Complexity Management and Modelling of VLSI Systems

Alex Dickinson B.E.(Hons)

A thesis submitted to the Faculty of Engineering
for the Degree of Doctor of Philosophy.

Department of Electrical and Electronic Engineering
The University of Adelaide
South Australia

January 1988

Contents

Abstract

The major objective of the research described in this thesis is to describe effective methods for the partitioning of Very Large Scale Integrated (VLSI) systems.

A comparison is made between the structural design of large programs and large VLSI designs. A methodology for VLSI structural design is proposed based on many of the precepts of structural program design. The methodology requires restructuring of the design process; a specific form of design representation; and the addition of computer aided modelling of both the algorithmic function and geometrical form of the structural design.

The design process is divided into a designer intensive top-down planning phase, and an automatic bottom-up construction phase.

A model of design is described in which the commonly used layered set of abstractions are replaced by a single structural description. The language introduced for this purpose incorporates a number of features to reduce the apparent complexity of the system description. The abstract representation of intermodule synchronization and communication allows a process of stepwise refinement to be applied to these as well as the more usual structural entities.

Two computer aided modelling tools have been developed that together constitute a facility for rapidly analyzing alternate structural partitionings in the search for an acceptable design.

The first computer aided modelling tool is a functional simulator incorporating a novel interprocess communication and scheduling mechanism. This allows the efficient implementation of the description language intermodule communication semantics. The interactive nature of the simulator facilitates initial debugging and qualitative evaluation of the design. A profiling facility allows for the quantitative evaluation of the partitioning based on data flow and module activity.

The second computer aided modelling tool is a hierarchical floorplanner that facilitates the evaluation of the embedding of the proposed structural design into the plane. An investigation of structured floorplan design shows that the process is inherently knowledge intensive, and that much of that knowledge is inexact. The floorplanner incorporates several novel knowledge representations that are used to express diverse classes of designer expertise. A representation for spatial reasoning provides for the

efficient manipulation of rectangles in a mosaic. Another representation has been developed for reasoning with the inexact knowledge used by designers in predicting the implementation of floorplan modules in a hierarchical design. A production system is used as a design manager to guide the overall development of the design.

A case study is presented that demonstrates the utility of the methodology and computer aided design tools in VLSI system design.

Preface

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University. To the best of the author's knowledge and belief the thesis contains no material previously published or written by another person, except where due reference is made in the text.

The author hereby consents to this thesis being made available for photocopying and loan if accepted for the award of the degree.

Alex Dickinson

Acknowledgements

Many people have contributed (sometimes unwittingly) to the effort represented by this thesis. Some of them are named here.

My Australian supervisor, Kamran Eshraghian, set me on this path, and provided the enthusiasm, confidence, and encouragement required to get me to the end. Neil Weste, my US supervisor, was instrumental in getting me there and provided much good advice. Special thanks to Neil and Avril (together with Miles and Ellen) for their support when I arrived zombie-like in New Jersey, and to John and Jenny for their friendship and lovely sweaters. Thanks also to Peter Cole, who acted as my supervisor in Kamran's absence, for being both appreciative and fun to work with over (though sometimes quite infuriating!).

Thanks to Integrated Silicon Design and The Association for Computing Machinery for providing the travel support that enabled me to flit about and greatly increase my knowledge (and time in the sun). The Commonwealth Department of Education provided the regular source of funds needed to make a PhD possible, and Tim Gent was stalwart in dealing with my odd requests. Bell Labs, and in particular the people of lab 1135 generously provided me with facilities and a whole new area of knowledge. Martin Levy patiently taught me the basics of Unix hacking and introduced me to the joys of the Redbank diner at 3am. Many thanks especially to Bryan Ackland, who seems to have endless patience with me, and whos calm, quiet advice I greatly value. Thanks to Alan Huang for his confidence in my abilities and for offering me a challenging opportunity.

Thanks to Charles Watson for reviewing the thesis, and Greg Zyner for his help with the case study. Thanks to the people at Bell Labs, Symbolics, and other institutions who gave up their time to drink coffee and discuss floorplan design.

Other far-away people that I would like to thank include Graham Birtwistle in Calgary and Wolfgang Fichtner in Zurich for their useful comments and travel assistance. Honestly, I didn't just visit to get up into the mountains...

To the residents of the postgraduate tea room and members of the Flat Earth Society go my thanks for providing hundreds (thousands?) of hours of entertaining conversation. On some days the only motivation I found to come into work was to sit and talk. I'd like to thank Mike Pope for his dry humor, his spelling ability, and

for joining with me in meandering hacker's conversations.

Mike Liebelt provided generous access to computer facilities (no matter how extreme and frivolous the demands I made on them!), together with teaching me how to confound a bureaucracy. More important though was his quiet confidence that I was going somewhere, his patience with me, and marvelously calm and sensible advice on dealing with many problems, both technical and personal. Thanks Mike.

To my parents and brother go my boundless thanks for their patience and support in dealing with an eternal student, both emotionally and financially, especially in my time away. Dad, I wish you could be here to see it finished, you were always so proud.

There's only one more. Thanks Pru for your faith, generosity, patience, understanding, caring, and love. This thesis is for you my friend.

Chapter 1

Introduction

The major objective of the research described in this thesis is to describe effective methods for the partitioning of Very Large Scale Integrated (VLSI) systems.

The philosophy of *structured programming* is regarded as one of the most effective methodological tools used to manage software complexity [Dijkstra, 1972]. The application of structured programming techniques to VLSI systems was proposed by Carver Mead at Caltech [Mead & Conway, 1980] and has since been widely adopted for the design of custom integrated circuits.

A central principle of this *structured VLSI design methodology* is the partitioning of systems into a hierarchy of modules. However techniques for generating such a structure are not well developed. In this thesis the inherent difficulties of partitioning VLSI systems are examined and methods proposed for solving the problem. These *structural* VLSI design methods are intended to complement those of the established *structured* VLSI design methodology.

The concepts involved in the structural design of programs [Yourdon & Constantine, 1975] are used in this thesis as an aid to the development of a methodology for structural VLSI design. It will be argued that the horizontally layered abstraction used for design representation in structured VLSI design are well suited to the structural planning task. An alternative method based on the parallel development of algorithmic *function* and physical *form* will be described.

Central to the methodology are a design specification language and two interlinked design aids for *modelling* the structural design. The architectural specification language provides for the abstract representation of the structure and function of the

proposed design. The first modelling aid is a simulator for the language that is used to evaluate the functional partitioning of the design. The second modelling aid is a knowledge-based floorplanner that assists in the evaluation of the partitioning on the two dimensional silicon surface. The use of knowledge-based techniques allows the floorplanner to operate in a top-down design style and incorporate principles of structured VLSI layout design. These two design aids constitute a facility for rapidly analyzing alternate structural designs in the search for an acceptable result.

In the remainder of this chapter the problem of VLSI design complexity is described together with the established structuring techniques used in its control. The deficiencies of such techniques are discussed and the relevance of *structural design* is introduced.

In Chapter 2 a description is given of the methods developed for the creation and evaluation of structural designs for programs. The difficulties of partitioning VLSI systems are examined, and the similarities and differences between the two domains identified. A structural VLSI design methodology is then described based on a reorganization of the VLSI design process. This allows software structural design techniques to be used in VLSI given the existence of appropriate languages and design aids.

A language designed for the abstract description of VLSI designs is described in Chapter 3. The language has a number of features that assist in the abstract description and refinement of structure and communication in the design.

The functional modelling of VLSI designs is discussed in Chapter 4. A simulator is described that models the behaviour of a design to aid in the evaluation of the functional partitioning.

The issues involved in producing a system partitioning suitable for physical design—floorplanning—are examined in Chapter 5.

In Chapter 6 a knowledge-based floorplanner is described that allows for the modelling of the structural design in terms of its realization as a floorplan.

An example of design using the previously described methodology and design aids is documented in Chapter 7.

Finally conclusions and suggestions for further research that arise from this thesis are given in Chapter 8.

1.1 VLSI System Design Complexity

In 1964 Gordon Moore predicted that integrated circuit complexity (as measured by device count) would continue to double every year: the ubiquitous "Moore's Law" [Noyce, 1977]. Since 1970 however this line has only been held to by memory devices. Less regular designs have fallen below the predicted complexity as shown in Figure 1.1(a) [Moore, 1979]. This lag exemplifies the problem that now faces VLSI design: the complexity of the design task has risen to the point where it is difficult for designers to create systems that make full use of the available fabrication processes. This has resulted in an exponential rise in design effort as illustrated in Figure 1.1(b).

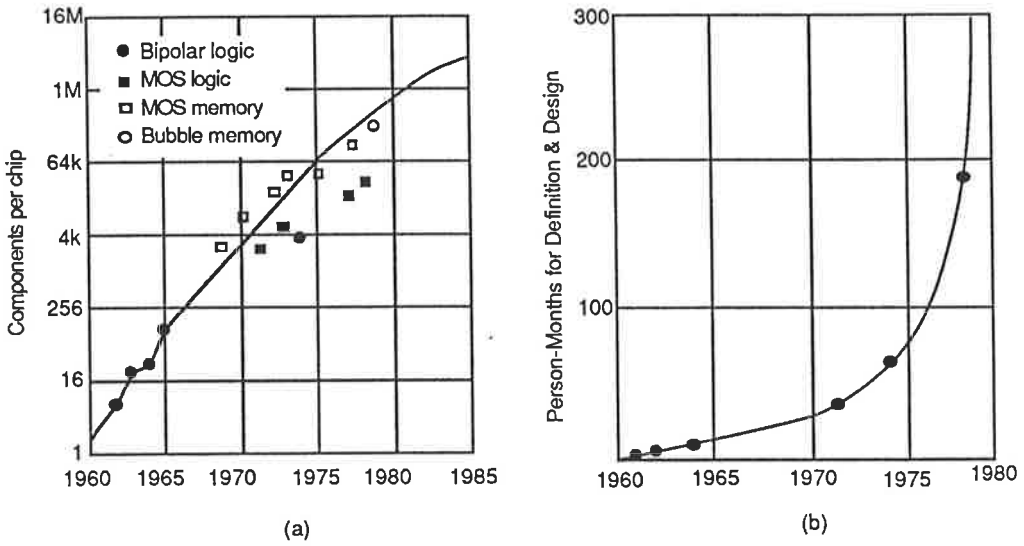

Figure 1.1: Trends in integrated circuit complexity [Moore, 1979].

The complexity of VLSI design may be attributed to two related phenomena: the lack of *structure* in, and difficulty of *communication* on, the silicon surface.

The structuring problem results from the essentially unstructured nature of the VLSI medium—there is no *a priori* partitioning imposed on the design. According to Séquin [Séquin, 1983] this may *result in a dangerous situation where the complexity within a large, unstructured domain simply overwhelms the designer.*

Large digital systems are typically exhibit a great deal of physical partitioning. Transistors are integrated into IC's, IC's are placed onto PC boards, PC boards are connected to mother boards, mother boards are packaged into cabinets and interconnected with backplane buses.

The partitioning of systems into such a packaging (or physical) hierarchy is guided by factors such as division of labor, clarity of design, ease of manufacture, functionality, communication requirements, reliability, testability and maintainability. This structure is an artifact imposed by *design* influenced by physical considerations, not an innate property of the medium — one could (however inadvisably) attempt to design and build a complex digital system on a single circuit board in a completely unstructured manner.

There is no innate structuring imposed by the physical nature the VLSI medium. The entire planar silicon surface is available, and any structure is imposed by the designer.

Some VLSI design problems are inherently well structured. Memory design, for instance, has achieved high circuit densities limited mainly by fabrication technology rather than design complexity [Taylor & Johnson, 1985]. This may be attributed to the their simple regular structure and consequent design in which a small number of circuit blocks are designed and replicated for placement in a regular pattern of interconnection. For designs with less inherent structure such as those based round instruction sets and protocols, the additional design complexity has a profound effect on designer productivity. This is illustrated in Figures 1.2(a) and 1.2(b). In memory design an economy of scale is achieved: as the number of devices increases, design productivity increases. In the design of less well structured elements, not only is productivity about an order of magnitude less, it actually decreases with circuit complexity.

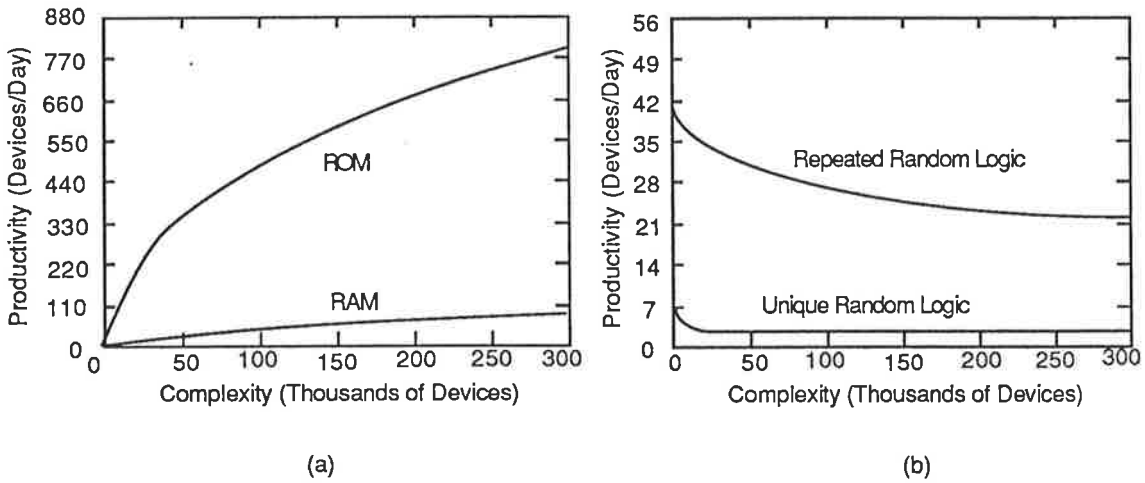

Figure 1.2: Designer productivity [Fey, 1985].

The general approach to improving designer productivity in such cases is to utilize the freedom that exists in the VLSI domain to impose different structuring schemes on the silicon surface in order to reduce design complexity. Prior to introducing two broad classes of structuring VLSI designs it is useful to briefly examine the issue that differentiates them: the design of the *interconnect* used for communication within the structure.

There are three primary costs associated with integrated circuit interconnect:

1. *Area.* Longer wires consume more design area. This results in less functionality being implemented in a given area.
2. *Speed.* Longer wires result in greater RC time delays for a fixed sized driver.
3. *Power.* Greater capacitive loads on drivers result in higher power consumptions.

Interconnect length and complexity has in fact been identified as one of the primary fundamental limits on circuit integration [Keyes, 1981]. Keyes also notes that it is one of the few such limits for which there is no underlying physical theory, further complicating design near that limit.

The need to reduce interconnect length must be traded off against the need to simplify design to increase productivity. This trade-off may be observed between the two major classes of VLSI structuring: *highly* partitioned and *functionally* partitioned systems [Ferry, 1985]. They are described in the following two sections.

1.1.1 Highly Partitioned Systems

It is possible to impose a structure similar to that use in conventional digital design onto unstructured silicon. *Gate array* and *standard cell* "semi-custom" design styles cluster the atomic components of the design, the transistors, into small functional units. These are then placed and routed such that area and interconnect length are minimized. The procedure is quite analogous to the production of printed circuit board designs, and a large number of algorithms have been created or adapted for the automation of this design task, a recent example being simulated annealing [Sechen & Sangiovanni-Vincentelli, 1985]. These design styles are termed *highly partitioned* in that they are structured into a large number of small partitions.

The primary disadvantages of such design styles stems from the expense incurred by communication between the components on the two-dimensional surface. Keyes refers to a typical gate-array design of 1496 gates in which $0.26cm^2$ of the total chip area of $0.32cm^2$ is used for the running of $4m$ of interconnect [Keyes, 1981]. This implies an active are of only 20%, the remaining 80% being utilized for long wire runs with their attendant disadvantages.

In summary, *highly partitioned* styles have the advantage of being amenable to design by automated techniques because of the simple formulation of design as place and route. However the large area consumed by wiring precludes them from achieving transistor packing densities sufficient for VLSI levels of integration ($> 10^5$ transistors per die).

1.1.2 Functionally Partitioned Systems

In order to achieve VLSI levels of integration it is necessary to resort to design styles based around *functional partitioning*. In these styles the system is decomposed into modules that are functionally related, resulting in a reduction in the amount of communication that occurs between partitions. The increased circuit densities achievable with this form of partitioning may be attributed to an associated reduction in interconnect. Ferry's results suggest that *average interconnect lengths in a VLSI circuit that is functionally partitioned do not continue to increase as device sizes are scaled down and chips become more densely packed* [Ferry, 1985]. He cites the functionally partitioned design of the HP 32b microprocessor. It contains 1.3×10^5 gates with $5m$ of interconnect, implying a factor of 70 reduction in the wiring length per gate ratio over the gate array design referred to previously. Ferry relates the success of functional partitioning to a reduction in information flow between design partitions, a concept discussed further in Chapter 2.

Functionally partitioned designs are generally produced using a *full custom* design style. Not only is the design functionally partitioned, but the design of the resulting layout takes into account the geometrical interactions of the connecting modules. This minimizes inter-partition communication costs. In this way modules may be designed so as to combine their function with the needs of system communication, a classic example being the barrel shifter (Figure 1.3). The interconnect required by the shifter matches the global interconnect strategy of the data-path illustrated in

Figure 1.9.

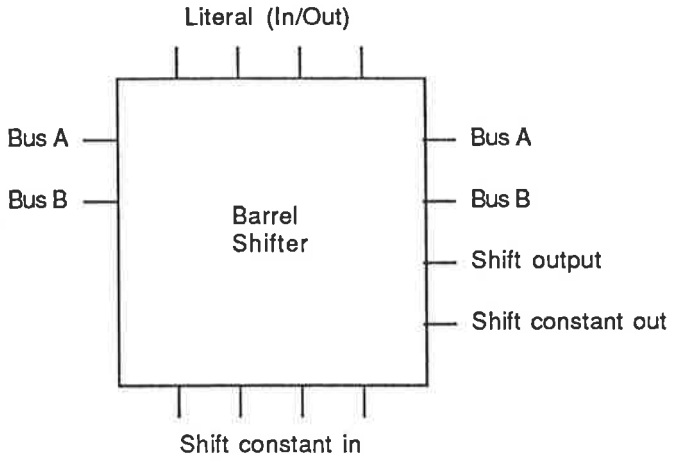

Figure 1.3: A structured layout design for a barrel shifter [Mead & Conway, 1980, p. 161].

The use of functional partitioning and custom design techniques adds considerably to the complexity of the design task compared to highly partitioned design styles. Not only must module interfaces be carefully designed to decrease interconnect costs, but the higher resulting device densities increase the number of components in the design.

Fey's research [Fey, 1985] supports the contention that the primary complexity problem of VLSI design is that of interconnect: the productivity model he describes predicts that a custom circuit of 10^6 devices will take about 400 man years of effort to design in 1989, however an increase in the complexity of interconnect could increase this by an order of magnitude.

In the following section an examination is made of some general complexity management techniques. The application of these techniques to functional partitioning in *structured VLSI design* is discussed in Section 1.3.

1.2 Complexity and Structure

In order to provide a basis for the discussion of structured design as a complexity management technique, the relationship between design complexity system structure shall be examined in this section.

The *complexity* of a system comprised of a number of parts may be defined as *the way in which a whole is different from the composition of the parts* [Van Emden, 1975]. In other words, complexity is that property of a system that arises from the fact that it is a composition of a number of parts: before the parts were composed they may be regarded as having zero complexity as a reference point. This definition emphasizes that aspect of complexity of particular interest in system design: *that the complexity of a system is due to the interaction of its parts.*

We can express the complexity of a system in terms of the interactions between its parts as follows. For the system S composed of atomic components $V_{1..n}$ illustrated in Figure 1.4 the complexity $C(S)$ may be expressed in terms of R, the interaction between atomic components as:

$$\begin{aligned} C(S) \;=\; & R(V_1, V_2) + R(V_1, V_3) + \cdots + R(V_1, V_n) \\ & + R(V_2, V_3) + \cdots + R(V_2, V_n) + \cdots \\ & \cdots + C(V_1) + \cdots + C(V_n) \end{aligned}$$

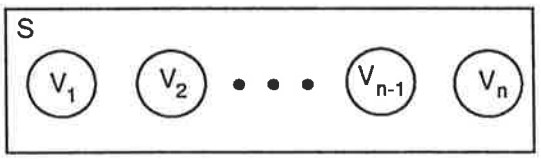

Figure 1.4: An unstructured system S composed of atomic components $V_1 \cdots V_n$.

In accordance with Van Emden's definition only the difference between the total complexity and the sum of the complexities of the atomic components is of interest:

$$\begin{aligned} & C(S) - C(V_1) - \cdots - C(V_n) = \\ & \quad R(V_1, V_2) + R(V_1, V_3) + \cdots + R(V_1, V_n) \\ & \quad + R(V_2, V_3) + \cdots + R(V_2, V_n) + \cdots \end{aligned}$$

The sum of the atomic component complexities may then be treated as a zero level of complexity finally giving the complexity of the system as a sum of the interactions of the parts:

$$C(S) = R(V_1, V_2) + R(V_1, V_3) + \cdots + R(V_1, V_n) + R(V_2, V_3) + \cdots + R(V_2, V_n) + \cdots \quad (1.1)$$

The *structure* of a system is the manner in which the component parts are organized to form a whole. We shall present a system as having two classes of structure: *implicit* and *explicit.*

Implicit structure. The implicit structure of a system is a property of its basic *construction*, not its description. The implicit structure tends to be intricate as it covers a systems function, physical partitioning and any other features of its operation and appearance. Systems have a corresponding *implicit complexity* [Séquin, 1983] that is a function of their construction: the number of components and their interactions.

Explicit structure. The explicit structure of a system is a property of a particular representation used to describe it to an observer. The *explicit complexity* [Séquin, 1983] implied by such a structure is a function of the representation, and as such is amenable to reduction by judicious choice of representation.

For example, a VLSI chip has an intricate implicit structure of perhaps hundreds of thousands of transistors each carrying out complex manipulations of charge carriers across junctions, and communicating via current carrying wires and capacitive coupling. When describing such a device, structuring and abstraction techniques may be used to generate a description that implies a much simpler explicit structure, and a correspondingly reduced explicit complexity.

