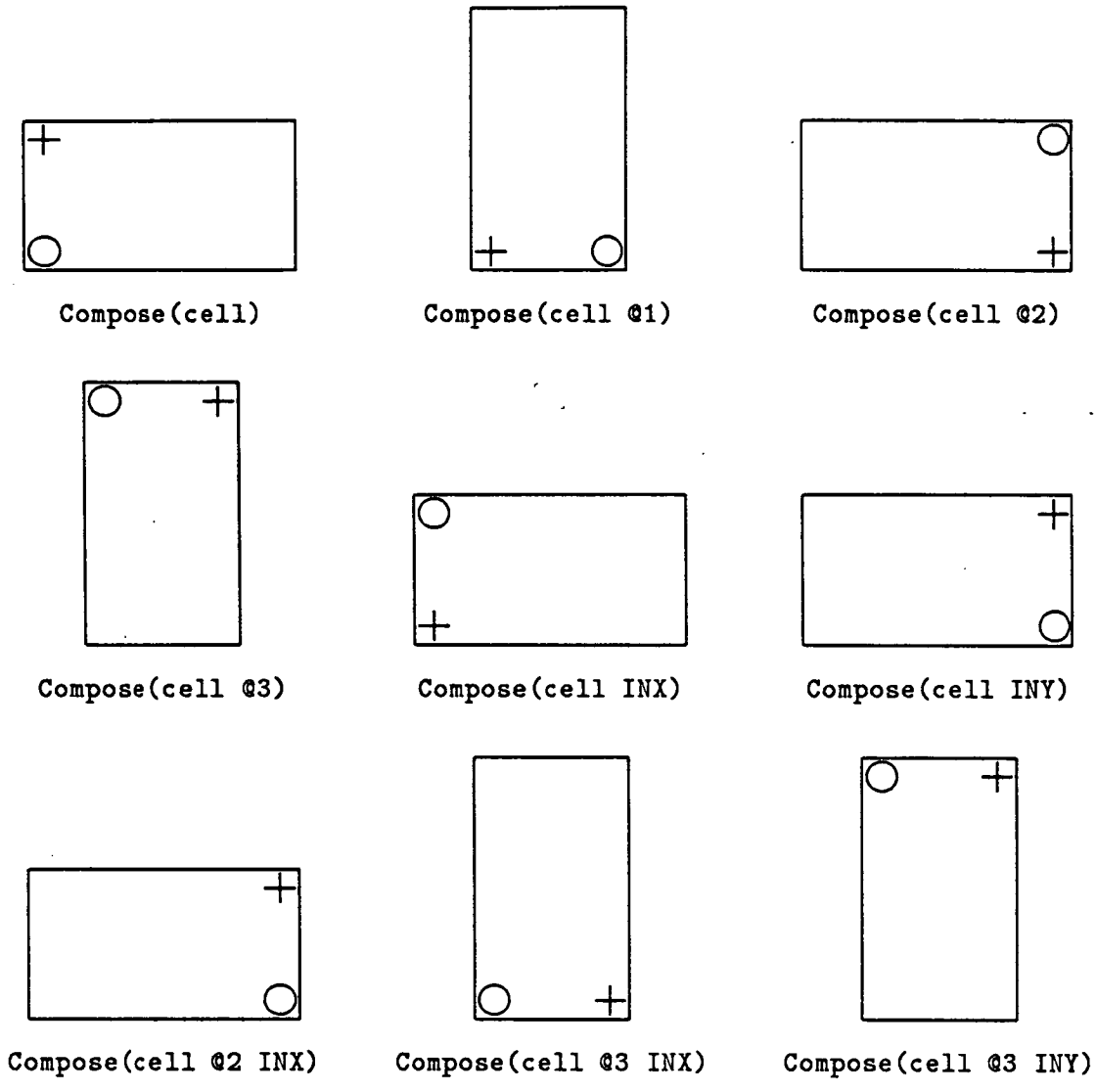


Figure 6-5: Examples of Rotation and Reflection

Transformations

Cells can be rotated anticlockwise or reflected in an axis. To rotate a cell, the symbol @0, @1, @2 or @3 is placed after the cell name to signify rotation by zero, one, two or three quadrants respectively. To maintain compatibility with earlier versions of LEGO R0, R1, R2 and R3 are kept as synonyms for @0, @1, @2 and @3.

Reflection is indicated by the reserved words INX and INY, for reflection of

a cell in the x and y axes respectively. Reflection is performed after rotation where the two operations are combined.

`Compose` statements for the most frequently used cell transformations are given in figure 6-5.

Conditional Composition

Two examples of a `Compose` statement have been given already: the first built a linear vector of `slice` cells, the second a square matrix of `systolic` cells. In reality, few applications use the same cell to tesselate a large area. Often, one cell is used for odd numbered slices and another cell for even numbered slices. In LEGO, this is done by testing the loop variables by a conditional expression and using the result of the test to select a cell.

Consider a counter made of odd and even slices. This counter could be composed by:

```
Compose(([a,Odd]oddslice, evenslice) N Wordlength)
```

The `[a,Odd]` part of the statement is a conditional expression. The expression is evaluated by pushing a (the loop counter for iterating from 0 to wordlength-1), onto an evaluation stack and applying the conditional operator `Odd` to test if a is an odd number. If the result is true then the cell `oddslice` is picked. If false, the next cell in the list is taken and the condition evaluated. Note that in the composition example, the cell `evenslice` is always chosen when the `oddslice` test fails. In a procedural language, this cell test would look like the following program:

```
proc select cell(symbol: oddslice,evenslice; ...
                 integer: wordlength)

    var  integer: a

    for a := 0,1,wordlength-1 cycle
                :
        if odd(a)=true then pick(oddslice)
        else pick(evenslice)
        fi
                :
    repeat
end { of procedure }
```

More Examples

Figure 6-6 gives examples of `Compose` statements for four other common cell configurations.

Composition Rules

There are several rules with which all leaf cells and compositions of leaf cells must comply. These are:

1. If a module uses power ports, then each power net must appear on at least two different sides of an assembled module. This is to ensure that the power net is always routable.

2. Externally connected ports should be separated by at least twice the minimum grid for metal to diffusion contacts, otherwise the routing channels may be inefficient. The compiler produces the most compact channels when the ports lay on a grid twice the minimum contact-to-contact grid, with the grid aligned with the sides of the cell. For the prototype NMOS2 library, twice the minimum grid is 14 lambda. A warning is posted if this requirement is not met. This rule and the rule given below, do not apply to power ports `Vdd` and `Gnd`, nor do they apply to `Internal` ports.

3. Cells may not have more external ports per side than is possible if they were placed two contact-to-contact grid units apart, starting and finishing one grid unit away from the corners.

Composition Checks

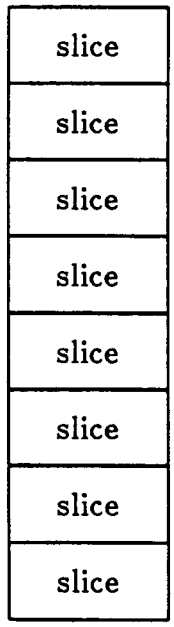
In LEGO, the result of a composition phrase is a module made up of leaf cell(s). The cell assembler responsible for compiling the modules checks that:

1. There are no gaps or missing cells within a bounding box formed by the outermost leaf cells: all of the module area must be covered. If an empty space is intended, then it must be declared as a cell without any geometry and the warnings switched off to stop messages about the bounding boxes declared not matching geometry.

2. No two leaf cells overlap. If it is intended that the leaf cells do overlap, then the bounding box in the LEGO description of each cell must be adjusted.

Vertical Stack of eight slices

`Compose(slice N Wordlength)` for `Wordlength = 8`



Cell matrix

`Compose((systolic slice E Wordlength) N Wordlength)` for `Wordlength = 4`

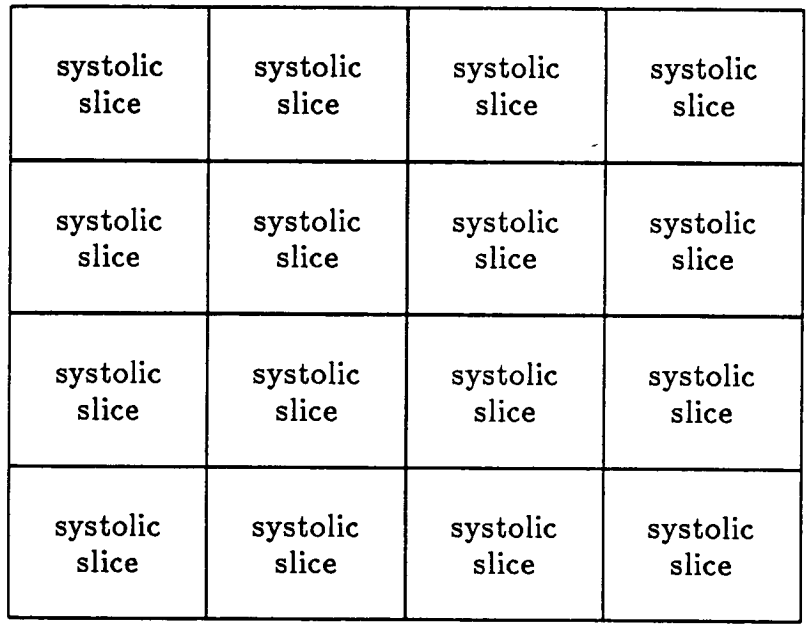


Figure 6-6: Composition Examples

Matrix with 'diagcells' across the rising diagonal

```
Compose(((([a,b,=]diagcell,normcell) E Wordlength) N Wordlength)
        for Wordlength = 4
```

norm cell	norm cell	norm cell	norm cell	norm cell	norm cell	norm cell	diag cell
norm cell	norm cell	norm cell	norm cell	norm cell	norm cell	diag cell	norm cell
norm cell	norm cell	norm cell	norm cell	norm cell	diag cell	norm cell	norm cell
norm cell	norm cell	norm cell	norm cell	diag cell	norm cell	norm cell	norm cell
norm cell	norm cell	norm cell	diag cell	norm cell	norm cell	norm cell	norm cell
norm cell	norm cell	diag cell	norm cell	norm cell	norm cell	norm cell	norm cell
norm cell	diag cell	norm cell	norm cell	norm cell	norm cell	norm cell	norm cell
diag cell	norm cell	norm cell	norm cell	norm cell	norm cell	norm cell	norm cell

Matrix with 'diagcells' across both diagonals

```
Compose(((([a,b,=]diagcell,[b,Wordlength,1,-,a,-,=]diagcell,normcell)
        E Wordlength) N Wordlength) for Wordlength = 4
```

diag cell	norm cell	norm cell	norm cell	norm cell	norm cell	norm cell	diag cell
norm cell	diag cell	norm cell	norm cell	norm cell	norm cell	diag cell	norm cell
norm cell	norm cell	diag cell	norm cell	norm cell	diag cell	norm cell	norm cell
norm cell	norm cell	norm cell	diag cell	diag cell	norm cell	norm cell	norm cell
norm cell	norm cell	norm cell	diag cell	diag cell	norm cell	norm cell	norm cell
norm cell	norm cell	diag cell	norm cell	norm cell	diag cell	norm cell	norm cell
norm cell	diag cell	norm cell	norm cell	norm cell	norm cell	diag cell	norm cell
diag cell	norm cell	norm cell	norm cell	norm cell	norm cell	norm cell	diag cell

Figure 6-6, continued

3. A connection is declared implicitly whenever two or more ports touch. Each connection is checked for type. This check detects:

   (a) floating ports, *i.e.* input ports that do not connect with the edges of the bounding box and are not driven internally.

   (b) ports shorted to Gnd or Vdd.

   (c) short circuits between two or more output ports.

   (d) ports aligned by less than the minimum linewidth for the lowest conductivity mask layer common to both ports.

   (e) ports aligned but on different and unconnected layers.

   (f) ports touching another port of incompatible type.

Many VLSI design languages compose cells by establishing a set of port constraints. For example, given three leaf cells `a`, `b` and `c`, it is possible to compose these by declaring the connections between ports and allowing the module assembler to deduce that `a` is to the left of `b` and that both `a` and `b` is to the left of `c`. These linear constraints declare port connections explicitly and positional information implicitly. The Small-SCALE language [Marshall 84] and the BLOB datastructure [Cownie 83] are examples of systems which do this. The opposite approach is taken in the LEGO language: Cell locations are declared explicitly and the port relations are implicit.

Implicit port association is used because it enables more errors to be detected than is possible when port connections are declared explicitly.

Cell Flexibility

The geometry produced by the compiler may not look even remotely like the arrangement declared by the `Compose` statement because the compiler is free to manipulate the composition to meet aspect ratio constraints. The compiler does this by first composing the cells as directed by the composition algebra and then extracts a net list. The compiler can then reorganise the layout by replacing abutted connections by box routing.

In general, the compiler will keep as close as possible to the layout specifed by the layout algebra, simply because abutted connections are cheaper than river routing.

Major recomposition is only undertaken when the assembled module is too big to fit on the chip or if the aspect ratio differs from the ratio wanted by the compiler by a factor greater than three. The theoretical basis for the magic

number *three* is that Leiserson has proved that for any layout, there exists an topologically equivalent layout with a 1:1 aspect ratio that is at most three times bigger than the original [Leiserson 82]. Therefore, the cost of completely rearranging a module to produce a better aspect ratio will cost, at most, three times the area of the original layout.

Minor recomposition to create space for *feedthroughs* may happen frequently. Feedthroughs are discussed in the Chapter 8.

Dynamic Species Definition

The `Interrogate` statement introduced earlier allows cells and species to be defined when they are used rather than when the LEGO is parsed. The `Interrogate` statement, in effect, provides dynamic definition facilities. Their use within a cell declaration has been described. In a Species declaration, the interrogation mechanism can be used to generate the material enclosed within the parentheses of the species header and in the `Parameters` and `Compose` statements.

There are various unpleasant side-effects if this facility is used thoughtlessly, most of which the compiler can detect easily. When the side-effect is detected, the experimental compiler U2.3 posts a (usually fatal) error message in the form:

```
***  Lib Prob:  A terse error flag
                - a less terse mesg stating the problem
```

Species Selection

Consider how the compiler interpretes some of the LEGO constructs, in particular, how the compiler selects species from a family.

In the UNIT language, parts are selected from the constraint library by `Instance` statements. These statements specify the family in which a behavioural element belongs, but do not necessarily specify which member of that family is to be used. For example, a counter called `acount` would be created by the UNIT statement:

```
Instance counter acount
```

There might be many members of the counters family: up-counters, up-down counters, loadable counters and variants of these in asynchronous and synchronous form. How does the compiler know which one to use?

It is possible to select species from a family by applying five filters in succession. Species must be allocated to families with these filters in mind.

At the start of the tests, all species which make up the members of the family are viewed as *contenders*. Each filter is applied in turn until there is only one contender left.

Two of the filters use *exclusion matrices* to discriminate between overlapping constraints. An exclusion matrix is a method for finding the best fit between a constraint vector and a matrix of freedoms. The constraints are generated from the UNIT specification, the freedoms are compiled from LEGO.

An exclusion matrix is a second order tensor for mapping port and parametric attributes into species names. Each instance of a species (created by an `Instance` statement in the UNIT language), has a set of attributes which are matched against those contained in the family attribute tensor. The species selection algorithm picks that species whose attribute vector is the minimal fit with the vector derived from the UNIT instance. An example of how these are built is given in the next chapter.

To iterate: the problem is one of selecting one species from a set of contenders. The compiler does this using five filters. The filters are:

Ports The instance has a unique set of ports. The ports filter finds the tightest fit between the ports used by the instance and those offered by each contender. For example, a counter with a load input and a data input is obviously loadable, and the converse is also true. The exclusion matrices provide discrimination between contenders even where their port demands overlap.

Fmax The maximum operating frequency demanded from each port is known. The Fmax filter removes contenders that are unable to meet the port speed requirements.

Parameters The instance has a unique set of parameters. The parameter filter finds the tightest fit between the parameters given by the instance and those demanded by each contender. The exclusion matrices provide discrimination between contenders even where their parameter sets overlap.

Environment Where there are both synchronous and asynchronous contenders, the compiler discriminates between them by searching

the environment around the instance. Synchronous modules are identified by their type Phi1 and Phi2 ports. If a parallel output of the instance connects to an asynchronous module then that instance must be synchronous. This filter works from the input pads, so modules connected to an input pad are always synchronous.

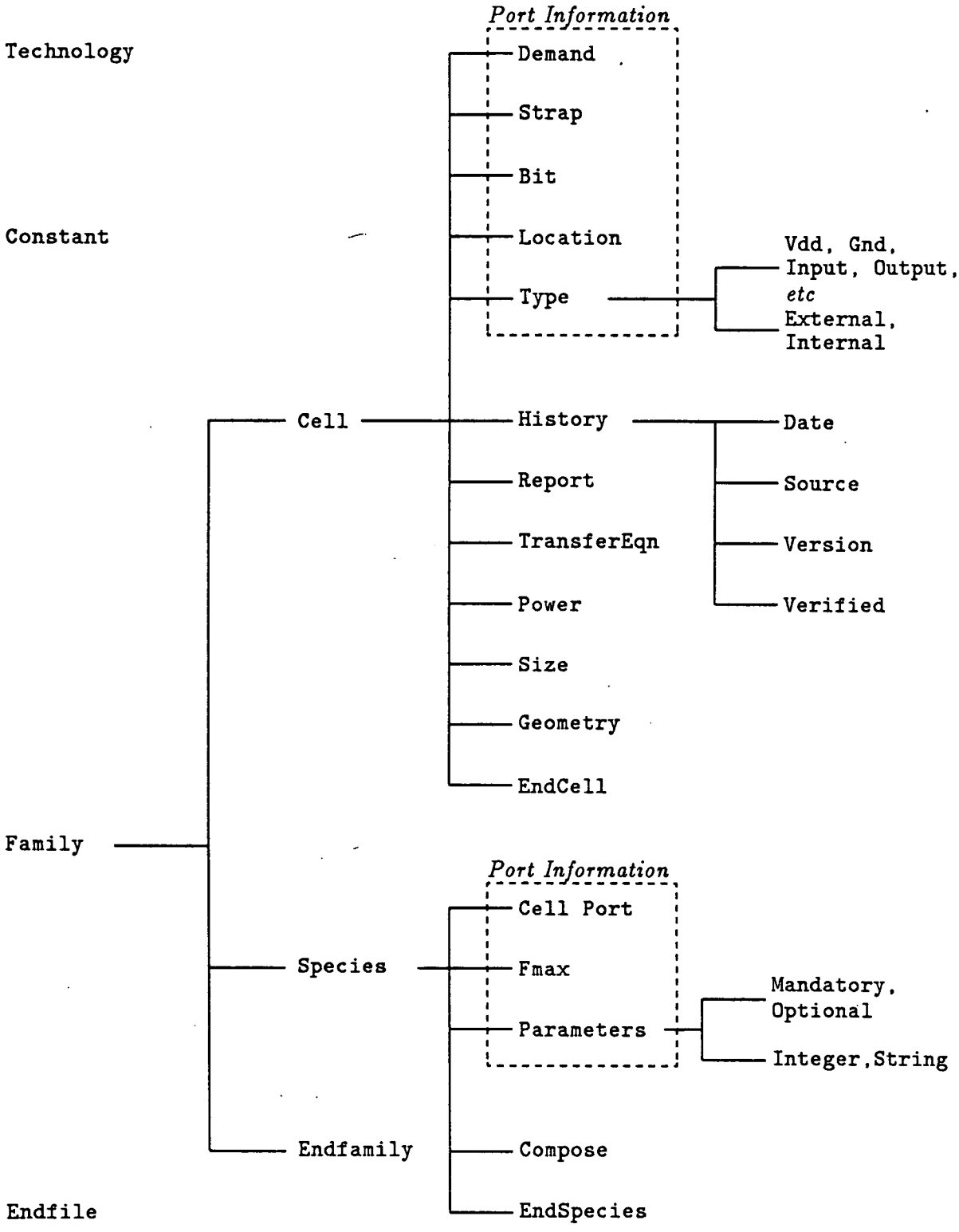
Size Contenders are ranked according to the area they occupy. The smallest contender is chosen.

Reserved Words Revisited

LEGO addresses a complex problem, that of describing design constraints to an automated VLSI CAD tool. The LEGO language is designed so that constraints can be described in precisely one way. The consequence of the former fact plus the latter rationalé is that LEGO is a large language: it contains many reserved words. You may have counted them. To simplify matters, each word is used for one purpose and no other, thereby imposing a hierarchy to the statements that make up a LEGO program, to the extent that the structure of a LEGO program might even be surmised from a knowledge of that hierarchy. This functional hierarchy is shown in figure 6-7

Summary

The LEGO language has been described. To support and reinforce this, the next chapter works through an example of how a new module family is described in LEGO.



Words available to arithmetic expressions are omitted from this diagram.

Figure 6-7: Statement Hierarchy of LEGO

Chapter 7

A LEGO example

Even after a silicon compiler has been in use for a long time, it is expected that its users will still find that the cell libraries lack parts. Perhaps a counter is wanted that can be loaded and reset. Using this counter example, this chapter demonstrates how leaf cells and collections of leaf cells are defined for inclusion in a Constraint Library. The discussion starts at the very beginning of the design process with a definition of the new counter family.

Definition

In its most basic form, a counter is a memory which increments the memory contents on receiving a clock signal. The contents of the memory are continuously available in parallel on the output.

Counters come in two main varieties: ripple counters, also known as asynchronous counters, and synchronous counters. In an asynchronous counter, the output changes ripple through each stage of the counter. In a synchronous counter, the output changes in synchronism with a change on the clock input. The difference between the two types is in the manner in which the carry is propogated: synchronous counters are faster than ripple counters because the carry chain propogates through less logic, in the extreme case, this produces full carry look-a-head.

Let us say we want a ripple counter built using series connected flip flips.

A Counter Family

An up counting, loadable, resetable counter is required immediately, but the other counters in the family have to be considered at this conception stage in order to keep the family port conventions consistent. It is thought that a family consisting of a simple counter, a loadable counter, an up-down counter and a loadable up-down counter should satisfy everyone.

Port Conventions

We have an idea of what each member of the counter family must do. The next step is to build a port *exclusion matrix*. An exclusion matrix indicates how many of each port is needed for each of the species in a generic family. The exclusion matrix is used by the compiler to identify a particular member of the family from the ports. The family of counters do not take any parameters when instances of the family are created, so it must be possible for U2 to determine exactly which species to use from a knowledge of what ports are used on a particular instance. The exclusion matrix for ports contains 0 when no ports are needed, 1 when one port is needed, 2 when two are needed etc., up to n when the number of ports used should match the wordlength of the counter. If a port *must* be connected, for example the clock input of a counter, then it is marked with a star. U2 forms its own exclusion matrix using these starred ports only, from the LEGO description of each species[1]. The exclusion matrix for the counter family is tabulated below.

	PORTS						
SPECIES	clk	qout	reset	load	datain	updown	borrow
std	1*	n	1*	0	0	0	0
loadable	1*	n	1	1*	n*	0	0
updown	1*	n	1*	0	0	1*	1
general	1*	n	1	1*	n*	1*	1

Figure 7-1: Exclusion Matrix for Counter Family

If the new family needs parameters when instanced, a separate exclusion matrix for parameters would have to be constructed. However, if species could be selected by ports alone, then the requirement that no two rows of the parameter exclusion matrix be identical would not be enforced.

Now that the general organisation of the counter family is decided, we can concentrate on the problem of defining the loadable counter.

Loadable Counter

The loadable counter will be a ripple carry counter composed of a series of flip-flips implemented in single layer poly, single layer metal NMOS. A block diagram of the counter is given in figure 7-2.

---

[1]Using Strap and Demand personality statements

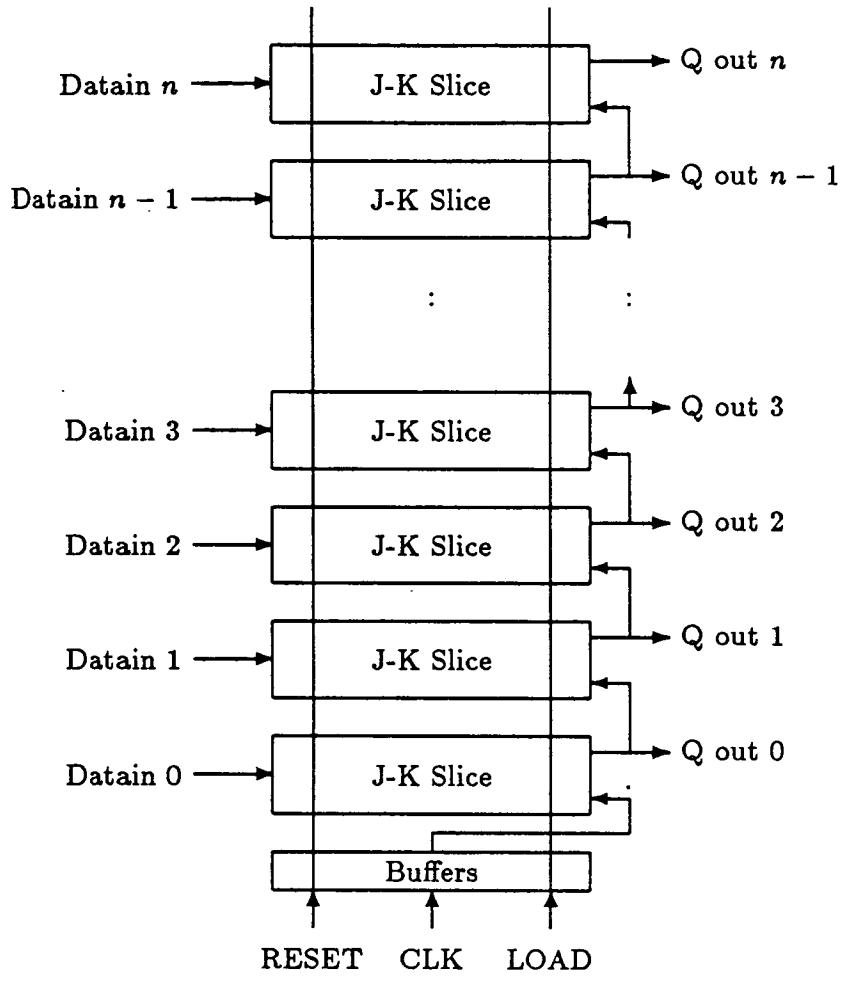


Figure 7-2: Block diagram of a ripple carry counter

Circuit Design

Enough of the loadable counter is defined for a start to be made on the circuit design. The actual circuit is not relevant to this discussion, so we shall simply assume that a counter circuit has been developed and laid out already.

Validation

It is important to validate every leaf cell by exhaustively simulating the cell for all possible inputs. It has been known for a cell as simple as the counter slice to contain five or six logical faults; so if cells are not validated, the resulting chips are unlikely to work.

Well designed leaf cells should be small in size and have few internal states, so simulating all binary inputs under all possible states is not a difficult problem.

It would be sensible to test also for the more obvious metastable conditions caused by synchrony between input lines, but in this case exhaustive testing is not feasible; discernment needs to be used in selecting test cases.

To validate the leaf cells, we build a 2 bit counter by connecting together two slices and a control part using a layout editor. In the Edinburgh CAD environment, SPICE and SDL files are extracted from the layout using the program EXTRACT. Simulation patterns are contrived and then run.

Port Positions

The next step is to encode the cell in LEGO. This involves finding the port positions.

Modern CAD systems label ports explicitly and so the port information can be obtained directly from the geometry file. However, at Edinburgh the layout system in most widespread use does not carry port information but the extraction tools produce a list of all geometry that abuts against the cell periphery. For readers not familiar with the Edinburgh tool set, an extractor comment for a two bit counter is shown below (abridged).

```
** EDGE INFORMATION *********************************
*    NODE  *    EDGE    *   LAYER   *    WHERE      *
*     1    *    EAST    *   METAL   *  149  153     *
*     1    *    WEST    *   METAL   *  149  153     *
*                        ...                        *
*                                                   *
*     1    *    WEST    *   METAL   *    1    4     *
*   208    *    EAST    *   METAL   *  136  139     *
*   207    *    EAST    *   METAL   *   68   71     *
*   208    *    WEST    *   METAL   *  136  139     *
*   207    *    WEST    *   METAL   *   68   71     *
*   130    *   SOUTH    *    POLY   *   38   40     *
*   188    *   SOUTH    *    POLY   *  127  129     *
*                        ...                        *
*                                                   *
*   111    *   SOUTH    *    POLY   *    3    5     *
*   100    *    WEST    *    POLY   *  146  148     *
*    98    *    WEST    *    POLY   *   78   80     *
***************************************************
```

Regardless of how the port information is obtained, the end product should be a picture of a cell with the port types and positions highlighted. We call this picture the *cell bounding box diagram.* The bounding box diagram for the two types of cell making up the counter corresponding with the unabridged EXTRACT listing are shown in figures 7-3 and 7-4. To avoid cluttering the diagrams with distracting details, only a few of the port labels are shown. In a real example, all the ports must be labeled.