A general approach to the problem of reducing explicit system complexity is to produce a simple explicit structure by describing the system in a *structured* manner: simplifying part (component) interactions by means of *hierarchy* and *regularity.*

1.2.1 Hierarchy

One of the simplest forms of structuring that may be applied to a system is that of partitioning. Recursive application of partitioning gives rise to a hierarchical system: *a system that is composed of interrelated subsystems, each of the latter being, in turn hierarchical in structure until we reach some lowest level of elementary subsystem* [Simon, 1962].

The components of a subsystem typically have non-uniform intensities of interaction. This non-uniformity is a useful basis for defining a hierarchical partitioning of the system. Strongly interacting components may be clustered into subsystems, and the resulting subsystems interact in relatively less complex manner. If a system is amenable to such a recursive partitioning it is known as *nearly decomposable* as opposed to a *decomposable* system in which the subsystems are effectively independent [Simon, 1962].

The judicious application of hierarchical structuring techniques to a system can reduce the explicit complexity of that system in a number of ways which may be qualitatively described as follows:

1. The task of dealing with the system is simplified because only the components and interactions of a single subsystem need be considered at a time.
2. Sufficient application of hierarchical partitioning results in simple *elementary* components ($V_{1..n}$).
3. In each subsystem, each child subsystem may be represented by an *interface* that hides information other than that relevant to the composition of the child subsystems.

Van Emden [Van Emden, 1975] presents the following analysis of the effect of using hierarchical structures. If the whole *is* different from the parts then this difference is due to the interaction of the parts. The magnitude of this interaction is equal to the difference between the complexity of the whole and the sum of the complexities of the parts. For the system S composed of subsystems S_{ij} and atomic components $V_{1..n}$ illustrated in Figure 1.5 the complexity $C(S)$ may be expressed in terms of R, the interaction between subsystems as:

$$\begin{aligned} C(S) &= R(S_1, S_2) + C(S_1) + C(S_2) \\ &= R(S_1, S_2) + R(S_{11}, S_{12}) + R(S_{21}, S_{22}, S_{23}) \\ &\quad + C(S_{11}) + C(S_{12}) + C(S_{21}) + C(S_{22}) + C(S_{23}) \end{aligned}$$

This decomposition can be continued to give $C(S)$ as a function only of subsystem interactions and $C(V_1) \cdots C(V_n)$, the complexities of the atomic components:

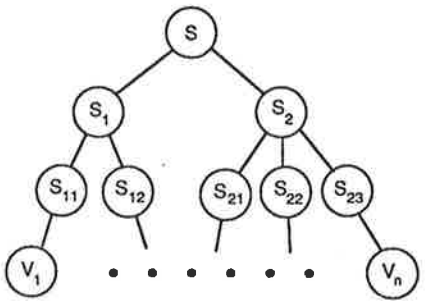

Figure 1.5: A simple hierarchically structured system.

$$\begin{aligned} C(S) &= R(S_1, S_2) + R(S_{11}, S_{12}) + R(S_{21}, S_{22}, S_{23}) + \cdots \\ &\quad \cdots + C(V_1) + \cdots + C(V_n) \end{aligned}$$

Under Van Emden's definition only the difference between the total complexity and the sum of the complexities of the atomic components is of interest:

$$\begin{aligned} &C(S) - C(V_1) - \cdots - C(V_n) = \\ &\quad R(S_1, S_2) + R(S_{11}, S_{12}) + R(S_{21}, S_{22}, S_{23}) + \cdots \end{aligned}$$

The sum of the atomic component complexities may then be treated as a zero level of complexity finally giving the complexity of the system as a sum of the interactions of the parts:

$$C(S) = R(S_1, S_2) + R(S_{11}, S_{12}) + R(S_{21}, S_{22}, S_{23}) + \cdots \tag{1.2}$$

The advantages of the hierarchical structuring compared to the non-hierarchical case (Equation 1.1) are:

1. There are typically fewer R (interaction) terms to be considered: only one for each subsystem. This contrasts to the non-hierarchical case in which there was an R term for each pairwise combination of atomic components.
2. The R (interaction) terms are independent: each subsystem S_{ij} appears only in one term. This simplifies the design of subsystems that minimize interaction with other subsystems.

Quantification of the interaction of the parts (R) is domain dependent, and will be discussed further in Chapter 2. The effectiveness of a partitioning at reducing complexity within a particular level of the hierarchy may however be expressed in terms of two basic concepts: *coupling* and *cohesion* [Yourdon & Constantine, 1975]. Coupling is a measure of the interaction between subsystems: lower coupling implies less interaction and hence less complexity. Cohesion is a measure of how well the components of a subsystem belong together. Coupling and cohesion may be used as criteria for designing system partitions, and will be discussed in detail in Chapter 2.

The vocabulary for the description of hierarchical systems varies with the domain under study and often even within a domain. In this thesis, general subsystems will be referred to as *modules*, subsystems composed of further subsystems will be referred to as *composition modules*, subsystems that are components of a composition shall be referred to as *submodules*, and elemental subsystems will be referred to as *leaf modules*. The terms *module* and *cell* are synonymous, although *module* will be preferred.

1.2.2 Regularity

The explicit representation of regularity in a system description is a powerful structuring technique. For example two systems A and B may be composed of a similar number of components with similar amounts of interaction and hence have similar implicit complexity. However if the structure of A can be expressed in terms of the simple replication of some module and its interconnections, then the explicit complexity of A relative to B is greatly reduced. Here we suggest that this reduction in complexity has two sources:

1. *Regularity of function.* Systems such as A contain a degree of redundancy in their implicit structure that may be made use of by judicious partitioning into identical modules. These may then be expressed as the multiple *instances* of a single module *definition.* Explicit complexity is reduced as only the definition must be considered in depth, each instance simply being a replication of the definition. This form of complexity reduction may be quantified by a simple regularity factor: the total number of components divided by the number that must be individually constructed. This is also illustrative of the difference between explicit and implicit complexity. The actual number of components

(implicit complexity) is unchanged, but their description (explicit complexity) has been simplified.

2. *Regularity of interaction.* In a system such as A that is amenable to division into identical modules, the interaction patterns between the identical instances will also tend to be regular. Such regular patterns of interaction reduce complexity simply by reducing R, the degree of interaction between subsystems.

The need to distinguish between these two classes or regularity arises because one may occur without the other. In particular, regular interaction patterns may be used to add structure to a system that has little functional redundancy in its implicit structure. Examples of functional and communication regularity in a VLSI system are given in Section 1.3.6.

1.3 Structured Design

The *structured design style* is a loose collection of techniques and principles that aid in the development systems in a structured manner in order to control design complexity.

The early formalization of structured design evolved in the domain of complex software systems. The increasing size of software projects in the 1960's gave rise to a situation in which few systems met goals, schedules or budgets [Brooks, 1975]. Researchers perceived that the problem was in part at least due to a lack of structure in the design domain, and this drove the development of a number of software structuring techniques that have become known collectively as *structured programming* [Dijkstra, 1972]. These techniques are quite diverse and based on number of principles including: abstraction, modularity, hierarchy, information hiding, regularity, and step-wise refinement.

The principles of structured programming have been widely adopted and are regarded as instrumental in controlling the complexity of the software problem [Brooks, 1975, p. 144].

The application of structured design to VLSI followed from the realization that in common with software design, VLSI design had reached a crisis point in complexity and this was to a considerable degree due to a lack of structure in the design process

(Section 1.1). The formulation of structuring techniques for VLSI design is typified by the *Caltech Structured Design Methodology* promulgated by Mead and Conway [Mead & Conway, 1980]. Many of these techniques have been adapted from structured programming to the two dimensional domain of VLSI design.

The remainder of this section provides an overview of the principles of structured VLSI design derived from a number of sources including [Mead & Conway, 1980; Buchanan, 1980; Rowson, 1980; Trimberger et al., 1981; Mudge et al., 1980b; Tucker & Scheffer, 1982; Lattin et al., 1981]. Where appropriate, parallels between structured VLSI design and structured programming will be drawn.

1.3.1 Abstraction

Structured system are particularly amenable to the application of *abstraction* as a means of simplifying their descriptions. Abstraction involves the development of a vocabulary that is well matched to the problem domain, suppressed irrelevant detail, that is translatable into the target vocabulary of the design.

In programming, high level languages (HLL) can be used to provide just such an abstraction. Each HLL construct is an abstraction of the underlying instruction set that performs some function appropriate to the problem domain. Examples include **if...then** for the expression of alternation and **for...do** for the expression of iteration.

The task of translating an HLL to machine code, though not trivial, requires only a small "conceptual distance" to be bridged as there is a simple relationship between many of the concepts in both the HLL and machine language: for example many architectures provide an subroutine call instruction, well matched to HLL procedure call constructs.

In VLSI, the conceptual distance between the system specifications and the target vocabulary (a mask description language) is far greater: the two have very few concepts in common. The response to this has been the development of a series of *levels of abstraction.* The levels suggested in the Caltech design methodology are:

1. *Behavioural.* Description of the system *function* without necessarily specifying any structure. For instance ISPS [Barbacci, 1981] represents behaviour in terms of an instruction set specification.

2. *Structural.* Description of the system as a set of interconnected components,

each contributing to the overall behaviour.

3. *Physical.* Description of the system as a set of interconnected *physical* components, each having a direct implementation as a mask entity.

In each case the uppermost level is intended to provide a vocabulary suited to expressing the system behaviour, and successive levels are designed to provide representations incrementally closer to the target mask level. The sequence of levels divides the large conceptual distance into smaller steps that can each be bridged by the human designers assisted by computer, or in some cases by translation or compilation programs alone.

Silicon compilers attempt to bridge the conceptual distance between system specification and mask in a single step, and typically this can only be done by narrowing the problem domain of a such compiler to a single target architecture: this allows the creation of a set of simple abstractions that can be used during the translation of specification to layout. Silicon compilers that generate high quality layout across a spectrum of architectures do not appear likely in the near future [Werner, 1982].

1.3.2 Modularity

The advantages of decomposing a system into a number of interacting components or modules was described in Section 1.2.1. In VLSI design the basic gains are in both ease of design and reduction in computation. The functional description can be partitioned into modules, reducing the complexity of individual design tasks, and allowing the application of multiple-person design teams working on independent problems. The actual amount of design work that needs to be performed may be reduced by the development of libraries of commonly used modules. The partitioning of a VLSI system is more complex than that of a software system because of the added physical constraints of shape, size and geometrical signal interface that exist in VLSI at the physical layout level.

Modularity achieves computational gains in the area of design verification: only modules that have been altered need be re-verified by for instance simulation and design rule checks, traditionally expensive procedures.

The motivations for the use of modularity in software are very similar. The main difference is the implementation of the principle in the two domains and the complexity

of the partitioning task. Re-use of modules in software (for instance math libraries) is more common as interfaces are simpler, having only logical and not physical manifestations. Typically an analogy is drawn between modules in VLSI and procedure calls in software, however a more appropriate analogy is that of coroutine or process structures [Hoare, 1978] in which the entities have a continuous existence and carry out computation in parallel. This hardware/software relationship can be used to advantage as will be shown in Chapter 4.

1.3.3 Hierarchy

The use of hierarchy makes it possible to decompose the design in order to control the number of submodules that occur in a particular composition module, allowing control of composition module complexity. In irregular design sections, the hierarchy may be made deeper with a lower branching factor to keep the module count in the realm of Miller's estimate of human information processing capacity of seven items [Miller, 1956]. In regular sections of design, the branching factor can be increased to make use of repetition of identical modules and communication patterns (Section 1.3.6).

A particular aspect of the use of hierarchy in structured VLSI design is the restriction to a *separated hierarchy* [Rowson, 1980]. In such a hierarchy a distinction is made between those cells that comprise the leaf nodes of the hierarchy and all other non-leaf nodes. Actual active circuitry may only be present in the leaf nodes, all other nodes consisting of simple interconnections of leaf or non-leaf nodes. This separation of hierarchy allows the definition of mathematical operators for operations on the hierarchy, treating it purely as a recipe for the combination of functional units (the leaves). Such operators have been defined for analysis [Rowson, 1980] and structural assembly [Watson, 1985] of separated hierarchies.

An unresolved issue in structured VLSI design is whether there should be identical hierarchies used in the description of the system at different levels of abstraction, or whether for the sake of simplicity, there should only be a single hierarchical structure. This issue is addressed further in Chapter 2.

In the software domain, hierarchical decomposition is used in design in a similar manner to reduce complexity. Typically the elements of the hierarchy are subroutines rather than processes.

1.3.4 Information Hiding

Closely tied to the principles of modularity and hierarchy is that of *information hiding*: in the construction of a composition module from a number of submodules, the submodules are represented by an interface that presents only that information relevant to composition task. The nature of these interfaces varies over the range of design abstractions. At the structural level an interface may consist only of the names and types of logical signal pathways into the module, while at the physical level the interface must include information about port positions, layers, and module size and shape amongst other things. Figure 1.6 illustrates some of the interface criteria at different levels of abstraction. At all times the interface should only specify the minimum necessary information to keep the intellectual complexity of the design task as low as possible. Information hiding by means of module interfaces plays a major part in top-down design: submodules may be used as components in a composition prior to their implementation.

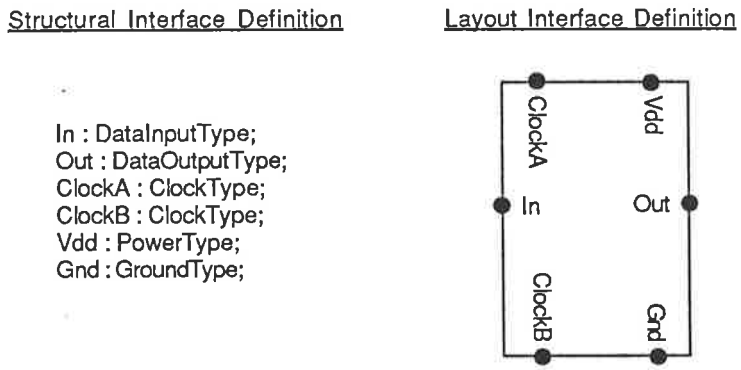

Figure 1.6: Information hiding by module interface design.

Information hiding in VLSI and software engineering are analogous: in software the details of a subroutine's implementation may be specifically separated from its interface.

1.3.5 Limited Constructs

One method for reducing the ease with which designers may create invalid layout structures is to limit the nature and number of the components that may be used in the design, and limit the means for composing them into larger structures. In

structured VLSI design this principle is applied in several areas. Firstly, designers are only permitted the use of "Manhatten" geometries in which all edges meet at right angles (Figure 1.7). This makes designs clearer in intent, reduces the margin for error by reducing for instance the number of configurations of transistors, and simplifies the construction and computational cost of design and verification tools. Secondly, layout modules may only have rectangular boundaries which must not overlap during composition and module interconnection may only occur over adjacent edges of modules via pre-defined ports: this simplifies the tasks of module composition and verification.

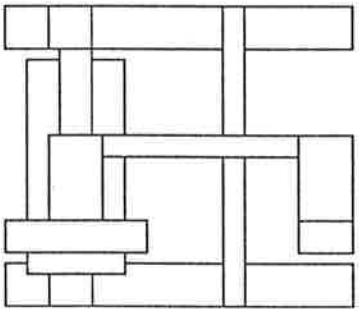

Figure 1.7: A layout using only "Manhatten" geometries.

The structured programming principle of limiting program structures to those of concatenation, limited selection and iteration based round the theoretic work of Böhm and Jacopini [Böhm & Jacopini, 1966] is analogous to the limited structures of structured VLSI design. In addition the syntax rules of a HLL preclude the generation of certain errors resulting from incorrect use of the limited constructs available. The limitation that layout modules communicate only through predefined ports is closely related to the software concept of parameter passing in function calls: data should only be transferred through explicitly declared channels, not via global accesses. Alternately, since in software modules communicate by means of control transfers in addition to data transfers, it is possible to equate communication through module ports with the avoidance of **goto** based global control transfers [Dijkstra, 1968].

1.3.6 Regularity

As discussed in Section 1.2.2, there are two aspects to regularity in structured systems: *function* and *interaction*. Regularity in function is achieved by the imposition of an appropriate partitioning on the design. Replicating modules has a number of

advantages. The complexity of the representation is reduced, leading to greater perpescuity. Design and computational savings are made in that only a single module need be designed/verified and then simply instanced a number of times. Figure 1.8 illustrates how functional regularity appears at the layout level, and how it may or may not be accompanied by regular communication patterns.

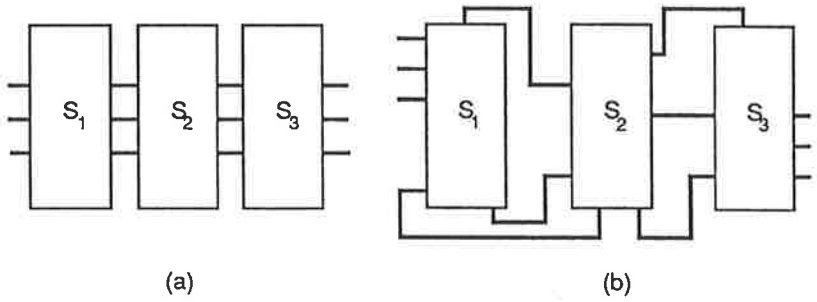

Figure 1.8: Functional regularity in a register set with (a) and without (b) regular interconnect.

Functional regularity tends to be a property of the low levels of large system hierarchies: within such elements as registers, adders and multipliers. At higher levels large systems (other than memories) are in general composed of non-identical modules. It is at this level that regularity in communication may be used to lower the complexity of module interactions. Figure 1.9 illustrates a data path in which the component modules have been designed with the aim of being interconnected according to a regular pattern.

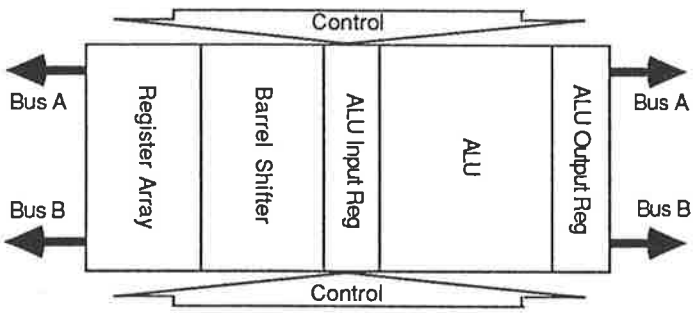

Figure 1.9: A data path segment: elements are connected in a regular fashion [Mead & Conway, 1980, p. 166].

The most appropriate software analogy to functional regularity is the application of

a function to a regular data structure. For example an image may be represented as a two dimensional array, and then the iterative application of a simple function to that array may be used to process the image—a single piece of code is used a number of times over a regular structure.

The use of interconnect regularity in software is less apparent, as the relative simplicity of intermodule communication makes it less necessary.

1.3.7 Design Procedures

Both structured VLSI design and structured programming are amenable to *top-down* design procedures in which the initial specifications are decomposed into smaller subproblems, a process that is applied recursively until the subproblems are small enough to be simply implemented by a block of layout or code. This process has been characterized in software as one of "stepwise refinement" [Wirth, 1971]. At each step in the decomposition, a set subroutines is proposed that when connected via their interfaces will perform the desired composite function. In this way the interfaces are designed first and guide the later implementation of the subroutines. Top-down design in structured VLSI is an analogous process in which modules are refined into interconnected submodules [Van Ginneken & Otten, 1984]. There are two primary differences in the application of stepwise refinement in the two domains:

1. The decomposition in the VLSI domain forms a *plan* that is used for the eventual bottom-up assembly of the layout blocks into a final design. In software there is no analogous assembly process, the "plan" simply being the procedure call sequence.

2. In VLSI systems, intermodule communication occurs through physical wires whose lengths affect the design quality. Additionally, these wires are constrained to share the same two-dimensional plane of the chip surface as the active circuit physical geometry. These additional constraints require that implementation be taken into account whilst the top-down decomposition process is carried out. Thus VLSI design tends to be a compromise between top-down decomposition to manage complexity and intermodule communication costs and bottom-up design to allow for performance effects and the difficulties of implementing active circuitry.

The primary advantage of applying top-down design to VLSI is that it emphasises the design of communication over other considerations. Lipp [Lipp, 1983] sites the following additional advantages of top-down design in VLSI:

1. It generates stable interfaces between functional components.
2. Design may be initiated before processes are stablized.
3. The bulk of a design will be technology independent, only the lower levels needing redesign for transfer of process.
4. It supports design by independent groups.

Not all structured VLSI design is carried out in top-down manner. In cases where modules have been predesigned for performance, or are created by generators, a bottom-up approach is required since the interfaces are not alterable. Most designs are actually created with a combination of top-down and bottom-up procedures. Even in the case of a purely top-down procedure, the requirement that a module be eventually implemented effects the design of the layout interface, introducing an element of bottom-up design.

1.4 Structural Design

Both structured VLSI design and structured programming are *methodologies*, not *algorithms*, for design. As such they share a number of problems:

1. *They are informal.* Both methodologies are comprised of a number of seemingly *ad hoc* techniques, advice and restrictions.
2. *They are not well matched to full automation.* Whilst it is possible to *assist* design (language directed and symbolic editors for instance) it is difficult to remove the designer from the process. Even attempting to automate only the layout phase is a complex task [Ackland et al., 1985].
3. *Their effectiveness is difficult to evaluate.* Even after a considerable research, there are still no widely recognized techniques for evaluating the quality of software (Section 2.1). Similarly the effectiveness of structured VLSI design appears to have evaded strict analysis.

Although some more rigorous analytic techniques have been applied to programming [Dijkstra, 1976] and VLSI design [Barrow, 1984] these are largely concerned with the *verification* of design correctness rather than the synthesis itself.

Even with their deficiencies, the structured design methodologies are regarded as being instrumental in the control of complexity [Séquin, 1986; Brooks, 1975], and work continues on their expansion and refinement.

For example although it provides a base for the design of complex software systems, structured programming is lacking in any precise procedures or criteria for the task of system partitioning. The design process was defined by step-wise refinement as a top-down development of interfaces followed by implementation. However the definition of these interfaces remained *ad hoc*, leading to low consistency and repeatability in software design.

The work of Yourdon and Constantine [Yourdon & Constantine, 1975] is aimed at finding more formal methods for partitioning software systems. Their techniques assume the underlying presence of *structured programming* to manage low level complexity, and introduces the concept of *structured design* for the management of higher level system complexity.

In order to avoid confusion in terminology, in this thesis the task of designing the *structure* of a system, that is the partitioning architecture, will be termed *structural design* after the suggestion of Yourdon and Constantine [Yourdon & Constantine, 1975, p. xvi].

Similarly to structured programming, structured VLSI design does not provide a basis for the design of partitions in VLSI systems, other than to suggest that it is generally achievable by way of a top-down procedure. The partitioning problem is far more complex in VLSI than in software because of the constrained nature of the VLSI medium. Not only must the problem be partitioned, but module interfaces must promote regular and space efficient function and interconnect patterns, and interconnect must meet performance requirements.

The need for the research into structural VLSI design described here has been motivated by several factors:

1. Prior to VLSI levels of integration, partitioning and module communication design were not difficult problems. The growing level of VLSI integration sug-

gests that larger complete systems are to be placed on single chips. Clearly this will produce greater complexity in the partitioning and module communication design [Ferry, 1985].

2. Some aspects of structured VLSI design do not encourage the creation of well partitioned designs. In particular the typical abstraction levels used make good interface design difficult as described in Section 1.3.1.

3. The major areas of VLSI CAD tool activity have been in the verification and assembly fields. There has been little software created to assist in the *planning* of design, their structural design in particular (Section 2.6).

This research will concentrate on the development of *structural* VLSI design techniques. In the next chapter an examination is made of the problems of software and VLSI partitioning. Changes and additions to the structured VLSI design methodology are then proposed that allow the use of software structural design techniques in the VLSI domain. Subsequent chapters describe a description language and related computer aided design tools that assist the modified design methodology.

1.5 Summary

The scale of current VLSI designs is limited not by technology but by design limitations. There are several properties of VLSI devices that make their design particularly susceptible to complexity problems:

1. The large number ($> 10^5$) of interacting components.

2. The inherently unstructured nature of the domain.

3. The planar nature of interconnect.

Without appropriate complexity management problems these factors my combine to overwhelm the capabilities of the designer and design tools. This is a particular problem with the functionally partitioned design styles that must be adopted in order to achieve VLSI circuit densities. Such styles are inherently complex as they are based on detailed design of module interactions.

The description of VLSI designs as structured systems offers the opportunity to manage such design complexity, and structured design methodologies have been developed for this purpose. Many of these techniques have been previously applied to software design in *structured programming.*

Structured VLSI design does not however address the issue of formalizing the partitioning process and it is this issue of *structural VLSI design* that provides the central theme of this research.

The work described in this thesis contributes to the field of VLSI design methodology and computer aided design in a number of ways:

1. It describes an alternative style of design representation that uses a single structural description hierarchy within a designer intensive planning phase. This focuses the designer's attention on system planning issues rather than distributing such decisions across a series of layers of design abstraction.

2. This representation is matched by a design methodology that emphasises the top-down specification of structure accompanied by parallel modelling of algorithmic function and physical form. One aim of the methodology is to encourage the use of software structural design techniques for partitioning the VLSI system.

3. It introduces a structural description language that supports the design methodology by encouraging the stepwise refinement of structural, communication, and functional primitives.

4. It describes a functional simulator that employs scheduling techniques that efficiently model the description language communication primitives. The simulator has facilities for aiding the qualitative and quantitative evaluation of the structural design.

5. A knowledge-based floorplanner is described that enables the designer to model the physical form of the structural design. The floorplanner uses inexact reasoning to perform physical design in a top-down manner without fully formed components. Appropriate domain knowledge enables it to incorporate structured design techniques in its results.

Chapter 2

Structural VLSI Design

As suggested in the previous chapter, there has not been a great deal of research in the structural design of VLSI systems. There is however a considerable body of literature associated with the structural design of software. It is only natural then to examine structural software design techniques in order to determine how they may be used in VLSI design. In this chapter, the criteria used in software partitioning are introduced, followed by a description of some of the more common structural software design methods. The criteria relevant to VLSI partitioning are then examined with the conclusion that the problems of VLSI partitioning present a *superset* of those of software partitioning. A design philosophy and associated design aids are then described. These allow the designer to select a structural design method appropriate to the problem at hand and apply the method to partitioning the design. Finally the methods described are compared to existing research in the field of VLSI partitioning.

2.1 Software Partitioning Criteria

There has been considerable research in the field of software engineering towards finding reliable metrics for the evaluation of the complexity of programs. Amongst other properties, such a metric should be sensitive to the quality of the structure chosen for the program. In the words of Evangelist [Evangelist, 1983], "we expect a good complexity metric to reward well structured programs by assigning to them a lower measure of complexity than would be given to equivalent, unstructured programs".

The two most studied metrics are those of Halstead [Halstead, 1977] and McCabe

[McCabe, 1976]. Another interesting measure is that of Henry and Kafura [Henry & Karfura, 1984] which is particularly sensitive to partitioning and resultant interface complexity.

Ideally such metric would provide quantitive guides as aids to the partitioning of programs. Indeed early empirical studies of these metrics indicated that there was a correlation between program complexity and the quantities derived [Baker, 1979; Henry & Karfura, 1984; Gordon, 1979]. More recent studies [Evangelist, 1983; Kearney et al., 1986] suggest however that the correlation is weak: the results of previous studies being misinterpreted as justification of the metrics. In fact Evangelist maintains that none of the metrics provides a better measure of complexity than a simple count of program source statements.

Given that there has been little success in deriving quantitative metrics program structure, it is appropriate to examine qualitative measures developed for the same purpose. Though not as useful for direct comparison of structures, qualitative measures may be used for both structural comparison and design, as they provide intuitive guides to partitioning.

Amongst the most useful [Bergland, 1981] of these qualitative metrics are *coupling* and *cohesion* introduced by Yourdon and Constantine [Yourdon & Constantine, 1975]. Coupling and cohesion may be used to evaluate the quality of a decomposition with respect to the complexity that results from the decomposition. Although developed to express concepts in the dataflow design methodology (Section 2.2.2), coupling and cohesion are relevant concepts in many domains and styles of structural design [Séquin, 1983].

In the remainder of this subsection the two concepts are introduced in terms of their original software design application.

2.1.1 Coupling

Coupling is a measure of the strength of interaction between modules in a partitioned design. Coupling is closely related to system complexity as defined in Section 1.2 as that property of a system that arises from the interaction of its parts. In structural design it is advantageous to reduce the interaction between modules, making them as independent as possible. In fact a zero point of coupling can be defined for two modules when they are completely independent of each other and the function of

each may be understood without reference to the other.

Coupling may seem at first a candidate for quantitative measure. For instance it might be possible to define coupling as the total data flow through an interface, and in fact Henry and Kafuras's complexity measure [Henry & Karfura, 1984] makes use of this. Closer examination reveals however that there are a number of different factors that influence coupling, and many of these are inherently qualitative [Yourdon & Constantine, 1975].

Complexity of interface. The more complex an interface is, the higher the coupling associated with it. This may be viewed as a measure of the "width" of the connection, the number of distinct paths that may be used to access the module. Yourdon and Constantine suggest that the number of parameters in an argument list is a reasonable estimate of this complexity. It is interesting to note that the measure is basically one of information *diversity*, without emphasis on the *mass* of the information.

Type of connection. Software modules pass information of two distinct classes between one another: *data* and *control*. There are two limiting types of connection for passing data and control between modules: *minimal* and *pathological*. Control is transferred over a minimal connection by naming the module to which control is being transferred to. That module has only one entry and exit point, and control return to the original calling point. Data is passed over a minimal connection by associating real values with named parameters in the called modules argument list. Minimally connected systems lead to low coupling because the interface of the module specifies all its external coupling: the control connection and the data connections. Pathological control connections are exemplified by the ubiquitous global **goto** statement which allows the transfer of control to some arbitrary point outside the module. Pathological data connections are made when a reference is made to a variable outside the module. In both cases in order to understand the function of the module it is necessary to refer to external modules, thus increasing the coupling.

Type of information flow on the connection. The simplest form of information that can be carried on a connection is *data* as it is clearly the minimal requirement for two data processing modules to cooperate. Data can exist in the absence of control if for example the modules operate synchronously. Adding control to a connection is often necessary, but does increase the coupling of the connected modules.

Binding time of the connection. The later that binding of variables to specific

referents occurs in the coding/compiling/linking/running cycle, the lower the inter-module coupling. For instance a screen editing program may have a module that contains all code referring to a specific terminal type. If modules may only by inserted at compilation time then to build the editor for a new terminal will require complete recompilation. If separate linking is available, a new terminal may be set at link time. If dynamic (run time) linking is available, the terminal need not be selected until execution time, obviously the preferable case for a text editor. Delaying binding in such a case clearly reduces coupling by reducing the number of steps that a connection must be processed through.

Common environments. When two otherwise independent modules both connect to a third module, then a common environment connection is created in which one module may well be implicitly coupled to the other via their common interest in the third module.

Yourdon and Constantine suggests that systems may be decoupled by:

1. Planning of the system structure.
2. Replacement of implicit references by explicit references: this reduces coupling as it easier to understand what can be seen than what is hidden.
3. Standardization of connections means that each new connection requires only an incremental amount of information to fully describe it, thus reducing the complexity of the interface.
4. Localization of the contents of common environments so that the amount of sharing and hence coupling is reduced.

2.1.2 Cohesion

The choices which guide the division of a system up into modules are not arbitrary: they can effect the structural complexity of the system. An important aspect of this is how closely the components of a particular module relate to each other: this is termed the *functional relatedness* or *cohesion* of the module. Clearly coupling and cohesion are related: in general, the greater the intramodule cohesion, the lower the intermodule coupling. Yourdon and Constantine suggest however that it is more useful to focus on cohesiveness in structural optimization.

Level of Cohesion	Scaling	Example
Coincidental	0	Repeated code
Logical	1	Input operations
Temporal	3	Initialization operations
Procedural	5	Iterative loops
Communicational	7	Checking & sorting
Sequential	9	Data flow
Functional	10	Square root

Table 2.1: Levels of Cohesion

In evaluating the functional relatedness of any two components in a module it is necessary to define a characteristic that the two share with respect to which their cohesion may be evaluated. These characteristics are quite general, and as such form a set of levels of cohesion. Yourdon and Constantine have defined these levels as shown in Table 2.1, their choices being guided by "...experiment, theoretical argument, and the practical experience of many designers" [Yourdon & Constantine, 1975, p. 146]. The scale attached to the levels is intended as a *relative* evaluation of the utility of the form of cohesion in reducing the complexity of a structured system.

Coincidental Cohesion. If the components of a module have little or no apparent relation then they are associated by *coincidence*. This occurs most often in cases where a piece of code is repeated, but the fact that the repetition occurs at all is coincidence. The code length may be reduced by replacing the repeated section with a subroutine call. The problem with this is that a change to that subroutine code may have undesirable effects as the call may have quite different meanings in different places. Coincidental cohesion increases the risk of errors in modification and reduces understandability.

Logical Cohesion. If the components of a module may be considered as being related at some "abstract" or "logical" level then they are *logically* associated. Examples of logical association classes are *input*, *output*, and *computation*. Logical cohesion is stronger than coincidental because it implies a structure that is more strongly related to the problem structure.

Temporal Cohesion. If the components of a module have a relationship bound

by *time*, then they are *temporally* associated. A typical example is an *initialization* module that performs a number of functions associated only by the fact that they need to occur at the same time. Temporal cohesion is stronger than logical because if logical associations are made in the absence of temporal associations, the code becomes convoluted as the program itself is basically a time ordered process.

Procedural Cohesion. If the components of the module are related by being sited in the same iteration loop or decision making clause then they are *procedurally* associated. The advantage of procedural cohesion is that it is quite strongly related to the problem structure. Its main weakness is that it tends to be fragmented in terms of function: a single loop often performs a number of parts of disparate functions, thus making it difficult to give a simple functional definition of such a module.

Communicational Cohesion. If the components of a module operate on the same input data and produce the same output data then they are *communicationally* associated. Communicationally cohesive modules are related to the problem structure through the dataflow graph, and thus present the possibility of being derived in a reasonably objective manner. A dataflow graph is one in which the nodes of the graph represent processes and the edges represent flows of data between processes (Figure 2.1).

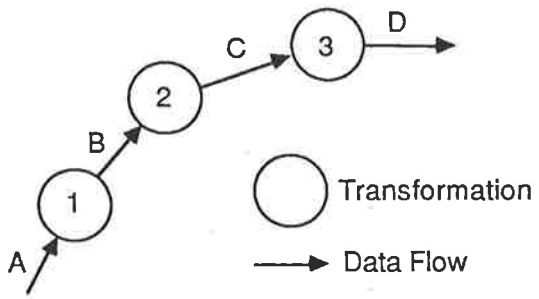

Figure 2.1: Example of a data flow graph.

Sequential Cohesion. If modules are designed such that the output of one module serves as the input to another, then the modules are *sequentially* associated. This is a higher level association than those considered so far and is clearly strongly problem related. A sequential module des however suffer from the fact that it may perform a number of functions and so still allows the possibility of division along non-functional lines.

Functional Cohesion. If the components of a module are related by the fact that

they perform a single function then there is said to be a *functional* association. Every component in the module contributes to and is necessary for the realization of the function. A simple class of functionally cohesive modules are those that perform mathematical functions: a subroutine for calculating a square root for instance has a single input, a single output and performs a single well defined operation. Yourdon and Constantine define a functionally cohesive module by negatives: if the associations in the module do not include any of those weaker association mentioned above, then it is functionally cohesive.

2.1.2.1 Summary

In an ideal system all modules are functionally cohesive. This ensures a strong relationship between the problem and the programming solution, greatly aiding designer understanding of the program. Making incremental changes to the program then becomes easier as altering one function is less likely to alter another by side effect as a high cohesion results in a low coupling. The practical problems of designing purely functional systems produce varying degrees of cohesion, all weaker than functional cohesion.

2.2 Structural Software Design

A number of design methodologies have been proposed for the structural design of software systems. Aspects of each of these methodologies are relevant to structural VLSI design. In this section a selection of three of these that are broadly representative [Bergland, 1981] of the methods in use will be presented. Bergland also discusses *programming calculus* [Dijkstra, 1976] however this is primarily a means of *verifying the correctness* of programs.

2.2.1 Functional Decomposition

Functional decomposition is the most intuitive software design methodology as it is based on the traditional problem solving technique of *divide and conquer*. The technique is well established in software design under a number of synonyms including "top-down design" [Dijkstra, 1972] and "step-wise refinement" [Wirth, 1971]. The

first formal application of functional decomposition to software design is often attributed to Dijkstra [Dijkstra, 1970]. The three basic phases of design in functional decomposition have been defined [Linger et al., 1979] as:

1. Clearly state the intended function.
2. Divide, connect, and check the intended function by reexpressing it as an equivalent structure of properly connected subfunctions, each solving part of the problem.
3. Divide, connect, and check each subfunction far enough to be comfortable.

This process can be viewed as one of *step-wise refinement* in which the programmer "solves" the problem using a small set of "proposed" high level problem oriented instructions. Each of these powerful instructions is then implemented in turn with less powerful ones, and the process repeated until the instructions of the target programming language are reached.

Advantages. The primary advantages of functional decomposition are its intuitive nature and wide applicability. Partitioning a system in such a flexible *top-down* manner encourages the designer to investigate alternative partitioning schemes: this is important because of the qualitative judgements that need to be made in designing a good partitioning. By Yourdon and Constantine's measures of cohesion (Table 2.1) the functional cohesion produced by functional decomposition is the most effective cohesion class.

Disadvantages. The main disadvantage of functional decomposition stems from its generality: by not specifying exactly *what* is the basis for creating a decomposition, the technique can give rise to a large number of alternate implementations of a design. A problem may be decomposed with respect to sequence, data access or data flow depending on the views of a particular designer.

Bergland suggests that functional decomposition serves as a base for many of the other programming methodologies. These are basically concerned with specifying less general criteria for carrying out decomposition, resulting in less variability and greater repeatability of structural designs.

2.2.2 Data Flow Design

The technique of data flow design was first embodied in Yourdon and Constantine's "transform-centered" design strategy [Yourdon & Constantine, 1975]. Data flow is a specialization of functional decomposition in which each module is a "black box" in the engineering sense: a device that transforms an input stream to an output stream. These "boxes" are then interconnected to solve the problem. The most important transformation that takes place in the strategy is that required to transform the "flat" data flow graph representation of the system into a hierarchical program structure. This is done by first identifying the most abstract input ("afferent"), output ("efferent") and processing ("central transform") modules. This process is applied recursively to each of the top level modules in order to construct a hierarchy. Ideally the resulting system morphology is one is fully "factored": all function is in the leaf modules, the composition modules simply being present to structure the system. This design procedure is illustrated in Figure 2.2 and may be summarised as:

1. Model the program as a data flow graph: a graph of single input/single output nodes.
2. Identify the two top level afferent and efferent modules and any number of central transform modules.
3. Factor the afferent, efferent and central transform modules to form a hierarchy. Factoring proceeds breadth first: all the subordinates of a module must be defined before any of them are further decomposed.
4. Write the code.

Advantages. Being based on data flow, the structure resulting from this technique is sequentially cohesive, and hence of high quality according to Constantine's measures. The greater formality of the decomposition should result in increased consistency of design.

Disadvantages. It is not clear how well this structure maps onto real problems and hence may result in poor problem/program structure correspondence. The more confined style discourages the designer from exploring alternatives. Bergland suggest that although the top level transformation from the data flow graph results

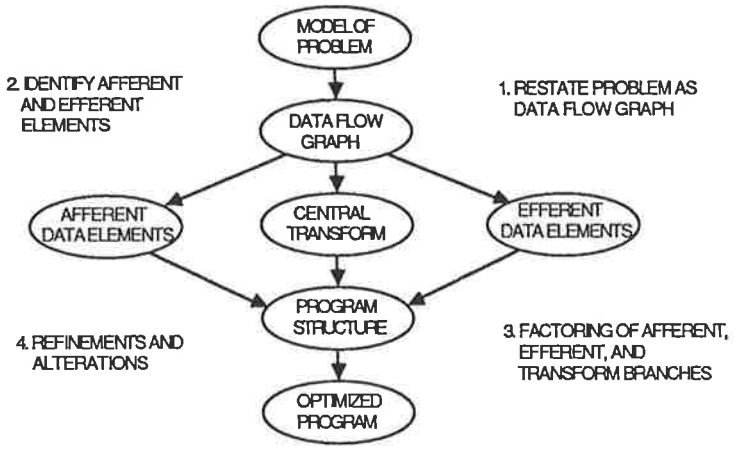

Figure 2.2: Procedure for data flow design.

in sequential cohesion, the factoring process may result in less desirable procedural cohesion.

2.2.3 Data Structure Design

The data structure design method developed by Jackson [Jackson, 1975] is perhaps the most systematic of the available structural design procedures. The basic procedure is to design a hierarchical set of data structures that closely relate to the real world problem. Program structures are then built around these data structures resulting program that contains a correct model of the real world. The procedure has the advantage that it consists of a number of independent steps, thus reducing the complexity of the design procedure as well as that of the design itself. The procedure may be summarised as [Cameron, 1983]:

1. Draw a system network diagram that models the problem.
2. Draw structure diagrams to represent each data structure input or output from the program.
3. Form the program structure diagram by merging the various data structure diagrams.
4. Make up a suitable set of elementary operations out of the programming language to be used and allocate these into the program structure.

5. Translate the program structure diagram into a program text.

This process is illustrated in Figure 2.3.

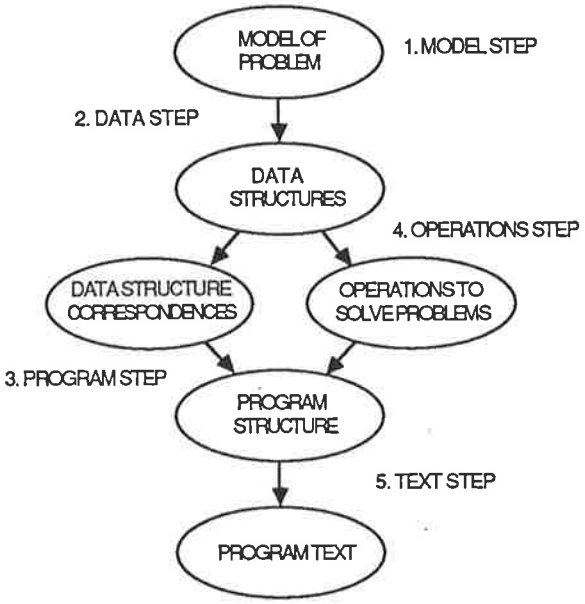

Figure 2.3: Procedure for data structure design.

Advantages. The primary advantage of the technique is that it is constructed of a number of well defined steps, resulting in reduced complexity of application and increased consistency of designs. Bergland suggests that it leads to functional or communication cohesion, both desirable.

Disadvantages. The main disadvantages are that the procedure has not yet been developed for application to large systems, and the initial choice of "correct" data structures is not always simple. In addition the real world modelling is done via the data structures, so it may not be clear how well the final structure corresponds to the problem.

2.2.4 Summary

Functional decomposition provides the greatest possible number of alternative structures thereby opening up the possibility of getting either a very good or very poor result, depending on the skill of the designer. Problem modeling and program construction are performed simultaneously, increasing the intellectual complexity of the

task.

Data flow design produces a good sequentially cohesive structure at the top level but otherwise shares similar properties to functional decomposition. The data flow chart provides a mechanism for separating problem modeling and code writing, but the exercise of converting the graph to a calling hierarchy may produce an inappropriate structure.

Data structure design is the most repeatable method as it is based on a well defined set of independent operations and relies least on the skills of the individual programmer. Basing the program structure on the data structures tends make analysis of the resulting structure in terms of cohesion rather difficult.

Bergland [Bergland, 1981] suggests that none of these methods is sufficient by itself, but that a combination of methods may be used successfully for most software design problems.

2.3 VLSI Partitioning Issues

The issues in the design of large software and VLSI systems are similar in many respects. VLSI does however present a relatively constrained medium, and this gives rise to a number of differences. In this section the similarities and differences in design between the two domains are examined. This will serve as an introduction to the structural design philosophy described in the next section.

2.3.1 Large Systems

Early LSI circuits integrated only a few gates onto a single chip. The device densities achievable with VLSI have introduced the integration of complete systems containing over 10^5 devices onto single dies. The large amount of functionality available on a chip has enabled designers to implement VLSI systems of similar complexity to conventional large software systems (Brooks defines a *large* program as having greater than 30,000 instructions [Brooks, 1975, p. 90]). Additionally, many of the systems now being implemented in VLSI are functionally similar to those that commonly appear in software. Function has been migrated to VLSI typically in order to achieve higher performance.. For example language interpreters [Sussman, 1981], high level

image processors [Petajan, 1986], and speech recognition devices [Mudge et al., 1984].