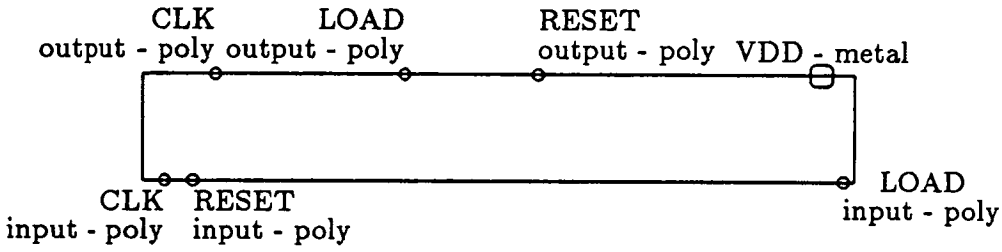


Figure 7-3: Bounding box of buffer (Scaled)

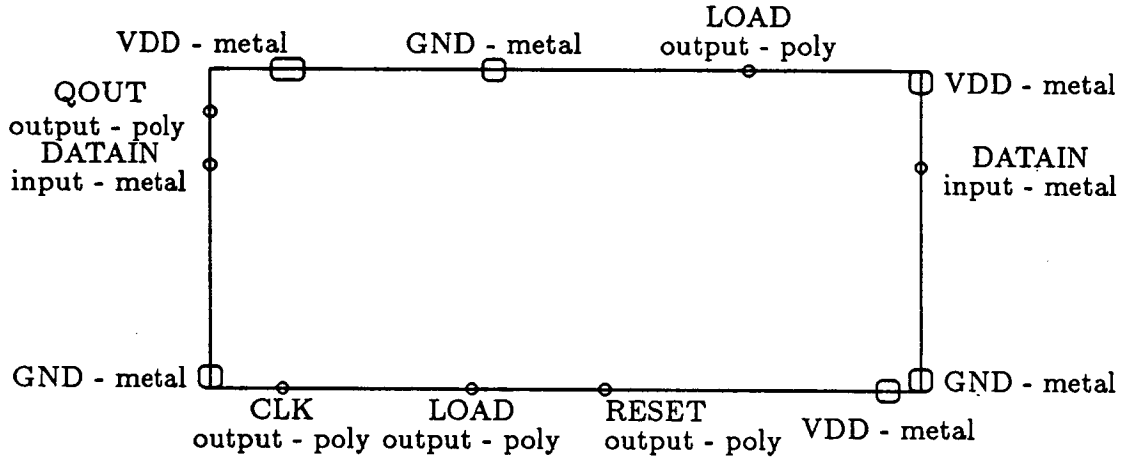


Figure 7-4: Bounding boxes of J-K slice (scaled)

LEGO Program

The LEGO program that follows was written directly from the bounding box diagrams.

```
Family counters
   Cell  loadableslice(
      botgnd(Type(External,Gnd),
         Location(1 W 4 M + 1 E 4 M)),
      topgnd(Type(External,Vdd),
```

```
      Location(62 W 4 M + 62 E 4 M)),
   datain(Type(External,Input),Bit(i),Strap(X),
      Demand(Wordlength),
      Location(49 W 2 P + 49 W 2 P)),
   out(Type(External,Output),Bit(i),Demand(1),
      Location(59 E 2 P)),
   clkin(Type(Internal,Input),
      Location(13 S 2 P)),
   resetin(Type(Internal,Input),
      Location(46 S 2 P, 70 S 2 P)),
   loadin(Type(Internal,Input),
      Location(97 S 2 P, 70 S 2 P)),
   clkout(Type(Internal,Output),
      Location(13 N 2 P)),
   resetout(Type(Internal,Output),
      Location(46 N 2 P, 70 N 2 P)),
   loadout(Type(Internal,Output),
      Location(97 N 2 P, 70 N 2 P)))

   History(Date(01/04/85),Validated(Yes),
         Source(EUCSD:AD),Version(2))
   TransferEqn("","")
   Power(3000)
   Size(143,68)
   Geometry(counter.sli)
EndCell  loadablebase

Cell  loadablebase(
   botvdd(Type(External,Gnd),
      Location(1 W 4 M + 1 E 4 M)),
   reset(Type(External,Input),Bit(0),Strap(X),
     Demand(1),Location(8 S 2 P)),
   clk(Type(External,Input),Bit(0),Strap(X),
      Demand(1),Location(3 S 2 P)),
   load(Type(External,Input),Bit(0),Strap(X),
      Demand(1),Location(127 S 2 P)),
   clkout(Type(Internal,Output),
      Location(13 N 2 P))
   resetout(Type(Internal,Output),
```

```
        Location(46 N 2 P, 70 N 2 P)),
      loadout(Type(Internal,Output),
        Location(97 N 2 P, 70 N 2 P)))

      History(Date(01/04/85),Validated(Yes),
            Source(EUCSD:AD),Version(2))
      TransferEqn("","")
      Power(1200)
      Size(143,19)
      Geometry(counter.bas)
   EndCell  loadablebase

   Species  loadable(
         Out(CellPort(loadableslice_out),
            Fmax(20,10,9,E,*,Wordlength,/)),
         Datain(CellPort(loadableslice_datain),
            Fmax(20,10,9,E,*,Wordlength,/))
         Load(CellPort(loadablebase_load),
            Fmax(20,10,9,E)),
         Clk(CellPort(loadablebase_clk),
            Fmax(20,10,9,E,*,Wordlength,/)),
         Reset(CellPort(loadablebase_reset),
            Fmax(20,10,9,E,*)))

      Compose(((([1,i,=](loadablebase)1 N),
               [2,i,=](loadableslice)Wordlength)1 N)
   EndSpecies  loadable
EndFamily  counters

EndFile.
```

It is as easy as that! Most of the work in describing a new cell is in the definition and simulation stages. Once a cell is layed out, it takes only a few minutes to write the LEGO cell description and just a few minutes more to compose it as a species. The program must still be checked to make sure the modules produced by the compiler do actually count, either by simulating them or by fabricating some test chips.

Chapter 8

The UNIT Language and Silicon Compilation

A silicon compiler must translate a behavioural specification into a topographical description of how that behaviour can be implemented in silicon. Using the framework described in this thesis, this means translating a UNIT program at one level of abstraction, into another program at a lower level.

The language used to represent a specification to the compiler, UNIT, was described in Chapter 5. This chapter introduces the remaining six members of the language group.

Parsing

A compilation begins with the UNIT program describing a design being parsed. The act of parsing encapsulates the source text in a *knowledge base.* A knowledge base is a database that incorporates semantic rules governing the data it holds, so the compiler could, for example, regenerate the original program from the knowledge base.

Knowledge Bases

Two knowledge bases are needed: one for storing UNIT programs, the other for storing design constraints. These two knowledge bases are known as the *Goal Base* and the *Constraint Library* respectively.

Initially, the Goal Base contains a specification of the desired chip in the form of an architectural description. During compilation, the compiler adds to the Goal Base in discrete stages until the Goal Base contains enough information to generate topography. At each stage the Goal Base may be edit-ed to modify decisions made by the compiler, or it can be dumped in one of the UNIT languages.

Unlike the Goal Base, the Constraint Library changes very little during the compilation. Its job is to store the technology dependent information specified by the LEGO file indexed from the UNIT program. The Constraint Library

supplies the compiler with rules: rules for picking the best synthesiser for any given situation, rules about where ports can go, rules that say how much stretch is allowed, rules that tell the compiler what to do with floating inputs, and so on. The Constraint Library is intimately tied to the target technology, hence different target technologies use different libraries. Any one copy of a compiler might have a library for an NMOS process, several libraries for various CMOS processes and perhaps a couple of libraries in another technology. When two or more NMOS processes are available, a separate Constraint Library exists for each.

This chapter is concerned with the Goal Base: what it contains and how it is modified during the course of a compilation.

Telescopic Languages

The dynamic nature of the Goal Base means that it describes not one, but a range of languages. In fact, there are seven UNIT languages, each describing a different level of design abstraction. The languages are enumerated below.

1. ROOT UNIT. This language is described in Chapter 5. A specification written in ROOT UNIT is known as the *Root Program.*

2. VALID UNIT

3. DEFINED UNIT

4. STRUCTURED UNIT

5. FLOOR UNIT

6. SLICED UNIT

7. PADDED UNIT. The PADDED UNIT program represents a topography. A specification that has been translated into PADDED UNIT is called the *Goal Program.*

The languages are all mutually consistent. That is, one UNIT program may mix different levels of language and the compiler will determine automatically which level is being used. In effect, the language definitions are nested inside one another. An analogy exists between these nested languages and the sections of a telescope, hence the seven languages that comprise UNIT are collectively termed the *UNIT Telescopic Language.*

This concept of a telescopic language is a powerful one: It allows designs to be compiled over a number of sessions rather than in one sitting, and it provides a mechanism by which a user can modify decisions made by a silicon compiler to introduce constraints the compiler does not know about, such as temporal behaviour, limits on power consumption or a pad ordering.

Compilation

The job of a silicon compiler is to translate a language low on the scale of languages (with a low number in the enumeration), into a program in a language high on the scale. A program in the language on highest rung of the scale, represented by PADDED UNIT, can be fed into a library of utilities to spawn geometry (CIF) or input files for timing verifiers and simulators.

Generally, there is one language for each compilation step. The exception is SLICED UNIT which covers five compilation steps. The correspondence between compilation steps and languages is illustrated in figure 8-1. The compilation steps are logically separated into two phases: compiling the original behaviour into a structure and then compiling the structure into the topography. The first phase is the true silicon compilation, the second phase is simply silicon assembly. Consider the two phases in turn.

Phase One

The first phase of a compilation translates the behavioural specification given by the user into a structural description of a specific implementation of that behaviour. A behavioural description differs from a structural description in that it deals with generic function families and incomplete connectivity nets, whereas a structural description deals with specific instances of low-level cells and has all connectivity defined explicitly.

There are four main stages involved in translating the behavioural description into a structural description, namely:

1. Design Validation
2. Cell Definition
3. Design Analysis
4. Floorplanning

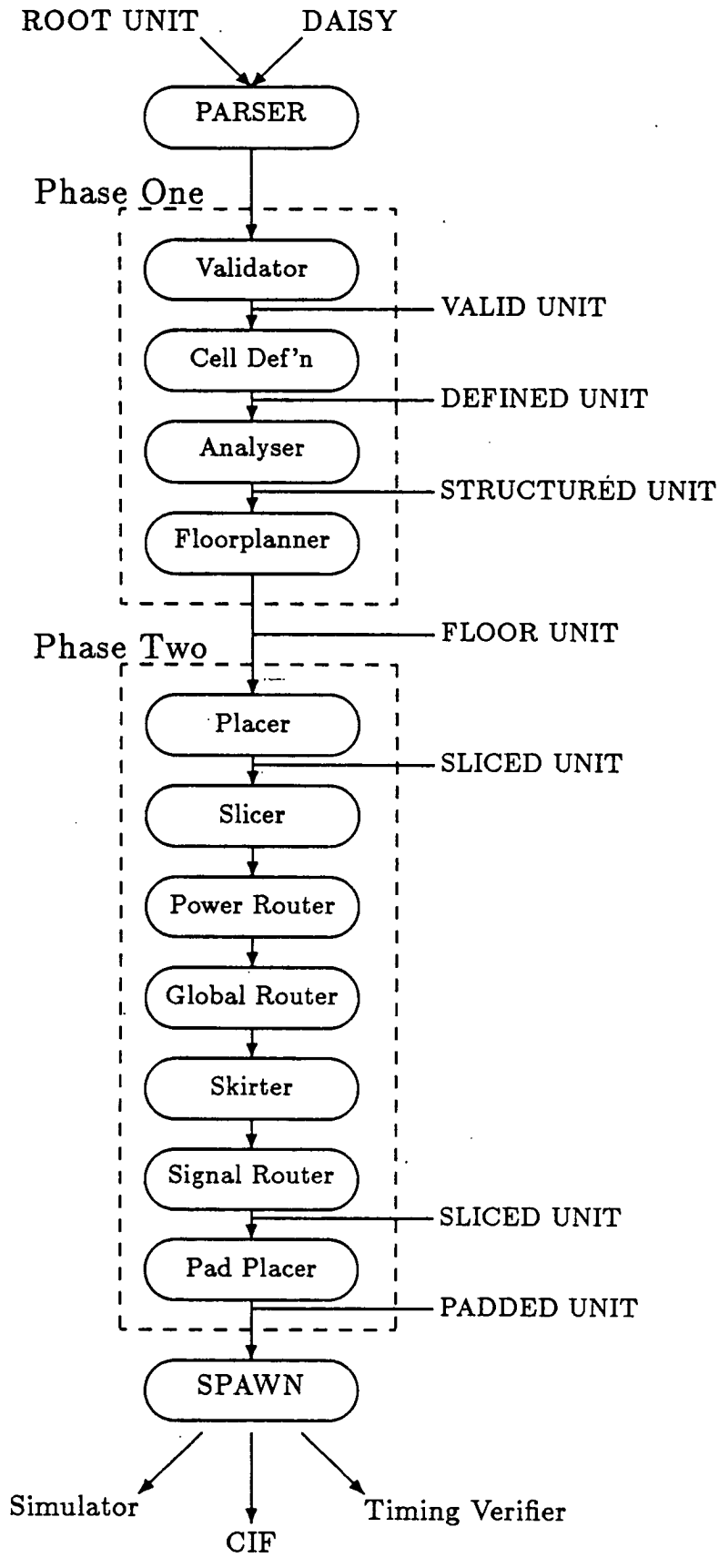


Figure 8-1: Hierarchy of UNIT Languages

The remainder of this chapter looks in detail at each of these design transformations and describes their support in UNIT.

Design Validation

While the specification is being parsed, a series of semantic checks can be carried out in an effort to catch design errors. A list of the checks U2 applies is given below, divided into two sections: Connectivity and Semantics.

1. **Connectivity**
   When the parser recognises the end of a block, the following checks on connectivity are performed:

   (a) The net of interconnected instances must be a complete graph. A Warning is posted if the check fails.

   (b) The minimum cut through a graph of interconnected instances, treating a port group as an arc of unit weight, must be greater than one. A Warning is posted if the check fails.

   (c) All ports into a block must be used within that block, that is, input ports must not be left floating. An error is posted if the check fails.

   (d) All ports into a block lower in the design hierarchy must be used. An error is posted if the check fails.

   (e) Nets must be driven by at least one source. A Warning is posted if the check fails.

   (f) Nets must drive at least one sink. An error is posted if the check fails.

   (g) All instances must have at least one input port and one output port connected. A Warning is posted if the check fails.

   (h) No net may shortcircuit two or more output ports. An error is posted if the check fails.

2. **Semantics**
   The following structural checks are carried out:

   (a) There must be no uninterrupted feedback loops: if a user really wants an oscillator, it must be designed as a leaf cell. An error is posted if the check fails.

   (b) The number of parameters needed by the Constraint Library for each instance must be given. An error is posted if the check fails.

(c) All ports must be known, either to the Constraint Library to the Goal Base as part of a block declaration. An error is posted if the check fails.

(d) If an instance uses a port for control purposes, it must be connected. An error is posted if the check fails.

(e) The set of ports used in the specification of an module in the Constraint Library must match precisely one species of module synthesiser. An error is posted if the check fails.

Warnings can be switched off by the `%Warnings` option. Errors cannot be switched off and the compiler will refuse to go onto the next compilation stage if any errors have been detected.

The VALID UNIT Language

Every compilation step in the first phase of the compilation has a corresponding UNIT language. VALID UNIT is the language used to represent a design after it has been scanned by the *Design Validator.*

The Validator takes a program written in ROOT UNIT and through a series of transactions on the Goal Base, translates it into VALID UNIT. The transactions perform the checking by way of a side effect.

VALID UNIT is identical to ROOT UNIT, except for the defaults: VALID UNIT does not have any. For example, ports defined by width in the Root Program are turned into a range definition in VALID UNIT. `Instance` statements in the Root Program that have a list of instances are expanded out into a series of statements, each declaring a single instance in VALID UNIT. To illustrate, if a Root program looked like:

```
Module multiplier(Input Operanda(8), Operandb(8),
                  Output Product(16))

    {Nested blocks}
        :
    Instance adder add1,add2
    {Further Instance Declarations}
        :
    {Connectivity}
        :
EndModule multiplier
```

After validation it would look like:

```
Module multiplier(Input Operanda(0:7), Operandb(0:7),
                  Output Product(0:15))
        :
    {Nested blocks}
        :
    Instance adder add1()
    Instance adder add2()
    {Further Instance Declarations}
        :
    {Connectivity}
        :
EndModule multiplier
```

Cell Definition

A program written in VALID UNIT contains very little structural information, if any at all. The first step in producing this information is to determine which module synthesiser is going to produce topography for each function block. The cell definition step selects a species of module synthesiser from the family of synthesisers specified in the Root Program. That is, the cell definition step chooses which synthesiser will produce each leaf module in the design hierarchy.