2.3.2 Unstructured Mediums

Both software and VLSI design take place within relatively unstructured mediums on which the designer is free to impose an appropriate structure. It is this freedom that raises the issue of structural design. The basic intent of this structure in software is to control complexity [Yourdon & Constantine, 1975]. Software systems are hierarchically partitioned to ensure that the complexity that must be dealt with by a designer at a particular time is kept low. The concepts of *coupling* and *cohesion* assist in the design of partitionings that reduce this complexity. The situation is similar in the case of VLSI design, with the additional problem that the constraints of the silicon medium must be taken into account as described in the remainder of this section.

2.3.3 Concurrency

In many software systems components are distributed across time, only one element performing a function at a time as directed by the calling hierarchy. VLSI systems are however inherently concurrent in nature. The system components are distributed across a surface, not time, and hence are capable of operating in parallel. The concurrent properties of a VLSI system allow the designer to maximize performance by partitioning the problem into the maximum number of simultaneous operations. Sequin has identified seven classes of concurrency that appear in VLSI designs [Séquin, 1983]. These are shown in Table 2.2.

The great scope for concurrent operations is VLSI is one of the reasons it is possible to achieve high performance implementations of algorithms but it adds greatly to the design complexity. Concurrent execution gives rise to a number of possible errors that do not occur in sequential systems. These errors are listed in Table 2.3.

The use of concurrent elements in a system increases its complexity by increasing the *coupling* between the components. The intermodule connections are not simple "one-shot" control and data transfers. Instead a complex interaction involving data being ready at a particular time is maintained continuously as VLSI modules exist continuously unlike software subroutine modules.

Class	Example
Bit Concurrency	n-bit parallel adder
Vector Operations	matrix multiplication
Pipelining	overlapped instruction execution
Set Concurrency	evaluation of alternatives
Specialist Functions	co-processors
Task Concurrency	communicating processes
Random Concurrency	everything else

Table 2.2: Classes of concurrency [Séquin, 1983].

Deadlock	cyclic data dependencies
Timing	data read before settled
Buffering	unequal rates of data production/usage

Table 2.3: Classes of errors in concurrent systems.

Whilst the bulk of software is sequential, the appearance of parallel architectures is driving the development of techniques for software development in concurrent environments. These are in the main notations and mechanisms for implementing parallel algorithms [Hoare, 1978; Hillis, 1986; Ahuja et al., 1986] that will be examined further in Chapter 4.

2.3.4 Communication

Communication between modules in a VLSI system is carried out at a quite considerable cost. Wires used for interconnection use up chip area, introduce signal delays and increase power consumption. This cost affects partitioning decisions in two ways:

1. The *number* of interconnections between modules should be minimized. This corresponds to minimizing module coupling and maximizing module cohesion during partitioning.

2. The *length* of interconnections between modules should be minimized. Given fixed position modules it may be possible to reduce the length of interconnect by migrating function across module boundaries. The issue of selecting module placement to minimize communication length is dealt with in the following subsection.

Intermodule communication in software has very little cost. The number of interconnections corresponds to the number of variables passed to a subroutine. The cost of variable passing is negligible unless they are to be copied for use by the receiver. Minimization of coupling is performed to reduce conceptual complexity rather than for performance reasons. Similarly there is no strong analogy to the *length* of connections except that accessing data widely distributed through memory may entail delays due to swapping.

2.3.5 Packaging

In a VLSI system, the modules and their interconnect share a two dimensional surface. This constraint gives rise to two interrelated issues that affect the partitioning of the system: module *placement* and module *interface design.*

Module Placement. Two primary factors guide the selection of positions and orientations of rectangular layout modules on the chip surface: *total area* and *interconnect length* minimization. The two are not independent so finding an optimal placement solution often involves a compromise between the two.

Software running on hardware with limited real memory may require a degree of manipulation of module placement in the memory space. This may be necessary in order to fit the code and data into available memory or it may be desirable to localize accesses to avoid paging from slow secondary memory. In general however software module placement is not a high level design issue.

Module Interface Design. The design of a module interface in a software system typically only requires the specification of the identifying name of the module and formal parameter names of the variables. This can be done without reference to the actual contents of the module or the underlying machine implementation. In VLSI layout design the situation is far more complex. The module interface design is influenced by:

1. The external connections of the module.
2. The external space in which it is intended the module should fit.
3. The internal connections of the module.
4. The internal space required by the layout contained within the module.

Design of a VLSI partitioning involves not only allocating function to particular modules, but also the physical design of the module interfaces. As illustrated in Figure 2.4 interface design affects both the packing density and area occupied by interconnect.

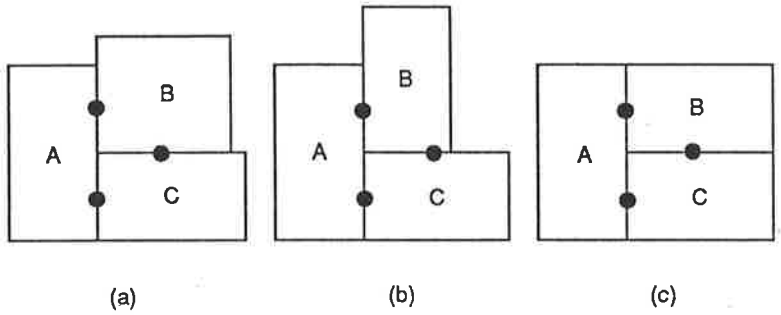

Figure 2.4: Interface design: (a) poorly designed for space packing, (b) poorly constrained by interconnection and (c) well designed for packing and interconnect.

2.3.6 Regularity

Regularity appears in software as multiple calls to the same subroutine. Such calls may be of general use and appear in functionally unrelated places, for example input-output routines. Some subroutines are more specific in use, but are called inside iterative loops to carry out repetitive functions on regular data structures. Both classes of software regularity provide leverage by reusing a single code section a number of times. In VLSI design, regularity appears in three classes: *functional* and *interconnect.*

2.3.6.1 Regularity of Function

In many large VLSI designs it is possible to reuse module definitions in a number diverse of places. In the TFB chip [Eshraghian et al., 1984] for example there are

multiple uses of the same adders, multipliers and memory blocks. This form of repetition reduces the functional and layout complexity of the system.

2.3.6.2 Regularity of Interconnect

A powerful method of reducing the complexity of a VLSI layout is to provide for regular communications between the various layout modules. This is one of the advantages of layout regularity: the use of identical modules gives rise to regular interconnect. At a higher level however it is possible to design the layout interfaces (port position and module sizes) in such a way as to provide regular module interconnect. The most common example of this is data path layout. Rather than design the components in isolation and *then* attempt to interconnect them, component modules are designed so as to have compatible interfaces.

2.3.7 Performance

The instruction sets of digital computers provide a clean and well characterised target for the translation of high level languages. Although the overall efficiency and performance of a software system must be considered during partitioning design, the execution times of individual instructions typically do not predicate the correctness or failure of a design. VLSI designs are far more dependent on timing issues. Individual modules perform their functions in finite times, and these results are communicated between modules also in finite times. The use of global clocks for synchronization reduces the complexity introduced by such delays (Chapter 4). Timing issues do however remain a major concern in meeting VLSI performance specifications, in particular the effect of long wiring runs in the layout design and the delays they introduce.

2.3.8 Discussion

The tasks of software and VLSI design share the following characteristics:

1. A large number of interacting components (instructions and devices respectively).

2. Complex functionality of a similar nature: exemplified by the migration of algorithms from software to VLSI for additional performance.

3. Unstructured domains in which the designer has control over the physical structuring that is to be used.

4. Concurrent functionality in which a number of computational elements perform calculations in parallel and pass data and control to one another. This is more common in VLSI, but a growing area of software design.

5. Regular functionality in which it is useful if the same *definition* of a component may be used a number of times.

The tasks of software and VLSI design differ in that:

1. Communication is expensive in VLSI: communicating components should be physically close. There is no strong analogy in software.

2. Area is expensive in VLSI: components must be designed and packed together such that they consume minimum area. Though space is a consideration in software, it is usually not a major design criteria for optimization.

3. Performance is a major issue in VLSI: all VLSI systems are "real time" and are required to meet timing constraints. Only a portion of software systems have similar constraints.

4. Regular interconnect is useful in VLSI: it reduces the complexity of layout module interfacing and reduces interconnect. The simpler and relatively inexpensive data passing mechanisms of software make this a less important issue.

These similarities and differences suggest that: *structural software design techniques are appropriate to the structural design of VLSI systems* given that *allowance can be made in the design process for the constraints implied by physical layout.*

2.4 Abstraction and Hierarchical Equivalence

As indicated in Section 1.3.1, structured VLSI design is typically carried out within three levels of abstraction: *behavioural, structural* and *physical.*

Horizontal abstraction schemes of this general form have the following advantages [Stefik et al., 1981b]:

1. The design description at each level that is free of certain classes of design errors ("bugs").
2. The form of design description at each level serves to narrow the designer's attention to particular concerns, reducing the complexity of the design task.
3. The powerful domain oriented abstraction provided by the higher levels support the rapid search by the designer through alternate solutions for an optimum.

Such schemes of horizontal levels of abstraction do however have a number of problems when viewed as environments for structural design:

1. Whilst initial partitioning may be specified at the highest level, actual geometrical layout information does not appear till the lowest level. The initial partitioning takes place without any geometrical information being available till a level transformation has taken place. Layout has been relegated to a very low level *detailing* task whereas in custom VLSI design it must be considered at an early stage, with floorplans providing a suitable abstraction away from the detailed mask geometry.
2. The increased chance of errors being introduced during the translation between the relatively several levels.
3. The lack of the descriptive capability to add in more detailed design information in a continuous manner. For instance if a designer wishes to add clocking strategies into a portion of a behavioural design, the entire module must first be translated to the structural description level.

An additional problem that arises from the use of horizontal description levels is that of *hierarchical equivalence.*

Rowson relates the function of composition modules in a separated hierarchy to combinators [Rowson, 1980]. He then proves that even given that two hierarchies operated on the same set of leaves, the issue of whether the two hierarchies are topologically identical is undecidable. This result implies that the only practical way of managing a design at multiple description levels is to require that the component

hierarchy be identical at each description level. If this is the case then it is possible to at least partition the cross-description verification problem to one of verifying the consistency of individual modules rather than the entire system.

The need for equivalent hierarchies can be seen to be contradictory to a methodology employing multiple levels of description. By definition the issues addressed in the design at different levels are dissimilar and are likely to lead to variations in the design partitioning.

2.5 Structural VLSI Design

In this section an approach to structural VLSI design is outlined that is intended to aid the development of complex VLSI systems. The approach is designed to support the application of software structural design techniques such as those described earlier in this chapter to the VLSI domain. In addition, the generality of the approach ensures that further developments in software structural design—those in concurrent system [Witt, 1985] for example—may also be adopted.

It is necessary to support a spectrum of structural design techniques rather than a single method because of the varied nature of the systems being implemented in VLSI. Some systems are amenable to a function decomposition style of design, a typical example being a 32 bit processor [Krambeck et al., 1982]. Signal processing systems fall into a data flow style of design [Ligtenberg & O'Neill, 1985; Jhon et al., 1985; Denyer et al., 1982]. Systolic array architectures are often well matched to a data structure style of design [Navarro et al., 87]. The designer should be able to select the appropriate strategy based on the *functionality* of the system under design, not the limitations of the VLSI design methodology.

The strategy to be adopted is based on dividing the VLSI design process into a designer intensive *planning* phase and an automatic *construction* phase. Within the planning phase the design is represented as a single structural hierarchy as is the case with software design. The particular constraints an optimizations required of a VLSI implementation of the structural design are examined by means of computer aided *modelling*.

2.5.1 Planning and Construction

Typically structured VLSI design is regarded as being a process of successively refining descriptions from the behavioural level to the layout level through one or more intermediate levels. The *planning* of the design occurs at each level of description at different times. By contrast, in software design, planning occurs in the top down successive refinement of the design hierarchy in *at a single level of description.* This description level is provided by the programming language itself. The transformation to a lower level of description, that of the machine instruction set, is carried out automatically by a compiler. As discussed earlier (Section 1.3.1) there are difficulties in creating a broad spectrum compiler for custom integrated circuit layout. There are however a number of techniques for *assembling* portions of designs into complete layouts given an appropriate plan [Watson, 1985; Wardle et al., 1984].

These factors lead us to observe that design be separated into two well defined phases: *planning* and *construction.*

The *planning phase* may be characterized by the rapid generation by stepwise refinement of possible design solutions in an abstract representation. Solutions are evaluated with respect to estimates of functionality, efficiency and performance. This process results in a set of *plans* that can be used for the *construction* of the actual layout. Plans are comprised of:

1. A structural description of the design.
2. A description of the algorithmic function of each module in the structural description.
3. A description of the physical form for each module in the structural description.

The physical form may be represented by module floorplans in the non-leaf modules, and circuit diagrams in the leaf modules.

These elements of a plan are illustrated in Figure 2.5.

The *construction phase* may be characterized by the use of the *plans* to design and assemble components to form a complete layout. The construction process involves:

1. Using leaf module floorplan and circuit diagrams to create leaf module circuit layout.

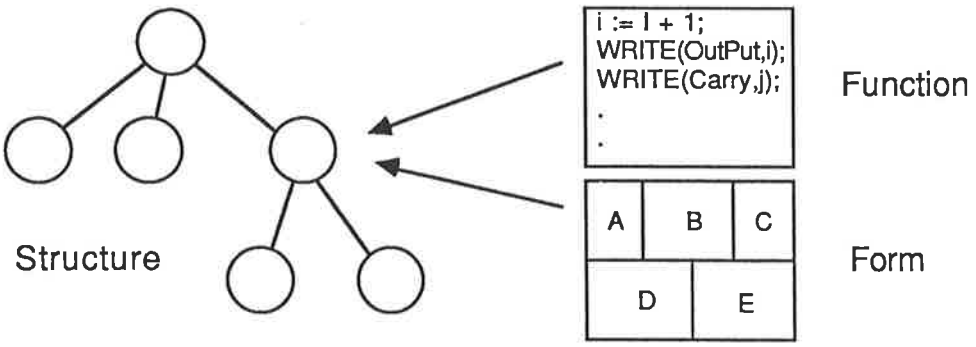

Figure 2.5: Elements of a plan.

2. Using the functional description of each leaf module to verify the correct function of each leaf module layout.

3. Using the floorplans as a guide to assembling the leaf modules into a final layout.

With existing CAD technology it is possible to automate the bulk of the construction phase. Expert systems are available for leaf module layout [Kimm et al., 1984; Kollaritsch & Weste, 1984; Watson, 1987]. Given a functional description, verification against a circuit simulation is feasible [Dickinson et al., 1987]. The construction of layout based on hierarchical floorplans is well documented [Watson, 1985; Wardle et al., 1984].

The planning phase is still largely a human activity. The complex tradeoffs involved in the design process are concentrated in planning. Lipp [Lipp, 1983] notes: *...manual design in general is not guided by a set of formalized criteria. Instead of pure metrical measures, a complicated mixture of metrically and intuitively based evaluation rules prevails which cannot be mapped successfully for automated design.*

This form of separation has the advantage that the designer's attention is focussed on system-level decisions that are made during the planning phase. Additionally, the bulk of technology dependent factors may be relegated to the automatic construction phase. This eases the designer work load and encourages the transfer of system design across technologies.

The implementation of a full design system based on this approach is described elsewhere [Dickinson et al., 1987]. The objective of this thesis is to examine in detail the requirements of the planning phase of the design.

2.5.2 Design Description

The primary element planning a complex system has been identified as the design of a *structure* for the system. We shall propose that this should be the central element of our approach to planning. The structure is a separated hierarchy (Section 1.3.3) of modules. The creation of the structure is guided by two primary considerations: the algorithmic *function* and physical *form* of that may be realized with that structure. In fact function and form will be regarded simply as aspects of the modules in the structure as suggested by Figure 2.5.

Unlike the more usual model of horizontal abstractions (Section 1.3.1), this presents a *vertical* arrangement in which a *single* hierarchy is incrementally developed. This removes the problems of multiple hierarchies for each abstraction level (Section 2.4). Such a structure is also the primary means of controlling the complexity of a system (Section 1.2).

The structure, when taken in conjunction with a functional description of each module, may be treated as a *program* that implements the same *function* as the VLSI system. This establishes a link to software design, and presents a base on which software structural design techniques may be applied. The application of these techniques does require that considerable effort be put into the appropriate design of the language to be used. Existing programming languages provide insufficient support for describing VLSI concepts in a manner appropriate to structural design: communication and regularity in particular. The design of such a language is described in Chapter 3.

2.5.3 Modelling Algorithmic Function

The nature of software design is such that after specification it is possible to compile and run the program that *is* the specification. The feedback from this assists in the discovery of errors, and evaluation of the design with various tools for analyzing the running code. In order to carry our analogy with software further, it is necessary to provide a means of compiling, running, debugging and profiling the VLSI functional specification as an aid to design evaluation.

The modelling of a VLSI structural design is particularly important as the cost of intermodule communication is often the limiting factor in the degree of integration

possible (Section 1.1.2). In Chapter 4 a functional simulator is described that aids the designer in evaluating relevant properties of the structural design, together with validating the correctness of the functional decompositions used in the design.

2.5.4 Modelling Physical Form

The risk of applying these software techniques is that the importance of *form* in VLSI design be ignored. It is not appropriate to insist that the designer specify the precise physical form of each module as required by some languages [Segal, 1981; German & Lieberherr, 1985]. This detracts from the application of good structural design techniques: the additional effort required by the designer to specify the physical form produces a reluctance to examine alternative solutions. The form of the proposed structure should be modelled by the automatic generation of floorplans (Chapter 6) as an aid to design evaluation.

2.5.5 Structural Design: Requirements

In summary, the support of this approach to structural design requires the following elements:

1. A *high level language* for the specification of the system structure and the function of each module in it.
2. A *simulator* that allows the *function* to be modelled as design proceeds.
3. A *floorplanner* that allows the *form* to be modelled as design proceeds.

The design of all three of these elements is affected by the need that their mode of use be compatible with software structural design techniques. This requires that their design vary from that found in existing hardware description languages, simulators and floorplanners. These variations will become apparent in subsequent chapters that describe the respective elements.

2.6 Related Research

Comparisons between this design approach and other research into VLSI partitioning is complicated by the fact that the area of structural VLSI design being addressed here is one that has received little direct attention. As this *structural* design philosophy is intended as to *supplement* rather than *supplant* structured VLSI design, a comparison is not appropriate between the two. The following subsections do however describe items of research that are at least partially connected to the partitioning problem described here. The particular design language and related design aids that are described later in this thesis will be compared to similar research in their respective chapters.

2.6.1 Graph Partitioning

There has been considerable research into the problem of general digital system partitioning [Kernighan & Lin, 1970; Schweikert & Kernighan, 1972; Breuer, 1976; Payne & Van Cleemput, 1982; McFarland, 1983; Lauther, 1979; Healey & Gajski, 1985]. In such systems an *existing* structural arrangement is re-arranged using a graph partitioning algorithm. The result is in general a set of clusters of the basic design elements. The elements in each cluster are related by some common factor: usually their connectivity [Schweikert & Kernighan, 1972] or their need to access a shared resource (an ALU for instance) [McFarland, 1983]. This tends to be *bottom-up* in style only, and can only improve the design within the constraints provided by the predesigned components. One of the primary problems is that in general the number and nature of the partitions must be chosen prior to the application of the procedure. The various design components are then allocated into one or other of the partitions in order to minimize the cost function.

2.6.2 Stepwise Layout Refinement

Otten [Van Ginneken & Otten, 1984] introduces the concept of the stepwise refinement of floorplans. This is a *top-down* process as is the case with program design. Otten does not however deal with the issue of relating function and form through the use of a structural description: the primary thrust is that floorplans may be stepwise refined by the use of hierarchical slicing trees. The modelling of function is

not introduced. The use of a slicing structure in the development of floorplans has the disadvantage that it encourages *overcommitment:* once a component has been allocated into a slice it may not be moved at a later stage in the design.

2.6.3 Partitioning Evaluation

Resnik [Resnik, 1986] describes an aid to system partitioning. It is a "spread sheet" program that assists the designer by allowing evaluation of the effect of various partitionings on a number of metrics of design quality. The program does not however provide a basis for design: neither form nor function are modelled.

2.6.4 Integrated Descriptions

A number of systems have been described [Buchanan, 1980; Segal, 1981; German & Lieberherr, 1985; Morel et al., 1982] that have design representations that allow the designer to specify the module physical form in the same textual description as the structure and function. Clearly this is useful in that it makes it easy for the designer to specify the physical form of a module in with its function. However this has several disadvantages in the context of structural design:

1. The designer is required to create a new floorplan for each alteration in structure. This discourages experimentation with structure.
2. There is no evaluation of the floorplan, it is simply specified and accepted.
3. The floorplan description is embedded in the structural description. There may however be a number of alternative floorplans for a particular module, and these evolve during the design as more constraints are created. Including the floorplan in the structural description precludes this mutability.

There are functional simulators associated with each of the languages, however they do not provide facilities for the specific analysis of *structure* other than to allow the specification to be executed.

2.7 Summary

The concepts of *coupling* and *cohesion* provide a basis for the qualitative evaluation of structural designs, particularly in respect to managing complexity.

There are a number of software structural design techniques, but none is regarded as completely general: often a mixture of the techniques is required in a design. If such techniques are to be successfully adopted to VLSI design, a design representation must be selected such that the task appears as similar to the software case as possible: this will ensure that present and future developments will be useful in VLSI.

There are a number of similarities between the tasks of software and VLSI structural design. In particular the issues involving algorithmic *function* in the two domains are closely related. Many of the complexity management issues are similar due to the unstructured nature of both domains. The two diverge however over the issue of physical *form*. Software in effect has no physical form: in most cases the translation to machine code is transparent to the designer. In VLSI however the constraints of the medium require that the form be considered from the *earliest* stages of design.

VLSI design may be divided into a designer intensive *planning* phase and an automated *construction* phase. This partitioning of the task allows the designer to concentrate on the critical planning phase of structural design.

A structural VLSI design philosophy may be based round the concept of a single structural hierarchy of modules. Associated with each module is an algorithmic *function* (a section of code) and a physical *form* (a floorplan).

The structural description together with the function of each module may regarded as a software implementation of the VLSI system, and software structural design techniques applied to it. In order to provide the designer with continuous feedback on the quality of the partitioning being developed it should be possible to model the structural design both in terms of its function and form. Function may be modelled with a functional simulation and analysis tool. Form may be modelled with a floorplan generator. Both modelling design aids must be suitable to use within the framework of software structural design techniques.

In subsequent chapters a structural and functional description language will be described, together with design aids for modelling the function and form of the design.

Chapter 3

A VLSI Description Language

3.1 Introduction

A central feature of any design system intended to support the structural planning methodology outlined in Chapter 2 is a *description language*. Such a language is used by the designer to formally specify a structural design as a precursor to modelling its function and form. There are a variety of requirements that must be fulfilled by such a language: of particular interest here is the ability to describe complex VLSI systems in a manner that both reduces explicit complexity and aids the application of structural design techniques.

This chapter describes *Pink*, a language for the hierarchical specification of the structure and function of VLSI systems. Whilst based on a conventional programming language, there is a set of additional features that support structural VLSI design:

1. A model of intermodule communication that allows information to be moved between modules with or without signal delay specification. This enables designs to be prototyped rapidly without the inclusion of intricate delay information, and successively refined into a more detailed delay based description if required.
2. A model of module functionality that allows for the preservation of control state between module invocations. This allows descriptions to include timed pauses at a particular point in the functional description.
3. A general syntax for expressing regularity in the structure of the design.

4. A model of structure and an associated hierarchical naming scheme that allows for the clean specification of module definitions, module instances and ports.

Function is specified in the Modula-2 [Wirth, 1982] language with a number of additional primitives. This approach has the advantages that designers require little additional expertise to program in the Pascal-like syntax, and the simulator (Chapter 4) can make use of Modula-2 compilers to produce an efficient simulation rather than having to interpret the functional specification.

As background to the language, an overview of existing research in VLSI hardware description languages will be given. The language itself (known as *Pink*) is then described, followed by some examples of its use in system description.

3.2 Hardware Description Languages

The need for designers to specify and simulate digital designs prior to fabrication has motivated the development of a variety of description languages. These languages are quite diverse in terms of the paradigms they use for design representation. In this section four overlapping classes of description language are described in terms of their basic philosophy, advantages and disadvantages.

3.2.1 Register Transfer Descriptions

Register transfer languages may be characterised as those that express the function of a system in terms of the transfer of data between registers: elements that may preserve state. Register transfer languages are largely functional: the level of description does not record a great deal of structural information, it is intended that this be added at a later stage in the design process. Instead the register transfer language description represents a *virtual architecture* without implementation details. One of the most widely known register transfer languages is ISP (Instruction Set Processor) [Siewiorek et al., 1982] and related languages including ISPS [Barbacci, 1981] and ISP' [Rose et al., 1983]. ISP is aimed specifically at the description of computer architectures. The basic execution paradigm is one of interpretation of instructions in an instruction register giving rise to data accesses, data transformations and data stores. A further example of a register transfer language is the clocked register level

of the *Palladio* system [Stefik et al., 1981b].

The basic advantages of register transfer languages stem from the high level of abstraction they provide. By supplying a single class of structural component (the register) and a single mode of operation (instruction interpretation) the description of a processor may be made clearly and concisely. Alternative implementations of the design may be rapidly proposed and evaluated.

There are however several disadvantages to this class of description:

1. *Lack of generality.* The languages are targeted at particular classes of hardware. For instance ISPS is useful for describing instruction set based machines, but may be less applicable to other architectures.

2. *Inflexible Abstractions.* The basic entities of the register transfer language are fixed. This reduces the ability of the designer to firstly build up higher level abstractions that may be used to represent complex designs more simply. Secondly it limits the designer's ability to *refine* the design down into an arbitrarily detailed form as required in a process of successive refinement. In some well structured cases (such as MacPitts [Southard, 1983]) the register transfer level may provide all the information needed to produce a suitable hardware architecture. In the general case however far more data on the detailed structure of the design will be required in order to construct a real system.

3.2.2 Token Passing Descriptions

Although register transfer descriptions are simplified to the extent that they are basically formed from descriptions of data transfers, they do not constrain the nature of those transfers: it is still quite possible create descriptions that contain errors. A means of detecting such errors in a description is that of *token passing.* Elements of the design may be specified in terms their input and output: inputs *absorb* tokens and outputs *emit* them. The token paths between modules are combined with a restricted set of composition rules. Correctly designed, such a system will exhibit the provable properties of a *Petri net* [Petri, 1966; Peterson, 1977]. A Petri net is a graph around which tokens are passed in a structured manner. Once a design has been described in a suitable form, the results of Petri net theory may be used to prove certain properties of the description: in particular that data is not being used

before it is settled and that deadlock conditions do not arise.

One widely known token passing language is that of Linked Module Abstraction (LMA) proposed for Palladio [Stefik et al., 1981b; Brown et al., 1983]. Other applications include general digital design [Azema et al., 1975] and asynchronous system design [Misunas, 1973].

Token passing techniques have the advantage that they guarantee that certain classes of errors will be detected in a description [Feldbrugge, 1980]. Unfortunately they are of little assistance in detecting a wider range of errors. They also suffer from the restrictions placed on the nature and composition of intermodule communications, reducing the designers freedom to express the design in the most perspicuous manner.

3.2.3 Provably Correct Descriptions

One of the most attractive description techniques is that of proving that a design meets its specifications. If the top level of a hierarchical description is taken as the specification, then it is desirable to prove that the subsequent hierarchical decompositions of that top level are equivalent to it in function. A number of approaches have been developed, one of the best known work being that of Gordon [Gordon, 1979]. In this work the function of each module in the design is specified in a Higher Order Logic (HOL) form. Given a particular module's specification in HOL, and the specification of its component parts, it is possible to prove that the function of the interconnected components is the same as that of the original module. This process may be repeated to verify the correctness of the entire design. Gordon's work has been extended by several groups [Birtwistle et al., 1986; Barrow, 1984] to the verification of reasonably complex designs.

The advantages of such techniques are clear: the hierarchical decomposition of the design into its components is proven correct. There are however a number of limitations in current techniques:

1. *Complexity.* The verification process is complex and as yet only partially automated. Considerable mathematical insight on the part of the designer is still required in order to prove equivalence.

2. *Language.* Expressing functionality in HOL is not likely to be well received by designers more familiar with conventional programming languages.

3. *Completeness.* Although techniques for purely functional verification have been developed, there are a number of areas in which proof techniques are still wanting. For instance the issue of detailed circuit function (timing in particular) and its equivalence to the functional description: given a functional description of a leaf cell and a circuit layout intended to emulate the function it is still difficult to prove equivalence.

3.2.4 Generalized Descriptions

The largest class of description languages are those that represent structure and function in some generalized way. These languages typically resemble programming languages, and in fact many are derived from such languages. They are based around the observation that many of the concepts and structures developed for programming are applicable to describing hardware systems. Examples of recent generalized description languages include: *VHDL* [Shadad et al., 1985]; *Zeus* [German & Lieberherr, 1985]; *KARL-III* [Hartenstein, 1986]; *FUNSIM* [Foyster, 1986]; *ICSYS* [Buchanan, 1980]; *Sakura* [Suzuki & Burstall, 1982]; *Conlan* [Piloty & Borrione, 1985]; and *SPAM* [Segal, 1981].

Generalized description languages have a number of advantages:

1. *Flexibility.* The generality of the languages make it possible to express most hardware structures in an intuitive form.
2. *Simulation.* The similarity of the languages to programming languages makes it simpler to write portable and fast simulators.
3. *Familiarity.* Designers are often familiar with conventional programming languages, making it more natural for them to express hardware designs in a related description language.

The disadvantages of generalized languages include stem from their general nature:

1. The often irregular syntax and semantics of the languages make automatic verification difficult.
2. There is no simple register/interpreter structure as in register transfer descriptions.

3. Data transfers are relatively unconstrained and do not exhibit the self-checking properties of token passing paradigms.

4. Limitations of the underlying programming language may sometimes make the syntax and semantics somewhat clumsy for hardware description.

3.3 Language Requirements

In this section an examination is made of the facilities available in a modern programming language, and their applicability to VLSI design description. This results in the identification of a number of points at which additional support is needed in order to facilitate the extension of a programming language to VLSI design.

3.3.1 Function

A general purpose programming language provides a useful basis for the expression of *function*. The various arithmetic and logical operators, together with comprehensive data structuring facilities allow for the concise expression of functionality. The basic block structured syntax that is used in conjunction with flow control operators such as `FOR`, `WHILE` and `IF` allows for the clear specification of flow-of-control within a functional description.

3.3.2 Structure

Both software and hardware rely on the concept of partitioning to control complexity, and many of the issues in developing a structural design are common to both domains (Chapter 2). The actual mechanism for representing the partitioning may however differ. The basic structuring mechanism within a typical programming language such as Pascal [Jensen & Wirth, 1978] is the *procedure*. Procedures are parameterized entities that are invoked with a procedure call in order to perform a particular function. There are several properties of procedures that make them unsuitable for the structuring of VLSI descriptions. Although we may regard the *definitions* of procedures as permanent with respect to time, they are *invoked* rather than *instanced*. This means that a procedure in general is called, performs its function, and then terminates, returning control to the caller. Thus the structure of a program is *sequential* in nature,

and a software hierarchy such as that shown in Figure 3.1(a) has an implied time domain unlike that of a VLSI structural design such as that shown in Figure 3.1(b). Hardware modules exist continuously and perform computation in parallel. Thus they have a continuous state, and are able to accept new data at any time.

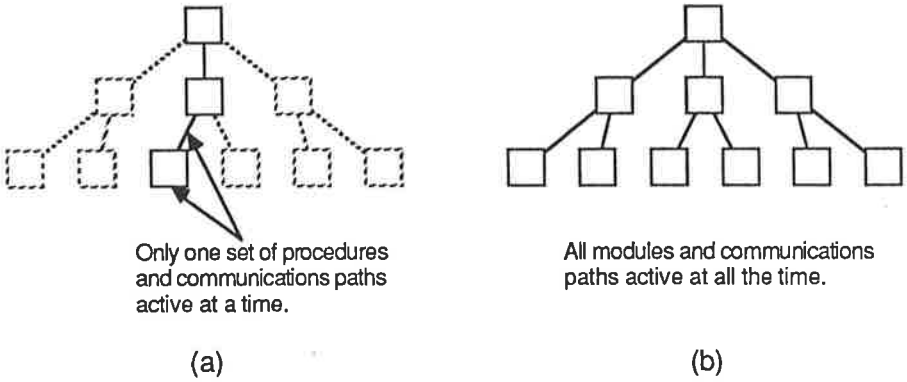

Figure 3.1: Software (a) and hardware (b) hierarchies.

3.3.3 Communication

Data is passed into a procedure by means of *parameters*. This may be represented as the creation of temporary paths along which data may travel in at the instant of invocation of the procedure and out at its termination. There is no analog to this in the hardware case: as modules exist continuously, the data connections to them must also exist continuously. Also, data communication in a programming language is synchronous with the procedure invocation, whereas in hardware there may be actual time delays associated with the transfer of data. It should be possible to declare these delays when appropriate as they comprise part of the design specification. This will not always be required, as the synchronous clocking methodology often applied in structured VLSI design [Mead & Conway, 1980] may be used to provide a level of abstraction behind which timing details are hidden and signals are presumed to arrive synchronously [Chen & Mead, 1983].

3.3.4 Reusability

A concept common to both hardware and software design is *reusability*. A definition acts not only as a means of partitioning a design, but also as a way of representing

it more succinctly. A single procedure definition may be *invoked* in a number of different places in a program in order to perform similar functions. In the same way, a hardware module definition may be *instanced* a number of times in different parts of the design to perform similar functions. This is particularly useful given the expense of communication in VLSI (Section 1.1.2). It may be less expensive to duplicate functional circuitry rather than provide interconnect to a single instance.

3.3.5 Regularity

There concepts of regularity in software and hardware description are loosely related. In software, regularity usually occurs in data structures such as arrays. A single function may be applied a number of times on each element in the array, and such an operation may be expressed succinctly with for instance a `FOR` loop.

Regularity in VLSI is similarly concerned with the regular use of modules. It is different to the software case in that the modules are all created at the same time in some regular manner rather than as a result of a sequential application. In addition it is necessary to represent the regular patterns of interconnect between the modules (Section 1.2.2).

3.3.6 Discussion

From the above observations it is possible to compose a set of requirements that were used in the design of the language to be described in the following sections. They are:

1. *Function.* The basic syntax of a block structured programming language provides a useful basis for expressing module function.

2. *Structure.* The structuring facilities provided by programming procedures are not appropriate to VLSI hardware representation. VLSI modules must be represented as entities that exist with continuous state and are able to perform function in parallel. Not that there is still a distinct hierarchical organization, and that its management will require additional facilities as it is not partitioned sequentially into a single stream of invocations.

3. *Communication.* The interconnect between VLSI modules must exist continuously as do the modules themselves. The management of the additional complexity introduced by synchronization and time delays must also be addressed in the representation.

4. *Reusability.* The need for module reusability suggests that as with a software procedure, a hardware module must have a *definition* which may be instanciated in a number of places in the design.

5. *Regularity.* The particular needs of regularity in VLSI designs suggests that special declarative constructs are needed to concisely describe regular instanciation and interconnect.

3.4 A Prototype of the Language

The need for a description language to aid in the specification and partitioning of a VLSI design was made apparent during the development of the TFB chip, a 200,000 device general purpose signal processing chip [Dickinson, 1984b; Eshraghian et al., 1985]. The *TicToc* [Dickinson & Eshraghian, 1984] language and simulator were rapidly developed in Modula-2 [Wirth, 1982] and used as a system design aid for that project [Schomburg, 1984; Murphy, 1984].

The design and implementation of *TicToc* had two fundamental limitations:

1. Intermodule communication could only take place across clock phases. Clocks were a special case of signals and modules could only input signals that had been output from another module in preceding cycles. This design decision was based on a limitation in the simulator implementation, and justified in terms of this being a "safe" methodology for clocked synchronous systems. Unfortunately this led in practice to designers "clumping" functionality into single modules that should have been decomposed.

2. There was no facility for the multiple instanciation of modules: each module had to have a textually separate definition which also acted as its instanciation. As well as requiring a great deal of source code, this did not allow for the explicit expression of regular structures in the design. Again this design problem was a result of an implementation decision. Hardware modules were mapped to

directly to Modula-2 modules, and although intuitively appealing, the static textual nature of the modules was inappropriate for the task.

These quite severe problems with *TicToc* motivated the complete redesign of a system structural design language, the result being *Pink*.

3.5 Selection of a Base Language

There are a number of languages that have potential as a basis for a VLSI description language. One of the simplest and important criteria is based on the observation in that the base language is more useful for the expression of function rather than structure. Thus a language with a clean, easily understood and sufficiently general syntax is appropriate. In addition it is useful if the language is widely available to ensure portability. The Modula-2 [Wirth, 1982] language fits these criteria well: it is based on Pascal and hence has a simple, well understood syntax, but has been improved to include systems programming and modularity concepts. It is also widely implemented, being available on a wide range of machines of different sizes. There are a number of other reasons for selecting Modula-2 over other languages, but these are based on the implementation of the *Pink* simulator, and are covered in Section 4.3.1. It is interesting to note at this point that *Pink* has been designed in such a way that translation from *Pink* to Modula-2 (for simulation purposes) may be achieved on a statement-by-statement basis. That is, the translation process need only look at a single line of *Pink* in order to produce the translated code it represents.

3.6 Structural Description

The descriptive properties of the *Pink* language may be classified into two catagories: description of structure and function. In this section the language syntax and the underlying semantics that support the description of structure is described. The mode of language description is informal as the syntax of the *Pink* additions to Modula-2 is quite straightforward. The bulk of the syntax is simply that of standard Modula-2 as described formally by Wirth [Wirth, 1982]. This follows the precedent of Buchanan [Buchanan, 1980] for embedded description languages.

Structural description in *Pink* may be regarded as *declarative*. Unlike other languages

such as Zeus [German & Lieberherr, 1985] and VHDL [Shadad et al., 1985], the structure of the design is not created by procedural code, but by specialized declarative statements that define the structure itself. This approach has the advantage that the structure is far more explicit in the description: there is no code that must be mentally "executed" in order to visualize the structure being implied.

3.6.1 Hierarchy

Basic to any language that is intended to describe system structure is the ability to express hierarchy in a concise and intuitive manner. In *Pink* the expression of hierarchy is based on there being two distinct partitioning entities: *definitions* and *instances*.

A *definition* is the "prototype" of a module in the design, defining its structure but not itself appearing as a component of the system description. All of the definitions in a single system exist in a flat namespace: they must each have a unique name. Definitions are analogous to procedure declarations in a programming language.

Definitions are created using the `DEFINITION` construct:

```
DEFINITION  <definition name>;
.
.
.
END <definition name>;
```

There are two major classes of definition: *topological* and *functional*. Topological definitions are used to create definitions that are *composition modules*: those that are created from the interconnection of a number of subordinate modules. Functional definitions are *leaf modules* that have no subordinate modules: rather they contain code that directly implements the function.

Within a definition a topology may be declared using the `TOPOLOGY` construct:

```
TOPOLOGY
.
.
.
ENDTOPOLOGY
```

Similarly a function may be declared using the FUNCTION construct:

```
FUNCTION
.
.
.
ENDFUNCTION
```

An *instance* is the "realization" of a prototype provided by a definition. Instances are created and named using the MakeInstance procedure in the TOPOLOGY section of a definition:

```
MakeInstance(<definition name>, <instance name>);
```

The naming of instances follows a quite different strategy to the naming of definitions. This is a result of the fact that a single definition may be instanced any number of times in different contexts. It is important that there be a consistent naming scheme that uniquely identifies each instance in a design. To facilitate this, instances are named according to a hierarchical strategy. Instance names need only be unique within the definition that declares the instances. Every module instance in a system is uniquely identified by its full "path" name, where a path is a concatenation of the ancestor instances, the separator being the "/" character.

For example (Figure 3.2) a definition A may contain two instances Y and Z of definitions B and C. Within the definition A the instances will be referred to as X and Y. The definition A may in turn be instanced twice as J and K. The resulting four instances of B and C will be named J/Y, J/Z, K/Y and K/Z.

With this naming scheme it becomes possible to access specific instances in either an absolute way, from the top instance in the hierarchy, or in a relative manner from a specific instance. In addition to being useful in specification, this allows simple examination of the structure in the simulator (Section 4.10.3).

3.6.2 Interconnect

Composition modules are constructed by creating instances of module definitions and interconnecting them. Connections are made to *ports* that are declared in a definition as the only means of providing data input and output to the module instance. Ports

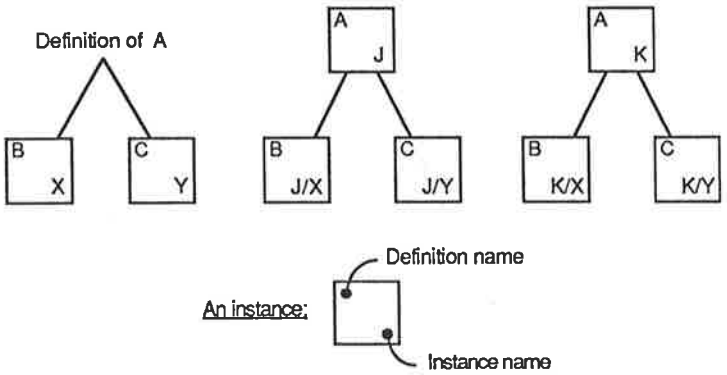

Figure 3.2: An example of the hierarchical naming scheme.

are analogous to procedure parameters in this regard, however they are more complex in that they allow a *flow* of data throughout the life of the module unlike parameters that only provide an initial input and final output function.

Ports have three basic attributes:

1. *Name.* The name of each port must be unique within a definition.
2. *Data Type.* Ports may be of two types: `BIT` and `INT`. A `BIT` type can assume three values only: `HI`, `LO` and `UN` representing high, low and undefined binary values respectively. An `INT` type can assume any integer value (the actual range being limited by the internal integer representation of any machine used as a simulation host). Only port of the same data type may be interconnected.
3. *Signal Type.* The type of data flow through the port: any one of `INP` (input), `OUTP` (output), `IOP`, (input-output), `PWR` (power), `GND` (ground), `CLOCK` (clock).

Ports are declared in the `STATE` section of a definition:

```
STATE
  <port name>, <port name>, ... : PortType;
```

and described in the `PORTS` section:

```
PORTS
  DeclarePort(<port name>, <port type>, <port direction>);
```

In addition to single channel ports, *Pink* provides a facility for describing multi-channel ports or buses. A bus is in effect an array of BIT type ports that may be treated syntactically as a single entity. Buses are declares in the STATE section:

```
STATE
  <bus name>, <bus name>, ... : ARRAY [0..<upper bound>] OF PortType;
```

and described in the PORTS section:

```
PORTS
  DeclareBus(<bus name>, <port direction>);
```

The naming of ports (and buses) is closely related to the naming of instances. Within a definition the ports of that definition are known by their declared names. However from the outside of a definition the ports are named by concatenating the instance name and the port name together. Thus if a port is named R in a definition A, then if A is instanced with an instance name J, the port may be referred to as J/R.

Ports are connected together to form composition modules inside the TOPOLOGY section of a definition using the Connect procedure:

```
Connect(<port name> & <port name> & ...);
```

A <port name> will appear either as a simple name referring to a port on the current definition or as a <instance name>/<port name> pair referring to a port on a subordinate instance to the definition. Additionally any name may be that off a bus with a single index of the form <port name>[<index>].

Entire buses and subranges of buses may be connected by specifying ranges as the indices as in:

```
Connect(<port name>[<lower>..<upper>] &
        <port name>[<lower>..<upper>]);
```

where the ranges must be off the same extent.

The connection mechanism in *Pink* differs significantly from that used in languages such as VHDL. In these languages a signal or net entity must be declared for each node that is to be created when one or more ports are connected. The connection is made by using the signal in instance declarations. This procedure requires that new entities be declared and is considerably less intuitive than the *Pink* strategy.

3.6.3 Structural Regularity

As indicated in previous chapters, a great deal of reduction in explicit complexity may be achieved by making use of the inherent regularity present in many structured VLSI designs. This regularity has a number of different facets, and in order to represent it fully it is desirable to incorporate a general and yet clear mechanism into the language. The representation should be *declarative* rather than *procedural* as the regularity is static and should be described as such. (Some languages such as VHDL [Shadad et al., 1985] require the designer to represent regularity using procedural constructs such as `for` loops).