The specification contains a reference to a generic group or family of module synthesiser. This is enumerated to specify a precise synthesiser in the family of synthesisers. A precise synthesiser is known by generic name plus a species name. For example, an instance declaration might look like:

```
Instance familyname  instancename
```

After cell definition, it would become:

```
Instance familyname.speciesname  instancename
```

Once the species name has been determined, the wordlength of each of the instances and the ports on the bounding edges of that instance, can be added to the instance declarations. The `Module` headers in DEFINED UNIT are coerced into the same notation, thereby keep the language consistent as well as preparing the way for the compilation stages that follow.

By way of an example of a program in DEFINED UNIT, the program given in the foregoing section would be transformed into:

```
Module multiplier(Ports(
   Operanda(Range(0:7),Dir(Input),Fmax(1)),
   Operandb(Range(0:7),Dir(Input),Fmax(1)),
   Product(Range(0:15),Dir(Output),Fmax(1))
   ))
      :
   {Nested Blocks}
      :
   Instance adder.full add1() [Wordlength(8),
        Ports(a(Range(0:7),    Dir(Input),  Fmax(1)),
              b(Range(0:7),    Dir(Input),  Fmax(1)),
              cin(Range(0:0),  Dir(Input),  Fmax(1)),
              sum(Range(0:7),  Dir(Output), Fmax(1)))
              ]
   Instance adder.full add2() [Wordlength(8),
       Ports( {port info like that above} )
       ]
   {Further Instance Declarations}
      :
   {Connectivity}
      :
EndModule multiplier
```

Notice that an instance port list includes all *input* ports for an instance, but only those *output* ports which are connected. To illustrate: the full adder add1 has a carry output but it is not in the port list because it is not used whereas the carryin (cin) port is included whether it is used or not. This is done to allow the second phase of the compiler to disable unused inputs by connecting them to power and ground.

Design Analysis

The Design Analysis step restructures the design hierarchy so that each module block contains one cluster of strongly interconnected instances. The resulting hierarchy is then classified into one of six classes.

In terms of Goal Base transactions, the Design Analyser compiles DEFINED UNIT into STRUCTURED UNIT. Invariably, a program in STRUCTURED UNIT will have more levels of hierarchy than the original. Also, the header of a STRUCTURED UNIT program will include the class of the block. By way of

an example, the DEFINED UNIT program given in the previous section would be transformed into the following STRUCTURED UNIT program:

```
Module multiplier(Class(2),Ports(
   Operanda(Range(0:7),Dir(Input),Fmax(1)),
   Operandb(Range(0:7),Dir(Input),Fmax(1)),
   Product(Range(0:15),Dir(Output),Fmax(1))))
      :
   {Nested blocks, perhaps structured differently}
      :
   {Instance Declarations as per DEFINED UNIT}
      :
   {Connectivity}
      :
EndModule multiplier
```

Floorplanning

The floorplanner computes the aspect ratio and a virtual grid position for every block in the design hierarchy and then fixes the location of ports that exist along the edge of each instance. This information is incorporated into the Root Program by enumerating instances and adding module declarations. Instances have the personality statements `Posn` and `Size` added to them, module declarations only have a `Size`. A module block with these statements is deemed to be in the FLOOR UNIT language.

The floorplanner determines where ports will be by composing and perturbing the bounding boxes of the cells that constitute an instance. The port locations are added to instances and module blocks using `Location` statements.

A post-processor to the floorplanner connects floating inputs to either `Gnd` or `Vdd` by adding connectivity statements to each module.

The example given in the previous section might be translated into the following program in FLOOR UNIT:

```
Module multiplier(Class(2), Size(647,3520),
   Ports(
      Operanda(Range(0:7),Dir(Input),Fmax(1),
         Location(
         0(12W4M + 12W2P),1(20W4M + 20W2P),
         2(40W4M + 40W2P),3(60W4M + 60W2P),
```

```
      4(80W4M + 80W2P),5(100W4M + 100W2P),
      6(120W4M + 120W2P),7(140W4M + 140W2P))
      ),

   Operandb(Range(0:7),Dir(Input),Fmax(1),
      Location(
      0(160W4M + 160W2P),1(180W4M + 180W2P),
      2(200W4M + 200W2P),3(220W4M + 220W2P),
      4(240W4M + 240W2P),5(260W4M + 260W2P),
      6(280W4M + 280W2P),7(300W4M + 300W2P))
      ),

   Product(Range(0:15),Dir(Output),Fmax(1),
      Location(
      0(  8E4M +   8P2P), 1( 18E4M +  18E2P),
      2( 38E4M +  38E2P), 3( 58E4M +  58E2P),
      4( 78E4M +  78E2P), 5( 98E4M +  98E2P),
      6(118E4M + 120E2P), 7(138E4M + 138E2P),
      8(158E4M + 160E2P), 9(178E4M + 178E2P),
      10(198E4M + 198E2P),11(218E4M + 218E2P),
      12(238E4M + 238E2P),14(260E4M + 260E2P),
      14(278E4M + 278E2P),15(300E4M + 300E2P))
      ),

   Gnd(Dir(Gnd),
      Location(0(0S647M + 0W4M + 0E4M))
      ),

   Vdd(Dir(Vdd),
      Location(0(0N647M + 318W4M + 318E4M))
      )))

{Nested Blocks}

Instance adder.full add1() [Wordlength(8),
     Size(114,29), Posn(29,180,0,0),
     Ports(a(Range(0:7),    Dir(Input),
        Location({Location list like that above})),
           b(Range(0:7),    Dir(Input),
```

```
            Location({Another location list})),
               cin(Range(0:0),  Dir(Input),
            Location({Another location list})),
               sum(Range(0:7),  Dir(Output),
            Location({Another location list})))
          ]

    Instance adder.full add2() [Wordlength(8),
        Size(114,29), Posn(29,180,0,0),
        Ports( {port info like that above} )
        ]

    {Further Instance Declarations}
       :
    {Connectivity}
       :
EndModule multiplier
```

The three new personality statements `Location`, `Size` and `Posn` are described in detail below.

Location Statement

The `Location` statement introduces a list of port locations indexed by their bit significance, *e.g.* bit 3 of a range would be written as:

```
3( ... )
```

Port locations are expressed in the same way as in the LEGO language, namely, as an abscissa along a given edge, of a certain width on a particular layer. For example, a port that represents bit 7 of a range, sited 32 lambda units along the west edge of a cell (measured south to north and west to east) in 6 lambda wide diffusion would be declared as follows:

```
7(32 W 6 D)
```

In the location statement, the delimiting spaces between fields of a port position can be omitted. If this was done, the above example would look like:

```
7(32W6D)
```

Ports that appear in more than one position are declared by a sequence of port positions separated by plus signs "+". For example, if the port at 32W6D was on both diffusion and metal layers and the port ran through the cell to appear on both lateral edges, then it should be declared as:

```
7(32W6P + 32W6M + 32E6P + 32E6M)
```

In the example given above, the net is assumed to be connected within the cell. If it is not internally connected and the compiler must complete the net by connecting the two ports together, then it is declared as:

```
7(32W6P + 32W6M, 32E6P + 32E6M)
```

This auto-connect facility is used frequently for connecting up power nets on bit slice modules.

The layer letters are given in the design rule file indexed from the LEGO library file. In this report it is assumed that the rule file assigns the letter `D` to the diffusion layer, `P` to polysilicon and `M` to metal. Other files may use further letters, such as `Q` for second layer polysilicon or `T` for second layer metal.

Posn Statement

The `Posn` statement records the x coordinate, y coordinate, rotation in anti-clockwise quadrants and the mirroring of an instance. Mirroring is in the form:

```
0 for no mirror
1 for mirror in x axis
2 for mirror in y axis
3 for mirror in x and y axis
```

The convention of using the X and Y coordinates to refer to the bottom left hand corner of a cell is used, so a cell whose bottom left hand corner was at lambda coordinate 0,20 after being rotated anticlockwise 1 quadrant (90 degrees), would be declared as:

```
Posn(0,20,1,0)
```

Similarly, a cell mirrored in the x axis after being rotated 2 quadrants (180 degrees) and then mirrored in the X axis would be declared as:

```
Posn(0,20,2,1)
```

Size Statement

The `Size` statement gives the X,Y size of an instance. For example, an instance 20 lambda wide by 100 lambda high would be declared as:

```
Size(20,100)
```

Floorplan Editor

Floorplanning is one of the hardest tasks there are to automate. Optimal floorplanning is NP-Hard, with many of the subtasks NP-Complete. The first part of this thesis was dedicated to addressing the floorplanning problem, but even the methods presented there are not perfect. Simply stated, no method, manual or automatic, can produce perfect floorplans.

A human designer will often see 'obvious' ways of improving a floorplan, so how does he go about implementing those improvements? Clearly, it is possible to modify a floorplan by dumping the Goal Base and editting the FLOOR UNIT with a text editor, but this approach is fraught with hazards. The prefered method is to use an interactive graphical floorplan editor.

As a safeguard against careless use of the floorplan editor, the compiler will not allow a design to proceed to the next stage of compilation unless the floorplan is both complete and planar. This means that all blocks must be placed without any overlaps.

Code Swelling

The multiplier example used so far is getting too large for comfort. From now on, bulky sections of code will be replaced by comments.

This phenomenon of code swelling out during the compilation should be considered for a moment.

A line in ROOT UNIT corresponds with thousands of lines in PADDED UNIT[1]. Obviously, if a Root Program is 300 lines long, the equivalent in PADDED UNIT is too long to peruse with a text editor. To get over this

[1]This is because PADDED UNIT describes behaviour, structure and topography. Most of the swelling is caused by the topography.

problem, the silicon compiler must provide graphical structure editors, preferably from a menu. The editors must include at least a cluster editor, a floorplan editor and a wire editor. These tools act directly on the Goal Base to perform all the functions that could be accomplished using a text editor, but in a far safer and more controlled fashion. Thus, a user should never need to examine low level UNIT. A knowledge of what each language describes is useful only in selecting the correct editor.

Phase Two

The second compilation phase must map the structural description into a topological implementation, an operation identified earlier as *silicon assembly.*

All of the four stages involved in the first phase of a compilation take a graph, perform some computation and return dimensionless results. Using the framework presented in this thesis, the input graphs are taken directly from the UNIT parse tree and the result of each stage of computation is used to enumerate the parse tree.

Unfortunately the use of the parse tree to hold intermediate results and for holding the specification graph is greatly restricted in silicon assembly. The silicon assembler needs to operate on large volumes of topographical information without sorting through enormous quantities of data – 500,000 or more geometric objects in current designs [Ousterhout 82]. To support topographical operations on these big data sets, special datastructures are required.

The design of a datastructure must be guided by the operations that are expected to be applied to the data contained within it. If a datastructure describes topographical information, then it must store information on the size of an object and information on what it is connected to. The operations that might be applied to this combination of geometrical and topological data include:

- **Point finding:** find all the objects encompassing a given X-Y point within their periphery.
- **Neighbour finding:** find all the objects that touch one side of a given object.
- **Topological sort:** recursively find all the objects to the left of or below a given object.
- **Bloat:** move objects to allow an object to be enlarged.

- **Compact:** press objects together so that the boundary of the region occupied by objects is minimised.

- **Channel finding:** divide the space between two objects into an oblong area that can be routed easily.

Existing datastructures

Very few datastructures support *all* of the operations listed.

For example, if everything is stored in a single linked list then to find the objects covering a point it is necessary to search through the entire list no matter how the list is ordered.

A popular method for reducing the amount of searching needed is to assign the linked lists to *Bins* by mapping intervals in the X and Y axes onto a two dimensional array. Each array element is a linked list connecting all objects that intersect the same X-Y interval. Whilst Bins reduce the amount of searching needed to find a point, they do not embody the concept of nearness: To find the neighbours of an object it is necessary to search outwards in a spiral fashion moving away from the point of interest.

Using explicit neighbour pointers speeds up localised operations such as finding points quickly and, naturally, of finding neighbours. However, if one object is moved along an axis, then it is necessary to search through the entire datastructure to ensure that all the pointers in the orthogonal axis point at the correct object.

Sedgewick describes a method for storing geometrical information in trees, where successive levels of the tree are used to index alternate axes [Sedgewick 83]. All branches from the root of the tree index an X interval. All second level branches index Y Intervals. Third level branches index X intervals, and so on. The problem with this and all other tree structures is that two objects adjacent to each other will be stored in different branches of the tree. In the worst-case, adjacent objects can be in two branches that split at the very root of the tree, making the path between them as long as can be.

So this was the situation at in October 1983, when the experimental work behind this thesis was started. People were using the four datastructures mentioned but none of these were up to the job at hand. Neighbour pointers seemed to be the nearest to what was needed in that it supported more of operations we expected to apply to the VLSI database than lists, bins or trees. Nevertheless, the shortcomings of neighbour pointers was a cause for serious concern.