The declarative mechanism for describing regularity in *Pink* is based on an integer subrange syntax that may be used in a variety of situations. The basic operation of the syntax is to take any language statement in which a regular subrange appears and expand it for each step of the subrange. The use of the facility is best illustrated by an example of it common usage with the `MakeInstance` and `Connect` procedure calls:

- `MakeInstance(adderbit, adderslice<0..31>);` declares 32 instances of the definition `adderbit`, named in the sequence `adderslice0, adderslice1, ..., adderslice31`.
- `Connect(adderslice<0..30>/cout & adderslice<1..31>/cin);` declares the connection of the carry out port of each instance to the carry in port of the subsequent instance.

The <> syntax may be used in many other positions including declarations. For example: `DeclarePort(cout<0..7>, BIT, OUT);` describes eight ports named `cout0, cout1, ..., cout7`.

The notation may be extended to more than a single dimension of regularity by using multiple ranges. The particular range is then identified by its position in a sequence of ranges, with the earlier ranges being cycled slower. For example: `MakeInstance(mult,mult<1..3><1..3>);` would generate instances named `mult11, mult12, mult 13, mult21 ... mult33`.

3.7 Specification of Function

The language has been designed so that the contents of the FUNCTION section of a functional definition appear as similar as possible to Modula-2 as defined by Wirth [Wirth, 1982]. Most *Pink* operators appear to be Modula-2 procedure calls, albeit with unusual argument syntax where an optional number of arguments may be required.

The remainder of this section describes the basic format of a functional description of a module. The issues of timing and communication are dealt with in the following section.

3.7.1 Declarations

Variables local to a definition may be declared in the STATE section of a definition. Standard Modula-2 syntax must be used. Two additional predefined types, STATEBIT and STATEINT are provided for declaring variables that hold values of the BIT and INT port types. Note that the variables declared in a definition are *static*: they may be treated as maintaining their values continuously.

The Modula-2 keywords CONST and TYPE may be inserted above the STATE keyword to allow for the declaration of constants and types.

3.7.2 Initialization

The INIT section of definition may contain any code that is suitable for use in the FUNCTION section. This code will however be executed only *once* for initialization purposes.

3.7.3 Functional Code

The syntax expected in the FUNCTION section is a superset of standard Modula-2. Function is expressed by using the *Pink* port communication operators (Section 3.7.4) in conjunction with standard Modula-2. A number of additional utility functions are provided to simplify the description of the function:

- `TOOBOOL( <BIT value> );` Converts a value of the `BIT` type to a `BOOLEAN`. Only valid if the `BIT` value is `HI` or `LO`.
- `FROMBOOL( <BOOLEAN value> );` Converts a boolean into a `HI` (`TRUE`) or a `LO` (`FALSE`) value suitable for writing to a port.

In some cases it is desirable to define Modula-2 procedures to perform often used functions in a description. *Pink* allows these to be defined in the normal manner anywhere outside a `DEFINITION` with the Modula-2 `PROCEDURE` construct.

Modula-2 library procedures [Wirth, 1982] for tasks such as textual input, output and maths may be imported with the Modula-2 `IMPORT` construct positioned after the `SYSTEM` statement.

3.7.4 Specification of Time and Communication

One of the most difficult aspects of representing hardware in a description language is that of intermodule communication and the related issue of timing. Most software systems are sequential in nature: data is passed into a procedure on invocation and passed out upon termination. Conversely hardware modules are constantly active and may input and output data at any time. The timing model adopted by most hardware description languages is constrained by the limitations of the underlying software simulator: each module instance must produce new outputs at a time *after* the inputs are changed. The disadvantage of this class of description language is that it requires that the designer begin to consider timing issues at an early stage in the design.

A primary motivation for the design of the *Pink* language was to enable a designer to rapidly develop and evaluate alternative abstract designs. The need to specify specific delays was regarded as an unsuitable constraint in for this class of language and hence a *demand driven* approach to data transfer has been developed based in part on Communicating Sequential Process (CSP) [Hoare, 1978] semantics. The result is a communication model that supports the *instantaneous* transfer of data between modules in addition to the more common *delayed* transfer. Additionally facilities may be built into the language simulator to detect pathological data transfers in a similar manner to that described previously for token based descriptions. The communication strategy may be implemented in a particularly efficient fashion for

simulation. These last two issues are discussed further in the next chapter.

The syntax and semantics of time and data transfer in *Pink* are described in the remainder of this section.

3.7.5 Description of Time

Time is represented in the description in terms of *ticks* relative to the current time. Ticks are *atomic*: they represent the smallest time that may pass between time separated events. Ticks are *generic* time units: the user may treat them as whatever real time units are appropriate to the system being described (e.g. nanoseconds, picoseconds etc). Module descriptions have no means of accessing *absolute* time: instead they must specify time relative to the current system time.

The `Pause` procedure provides a mechanism for an instance to pass time without any activity: `Pause(<ticks>);` causes the instance to suspend activity for `<ticks>` amount of system time after which activity is resumed with the subsequent statement.

3.7.6 Outgoing Communication

There are two primitive operations for the output of data from a module through one of its ports: `WRITE` and `DELAYEDWRITE`.

The `WRITE` procedure allows for the writing of a value to a port *instantaneously*. Any module connected to the port will receive the new data without any apparent delay. The syntax of the command is:

```
WRITE(<port name>, <value>);
```

The `DELAYEDWRITE` procedure allows for the output of a value to a port after some finite *delay* specified in ticks. The effect of a delayed write is that any modules connected to the written port will not receive the new data till the specified delay has elapsed. The activity of the writing instance is not halted however: activity continues immediately with the subsequent statement.

The statement has the following syntax:

```
DELAYEDWRITE(<port name>, <value>, <ticks>);
```

3.7.7 Incoming Communication

The basic entity for data input to an instance is the `READ` function which returns a value read form a port. The semantics of a `READ` require that the value it immediately returns is the final value that will be assigned to that port in the current time interval (tick) rather then the value that it might have at the current time. This raises the issue of deterministic as opposed to non-deterministic scheduling which will be dealt with in the next chapter. From the point of view of the description however, it is sufficient to assume that the value returned by a `READ` call is simply a valid value for the port at the current time.

The statement has the following syntax:

```
<variable> := READ(<port name>);
```

3.7.8 Synchronization

The various modules in a system design must be synchronized in order to function in a coherent fashion. The *Pink*language provides a set of explicit *waiting* operators in order that instances be able to operate in synchronism with each other by pausing till particular ports are written. By far the most common mode of use for these operators is to wait for a clock signal to be asserted.

There are two related procedures for waiting. The first, `WaitFor`, causes activation of the instance to be halted till one of the named ports is written. The statement has the syntax:

```
WaitForCase(<port name 1> | <port name 2> | ...)
```

The second and more comprehensive form, `WaitForCase`, allows for the specification of different actions depending on the name of the written port:

```
WaitForCase(<port name 1> | <port name 2> | ...)
  Signal <port name 1> Has Action
    .
    .
    .
  Signal <port name 2> Has Action
```

```
.
.
.
END;
```

The presence of explicit synchronization calls in the description ensures that the style of synchronization being used is explicitly represented, thus aiding comprehension of the description.

3.7.9 Bus Operations

The writing and reading operators described above may be used on individual elements of buses as each of these is a simple BIT port. In addition however several bus-specific operators are provided. The BUSWRITE and DELAYEDBUSWRITE procedures are identical to their BIT counterparts except they expect decimal integer values as arguments that are then converted to binary BIT values and written onto the bus itself. Similarly the BUSREAD procedure reads the bus and returns a single decimal integer value representing the binary bus value.

```
WRITEBUS(<bus name>, <value>);
DELAYEDWRITEBUS(<bus name>, <value>, <ticks>);
<variable> := READBUS(<bus name>);
```

3.8 Systems, Subsystems and Libraries

A *Pink* description is contained in the basic context of a *system*. A system is declared with the SYSTEM construct:

```
SYSTEM <system name>;
.
.
.
BEGIN
.
.
.
END <system name>.
```

In any *system* one and only one instance must be declared as the *root instance* to which all other instances are subordinate. The root instance is declared with the MakeTopInstance procedure which must appear thus:

```
SYSTEM <system name>;
.
.
.
BEGIN
  MakeTopInstance(<definition name>, <root instance name>);
END <system name>.
```

In addition to complete systems specified in a single file, it is desirable to have a facility that allows the description to be spread across a number of files to aid structuring of the description and separate compilation. This same facility may be used for the collection commonly used definitions in libraries. The SUBSYSTEM facility provides this by allowing definitions to be created and then imported into the main SYSTEM description for instancing:

```
SUBSYSTEM <subsystem name>;
.
.
.
  DEFINITION <definition name>
     .
     .
     .
  END <definition name>;
.
.
.
END <subsystem name>.
```

The definitions contained in the subsystem may be imported into the system description with the Modula-2 IMPORT facility:

```
SYSTEM <system name>;

FROM <subsystem name> IMPORT <definition name>;
.
.
.
END <system name>.
```

Note that definitions do not have to be explicitly exported from the subsystem: all definitions in a subsystem are automatically exported.

3.9 Parameterization

Several researchers [Buchanan, 1980; German & Lieberherr, 1985] have emphasized the importance of *parameterization* of hardware modules. This refers to the association with a definition of parameters that set certain numerical features of that definition. The number of bits in an adder for example. In such a scheme, when a module is instanced, the number of bits may be set at the same time. This is useful in that it reduces the number of different definitions that may be required. There are two observations that may be made on this issue:

1. The run-time parameterization of modules disrupts the clean mapping between definitions and instances. A number of physically different instances may have the same definition, varied only by parameters. When decomposition of such a definition takes place, the decomposition must be parameterized, and this may be difficult. When the definition comes to be realized at the circuit level, similar allowances must be made for parameterization, and quite different circuits may be required for different values of the parameters.

2. The range of different parameters used in a definition is likely to be small. For instance within the one system there are unlikely to be more than say 8, 16 and 32 bit adders in use.

3. The parameterization only refers to structure, and this is a static property of the description. There is no need to carry out parameter expansion of definitions in a fully executable form.

These observations led to the inclusion of *textual* parameterization of *Pink* definitions. The *C* preprocessor [Ritchie & Kernighan, 1978] is used to provide a "macro" facility that can be used to create several versions of a definition, one for each set of parameter values. For example an adder definition may be parameterized as:

```
#define AdderDefinitionMacro(Name,MaxBit)
  DEFINITION Name;
    .
    .
```

```
MakeInstance(AdderBit,Bit<0..MaxBit>);
.
.
END Name;
```

A 32 bit version of the adder definition named `Adder32` and a 16 bit version of the adder definition named `Adder16` may then be created:

```
AdderDefinitionMacro(Adder32,32)
AdderDefinitionMacro(Adder16,16)
```

This mechanism is appropriate for several reasons:

1. Its use as an aid to speeding the construction of a description is explicit: there are still different definitions for different types of instances, so the mapping between definitions and instances is not confused.
2. There are only likely to be a small number of variations on a particular definition, so the additional textual overhead of processing separate code for each definition is not high.
3. The expansion takes place during preprocessing, thus a simulator need not deal with the additional complexities of parameterized definitions.

3.10 Limitations

The primary limitation of the language arises in the description of feedback paths without delays. Any such path may give rise to a situation in which it is not possible to find a stable value for a given node: an inverter with its input and output connected together for instance will cycle between `HI` and `LO` states within the same instant of time. Given some delay in the feedback path however means that the circuit values will alternate within each delay interval.

This is not in general regarded as a critical restriction. The normal timing methodologies of structured design suggest an even more significant restriction: that feedback paths be interrupted by a clocking signal to ensure that only fully clock-synchronous operation may occur. This decreases the complexity of circuit function, and is a basic rule in design for testability [Gerner & Johansson, 1986].

3.11 An Example

In this section some of the description concepts outlined in this chapter will be illustrated with an example: the various levels of *Pink*description of a thirty-two bit adder. An example of more significant complexity is described in Chapter 7.

Note that there are two comment mechanisms in the language:

1. The Modula-2 based multiline comment initiated with "(*" and terminated with "*)".
2. The single line comment initiated with "--".

Five different levels of description of a simple thirty-two bit adder will be described. Each level is a successive refinement of the previous level created by refining abstractions in the previous level.

The first description is comprised of only a single functional block with integer inputs and output:

```
DEFINITION Adder32;
  STATE
    Ainput, Binput, Output : PortType;
  PORTS
    DeclarePort(Ainput, INT, INP);
    DeclarePort(Binput, INT, INP);
    DeclarePort(Output, INT, OUTP);
  FUNCTION
    WaitForAny(Ainout | Binput);
    WRITE(Ouput, READ(Ainput) + READ(Binpout));
  ENDFUNCTION;
END Adder32;
```

The second description may be created by refining the `INT` ports into bitwise descriptions:

```
DEFINITION Adder32;
  STATE
    Ainput, Binput, Output : ARRAY [0..31] OF PortType;
    -- Now use bitwise buses rather than integers.
  PORTS
```

```
    DeclareBus(Ainput, INP);
    DeclareBus(Binput, INP);
    DeclareBus(Output, OUTP);
  FUNCTION
    WaitForAny(Ainout | Binput);
    WRITEBUS(Ouput, READBUS(Ainput) + READBUS(Binpout));
  ENDFUNCTION;
END Adder32;
```

In the next refinement, clocks are added to provide explicit synchronization. In this case the clock only generates HI signals, and the modules that use this clock use it only for synchronization, and never read the actual value. Usually the clock would be obtained from a definition library, however here it is explicitly defined for illustration:

```
DEFINITION TwoPhaseClock;
  STATE
    Phase1, Phase2 : PortType;
  PORTS
    DeclarePort(Phase1, BIT, CLOCK);
    DeclarePort(Phase2, BIT, CLOCK);
  FUNCTION
    WRITE(Phase1, HI);
    Pause(10); -- The module idles for 10 ticks.
    WRITE(Phase2, HI);
    Pause(10);
  ENDFUNCTION;
END TwoPhaseClock;
```

Such a clock could be connected to an instance of the following definition in order to provide synchronization:

```
DEFINITION Adder32;
  STATE
    Phase1, Phase2 : PortType;
    Ainput, Binput, Output : ARRAY [0..31] OF PortType;
    Temp : STATEINT;
  PORTS
    DeclarePort(Phase1, BIT, CLOCK);
    DeclarePort(Phase2, BIT, CLOCK);
    DeclareBus(Ainput, INP);
    DeclareBus(Binput, INP);
    DeclareBus(Output, OUTP);
  FUNCTION
```

```
    WaitForAny(Phase1 | Phase2);
    -- Wait until there is some activity on Phase1 or 2.
    WRITEBUS(Output, READBUS(Ainput) + READBUS(Binpout));
  ENDFUNCTION;
END Adder32;
```

The fourth description may be constructed by creating a partitioning of the adder into thirty-two components:

```
DEFINITION AdderSingleBit;
  STATE
    Phase1, Phase2, A, B, CarryIn, CarryOut, Out : PortType;
    av, bv, cinv, coutv, outv : BOOLEAN;
  PORTS
    DeclarePort(Phase1, BIT, CLOCK);
    DeclarePort(Phase2, BIT, CLOCK);
    DeclarePort(CarryIn, BIT, INP);
    DeclarePort(CarryOut, BIT, OUTP);
    DeclarePort(A,BIT, INP);
    DeclarePort(B,BIT, INP);
    DeclarePort(Out,BIT, OUTP);
  INIT
  FUNCTION
    WaitFor(phase1 | phase2);
    av := TOBOOL(READ(A));
    bv := TOBOOL(READ(B));
    cinv := TOBOOL(READ(CarryIn));
    outv := (av AND bv AND cinv)
            OR
            (av AND NOT bv AND NOT cinv)
            OR
            (NOT av AND NOT bv AND cinv)
            OR
            (NOT av AND NOT cinv AND bv);
    coutv := (av AND bv) OR (av AND cinv) OR (bv AND cinv);
    WRITE(Out, FROMBOOL(outv));
    WRITE(CarryOut, FROMBOOL(coutv));
  ENDFUNCTION;
END AdderSingleBit;

DEFINITION Adder32;
  STATE
    Ainput, Binput, Output : ARRAY [0..31] OF PortType;
  PORTS
    DeclarePort(Phase1, BIT, CLOCK);
```

```
    DeclarePort(Phase2, BIT, CLOCK);
    DeclareBus(Ainput, INP);
    DeclareBus(Binput, INP);
    DeclareBus(Output, OUTP);
 TOPOLOGY
    MakeInstance(AdderSingleBit, BitSlice<0..31>);
    Connect(BitSlice<0..31>/Phase1 & Phase1);
    Connect(BitSlice<0..31>/Phase2 & Phase2);
    Connect(BitSlice<0..31>/A & Ainput[<0..31>]);
    Connect(BitSlice<0..31>/B & Binput[<0..31>]);
    Connect(BitSlice<0..31>/Out & Output[<0..31>]);
    Connect(BitSlice<0..30>/CarryOut & BitSlice<1..31>/CarryIn);
    Connect(BitSlice0/CarryIn & FixedLo);
  ENDTOPOLOGY
END Adder32;
```

In the final stage of refinement, delay information is added into the adder bit slice output (all other text is unchanged):

```
DEFINITION AdderSingleBit;
    .
    .
    .
    DELAYEDWRITE(Out, FROMBOOL(outv), 40);
    DELAYEDWRITE(CarryOut, FROMBOOL(coutv), 40);
  ENDFUNCTION
END AdderSingleBit;
```

3.12 Comparison

It is not practical to carry out an extensive comparison between *Pink* and the large number of current hardware description languages. However an extensive comparison between VHDL and eight other languages clearly demonstrated that it is broadly representative of current research [Aylor et al., 1986]. It is appropriate then to use VHDL as a basis for comparison of the features of *Pink* intended for design complexity management and structural design. The information on VHDL used in this section for comparison is drawn from several sources [Aylor et al., 1986; Lipsett et al., 1986; Nash & Saunders, 1986]. VHDL is a hardware description language developed for the Very High Speed Integrated Circuit (VHSIC) program and has many of the features

and much of the syntax of the Ada language [Shadad et al., 1985]. VHDL is the result of a large research and standardization effort, an the language is very rich in data typing and software organization facilities such as *packages*. It is not the intent here to show that *Pink* can compete on the basis of size and flexibility. Rather that certain features of *Pink* do not appear in VHDL, and that such features may provide superior facilities for structural design.

3.12.1 Structural Description

In VHDL a definition (declaration) of a module may be instanced and connected in order to create structural entities. The instanciation of a module is expressed in a similar manner to that used for procedures in software. Each "parameter" to the instanciation is a "signal" that is declared to act as a node in the design. The signal is bound to the relevant port in the definition. For example, the following is a structural description (down to the gate level) of an adder single bit:

```
architecture STRUCT of HADDER is
  block
     component nandgate port (A,B: in BIT; C: out BIT);
     component xorgate port (A,B: in BIT; C: out BIT);
     component inv port (A: in BIT; B: out BIT);
     signal T1;
  begin
    Z1: xorgate port (X,Y,SUM);
    Z2: nandgate port (X,Y,T1);
    Z3: inv port (T1,Cout);
  end block
end HADDER
```

The components may only be connected to declared signals or ports on the parent module. To actually comprehend the structure of the design, the connections must be derived by checking each instanciation statement for common signal names. In *Pink* the connections are made explicitly with a `Connect` statement.

It is not clear from the literature whether or not there is any form of general hierarchical naming system for VHDL instances and ports.

Regularity is expressed in VHDL with the `generate` statement. For example:

```
for i in 0 to 7 generate
  BIT: adder( ... );
end generate;
```

This style of *procedural* description of regularity is very general, but has the disadvantage that the structure it describes is implicit rather than explicit. The *Pink* equivalent would be:

```
MakeInstance(adder, Bit<0..7>);
```

which is both more explicit and concise.

3.12.2 Functional Description

Functional description in both languages is similar: VHDL uses an Ada-like syntax, and *Pink* uses Modula-2 for the description of algorithms. The VHDL syntax is complex, but provides a very rich programming environment. The *Pink* syntax is simple, but does not have the same data structuring facilities as VHDL for example. As VHDL is not however a true superset of Ada, there are in fact some facilities missing: dynamic objects may not be created for example, and this is a disadvantage in some applications [Nash & Saunders, 1986]. *Pink* allows the incorporation of all such Modula-2 facilities in a description.

3.12.3 Timing

Although delays may be specified on VHDL signals, there is no mechanism in the language analogous to the *Pink* `Pause` statement. Thus it is not possible for a module to simply suspend operation for a fixed amount of time. This is seen by most researchers as a major omission in the language [Nash & Saunders, 1986].

VHDL provides means of specifying both delayed and undelayed signals. An undelayed signal is simply delayed by a very small amount of time referred to as "delta". This is quite different from the concept of an instantaneous signal in *Pink*. The later implies a higher level of abstraction. The data driven semantics of an instantaneous signal are such that an abstraction is formed that does not deal with time or sequence at all: the selection of correct sequencing within the instant becomes a property of the language, not the particular design description.

3.13 Summary

The *Pink* language provides a number of features that facilitate its use as a structural design language:

1. It provides *definition* and *instance* constructs that are suitable for the structural description of VLSI systems.
2. It is based on Modula-2 which provides a suitable means of expressing algorithmic function.
3. It provides a means of interconnecting module instances with typed data carrying paths.
4. Data input and output operations are provided that may be used to express abstract (instantaneous) or detailed (delayed) data transfers between modules.
5. Preservation of control state allows the description to incorporate pausing for a period of time at specific points in the code.
6. A syntax for the general expression of regular structures allows for the declarative rather than procedural description of regular structural entities.
7. A macro expansion facility is used for the construction of parameterized module definitions.

The intent of these features is to provide the designer with a facility for expressing a design with the minimum explicit complexity. In addition, the software flavour and refinable abstractions of the language encourage the use of software structural design techniques in the creation of a VLSI design.

Chapters 4, 5 and 6 examine the issues involved in *modelling* the algorithmic function and physical form of a *Pink* description. The overall structural design procedure that may be used given the *Pink* language and modelling aids is described along with a system design example in Chapter 7.

Chapter 4

Modelling Function: Simulation

4.1 Introduction

The *Pink* language described in the previous chapter provides a basis for the abstract description of a *structural* design of a VLSI circuit. In addition it enables the designer to specify a *function* for the leaf modules in the design. By itself, such a description may serve to unambiguously define the structure and function of a design. The utility of describing a system in such a way may be greatly increased however by *modelling* the system description. This involves actively simulating system function and thus providing information about its behaviour that may be used in evaluation of the proposed design. Additionally, the modelling process may be used as an aid to verification of the functional decomposition and overall system integration.