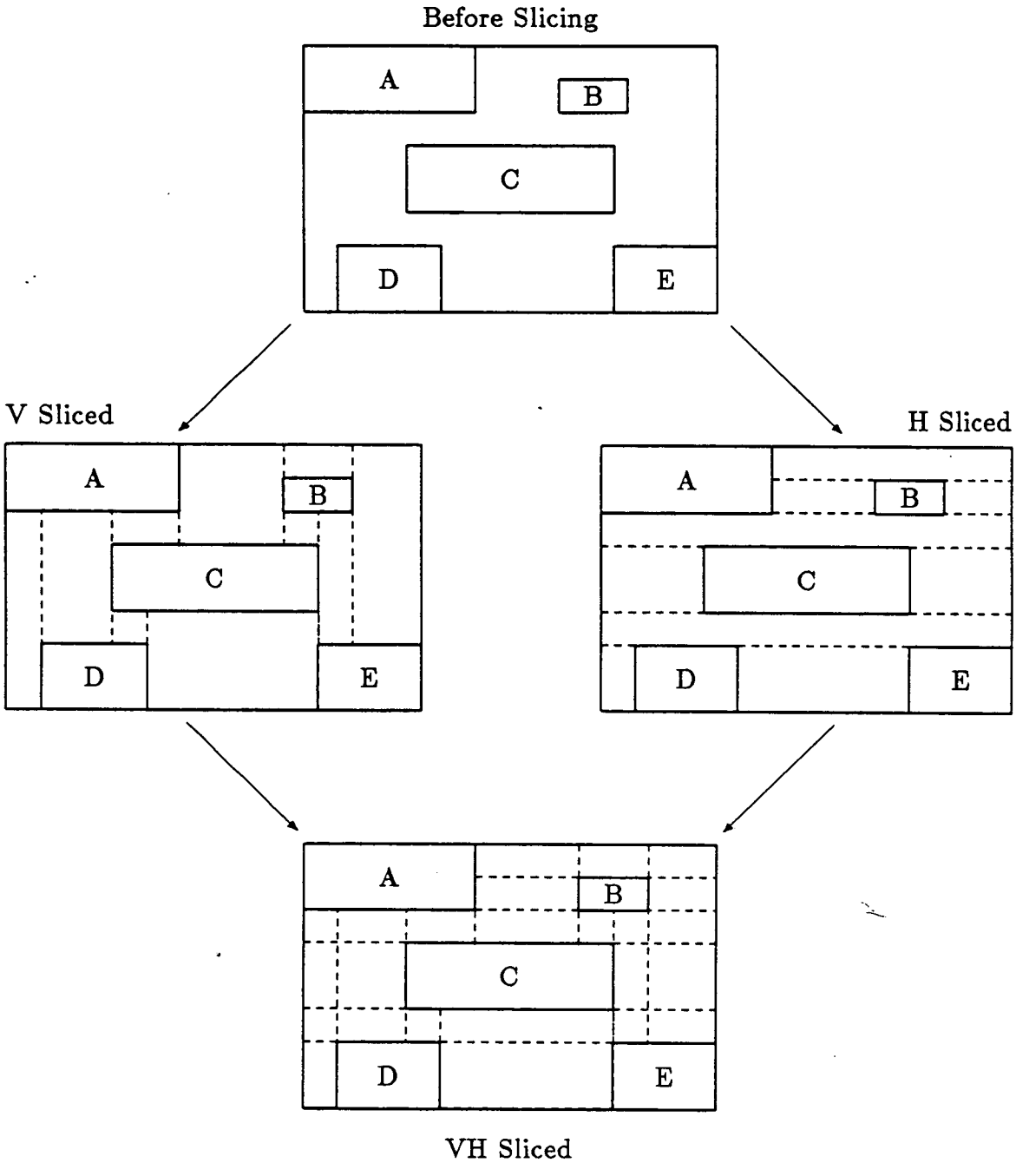


Figure 8-2: Slicing

Bloating cells is a very common transaction on a silicon compiler's database, and if every bloat triggers an exhaustive search through the database then the compiler would grind to a halt on large designs. Also, the difficulty of defining wiring spaces using the neighbour pointer scheme presents enormous difficulties when it comes to writing a global router. These problems were so serious that some variations on the neighbour pointer theme were explored.

It was obvious from the start that to avoid ambiguities in the structure, all pointers had to come from the corners of objects rather from their centres. This meant that every object in the database had four pointers, two from the north west corner and two from the south east corner, pointing at the nearest neighbours in the X and Y directions.

Just after this scheme was implemented, Ousterhout published details of his *corner-stitching* datastructure, used in the MAGIC system [Ousterhout 84], [Ousterhout 84b]. It was apparent immediately that Ousterhout's new structure did support *all* the operations listed earlier. In fact, the only disadvantage of using corner-stitching is that it consumes about eight times more space than a linked-list[2].

Corner-Stitching

Using corner-stitching, the design space is organised into planes. Each independent collection of masks are given their own plane. In Ousterhout's original scheme there were two planes, one for objects that interact with metal, the *m-plane*, and another for objects that interact with polysilicon or diffusion, *the pd plane*. Of course, contacts interact with both planes so whenever a contact is created on one plane its counterpart must be created on the other.

Ousterhout calls the objects on the planes *tiles* and a logical collection of tiles, such as a contact, a *log*. A log might be only one tile.

Tiles are linked to their neighbours by pointers at their corners, hence the name corner-stitching. What makes corner-stitching different from the simple neighbour pointers is that corner-stitching declares all space *explicitly* and *all* of a plane is covered with tiles. The design rules say that some tile types can merge with their neighbours, and space tiles are one such tile type. The merging

---

[2] A 40K transistor chip would take roughly 40MB of space in a corner-stitched datastructure if it was "flattened". That is, if the chip had the natural design hierarchy removed.

is done is so as to maximise the height of the tile[3]. This merging process means that as a design is created, it maintains the V-Sliced appearance shown in figure 8-2.

One weak aspect of Ousterhout's original corner-stitching scheme is that background design-rule checking (DRC) is done by looking for patterns at the corners of each tile. This method is slow, complicated and inhibits the adaptation of the design tool for different fabrication processes. For the silicon assembler extra planes were maintained for DRC. Whenever a tile is created on the m-plane or pd-plane, tiles are also created on one or more of the DRC planes. Tiles on the DRC plane can be bloats or shrinks of the original pd or m-plane tile and the DRC plane tiles are also typed like all other logs: The type information dictates whether one log merges or replaces another. Generally a tile can only displace logs with a lower displacement priority. Using this DRC scheme, when for instance an attempt is made to create a contact too close to another contact then the DRC tiles would interfere and the appropriate error message issued.

In addition to the pd-plane, the m-plane and the DRC planes, the silicon assembler also has a symbol plane. The symbol plane is used to capture the information on the bounding boxes of cells without analysing all the low level geometry. Tiles on the symbol plane can have child planes, allowing the datastructure to describe design information hierarchically.

What if there are two layers of metal, or multiple polycide layers? No problem. Every independent layer has its own plane, so if there are two layers of metal then there are two metal planes. In fact, the corner-stitching scheme implemented for the silicon compiler has a design rule server attached to it, allowing the number of planes, the log types, the DRC information, the drawing and CIF information, the connectivity, resistivity, interlayer capacitance and every other aspect of the design system that might possibly be affected by a change in the design rules to be set up dynamically. In this way the silicon assembler, and thus the silicon compiler, is not restricted to a particular process technology, nor is it restricted to using the conservative lambda rules systems to achieve portability. By setting up all design rules dynamically, one user of the compiler could be using, say, a bipolar technology whilst at the same time another user of the same compiler might be using the latest submicron CMOS process.

---

[3]Ousterhout's implementation maximises the *width* of tiles rather than the height, so his planes are naturally H-Sliced instead of V-Sliced. The difference is of no consequence.

Silicon Assembly

The Silicon Assembler has eight stages:

1. Placement
2. Perturb and Slicing
3. Power Routing
4. Global Routing
5. Skirting
6. Signal Routing
7. Pad Placement
8. Spawn

Placement

The Placement routine converts the virtual grid coordinates computed by the floorplanner into coordinates in lambda space.

Many algorithms for placing rectangular objects are described in the literature, for example [Khokhani 85], [Rivest 82], [Cheng 83], [Khokhani 81], [Kozawa 83], [Lauther 79] and [Preas 78]. However, the algorithm used in U2 is much simpler than any of these because in the context of the framework being discussed here, a floorplanner has already determined the relative location of each block. All that needs to be done is to plow blocks to make space for power and signal wiring. One such plowing algorithm is given by Ousterhout [Ousterhout 84], but a much simpler algorithm is possible, based on summing the X and Y displacements of neighbouring blocks. The improved method is described by Chung in [Chung 85].

Perturb and Slice

The Perturber modifies the placement to allow room for power routing and wiring channels. The perturbed placement is then sliced horizontally using the algorithm developed by Kinnear [Kinnear 85] to produce the outlines of individual channels. The slicing process is illustrated in figure 8-2.

The wiring modules created by the slicer are added to the list of modules blocks as siblings and then instanced. These wiring modules have no connectivity for the moment, so they are treated differently from other modules, (other module blocks are illegal if they have no connectivity). To distinguish these wiring modules from other modules and to prevent any name clashes, the wiring modules take the name of their parent block, suffixed by an ampersand "&" and a channel number. The ampersand is a feature of the SLICED UNIT language.

Power Routing

Power nets must be routed differently from signal nets because:

- Both Vdd and Gnd nets must be routed on a single layer of metal, without any crossovers
- The widths of the conductors must be tapered as a function of the current density
- Each net connects to every function block at least once

Power nets are routed using the method developed by Morton and described in his Masters Thesis [Morton 85]. Briefly, Morton's method involves producing two spatially separable graphs whose edges hug the boundaries of the function blocks. Morton's work extends the method first described by Hassett [Hassett 82] and independently, but in less detail, by Lie [Lie 82].

A power route is described in SLICED UNIT as parameterised instances inside the wiring modules. These parameterised instances are viewed as topological entities that connect by abutment. For example, for the wiring module marked "&6" in figure 8-3, the following statements would be generated:

```
Module name&6(Size(32,64),
   Ports(
      Gnd(Dir(Gnd),
         Location(O(4S14M + 24W14M))
         ),
      Vdd(Dir(Vdd),
         Location(O(20N8M + 3E8M))
         )))

   Instance  topography.wirem gnd1() [
```

```
        Size(14,61), Posn(4,0,0,0),
        Ports(Dir(Gnd),
            Location(0(0S14M + 0N14M + 24W14M)))
        ]

    Instance  topography.wirem gnd2() [
        Size(4,14), Posn(0,24,0,0),
        Ports(Dir(Gnd),
            Location(0(24W14M + 24E14M)))
        ]

    Instance  topography.wirem vdd1() [
        Size(8,61), Posn(20,3,0,0),
        Ports(Dir(Vdd),
            Location(0(0S8M + 0N8M + 3E8M)))
        ]

    Instance  topography.wirem vdd2() [
        Size(4,8), Posn(28,3,0,0),
        Ports(Dir(Vdd),
            Location(0(0S8M + 0N8M)))
        ]
EndModule name&6
```

Global Routing

The communications between blocks are decomposed into a series of wires allocated to specific wiring channels. The wires are allocated to the channels by adding ports to their headers and connectivity to their bodies.

Figure 8-4 illustrates how some instances can allow wires to pass through them as *feedthroughs*. The feedthroughs are inserted in the UNIT language by adding ports to instances and updating the connectivity. The new ports have names such as `name&ft0`, `name&ft1` etc.

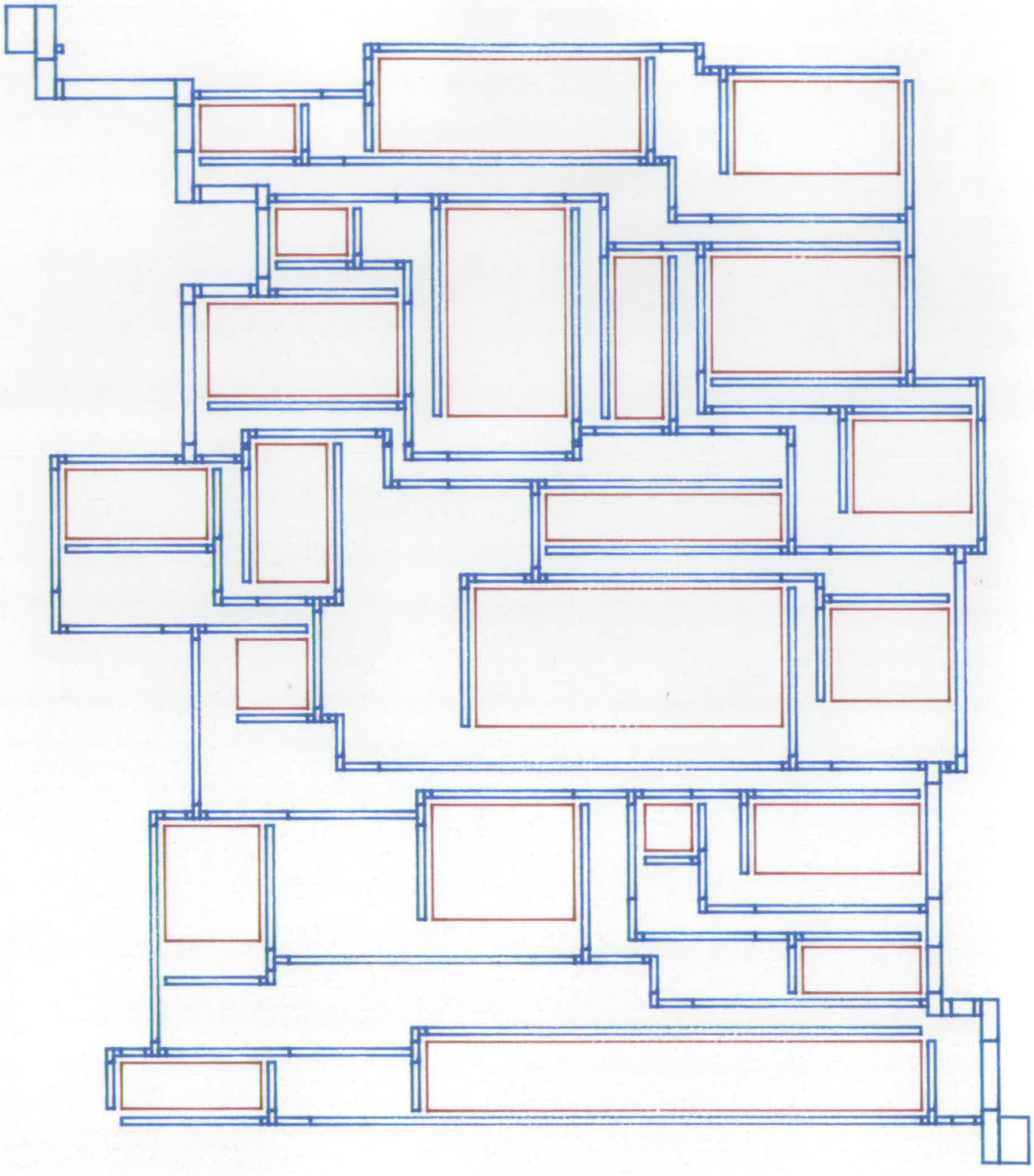


Figure 8-3: Power Routing Example

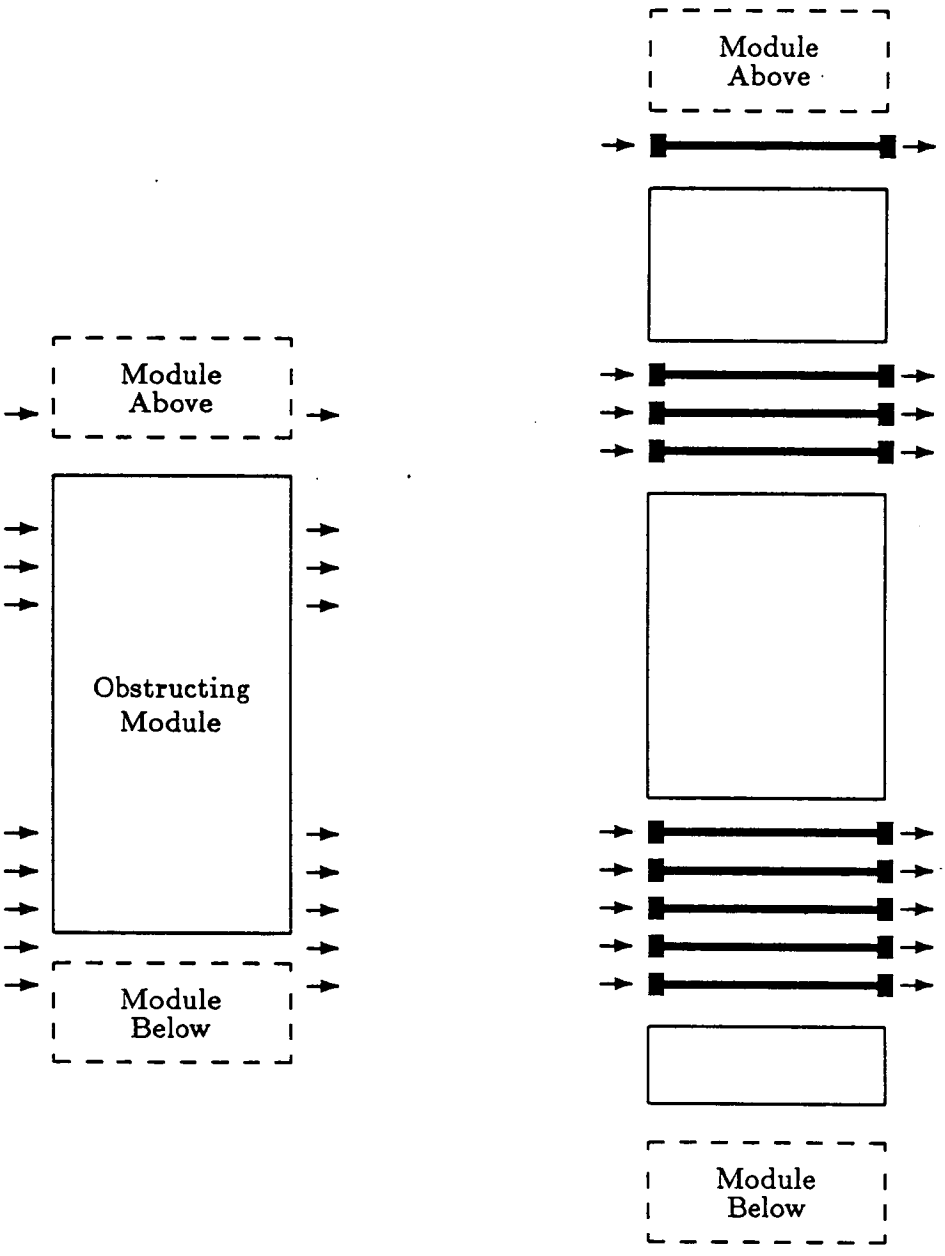


Figure 8-4: Feedthroughs

Skirting

The skirter adds river routing to the boundaries of each cell to align all ports on a grid so that as much river routing as possible lies underneath the power routing. The skirts are added as topological instances to the appropriate wiring modules.