The evaluation of the quality of a software design is eased by the relationship between the design and the result. A program is typically designed in the implementation language, and may be incrementally decomposed and executed to provide feedback. Clearly the fabrication delays involved in VLSI make this form of design difficult. However, given an appropriate description of the design, and suitable tools, it is possible to simulate the design in order model its behaviour.

This chapter will examine the task of actively modelling the composite *function* of a *Pink* description with a functional simulator. Primary design motivations for the simulator are to encourage interactive investigation of the system partitioning by the designer, and to provide facilities for the evaluation of competing designs. The first is catered for by supplying the simulator with a user interface based on familiar and

intuitive concepts adapted from various sources. The evaluation of of alternative structural partitionings is in turn catered for by facilities for the *profiling* of system performance: the collection of statistics on data flow activity.

The simulator implementation is based on a process based representation of module instances. Process scheduling is directed by the pattern of data flow between modules, allowing the implementation of the *Pink* language *instantaneous* mode of intermodule communication. A simple extension of this scheduling algorithm is used to introduce the concept of time into the simulation, thus implementing *delayed* communication semantics. This implementation results in efficient simulation as the invocation of functional module code is kept to a minimum.

The implementation makes use of a number of features of the Modula-2 [Wirth, 1982] language to provide a facility for user written extensions to the simulator: a parsing facility that enables users to rapidly define translations between *Pink* and other languages; and a structure that allows users to specify new commands to the simulator command interpreter.

This chapter will firstly provide an overview of the simulator requirements, followed by a description of the implementation. The different modelling modes made available by the simulator are then described. An example of functional modelling in a structural design procedure is given in Chapter 7.

4.2 Functional Modelling Requirements

The basic requirement on a functional simulator is that it be able to precisely and efficiently model the design as it is described in the description language, in this case *Pink*. There are however a number of additional activities that must be supported by the functional simulator in order to facilitate the modelling of the design. In this section the various facilities required to perform these activities are described.

4.2.1 Architectural Evaluation

The primary aim of the *Pink* modelling facility is to allow the designer to examine and evaluate the partitioning proposed in the *Pink* description of a system. There are two basic means of using the simulator for this purpose: by gathering statistics

on system activity that may be used as a basis for quantitative evaluation, and by interactive examination of the system behaviour for qualitative evaluation.

4.2.1.1 Quantitative Evaluation

Ferry [Ferry, 1985] presents data that suggests that the quality of custom structured design layout is a property of the *flow of information* in the system design. By partitioning the design by function, the amount of information flowing between partitions is reduced, and this corresponds to a reduction in the average interconnect length of a design. This suggestion that partitioning based on functional issues and minimization of information flow is well matched with the concepts of cohesion and coupling in software structural design (Section 2.1). The greater interactions resulting from low cohesion and high coupling result in:

1. *Greater functional interdependence* between modules resulting in greater conceptual complexity and less reusability of module designs.
2. *More interconnections* resulting in greater area and power use.
3. *Longer interconnections* resulting in greater area and power use, and slower signal propagation.

The slower signal propagation speeds that are a result of (3) are exacerbated by the fact that with greater interactions, more data is required to be transferred over interconnect per unit time. This places even greater constraints on maximum clocking speeds and maximum interconnect lengths.

To aid in the qualitative evaluation of a proposed partitioning, particularly with respect to coupling, the following statistics may be gathered during a simulation:

1. *Link Volume:* The total volume of data that passes through a link in a given time.
2. *Link Saturation:* The percentage of the total time that a link is being used for the transfer of data. A saturation of 100% implies that the link is in continuous use.

Typically the gathering of statistics would be done whilst running a set of verification test vectors (Section 4.2.2) as they are typically chosen to strongly exercise module function.

This *profiling* activity has a close analogy to that carried out in software design. In order to analyse the computational performance of a program, designers often use a *profiling* design aid to provide statistics on how much *time* is spent by the program in each procedure under typical circumstances. This acts as a guide to finding the bottlenecks in the algorithm and its implementation. Once identified, these areas may be given special attention in order to improve the performance of the program.

The cost factors in VLSI design are quite different to those in software design (Section 1.1.2). In software the limited resource is *computation* as the host has finite processing power. In VLSI the limited resource is *communication* as the chip has limited area. Hence the proposal to primarily profile *data flow* rather than computation time. The resulting statistics assist the designer in identifying:

1. Tradeoffs between wide/slow and narrow/fast communication paths.
2. Tradeoffs between additional communication/duplicated functionality.
3. Underutilized (space consuming) communication paths.
4. Overutilized (time consuming) communication paths.
5. Heavily interacting modules.

In addition to collecting communication statisitics, it is also useful to have a measure of computation load for each module similar to that found in conventional software procedure profiling. The equivalent here is to measure percentage of the total during which each module is activated. This figure assists in identifying parallelism in the design. In a highly parallel design the modules will all have similar percentages of activation. A relatively low activation measure for a module suggests that it has a low computational utilization and may require further investigation.

Interpretation of the statistics may require an interactive simulation session with test vectors being selected to highlight the problem area. Eventually enough information will be collected to allow the proposal of a new partitioning, or if a comparison of several alternative partitionings is made the highest quality partitioning may be selected.

An example of using the profiling statistics as a form of quantitative evaluation may be found in Chapter 7.

4.2.1.2 Qualitative Evaluation

The commands provided in the simulator should enable the designer to interactively examine the static structure created by the description. The connections between ports have been resolved into single nodes in the simulator and hence the ports that share the same values become explicit. These operations are an aid to the designer in observing the total interconnect structure and the qualitative partitioning that implies. This function is also an aid when the communication metrics described in the following section are being used to pinpoint problem areas in the partitioning.

The interactive dynamic simulation facilities such as vector stimulus, watches, breaks and plots may be used by the designer to observe system communication dynamics as the simulation proceeds, aiding in the evaluation of communication traffic patterns and densities.

4.2.2 Functional Verification

In its most broad sense, functional verification refers to the need to verify that a description performs its intended function. Design with *Pink* involves the successive refinement of abstractions into more detailed descriptions. One of the most common refinements is the hierarchical decomposition of an abstract module into a set of interconnected submodules. Another class of refinement occurs when detailed time delay information is added into a data communication description. In either case it is necessary to verify that the new refinement is equivalent in function to the previous more abstract description.

There are two basic approaches to the verification problem:

1. *Proving Correctness.* This is the most intellectually satisfying method of functional verification: proving that the composition of the functions of the subordinate components is logically identical to the parent module's function. This task is itself an area of considerable ongoing research [Birtwistle et al., 1986; Barrow, 1984; Gordon, 1979]. The structure of the *Pink* language does not

preclude the use of proof techniques, but such verification is seen as beyond the scope of this research and in the area of challenging future work.

2. *Validating Correctness.* A less satisfying but more accessible method of functional verification involves the use of a simulation of the description. Upon defining a module, the designer specifies (perhaps with the aid of specific design tools) a set of input and output test vectors that purport to exercise the module's function. The assumption is then made that any entity that claims to emulate the module's function (a decomposition of that module in terms of interconnected subordinate modules, for instance) must produce the same output vectors upon being stimulated by the input vectors. Thus in order to verify a decomposition the designer feeds the test vectors to a simulation of that decomposition and compares the output with the specified vectors. If there is no difference, the designer has *a degree of confidence* that the decomposition is correct. Clearly unless the set of test vectors is complete (usually impractical due to the large number of states of a module) this *validation* process is not as reliable as a proving process. There is considerable ongoing research in the area of automatic generation of test vectors [Gerner & Johansson, 1986].

   Although less formal than a proof, validation does however have several advantages:

   (a) Implementation is *much* simpler.

   (b) The description language does not have to cater to the requirements of an automatic theorem prover.

   (c) The test vectors provide a general means of verifying more than simply functional descriptions: they may be used for verifying the correctness of circuit layouts, and the functionality of the fabricated circuit itself.

In this case of *Pink* description verification, the validation process is regarded as having two aspects:

1. The informal, interactive simulation of a description by a designer that is typically used to discover straightforward errors in the design.

2. The more formal verification produced by the use of test vectors that the design performs its intended behaviour with respect to some other model of the design.

4.2.2.1 Informal

The input of a system description involves the designer creating the text that described both the function of modules and their interconnect. Such a process is, like programming, inherently error prone and likely to lead to syntactic and semantic errors. Basic syntactic and structural errors may be caught by the compiler and run time system. To aid the designer in finding more subtle errors, the interactive simulator interface allows the designer to treat the description as a program to be debugged. Commands are available to stimulate the system, observe system activity in a running simulation, halt the simulation and examine its state. This environment is similar to that provided by an interactive source level debugger for software, and in fact the intent is the same: detection and tracing of specification errors.

4.2.2.2 Formal

Whilst the informal interactive procedure outlined above is useful for the initial debugging of a description, it does not address the problem of more formal functional verification. For this purpose a process of *validation* in the form outlined above has been adopted. The file based stimulation (4.10.5.2) and observation (4.10.7.1) facilities of the simulator may be used to feed test vectors into a module and record the result. It is a simple task to automate the process using operating system command files that appropriately invoke the simulator with the test vectors as input, then compare the output files generated with the required result, flagging any differences between the two. The interactive facility may then be used to debug the description. Test vector generation and validation may also be carried out by testing modules defined in *Pink* code and connected to the module being validated. An example of this is shown in Section 3.11.

4.2.3 System Integration

A VLSI circuit is usually only a single component of a larger system that consists of a number of hardware and software components. A working simulation of the system such as that supplied by the *Pink* simulator may be substituted for the actual chip during the development of software for the chip and the design of other hardware that must interface to the chip. This allows system hardware and software development

to be carried out in parallel to the chip design and fabrication.

Hardware interface design may be carried out by creating *Pink* descriptions of the interconnected system components and connecting them up to the *Pink* description of the chip. The success of the interface design may then be verified by simulation.

Software development may be carried out by downloading the code into the memory part of a simulation (using the file based stimulation facilities) and running the simulation to observe software behaviour.

4.3 Design and Implementation Overview

There primary issues in the design and implementation of the *Pink* simulator are:

1. *Accurate* modelling of the language.
2. *Efficient* modelling of the language.

These affect all areas of the simulator design, but the major impact is in the area of communication and scheduling (Section 4.8). It is in this area that this implementation most differs from existing functional simulators.

The implementation of a simulation of a *Pink* description involves the the following phases:

1. Translation of the *Pink* description into Modula-2.
2. Compilation of the Modula-2 source into object code.
3. Linking the object code derived from the description with both the run time system object code and the object code representing any library modules used in the description. The linking process results in an executable image.
4. Running the executable image as the simulation.

The Modula-2 code produced by the translation process assumes the existence of a number of external routines that serve to connect the code to the *run time system.* The run time system is a large body of code that provides:

1. Data structures that represent the interconnections between modules.

2. A means of implementing instances as processes.

3. A scheduling mechanism that controls the activation of instances given specific patterns of data transfer between the instances.

4. Monitoring of data flow and module activation.

5. A command interpreter that enables the user to control the simulation.

6. Interactive and hardcopy display of system activity.

7. User accessible facilities for language parsing and additional command definition.

4.3.1 The Implementation Language

Several reasons for choosing Modula-2 [Wirth, 1982] over other available programming languages as a basis for *Pink* descriptions were given in Section 3.5. There are a number reasons that Modula-2 is an appropriate language based on its suitability as an implementation language for the simulator:

1. The language provides excellent facilities for developing large, well structured programs. In particular there is strong control over the import and export of entities across software module boundaries. Module interfaces are defined and the module implementations hidden, thus reducing the interaction between software modules to well defined routes.

2. The language provides facilities for the programmer to create and manipulate simple *processes*. A process may be created within which a procedure may execute with its own state: variables and program counter. This facility is typically implemented as *coroutines* on a uniprocessor. Processes are used in the simulator as a basis for the implementation of instances.

3. Procedures are defined as a first class data type and thus may be assigned to variables. This is required as procedures are to be assigned to be executed in processes.

4. The language is strongly typed, a property that is generally regarded as leading to the creation of more robust software as a number of classes of errors may be detected syntactically.

5. There are a number of implementations of the language on different machines and operating systems ensuring the portability of the simulator.

Modula-2 has several disadvantages as an implementation language:

1. Procedures may only have a fixed number of actual parameters. Variable parameter lists for commands such as `Connect` must be implemented by the macro preprocessor.
2. The terminal and file input/output procedures are not standard, being provided by implementation-dependent libraries. This may cause portability problems.
3. There is no standard method of declaring text strings of dynamically varying length. The nature of the task is such that this would be a useful attribute.

In balance however the advantages of Modula-2 were judged to outweigh its disadvantages.

In addition to Modula-2 there are a number of languages that might be used to form the basis of a VLSI description language. The following is a brief list of these languages and their characteristics:

1. *Lisp:* The macro facility is very useful for embedded language definition, and symbolic processing may be appropriate. Object oriented extensions such as *Flavors* [Weinreb & Moon, 1981b] would provide a basis for instance state representation. Automatic memory management with garbage collection is an advantage, but has an efficiency overhead as do tagged data types. There are no standard means for specifying processes.
2. *Simula:* The class structure is particularly valuable in design representation [Buchanan, 1980]. The syntax is Algol-like and somewhat lacking in familiar control structures. There is not a great deal of availability on a range of machines.
3. *C:* Very widely available. Not a simple syntax, and has poor software management facilities. Has a useful preprocessor for macro expansion. Does not have coroutines facilities.
4. *C++:* An object oriented version of C, it has a number of facilities that make it suitable for embedded language applications. It was not available at the time of the design of the *Pink* simulator.

5. *Pascal:* Simple syntax, but has no software management facilities, nor coroutines.

6. *Ada:* Large enough to provide virtually all of the functionality of the previously described languages. Does not however have a familiar syntax, and is not as yet available on a wide range of machines.

4.4 Translation to Modula-2

One of the considerations in the design of the language was to make it syntactically similar to the Modula-2 base language in order to simplify and hence speed the translation process. This simplification was aided by the close relationship between the structural specification of hardware and software. The result is that the translation to Modula-2 is quite simple with most *Pink* constructs being implemented simply as syntactic *macros*. Macros are a means of specifying a template for translating one textual entity into another.

The macro definitions fall into three classes:

1. Those that translate into a simple token in the result, and thus only provide substitute a preferred keyword. For example the *Pink* `DEFINITION` keyword translates directly to the Modula-2 `PROCEDURE` keyword.

2. Those that translate into a number of statements. For instance the *Pink* `STATE` keyword translates to:

```
VAR
     INSTANCENAME : string;
     READYTODODECS : BOOLEAN;
     SIGNAL : node;
```

   The Modula-2 `VAR` keyword initiates variable declarations, the three subsequent declarations are for variables which are used in the result of the translation but never seen by the user.

3. Those that serve to translate argument lists from the form used in *Pink* to valid Modula-2. For example the *Pink* procedure call `DeclarePort(CarryOut, BIT, OUTP)` translates to:

```
declareport(INSTANCENAME, "CarryOut", CarryOut, BIT, OUTP)
```

The `INSTANCENAME` variable is used to pass the name of the instance the call occurs in (only known at compile time) to the run time system. The string `"CarryOut"` passes the *name* of the port to the run time system. The variable identifier `CarryOut` passes the *variable* representing the port to the run time system. The identifiers `BIT` and `OUTP` are passed to the run time system unchanged.

In the current implementation of the simulator the C language [Ritchie & Kernighan, 1978] macro preprocessor is used to perform macro expansion. Only two pages of macro definitions are required to perform the entire translation. The C preprocessor also perform the expansions of user defined macros for parameterization of definitions (Section 3.9).

One aspect of the *Pink* language that cannot be translated with simple macros is the syntax developed to express regular structures (Section 3.6.3). This syntax is unlike any existing Modula-2 construct and so must undergo a more complex translation. Additionally the syntax is quite general: it is intended to be used even in user defined procedure calls. The chosen solution is to implement the syntax as an unusual form that is similar to a macro in that involves text-rewriting, but unlike a macro in that it does not follow the conventional form of identifier and argument list. Instead the source text is processed unchanged statement by statement, until one or more `<a..b>` type constructs are identified in a statement. The statement is then re-written for each step of the expansion, with the `<a..b>` token replaced by an integer index in the stated range. For example the statement:

```
Connection(AdderBit<0..6>/CarryOut & AdderBit<1..7>/CarryIn);
```

is translated to:

```
Connection(AdderBit0/CarryOut & AdderBit1/CarryIn);
.
.
.
Connection(AdderBit6/CarryOut & AdderBit7/CarryIn);
```

In the current implementation, the expansion of regular syntax expressions is performed by a program written in C in conjunction with the LEX lexical analysis facility [Leske & Schmidt, 1975].

The complete *Pink* to Modula-2 translation process consists of first passing the source through the regular syntax expansion program and then the macro expansion program.

4.5 Internal Representations

An fundamental task in creating a simulation of a description is to represent the structure and function implied by the description in the memory of the machine to be used for the simulation. The choice of these representations are critical as they must support algorithms that implement the semantics of the description language. They must do so efficiently in terms of both memory space and compute time, as the intent is to support the modelling of large, complex systems.

In the following subsection some of the typical simulation representations and their limitations will be described in order to place the decisions made in this implementation in perspective. This is followed by a description of the methods chosen to represent definitions and instances in the *Pink* simulator. This section is intended to describe those aspects of the system that are in a sense *static* during the simulation. In a subsequent sections the more *dynamic* aspects of the simulation process will be described.

4.5.1 Conventional Representations

The conventional means of representing an instance in a functional simulation is as a normal procedure. Each time one of the input ports receives a new value the procedure is called. It must then run through to completion and terminate, returning control to the procedure that invoked it. Although almost trivial to implement, this structure has a number of severe limitations including:

1. *Inefficiency.* Every time a module input changes the relevant procedure must be invoked, typically generating some new output which in turn causes other procedures to be invoked. In many cases it should be possible to only execute a section small section of code before suspending till further required data arrives. This would reduce execution time within the instance, and reduce the amount of unnecessarily propagation which occurs between instances.

2. *Pausing.* It is often convenient to express function in terms of *time pauses* in which the module's function is suspended for a specified length of time (as lacking but suggested for the VHDL language [Nash & Saunders, 1986]). As it is not possible to suspend a normal procedure midway through its execution and restart another, it is not generally possible to implement pausing with a procedural representation.

The following subsections describe the process based representations used in the *Pink* simulator that do not share the above limitations.

4.5.2 Definitions

A *Pink* definition serves as a template for the creation of instances, and as such does not appear as a connected part of the internal representation of the structure implied by the description. Instead the definition is translated into a procedure: a piece executable code which may then be used as a template for constructing instances each time a `MakeInstance` of that definition appears.

The *definition procedure* for a topological definition has the form:

```
PROCEDURE <definition name>;
VAR
  --- Declare internal variables.
  --- Declare user variables for ports and state.
BEGIN
  --- Wait for "initialization" signal.
  --- Describe ports.
  --- Describe instances to be made.
  --- Describe interconnection of ports.
  --- Wait for the end of the simulation, doing nothing.
END <definition name>;
```

The *definition procedure* for a functional definition has the form:

```
PROCEDURE <definition name>;
VAR
  --- Declare internal variables.
  --- Declare user variables for ports and state.
BEGIN
  --- Wait for "initialization" signal.
```

```
  --- Describe ports.
  --- Wait for "simulation begin" signal.
  LOOP
      --- Execute code that emulates module function.
      --- Repeat the execution for the duration of the simulation.
  END;
END <definition name>;
```

The following two subsections describe the means by which definition procedures are used to create instances and interconnect respectively.

4.5.3 Instances

A instance is an instanciation of a definition. For reasons that will become clearer later in this chapter, there are two basic requirements for such as instanciation:

1. *Preservation of control state.* A single instanciation should be preserved for the entire life of the simulation and have the capability of being activated and deactivated as required for the simulation.
2. *Preservation of variable state.* The values held in the variables tied to an instance should be preserved for the entire life of the simulation.

Conventional *procedure call* semantics do not fulfill these requirements as they imply the existence of only a single linear calling hierarchy (usually implemented with a single stack). This implies firstly that the only means of creating an activation of a procedure B is to call it from the currently active procedure A. B may call subsequent procedures, however eventually it must terminate if procedure activation A is to continue. Hence there is neither a means to activate and deactivate a single activation of a procedure, as required for preservation of control state, nor is there a means of preserving variable state as when a procedure terminates, all state is lost.

A set of semantics that do fulfill the requirements outlined above are those associated with *simple processes*. Here a simple process is defined as providing an environment in which a procedure may be activated and may preserve its control and variable state indefinitely. Simple (or *light weight*) processes are typically implemented by providing a procedure with an individual stack upon which it may allocate space. Control is switched between procedure activations by a switching of stacks, only one

being active at a time. Light weight processes differ from *heavy weight* processes in that the later also provide a complete memory space for the procedure to execute in. Light weight processes are exemplified by those in the Lisp Machine [Weinreb & Moon, 1981a] operating system. Heavy weight processes are exemplified by those in operating systems such as UNIX [Ritchie & Thompson, 1974].

The Modula-2 language provides a facility for the creation of light weight processes. This facility is independent of the operating system as the processes are implemented using a simple *coroutine* structure on uniprocessor machines [Wirth, 1982]. The interface to the processes module is via two procedures:

1. `NEWPROCESS` Allows for the creation of a new process running a user specified procedure.
2. `TRANSFER` Allows for the transfer of control from the current process to some other specified process.

The `NEWPROCESS` procedure may be used to create an instance of a definition by passing it the name of the definition procedure for the required instance. The `TRANSFER` procedure may be used as the basis for process scheduling and signaling as described in a later section (4.8). For the moment it is sufficient to assume that processes may be created within which procedures may be run, and that there is a signaling mechanism with which processes may suspend their execution and await the arrival of synchronizing signals.

Whenever a `MakeInstance` appears in the description, a simulation representation of the instance is created by passing the instance's definition procedure to `NEWPROCESS` which created a light weight process within which the instance may execute.

4.6 Initialization

The initialization phase of the simulation is defined as that between the initial invocation of the simulator and the appearance of the command line interpreter prompt that indicates that the program is ready to begin simulation. During initialization the definition procedures are used to create the internal representations of structure and interconnect.

This process is described in the following sections.

4.6.1 Genesis

The program begins execution with the statements in the top level block of the main software module. This contains only one statement: a call to `MakeTopInstance`. In effect this simply calls the more general `MakeInstance` procedure with the name of the top level definition to be instanced and the name to be given to that instance (the operation of `MakeInstance` is described in the following subsection). This call takes place within the only process that exists at that time which is named "`ParentProcess`". For the duration of the simulation `ParentProcess` serves as the *control process* or *scheduler*: all other processes are associated with particular module instances.