Grid alignment is necessary even though the routers can operate in a grid-free mode, because a channel constrained by two cells each with ports very close together could cause the router to collapse. Grid routing prevents this happening. The grid unit is set by the minimum distance between two contacts, currently 7 lambda for the prototype nmos2 library. Ports may be one grid unit apart when there are no other ports on the directly opposite section of the channel, otherwise they should be 2 units apart.

An important advantage of grid-aligned routers over the grid-free routers is that a skirter removes cyclic constraints by offsetting ports on the east or north edges of a channel by one grid unit. The river routing is usually narrower than the power lines that run over the top, so rather than reducing total area efficiency by eating up useful channel space, skirting actually increases efficiency by removing cyclic constraints.

Signal Routing

Global wiring has been decomposed into wires allocated to wiring channels by the Global Router. The Global Router sees a wiring channel as an oblong box with ports on the edges. The Signal Router converts the *wiring box* into a topological entity by adding further topological instances to the wiring modules.

This means that very exacting demands are placed upon the signal router. Wires may enter a routing region from any side and they may be connected to any number of other wires in an arbitrary manner. The framework for a silicon assembler described here makes the demand that any such switchbox of wires be routed in the minimum possible space. This wiring capability did not exist at the start of the experimental work associated with this thesis so new methods had to be developed.

A group of three was formed to investigate the problems of wire routing. A channel router based on Deutsch's algorithm was implemented [Deutsch 76], Thomlinson implemented Burstein's routing algorithm [Thomlinson 85], [Burstein 82], [Burstein 83], [Burstein 83b], [Burstein 83c], [Burstein 83d], [Burstein 84], and Waring produced a switchbox router of his own [Waring 85].

The only changes made to Deutsch's algorithm involved removing the GOTOs and to make it work on sorted data. Sorting the data – an $O(T.log(T))$ operation, where T is the number of terminals – means that the actual routing achieves the lower time complexity bound in time for routing any channel using a heuristic method (almost $O(T)$).

All algorithms were implemented on the same machine and using the same programming language. To prevent the net order effecting the results it was decided to sort all nets into ascending abscissae order.

The modified Deutsch's algorithm proved to be the fastest. Burstein's algorithm was by far the slowest and produced the poorest quality routing. Waring's router achieved the initial objective by routing all benchmarks in the smallest possible space, that is, in the theoretical lower bound of space.

Burstein's algorithm achieved fame by producing what appeared to be the most compact and efficient routing for any published algorithm, and also by defining a hard switchbox routing benchmark, a benchmark that Burstein was unable to route. Burstein's algorithm has been published in many different publications.

Burstein's benchmark was first routed Malgorzata Marek-Sadowska using a rule based system specially coded to route this benchmark [Marek 85]. The only general purpose part of Marek's method, that of unique path propogation, is also used in Waring's method (developed quite independently), but Waring also uses other heuristics to route both Burstein's benchmark and the larger Deutsch's benchmark.

The four algorithms are compared with each other in the following table.

	Deutsch's Benchmark		Burstein's Benchmark	
	Tracks	Time	Completed	Time
Deutsch's router	22	1.6 Secs	Not suitable	-
Burstein's Router	19	24 IBM Secs	Fails	-
By Thomlinson	26	20 Mins on 68000	Fails	-
Marek's Router	Fails	-	In +1 column	5.2 VAX Secs
Waring's Router	19	30 Mins	In min space	5 Secs
	20	3 Mins		

Notes:

1. All timings were performed on a 10MHz 68000, except for Marek's which is the published figure using a VAX 11/780 and Burstein's 19 track result

which uses an IBM 370/3033. With the sorted input data, Thomlinson was not able to reproduce, nor even come near, the results published in Burstein's papers.

2. Thomlinson's implementation of Burstein's algorithm used two dimensional arrays and real numbers. The IMP compiler code for array accesses is up to 6 times slower than it should be, possibly due to a compiler bug, and real numbers are also inefficient – there was no floating point unit on the machines used for these experiments. However, even dividing Thomlinson's timing figures by 10, Burstein's algorithm is at least two orders of magnitude slower than the others. Also, Thomlinson was unable to get Burstein's router to reproduce the published results – Deutsch's benchmark needed 26 tracks!. Burstein admits to ordering the input data very carefully to produce his 19 track result. For these benchmarks all methods were given the same sorted data.

3. The discrepancy between Deutsch's published best (21 tracks) and the figure here (22 tracks) is probably due to Deutsch rearranging his input data to produce the best route – Deutsch mentions methods for doing this automatically in his paper.

It was stated earlier that the ability to route any possible switchbox in the minimum space was a prerequisite to the silicon assembly framework described here. To prove that this ability now exists, Burstein's largest switchbox benchmark is shown routed by Waring's algorithm in figure 8-5. Other benchmarks and a full description of the algorithm can be found in Waring's Master's thesis *ibid.*

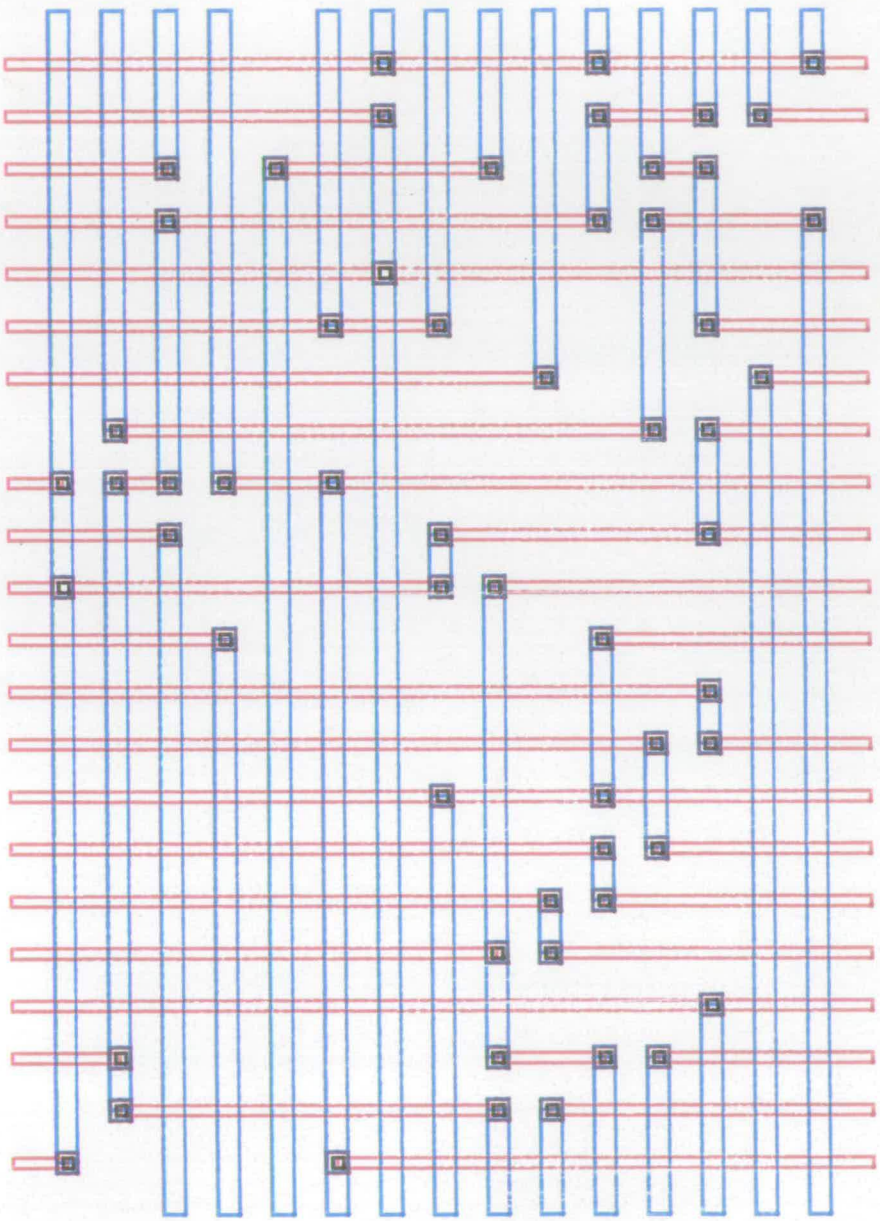


Figure 8-5: Burstein's switchbox benchmark routed by Waring

Routing Failure

Power routers, global routers and signal routers are not guaranteed to succeed first time. The channels may not wide enough on the first pass or perhaps the power routing demands too much of the channel space. If this happens, the routers can call the placement routine which will increase the channel widths to whatever level is needed and rerun the entire second phase of the compilation. Note that the placer can only increase channel width, to circumvent the classic channel convergence problem.

Cell Creation

For each reference to a module synthesiser, the Cell Creator adds a new module to the block in which the generated part was found and fills the new block with topography. Topography is declared in the same way that wires and contacts are by the routers, namely, by using species in the topography family.

At the end of the compilation there are no parts which call module synthesisers, everything is a module block. At the leaf level all modules contain topography.

From LEGO to UNIT

It is worth spending a moment discussing how the ILAP geometry provided by the LEGO library is converted into UNIT topography. The UNIT topography has ports and other fancy things that may not be in the LEGO. Where does it all come from?

The geometry indexed in the LEGO is used to produce corner-stitched log planes. U2 uses the slices inherent in the log planes to generate topological objects with ports on their edges where they abut with other logs. The ports are generated only when abutting logs share a low resistance mask layer.

Pad Placement

The Pad Placer finds the best tradeoff between bondability constraints and minimum area constraints, using channel routing to produce pad patterns specified by the user. The pad generator, in fact, just places pads as near as possible to the ports on the root block, hence changes to pad postion are effected by altering the floorplan for the root block.

The pad placer creates a module called pad&0 in the outermost module block. The compiler inserts into pad&0 the topography for the pads and for the wiring channels that connect ports on the outermost block to the pad positions.

Spawn

The compilation is complete. The high level behavioural description has been compiled into topography. Utilities now become available to allow geometry (CIF), and at some future date, files for timing verifiers and simulators to be spawned from the Goal Base. These utilities are accessed from U2's menus.

Upon finishing, the compiler will dump a table giving information on the expected yield, power consumption, the percentage of routing, the pad order, statistics about the number of warnings and errors etc. The compiler will also dump the Goal Base.

Language	Produced by	Change to previous language
ROOT UNIT	User or DAISY	–
VALID UNIT	Validator	Expansion of shorthand.
DEFINED UNIT	Cell Def'n	Insertion of species names. New port declaration format.
STRUCTURED UNIT	Analyser	Classes added to modules. Design hierarchy restructured.
FLOOR UNIT	Floorplanner	Size, Posn and Location of instances and ports determined.
SLICED UNIT	Silicon Assembly Subsystem	Floating connections grounded. Topography modules added. All lib references replaced by topography modules. Gnd and Vdd connections added. Feedthroughs added. Connectivity decomposed.
PADDED UNIT	Spawner	Pad Placement and pad block added.

Figure 8-6: Dialects of UNIT

Summary

The framework for a silicon compiler has been described in terms of a series of compatible languages to capture the intermediate stages of the translation from

a behaviour to a topography. Once physical information had been generated by the floorplanner, the framework developed around a datastructuring technique called *corner-stitching.* The corner-stitched structure incorporates the information needed to spawn mask geometry or a switch-level net-list very quickly, without the need for sophisticated extraction methods.

The use of the UNIT languages enables the compiler to be interrupted by a user, the information the user contributes to be captured at any stage of the design process and then allow the compiler to continue automatically. As figure 8-6 shows, each step of a compilation alters the Goal Base and different languages are needed to represent these changes.

Chapter 9

Very high level optimisations

The higher the level at which optimisation is performed then the greater the potential benefits.

At a low level of abstraction, it may be possible to reduce the area taken by a wiring channel, reduce the number of vias, maximise the amount of metal used *etc.* A lot of effort is involved in making this sort of optimisation and the benefits are poor. Take for example the problem of minimising the channel area. Even a very simple channel router will usually get within 4% of the theoretical minimum area and typically within 1%. A sophisticated channel router might attain the minimum area but at the cost of taking 100 times more computer time, 20 times more coding effort and three times more memory than the simple router. Even if wiring takes 50% of a chip's area, then the 0.5% reduction in the total area achieved by using a sophisticated router will have been bought at a high price.

At a slightly higher level, a crude floorplanner which lays everything out in rows may consume four times the silicon real-estate of a 'perfect' floorplanner. The perfect floorplanner does not exist and even highly sophisticated floorplanning techniques such as those described in this thesis need typically 10% more space than is really necessary. An idiomatic floorplanner takes over 100 times more computer time, and needs 50 times the coding effort of the crude row by row method. But then it does produce a 400% reduction in area.

At an extremely high level it is possible to optimise architectural information by minimising the number of function units, that is, by trading off the number of times a function unit is instanced against temporal performance criteria. Going back to the MC68000 microprocessor example, the performance criteria could be expressed as "a computer executing a given instruction set enumerated with the maximum number of clock cycles taken by each instruction". No-one would be impressed by an engineer who given this specification, delivers a 200MIP supercomputer. Likewise for a silicon compiler. The specification given to the engineer can be expressed as a maximally parallel dataflow graph in the UNIT language; but if the silicon compiler was to translate the graph directly into

silicon then it would need an acre-scale fabrication technology with submicron linewidths to manufacture the result. Yet for just a small cost in programming effort, the dataflow graph could have been optimised, reducing the chip area by many orders of magnitude.

For the three levels of abstraction touched upon, the possible benefits of optimisation have been 0.5% for the wiring channels, 400% for the floorplanner and perhaps 10000% for the architectural optimiser.

This does not mean a CAD tool builder can afford to be complacent when it comes to low level optimisation. No indeed. Throughout this thesis a great deal of attention has been given to how even the lowest level aspects of a design can be optimised and produced efficiently. As part of the experimental work described by this thesis, new methods of channel router were developed, methods far superior to those presented previously. Methods for producing perfect power routers have been developed, methods for using the normally wasted space underneath the power rails have been described. New floorplanning methods have been developed. No, there has been no complacency about making low level optimisations but it would be a shame to waste this effort by being sloppy or careless in the way very high level information is handled. The importance of making the very best use of the opportunities available to optimise very high level information has prompted this chapter to be written, to deal with methods of performing functional level optimisation and the problems of synthesising behavioural information.

Very high level specifications

At a very high level a CPU might be specified by a macro-instruction set. Of course, at an even higher level, one might start with a suite of programs that must be executed in a given time. There are some dangers in taking a specification to this ultimately high level. For instance, the machine that results may not be general purpose as intended because the suite of programs do not use particular instructions. Kean [Kean 85] mentions a number of similar problems relating to the MIMOLA system which does attempt to do this ultra-high level compilation task [Marwedel 84].

Assuming the suite of programs have been analysed by hand and an instruction set has been defined, the macro-instructions can be encoded in the UNIT language as a collection of dataflow graphs. In this form, the machine is maximally parallel: Every instruction has its own pipeline of functional operators

and the control logic enables an entire pipeline as soon as an instruction is recognised. Of course, nobody could build a machine left in this form: It would be too big and expensive. In fact, it would be even bigger than the supercomputer mentioned earlier. Some optimisation of the dataflow graphs is needed to produce the datapath section of a CPU. Once the datapath has been defined, it can be analysed to recover microcode and timing information.

Defining the datapath

An instruction set described in UNIT is a set of disjoint dataflow graphs. Each instruction is fired by a horizontal control sequence.

The first step in compiling the dataflow graphs into a chip is to merge the many disjoint graphs into one, or a few, much larger instruction graphs. The best way of describing this process is probably through an example.

Consider the SUBI (Subtract Immediate), LINK (link and allocate) and the ADDI (Add immediate) instructions of the Motorola 68000 [Motorola 82]. The dataflow and timing information for each of these instructions is tabulated below.

Mnemonic	Generation
SUBI	Destination = Imm data − (Destination)
LINK	An → −(SP); SP → An; SP + d → SP
ADDI	Destination = Imm data + (Destination)

Every instruction can work with a predefined set of data address modes. An extract from the table given in the Motorola 68000 user manual is reproduced overleaf to show that the address modes multiply the number of disjoint dataflow graphs representing the instruction set, and that many of these graph extensions use the same hardware functions as the instruction graph itself. This means that the first step in merging the disjoint graphs together is likely to be reduce the number of function operators most if it separates the addressing mechanism from the instructions themselves and then merges the instructions together whenever there are identical operators in two disjoint graphs. By applying this merge process until there are either no more disjoint graphs or until there are no more operators common to more than one graph, a large dataflow graph representing all the instructions of a CPU would be produced.

Mode	Generation
Register Direct Addressing	
Data register direct	EA = Dn
Address register direct	EA = An
Absolute Data Addressing	
Absolute short	EA = (next word)
Absolute long	EA = (next two words)
Program Counter Relative Addressing	
Relative with offset	EA = (PC) + d_{16}
Relative with index and offset	EA = (PC) + (X_n) + d_{16}

For the 68000 example, the merging process might identify the addition and subtraction operators in the generation sequences for the ADDI, LINK and SUBI instructions to produce a dataflow graph with just a single add and subtract operator instead of the three existing before. This type of interinstruction merge does not slow the CPU down at all but the second type of merge, the merging of intrainstruction operators certainly does.

Intrainstruction mergers

Intrainstruction mergers, that is, merging operators within a single instruction and then cycling the data through the combined operator in separate clock cycles, is a direct area-time tradeoff. The more merges then the more clock cycles are needed but the smaller the chip. Different merges cause different effects on the timing and different operators require differing amounts of silicon area.

Clearly, the intrainstruction merges should identify those operators which take up the most area but whose merging has the smallest effect of the temporal performance. So an optimiser must assess the value of at least two goodness factors before accepts a merge, the area that would be released by making the merge and the number of clock cycles needed to perform the merged instruction. If the number of cycles exceeds the predefined upper limit for that instruction, then the optimiser must reject the merge no matter how much area it saves. This means that the largest operators, such as multipliers, must be merged first and the smaller operators merged later.

The area advantage of a merge is trivially easy to assess, but what about the number of clock cycles?

One way of assessing the implications of a merge on the number of clock cycles is to find a proper vertex colouring of the the dataflow graph G by assigning the two operators to be merged the same colour. The number of colours

needed, $\chi(G)$, is the number of clock cycles, or the number of buses depending on what architectural assumptions are being made. This correspondance is true by virtue that the bus allocation problem, that is, the problem of making sure that no two operators attempt to use the same bus in the same clock cycle, can be reformulated as the vertex colouring problem where no two communicating vertices share the same colour. Other methods for allocating clock cycles to operators are described in [Nagle 80], [Park 84] and [Park 85].

For the colouring method to be valid, the following assumptions and restrictions must be made:

1. It is forbidden to merge adjacent operators, otherwise a a contradiction would occur in the vertex colouring.

2. The total number of control points around an operator must occupy less area than the operator itself. Unless this assumption is made, the optimiser could never merge to operators because the area taken by the control points introduced by the merge would exceed the area saved by eliminating one instance of the operator itself. For a practical optimiser, it is useful to assume that control points take up zero space.

3. A bus can carry only one signal per clock cycle.

4. Buses have intrinsic storage for one cycle, otherwise the optimiser would have to introduce extra registers.

5. Buses occupy a small but finite area. Without this assumption, the system would plaster a dataflow graph with buses.

Recovering microcode and clock information

It must be possible to recover the microcode information from the merged dataflow graph. This means a running log must be maintained from which the lineage of each merged operator can be determined. A simple way of doing this is to label the instructions and number the clock cycles. Everytime a merge is made, the instruction labels and clock numbers allocated to each of the merged operators are appended to a string attached to the new operator. For example, if the letters A, B, C, D and E are used to label five operators within a single graph which are subsequently merged into one operator, then the string might look like:

```
"A 1 B 2 C 1 B 3 C 5 E 1 D 4"
```

This says that the operator is enabled during Phi1 of instruction A, Phi2 of instruction B, Phi1 of instruction C, and so on to Phi 4 of instruction D. To find the number of clock cycles taken by an instruction, it is necessary to examine all the log strings and find the highest clock number associated with the instruction label. Microcode information can be recovered by replacing the instruction labels by operator control point labels. For example, the log string

```
"A 1 B 2 C 1 B 3 C 5 B 5 3 D 4"
```

demands that the operator control points be switched on during Phi 2, Phi 3 and Phi 5 to execute instruction B.

How useful is very high level optimisation?

From the discussion so far, it is obvious that high level optimisations must be developed with a particular field of application in mind. The framework presented in this thesis allows behavioural optimisation, but it maintains the temporal function of a system. Functional level optimisation however, can alter the function being performed.

It is difficult to see how ultra-high level optimisation could ever be completely automated: there must always be some feedback to the designer for him to decide whether the change in function is acceptable or not. This does not mean that functional level compilers will not exist, nor that they will be useful. But it does mean that the compilers must be highly specialised and interact closely with the designer. In this way, the optimisers would be designer's assistants interfacing to some preexisting tool set rather than stand-alone chip generators.

It is for these reasons that the framework presented in this thesis carries out no optimisation above the behavioural level. At the outset, it was decided that the value of a general purpose silicon compiler was greater than that of a specialised tool which achieves only a slightly higher level of design abstraction.

Summary

The framework presented in this thesis could act as the back end of a more specialised front end. The library facilities allow complex modules to be recognised, generated, decomposed and then reconstituted. For example, an adder might be specified. The compiler might decide to use a fulladder with full look-ahead across two sections of the adder. The floorplanner is given the hierarchy

of cells needed to build the adder so the floorplanner might decide that splitting the adder in two, putting each part on the opposite sides of a chip and running the carry lines across the chip might be the best way of resolving the constraints imposed upon it. In this way, the framework presented here does include very sophisticated optimisations and does take a truly behavioural level of input. Functional level optimisation has been deliberately excluded from the framework, but the subject of functional optimisation in one area in which the compiler may find service has been discussed.

Chapter 10

Conclusion

We set out on a quest to overcome the hurdles obstructing our view of some ultimate CAD tool, a silicon compiler. A homogeneous set of techniques for compiling a specification of the behaviour of a system into the topography needed to implement it has been described.

A new floorplanning method was developed. Unfortunately, the expert nature of the system means that it is not possible to prove the system through benchmarks: if there is ever a floorplanner that can do better than the idiomatic floorplanner in a particular domain then that floorplanner can be added to the library of floorplanners used by the idiomatic method. This has meant that a sound theoretical basis for the method has had to be developed. The theory is not reliant on the sole conjecture that floorplanning idioms can be classified according to how they handle wires, but it includes support for functional classification when the function or other non-systematic aspect of a design has to be considered to produce a high quality floorplan.

Once the obstacles of floorplanning were overcome, the problems of library assignment and silicon assembly remained. A significant contribution is made by this thesis in these areas. The use of design-rule servers and the multiplane corner-stitching system are important contributions, paving the way for technology independent design tools.

With this framework in place, others working on the U2 silicon compiler project have been able to produce automated power routers, outstanding switchbox routers, new bloating algorithms and so on. These achievements by other members of the silicon compiler project demonstrate the value of this framework.

An important reason for developing this framework is that once a framework is established it becomes very much easier to interface autonomous programs with one-another. For example, the power router, global router, switchbox router and the bloater all form part of a single convergence loop. By allowing the framework to take the load off the separate subsystems, these tools may be integrated into a working whole much more easily than if they all had their own datastructures and their own systems for controlling feedback.

Areas for further research

Needless to say, there are several areas related to the work described here which require further research to fully investigate.

The problems of synthesising chips has been tackled, but the problems of how to test the chips has been totally ignored. Designing test patterns to achieve a high fault coverage is only a part of the problem: unless the chip has been designed with the problems of testing in mind it could turn out be untestable, and therefore useless.

In the context of silicon compilation, it might be possible to use the same divide and conquer approach to synthesising test structures as is described in this thesis for doing automated floorplanning. Ideally, every type of module would be served by a one of a collection of test strategies. Each strategy would have its own test structures, such as scan paths or signature analysis, and each module in a design could be augmented by the hardware needed to support the test function.

Another area that needs further work is that of synthesising multi-chip systems. The framework for silicon compilation described in this thesis is limited in a practical sense to the synthesis of single chips. If progress is to be made on allowing laymen to specify chips, there must be a mechanism for breaking very large designs into logical subunits and then proceed to make some physical division between the set of chips needed to implement the desired function.

Finally, the work described in this thesis remains experimental. A great deal of work remains to be done to build a reliable full-size silicon compiler. This engineering phase is now well underway but without the framework provided by this thesis, the engineering would have been almost impossible.

Appendix A

List of Symbols

$CS(v,m)$	The mth degree *connection set* of the vertex v
$e_{i,j}$	An edge from vertex v_i to vertex v_j
G	A simple graph
$\|G\|$	The *order* of the graph G
G^n	An arbitrary simple graph of order n
$H(\gamma)$	Heawoods function
$K^{a,b}$	A complete bi-partite graph with a vertices in one set and b in the other
K^n	A complete graph of order k
$MC(v,m)$	The mth degree *mutual connectivity* of the vertex v
γ	The genus of a plane
$\varphi(G)$	The *planarity* of the graph G – see Chapter 2
$\varphi'(v_i)$	The *local planarity* of the vertex v_i – see Chapter 2
$\chi(G)$	The *chromatic number* of the graph G.
$\chi'(G)$	The *edge chromatic number* of the graph G.
$\emptyset$	The empty set
$\cup$	Union of two sets
$\cap$	Intersection of two sets
$G \rhd H$	Contraction of graph G to the subgraph H
$G \succ H$	Subcontraction of graph G to the subgraph H

Bibliography

[Alle 82] J. Allen.
Introduction to VLSI design.
In *VLSI Architecture*, pages 1-23. Prentice Hall, 1982.

[Appe 76a] K. Appel and W. Haken.
Every planar map is four-colourable, Part I: discharging.
1976.
University of Illinois.

[Appe 76b] K. Appel, W. Haken and J. Koch.
Every planar map is four-colourable, Part II: reducibility.
1976.
University of Illinois.

[Barr 83] I. Barron.
Fifth generation components.
In *The INMOS Seminars, Edinburgh*. INMOS, 16 March, 1983.

[Basi 84] V. R. Basili and B. T. Perricone.
Software errors and complexity: an empirical investigation.
CACM 27(1):42-52, 1984.

[Berg 83] N. Bergmann.
A Case study of the F.I.R.S.T. silicon compiler.
In *Proceedings of 3rd Caltech Conference on VLSI*, pages 379-394.
Computer Science Press (ed. Bryant), 1983.

[Boll 78] B. Bollobas.
Extremal graph theory.
Academic Press, 1978.

[Breb 84] G. Brebner.
Private communication on NP-Completeness of CS(v_i,m) and proof thereof.
December 1984.
University of Edinburgh.

[Brel 79] D. Brelaz.
New methods to colour the vertices of a graph.
CACM 22(4):251-256, April, 1979.

[Broo 75] F. P. Brooks, Jr.
The mythical man-month.
Addison-Wesley Pubs., 1975.

[Buch 83a] G. Brebner and D. S. Buchanan.
On compiling structural descriptions to floorplans.
In *ICCAD 83*, pages 6-7. IEEE, 1983.

[Buch 83b] D. S. Buchanan.
VLSI layout planner.
BSc. Hons. Thesis, Dept of Computer Science, University of Edinburgh , 1983.

[Buri 83] M. R. Buric et. al.
The Plex Project: VLSI layouts of microcomputers generated by a computer program.
In *ICCAD 83*, pages 49-50. IEEE, 1983.

[Burk 83] K. P. Burkhart, M. A. Forsyth, M. E. Hammer and D. F. Tanksalvala.
An 18MHz, 32 Bit VLSI microprocessor.
The Hewlett Packard Journal :7-11, August, 1983.

[Burs 82] M. Burstein and R. Pelavin.
Hierarchical channel router.
Technical Report 9715, IBM T.J. Watson Research Centre, 1982.
The Author (A.Deas) has been unable to obtain this report.

[Burs 83a] M. Burstein and R. Pelavin.
Hierarchical channel router.
Integration 1(1), April, 1983.

[Burs 83b] M. Burstein and R. Pelavin.
Hierarchical channel router.
In *20th D.A. Conference*, pages 591-596. IEEE, 1983.

[Burs 83c] M. Burstein and R. Pelavin.
Hierarchical wire routing.
IEEE Trans. on CAD 2(4), October, 1983.

[Burs 83d] M. Burstein and S. J. Hong.
Hierarchical VLSI layout: Simultaneous Placement and wiring of gatearrays.
In *Proceedings of the VLSI '83 Conference*. Olso, Norway, 1983.
This version of Burstein's paper is the most readable.

[Burs 84] M. Burstein and R. Pelavin.
Hierarchical channel router.
Computer Aided Design 16(4), July, 1984.

[Camp 84] R. Camposano, A. Kunzmann and W. Rosentiel.
Fifth generation components.
In *VLSI: Algorithms and Architectures*, pages 233-242. North Holland Publishing Co., May, 1984.

[Cane 83] M. Canepa, E. Weber and H. Talley.
VLSI in focus: designing a VLSI chip.
VLSI Design :20-24, January/February, 1983.

[Cheng 83] C. K. Cheng et. al.
Partitioning and placement based on network optimisation.
In *ICCAD 83*, pages 86-87. IEEE, 1983.

[Chun 85] K-Y Chung.
How to place n blocks in O(n) time.
1985.
Course project report, submitted for the VLSI for CAD MSc Unit, University of Edinburgh.

[Cole 84] M. I. Cole.
Partitioning programmable logic arrays.
BSc. Hons. Thesis, Dept. of Computer Science, University of Edinburgh , October, 1984.

[Cown 83] R. C. Cownie.
The BLOB datastructure for VLSI design.
Master's thesis, University of Edinburgh, 1983.

[Deas 83] A. R. Deas.
UNIT: a silicon compiler.
Master's thesis, University of Edinburgh, 1983.

[Denn 84] I. C. Dennison.
A Sticks front-end for SCALE.
Technical Report CSR-163-84, University of Edinburgh, September, 1984.

[Deut 76] D. N. Deutsch.
A 'dogleg' channel router.
In *13th D.A. Conference*, pages 425-433. IEEE, 1976.

[Dira 52] G. A. Dirac.
A property of 4-chromatic graphs and remarks on critical graphs.
Journal of the London Mathematical Society 27:85-92, 1952.

[Dira 57a] G. A. Dirac.
Short proof of a map-colour theorem.
The Canadian Journal of Mathematics 9:225-226, 1957.

[Dira 57b] G. A. Dirac.
Map colouring theorems related to the Heawood colour formula II.
Journal of the London Mathematical Society 32:436-455, 1957.

[EDIF 84] EDIF Steering Committee.
EDIF specification.
Technical Report Report Version 0 9 5, EDIF Electronic Design Interchange Format, November, 1984.

[Gajs 83] D. Gajski and R. Kuhn.
Guest editors' introduction: New VLSI tools.
Computer 16(12):11-14, December, 1983.

[Gare 79] M. R. Garey and D. S. Johnson.
Computers and Intractability: a guide to the theory of NP-Completeness.
Freeman, 1979.

[Gray 82] J. P. Gray, I. Buchanan and P. S. Robertson.
Designing gate arrays using a silicon compiler.
In *19th D.A. Conference*, pages 377-383. IEEE, 1982.

[Hadw 43] H. Hadwinger.
Uber eine klassifikation deer steckenkomplexe.
Vierteljschr. Naturforsch. Ges. Zurich. 88:133-142, 1943.

[Hadw 58] H. Hadwinger.
Ungeloste Probleme.
Elementary Mathematics 13(26):128-128, 1958.

[Hami 83] A. R. Hamilton.
PLA generator extensions.
BSc. Hons. Thesis, Dept. of Computer Science, University of Edinburgh , 1983.

[Hass 82] J. E. Hassett.
Automated layout in ASHLAR: An Approach to the problems of 'general cell' layout for VLSI.
In *19th D.A. Conference*, pages 777-784. IEEE, 1982.

[Heaw 90] P. J. Heawood.
Map colouring theorems.
The Quarterly Journal of Mathematics 24:332-338, 1890.

[Hell 79] W. R. Heller.
An algorithm for chip planning.
Technical Report Memo 2806, Caltech Silicon Structures Project, May, 1979.

[Hell 82a] W. R. Heller, G. Sorkin and K. Maling.
The planar package planner for systems designers.
In *19th D.A. Conference*. IEEE, 1982.

[Hell 82b] W. R. Heller, K. Maling and S. Mueller.
On Finding the most optimal rectangular package Plans.
In *19th D.A. Conference*. IEEE, 1982.

[Hugh 81] J. G. Hughes.
VLSI Design Tools: the ILAP library.
1981.
University of Edinburgh, Internal Report.

[Jame 85] P. D. James.
A Design rule generator.
Master's thesis, University of Edinburgh, 1985.

[Joha 79] D. Johannsen.
Bristle Blocks: a silicon compiler.
In *16th D.A. Conference*, pages 310-313. IEEE, 1979.

[Kahr 85] M. Kahrs.
An overview of SILI.
In *Proceedings of the VLSI '85 International Conference*. Tokyo, Japan, August, 1985.

[Karp 72] R. M. Karp.
Reducibility among combinatorial problems.
Complexity of Computer Computations.
Plenum Press, 1972, pages 85-103.

[Kean 85] T. Kean.
Processor compilation from behavioural descriptions.
BSc. Hons. Thesis, Dept. of Computer Science, University of Edinburgh , 1985.

[Kern 78] B. W. Kernighan and D. M. Ritchie.
The C programming language.
Prentice-Hall, 1978.

[Khok 81] K. Khokhani, A. Digiacomo and D. Weinel.
Placement of variable size circuits on LSI masterslices.
In *18th D.A. Conference*, pages 426-432. IEEE, 1981.

[Khok 85] K. Khokhani, A. Digiacomo and D. Weinel.
Automatic placement of non-uniform rectangular components.
VLSI Design :50-70, May, 1985.

[Kinn 85] C. F. Kinnear.
An automatic power router.
1985.
Course project report, submitted for the VLSI for CAD MSc Unit, University of Edinburgh.

[Koza 83] T. Kozawa et. al.
Automatic placement algorithms for high packing density VLSI.
In *20th D.A. Conference*, pages 175-181. IEEE, 1983.

[Kuzm 83] W. Kuzmicz et. al.
CAD System for automated thick film hybrid IC layout design.
In *ICCAD 83*, pages 44-45. IEEE, 1983.

[Laut 79] U. Lauther.
A min-cut placement algorithm for general cell assemblies on a graph representation.
In *16th D.A. Conference*, pages 1-10. IEEE, 1979.

[Leis 82] C. E. Leiserson.
Area-efficient VLSI computation.
ACM Doctorial Dissertation Award 1982, MIT Press, 1982.

[Liem 82] M. Lie and C. Horng.
A Bus router for IC layout.
In *19th D.A. Conference*, pages 129-132. IEEE, 1982.

[Lipt 83] R. J. Lipton.
ALI: a procedural language to describe VLSI layouts.
ACM Trans on Prog Languages and Systems 5(3), July, 1983.

[Loca 78] B. Locanthi.
LAP: A Simula package for IC layout.
Technical Report Caltech Display File 1862, Caltech, 1978.

[Mach 83] J. W. Machar.
An investigation of Weinberger arrays.
BSc. Hons. Thesis, Dept. of Computer Science, University of Edinburgh , 1983.

[Marc 84] P. Marchal, M. Nocolaidis and B. Courois.
Microarchitecture of the MC68000 and evaluation of a self checking version.
In *Microarchitecture of VLSI Computers*. NATO Advanced Study Institute, Sogesta, Urbino, Italy, July, 1984.

[Mare 85] M. Marek-Sadowska.
Two-dimensional router for double-layer layout.
In *22nd D.A. Conference*, pages 117-123. IEEE, 1985.

[Mars 84] R. M. Marshall and I. Buchanan.
SCALE, A Language for VLSI design.
Technical Report CSR-158-84, University of Edinburgh, January, 1984.

[Marw 84] P. Marwedel.
The Mimola Design System: Tools for the design of digital processors.
In *21st D.A. Conference*. IEEE, 1984.

[Math 83] T. G. Matheson et. al.
Embedded electrical and geometric constraints in hierarchical circuit-layout generators.
In *ICCAD 83*, pages 3-5. IEEE, 1983.

[McGr 82] D. J. M. McGreivy.
VLSI approaching cost performance saturation.
In *International Conference on Circuits and Computing*. New York, Sept, 1982.

[Mead 80] C. Mead and L. Conway.
Introduction to VLSI.
Addison-Wesley Publishing Company, 1980.

[Moor 75] G. E. Moore.
Digest of Papers.
In *International Electron Devices Meeting, Washington DC, USA*. 1975.

[Moor 79] G. E. Moore.
Are we really ready for VLSI.
In *CALTECH Conference on VLSI*. January, 1979.

[Mort 85] R. W. R. Morton.
A Hierarchical power router for general cells.
Master's thesis, University of Edinburgh, 1985.

[Moto 82] 16 Bit microprocessors user manual (Third Edition).
MC68000UM.
Motorola Inc., 1982.

[Mudg 80] J. C. Mudge et. al.
A VLSI chip assembler.
In *Design Methodologies for VLSI circuits*, pages 329-355. NATO Advanced Study Institute, July, 1980.

[Nage 75] L. W. Nagel.
SPICE2: A Computer program to simulate semiconductor circuits.
Technical Report ERL-M520, University of California, Berkeley, May, 1975.

[Nagl 80] A. Nagle.
Automatic design of sequencers for the control of digital hardware.
PhD thesis, Carnegie-Mellon University, 1980.

[Nanu 64] B. Nanus and L. Farr.
Some cost contributors to large scale programs.
AFIPS Proc. SJCC 25:239-248, Spring, 1964.

[Napp 84] D. Napp and A. Parker.
A Datastructure for VLSI synthesis and verification.
Technical Report Report DISC/83-6a, EE Systems Dept., University of Southern California, March, 1984.

[Nixo 84] I. M. Nixon.
IF: An idiomatic VLSI floorplanner.
Master's thesis, University of Edinburgh, 1984.

[Nixo 85] I. M. Nixon.
Private communication on time taken by Heller's floorplanning algorithm.
January 1985.
University of Edinburgh.

[Obre 81] M. Obrebska.
Comparative survey of different design methodologies for control parts of microprocessors.
In *Proceedings of 1981 CMU Conference on VLSI*. Carnegie Mellon University, October, 1981.

[Obre 82] M. Obrebska.
Efficiency and performance comparison of different design methodologies for control parts of microprocessors.
Microprocessing and Microprogramming 10:163-178, 1982.

[Oust 82] J. Ousterhout and D. M. Ungar.
Measurements of a VLSI design.
In *19th D.A. Conference*, pages 903-908. IEEE, 1982.

[Oust 84a] J. K. Ousterhout.
Corner Stitching: a data-structuring technique for VLSI layout tools.
IEEE Trans. on CAD of Integrated Circuits and Systems 3(1):87-100, Jan, 1984.

[Oust 84b] J. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott and G. S. Taylor.
Magic: a VLSI layout system.
In *21st D.A. Conference*, pages 152-165. IEEE, 1984.

[Park 84] N. Park and A. Parker.
Synthesis of optimal clocking schemes for digital systems.
Technical Report Report DISC/84-1, University of Southern California, May, 1984.

[Park 85] N. Park and A. Parker.
Synthesis of optimal clocking schemes for digital systems.
In *22nd D.A. Conference*. IEEE, 1985.

[Patt 81] D. Patterson and C. H. Sequin.
A Reduced Instruction Set VLSI Computer.
In *Proceedings of the Eighth Annual Symposium on Computer Architecture*. Minneapolis, May, 1981.

[Prea 78] B. T. Preas and W. M. VanCleemput.
Methods for hierarchical automatic layout of custom LSI circuit masks.
In *15th D.A. Conference*, pages 206-212. IEEE, 1978.

[Rive 82] R. L. Rivest.
The PI (Placement and Interconnect) system.
In *19th D.A. Conference*, pages 475-481. IEEE, 1982.

[Robe 80] P. S. Robertson.
The IMP-77 language.
Technical Report CSR-19-77, University of Edinburgh, November, 1980.

[Sato 79] K. Sato, T. Nagel, H. Shimoyama and T. Yahara.
MIRAGE: a simple-model routing program for the hierarchical layout design of IC masks.
In *16th D.A. Conference*, pages 297-304. IEEE, 1979.

[Scho 82] J. P. Schoellkopf et. al.
CAPRI: A Silicon compiler for VLSI circuits specified by algorithms.
In *VLSI Architecture*, pages 149-154. Prentice Hall, 1982.

[Scho 83] J. P. Schoellkopf.
Lubrick: a silicon assembler.
In *VLSI 83*, pages 435-445. North-Holland Pubs., 1983.

[Sedg 83] R. Sedgewick.
Algorithms.
Addison-Wesley, 1983.

[SERC 84] SERC.
UK5000 design reference manual.
October, 1984.
Literature issued during UK5000 design course held at Rutherford Labs., Oxfordshire, England.

[Shro 82] H. E. Shrobe.
The datapath generator.
In *Conference on Advanced Research in VLSI*, pages 175-181. MIT, 1982.

[Sisk 82] J. R. Southard, K. W. Crouch and J. M. Siskind.
Generating custom high performance VLSI designs from succinct algorithmic descriptions.
In *Conference on Advanced Research in VLSI*. MIT, 1982.

[Syed 82] Z. Syed and A. E. Gamal.
On routing for custom integrated circuits.
In *19th D.A. Conference*, pages 887-893. IEEE, 1982.

[Szep 80] A. A. Szepieniec and R. H. Otten.
The geneological approach to the layout problem.
In *17th D.A. Conference*, pages 535-542. IEEE, 1980.

[Szep 82] A. A. Szepieniec.
SAGA: An experimental silicon assembler.
In *19th D.A. Conference*, pages 365-370. IEEE, 1982.

[TED 83] D.J.Rees.
Undocumented computer program.
1983.
Department of Computer Science.

[Thom 85] J. G. Thomlinson.
Implementing the hierarchical channel router.
Master's thesis, University of Edinburgh, 1985.

[Wald 82] K. Waldschmidt.
Microelectronics technology: an introduction.
Microprocessing and Microprogramming 10:71-76, September, 1982.

[Walk 85] R. A. Walker and D. E. Thomas.
A Model of design representation and synthesis.
In *Proceedings of 22nd Design Automation Conference*, pages 453-459.
IEEE, 1985.

[Wari 85] T. J. Waring.
On automatic switchbox routing.
Master's thesis, University of Edinburgh, 1985.

[Wein 65] G. F. Weinwurm.
Research in the management of computer programming.
Technical Report SP-2059, System Development Corp., Santa Monica, 1965.
Cited by Brooks.

[Wein 67] A. Weinberger.
Large scale integration of MOS complex logic: A Layout method.
IEEE Journal of Solid State Circuits 2(4):182-190, December, 1967.

[Will 78] J. D. Williams.
STICK - A Graphical compiler for high level LSI design.
In *National Computer Conference*. NCC, 1978.

[Wood 69] D. C. Wood.
A technique for colouring a graph applicable to large timetabling problems.
Computing Journal 12:317-319, 1969.