The code executed by the parent process during the initialization phase may be summarised as:

1. Create the top instance, thus recursively creating the instance process hierarchy.
2. Send the initialization signal, causing all instance processes to execute their port declarations, thus completing the interconnect data structure.

These phases are described in more detail in the following sections.

4.6.2 Instance Creation

A call to `MakeInstance` has the effect of creating an *instance process*. This process is set to run the definition procedure of the instance requested. Just before it is about to return to the caller, the *MakeInstance* procedure suspends the current process and activates the instance process it has just created, thus beginning execution of the associated definition procedure. `MakeInstance` appends the new instance name to the name of the instance process that made the `MakeInstance` call and passes the resulting name to the new process in order that it may be have a record of its full name (Figure 4.1). The name is stored in the `INSTANCENAME` variable of each instance process. This is passed as a parameter in most procedure calls into the run time system in order that the system have knowledge of the name of the calling instance process. This is particularly important in dealing with ports, as within a definition they appear only as simple names such as `CarryOut` however if an operation

on that port is to be described usefully to the simulation user it must be appended to the full name of the instance, `Adder/AdderBit15/CarryOut` for example.

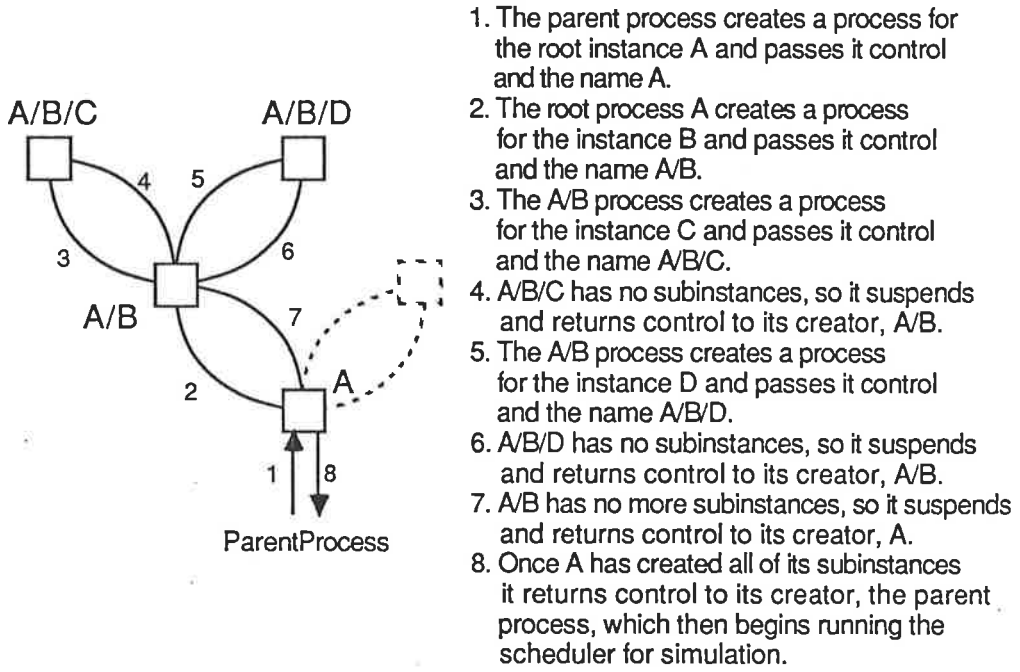

Figure 4.1: The recursive assembly of the instance process hierarchy.

4.6.3 Instance Hierarchy Creation

If the newly created instance process has an associated topological definition procedure, the process immediately begins to execute any `MakeInstance` calls contained in that definition. The effect then is to recursively build the entire instance hierarchy up in a depth first manner. This process is illustrated by example in Figure 4.1. Eventually in any given instance process, processes will have been created for all the subordinate instances and control will return to the given instance process.

4.6.4 Connection Creation

Once all of the `MakeInstance` calls have been completed by an instance process involved in creating a topology, any `Connection` procedure calls are proceeded with. The names of the ports be be connected are passed in as a single string. This string is parsed by the run time system. If any ports are listed that have not yet been

created, they are created on the appropriate instance. The ports are connected by the creation of a node data structure that is pointed to by all connected ports. The connection process is illustrated in Figure 4.2. After completion of connections, the instance process suspends its operation pending an initialization signal.

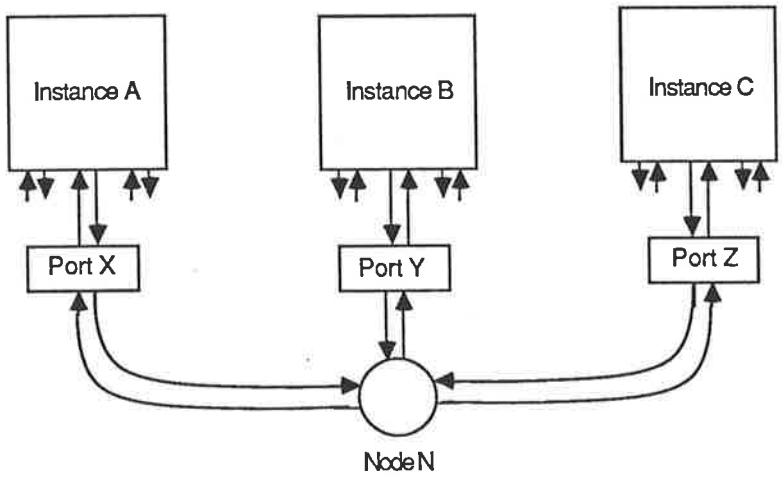

Figure 4.2: The statement `Connect(A/X & B/Y & C/Z);` creates the above data structure. The node N is implicitly created in order to provide a common data area for values written and read by the three ports.

4.6.5 Port Declaration

Once all the instance processes have completed their instancing and connection tasks their suspension causes reactivation of the parent process. The next operation carried out by that process is to broadcast the initialization signal that causes all instance processes to execute their `DeclarePort` procedure calls. Although it seems counter-intuitive to do this *after* the ports connections have been made, this ordering is necessary. Completing all the connections in the system ensures that for every set of connected ports, a single node structure now exists, to which a pointer has been generated. One of the functions of `DeclarePort` is to now set the local variable representing the port to point at that node. As a result, any read or write operation that is carried out on a port identifier will cause a direct reference to the *node* connecting the ports, not to a single port. This data structure is illustrated in Figure 4.2. In addition of course the `DeclarePort` call provides information on port type and data direction. If any connections have been made to ports that now found

never to have been declared, or there are port type and direction mismatches, fatal errors are generated.

At this point the entire function of all the topological instance processes has been completed. They proceed to suspend their execution and wait on a signal that is only sent out at the termination of the simulation session that causes them to terminate with all other processes.

4.6.6 Function Creation

Instance process that have functional rather than topological definition procedures move directly to waiting for the initialization signal as soon as they are created. On receipt of this signal they execute their port declarations and then rather than suspending and waiting for a termination signal, they suspend and wait for a "simulation begin" signal which is not sent out till it is desired that the functional code begin execution. The signalling mechanism will be described in Section 4.8.

4.7 Active Simulation

Once all instance processes have completed their initialization code and are suspended waiting for signals, control again returns to the parent process. The parent process then invokes the command line interpreter to allow the user to set up the simulation parameters before initiating the simulation. Once the user has used the command interpreter to set up the environment for a particular simulation session, the run command is given. This causes the command interpreter to return and the parent process proceeds to send out the "simulation begin" signal causing the functional instance processes to begin operation. The parent process then begins running the process scheduler code and continues to do so for the remainder of the simulation session whenever when it is activated. The basic mechanisms of the scheduler are described in the following section. The scheduler is also responsible for invoking the command interpreter upon user request, and continuing execution whenever the command interpreter is exited (the `run` command). The simulation is terminated by the user giving the `quit` command to the command interpreter, which results in the execution of the Modula-2 `HALT` procedure in the main process, causing an exit from the program.

4.8 Communication and Scheduling

Given that instances are represented as processes as described in the previous sections, the major remaining simulator design issues in the simulator implementation become:

1. Providing a *communication* mechanism that allows data to be transferred between instance processes in fashion that accurately implements the *Pink* communication semantics.
2. Providing a *scheduling* mechanism to select which of the instance processes is to be next activated in order that the simulation proceed in a deterministic fashion.

In fact the provision of these two mechanisms is a closely related problem, and the that data flow patterns within a particular design may be used to guide the scheduling in such a way as to produce an efficient implementation of the *Pink* communication semantics.

The mechanism developed is related to several techniques in parallel programming research: Hoare's *Communicating Sequential Process* (CSP) notation [Hoare, 1978] and *data flow* programming [Dennis, 1980].

The following subsections discuss the *Pink* communication semantics, the problems that occur in implementing these semantics, and the mechanism developed to implement them efficiently in the simulator.

4.8.1 Communication Semantics

The basic *Pink* communication semantics may be regarded as being associated with three major activities: *waiting*, *writing* and *reading*. The semantics associated with each may be concisely defined as follows:

Waiting. Suspend operations until one of the listed ports has a value written to it from outside the module.

Writing. Write a new value out through the port to all connected external ports, perhaps after a specified delay.

Reading. Read a value from a port such that the value is the correct value for this instant in time.

These semantics vary from this in existing functional description languages in that a delay need not be specified or implied in the write operation. A write without a delay implies that the node connecting the ports changes its value instantaneously, thus stimulating the connected instances at this same instant, perhaps causing them to generate new outputs at the same instant. In this manner a complete system may be stimulated a number of times without any passing of simulation time (the utility of such a timing abstraction is discussed in the previous chapter).

4.8.2 The Scheduling Problem

The scheduling algorithm is responsible for selecting which of the instance processes is to be made active and allowed to proceed with execution. The current implementation of the simulator is on a uniprocessor, thus limiting the size of the set of active processes to one. However in a multiprocessor implementation this set could be greater in size with active instance processes being allocated to processors.

The scheduling of instance process should not be visible to the designer, rather the designer should only be able to perceive that the semantics of the communication operators has been correctly emulated. This is necessary as modules in a VLSI system operate with complete parallelism and any appearance of sequential operation in the simulation is highly undesirable as it would violate the parallel nature of the module entities in the description.

The *instantaneous* communication semantics in *Pink* present a challenging implementation problem: *when an instance requests the value of a node by reading from a port at a particular instant in time, the value returned must show the result of any writes that take place to that node during the same instant.*

How then is scheduling to be directed? Instance processes *must* be activated by the scheduler if simulation is to proceed, but no process must be activated that uses the value of a node if that node may be have a new value written to it during the same instant. This dilemma is illustrated by example in Figure 4.3.

The solution to this problem presented in the following subsection relies on various properties of the description language, properties of VLSI systems design and a

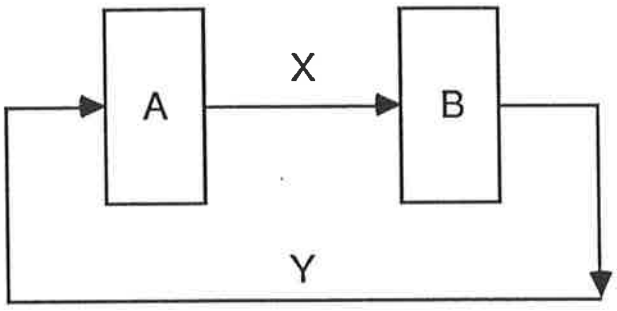

Figure 4.3: The scheduling dilemma. Instance A may only be activated if node Y is not to be written by instance B in the current instant. Similarly Instance B may only be activated if node X is not to be written by instance A in the current instant.

number of software techniques.

The *delayed* communication semantics in *Pink* may be implemented as an extension of the mechanism for implementing *instantaneous* semantics and are described in subsection 4.8.5.

4.8.3 Interprocess Communication Techniques

As a prelude to describing the scheduling algorithm, some of the parallel process communication structures that have influenced the design will be introduced.

One of the most widely know models for interprocess communication is that proposed by Hoare know as Communicating Sequential Processes (CSP) [Hoare, 1978]. Communication is based on two operators, one for input ("?") and the other for output ("!"). Interprocess communication takes place when an output operation specifies that data be output to a second process and the second process requests input from the first process. The operation (input or output) which is requested first is delayed by suspending the process till the other is requested, thus ensuring synchronization (Figure 4.4).

Although not strictly only a parallel processing method, *data flow* techniques have been used for the programming of parallel machines [Dennis, 1980]. In a data flow language, rather than evaluate expressions in a fixed sequential order, operators are evaluated only when their operands are ready to be processed. Thus as operands are generated as the result of operations, they propagate outward and allow further

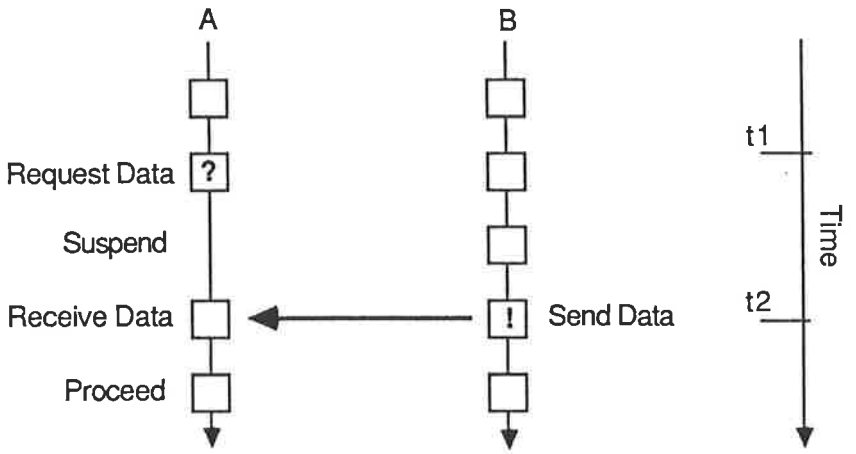

Figure 4.4: An example of CSP semantics. The process A requests input from B at time $t1$. The data is not available, so A is suspended. The process B does not become ready to output data till time $t2 > t1$. As A has indicated it is ready to receive data, the transfer takes place and both processes continue.

operations to take place (Figure 4.5).

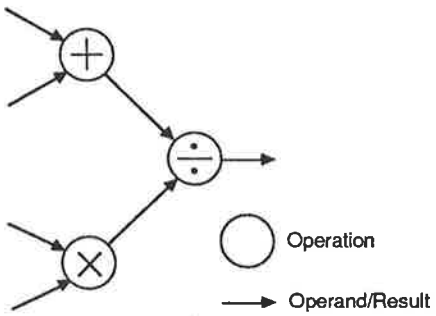

Figure 4.5: Example of a data flow evaluation. Each arithmetic operation only executes when all of the required operands have arrived.

4.8.4 Demand Driven Scheduling

The algorithm developed for scheduling and communication in the *Pink* simulator is know as *demand driven scheduling*. The majority of the scheduling function is in fact distributed into the implementation of the port writing and reading operations.

When an instance process attempts to *read* from a port one of two events may take

place:

1. If the attached node has been written during the current time (*tick*) then the read returns with the current value of the node as the result. This is the correct action because the language semantics require that a node not have more than a single value written to it during a single time interval. Multiple writes may occur, but they must place the same value onto the node as any previous writes in that interval. Thus it is safe to return the node value as there is no way it may change to another value during the current instant.

2. If the attached node has *not* been written during the current time (*tick*) then it is not yet possible to return from the read as in this some instant another instance may yet write a new value to that node. The problem is deferred by suspending the process that requested the read operation. A note is attached to the node that this process (and any others) are in a state of suspension pending a write to this node.

When an instance process attempts to *write* to a port several events take place:

1. The value of the node is updated to the new written value and a note of the writing time is made on the node.

2. The list of process that suspending pending activity on this node is examined. Each process is resumed in order that the original read each requested may return the new value.

As each of the reads in (2) above completes, the instance that is reactivated may in turn generate writes, and these in turn may cause the activation of other suspended process. Thus the effect of a single write (such as to a clock line) may propagate out thorough the structure causing processes to execute functional code.

The `WaitFor` operation is implemented simply by suspending the requesting process and making a note on the specified node that the process is to be resumed when the node is next written. `WaitForAny` extends this to a number of nodes, the writing of any one of them causing a reactivation. The wait operations, when combined with the write operations may be viewed and used as an interprocess signaling mechanism. In fact the internal control signaling referred to in Section 4.5.3 uses this mechanism, the signal channels being simple nodes as used in circuit descriptions.

The relationship between the *Pink* scheduling mechanism and CSP semantics stems from the behaviour of the read operation. When a read occurs, if there is no write in place on the node, the reading process is suspended. This is similar to the way in which an input and output operation are synchronized in CSP. They two diverge on write/output semantics however. Whilst a CSP process will suspend till an input is found to match output, a write in *Pink* is allowed to complete immediately if there is no matching read. Here the *Pink* semantics approach those of data flow: if there *are* outstanding reads on a node, and that node is written to, the immediate response is to allow the waiting processes to proceed as they now have the values they require to continue operation.

The basic scheduling is then based on interactions between the writing, reading and waiting operations. In most specifications this process will continue for an interval until there are no more instance processes that are ready to run: the system requires a write operation to occur before further progress may be made. Control automatically reverts to the parent process which is at this stage running the scheduler code. There are two distinct situations that may give rise to this situation, and two corresponding responding responses:

1. *There are no outstanding reads on nodes.* All simulation of this time instant has been completed. The event list (subsection 4.8.5) is examined for the next event scheduled at a future time, the current simulation time is updated to that time and the event processed causing the simulation to proceed as required.

2. *There* are *outstanding reads on nodes.* Simulation of this time instant has not been completed: one or more instances that required a node value as part of their computation did not receive a value because the node was not written during this instant. It is assumed that at least one of the reads was intended to return a value written to the nodes at a previous time. Once this node has been identified the current value of the node is in fact returned and the suspended instance processes reactivated causing the resumption of simulation during this instant.

The situation described in (2) above requires that a means be found of identifying a node (or set of nodes) that may have the old value of a node returned to them. The procedure adopted is to examine all of the ports that are connected by the node and disqualify the node if any one port is not an input port. An output or

bidirectional port might yet cause a write to the node in this instant, *unless* the module is suspended pending a write to *this* node. It will not write to this node if it has just read from it in the same instant.

The above procedure does not guarantee that a suitable node will be found. If not, and arbitrary choice is made between the nodes and one is returned with old values. If this choice is in error, the node will in fact be written during this instant after it has been read (for its old value). Such a sequence is however illegal (a node must not be used before it is written in any one instant) and an error will be signaled to the user. In the unlikely event of this happening the user is required to select a port in the description and use a simulator command to label it as an "old value" port that when read always expects to be returned a value that was written to the node at a previous time. This removes the ambiguity from selecting such a node, and at the same time does not effect the description: the exercise is associated with a limitation of the simulator, not the description.

4.8.5 Delay Driven Scheduling

The previous section has described the mechanism by which the instantaneous write, read and wait operations are implemented. Time must of course be introduced into the simulation since the description allows the use of time specifications in delayed write and pause operations. The concept of time is introduced into the scheduling/communication mechanism by means of two related entities: *current time* and an *event list.* The current time is simply an integer representing the current tick of system time. The event list is comprised of records of future events that are sorted on the basis of the time at which they are scheduled to occur, the soonest at the head of the list. There are two types of simulation event that may appear on the list:

1. *Write events.* Whenever a delayed write is requested by an instance process, the delay interval is added to the current time, a record of the resulting time and the details of the write is created, and the record is sorted into the event list according to its scheduled time. The delayed write request then returns allowing the instance process to continue immediately. When a write event is executed at a later time, it simply results in an instantaneous write thorough the specified port.

2. *Resume events.* Whenever a pause is requested by an instance process the delay interval is added to the current time, a record of the resulting time and the details of the process to be resumed is created, and the record is sorted into the event list according to its scheduled time. The instance process is then suspended. When a resume event is executed at a later time it simply results in the suspended process being resumed from the pause request.

As noted in the previous section, whenever there are no more instance processes in a state to run, control is returned to the parent process which is running the scheduler. The scheduler examines the next event on the event list. If the next event is scheduled to occur at the current time, that event is taken from the head of the list and executed, causing a write or resume operation to take place. If on the other hand the next event is scheduled to take place at a time greater than the current time, the scheduler first ensures (using the mechanism described in the previous section) that all outstanding events in the current instant have been performed. The head of the event list is then examined again and the process repeated until there is an event scheduled for a future time on the list and the operations for the current instant are complete. This state implies that the next simulation event must be that specified by the record at the head of the list, and that it must occur at the scheduled time. The current time is update to that time, and the event removed from the head of the list and executed.

In this manner simulation time progresses forward, the current time only ever taking up values associated with specific events: if nothing is scheduled to happen at a particular time, that time is skipped over.

Note that in addition to write and resume simulation events the event list is used for scheduling simulation control events such as user requested interruptions of the simulation.

4.8.6 Discussion

There are several advantages to the scheduling algorithm that has been described above:

1. The algorithm allows for the implementation of the *Pink simultaneous* communication scheme. By avoiding the arbitrary scheduling of instant processes

within a particular instant, it is possible to model design descriptions that do not specify delays.

2. Instance processes preserve control state as well as data state between activations. Thus an instance process may express interest only in a subset of the instance ports, and will only be activated when one changes. This reduces the number of instances that must be activated.

3. The bulk of the scheduling is distributed, the scheduler only being activated when there are no more activities that may be carried out within the current instant. Ghosh [Ghosh, 1986] suggests that such distributed scheduling schemes are applicable to multiprocessor implementations.

4.9 Software Organization

In the current implementation, the *Pink* to Modula-2 translation process is carried out by passing the source code through the C program that performs the regular syntax expansion, and then through the C preprocessor to expand both predefined and user defined macros.

The run time system is comprised of software modules that when linked together form the complete system. This may then be linked with the user code that describes the design. The software modules and associated functions are:

kernel The kernel module implements memory allocation, process creation, process scheduling, data transfer and simulation monitoring.

userifc The user interface module implements the command interpreter and associated utility routines.

ute The utility module implements widely used routines that are not functionally interrelated

hash The hash table module implements a fast facility for hash table access of port records by way of their names.

plot The plotting module implements procedures for plotting graphs of the simulation

vis500, vis550 Modules that implement terminal drivers for the respective terminals.

osint The operating system interface module implements calls into the operating system. In the current implementation this is UNIX.

floyd The Floyd parser module implements a back end for translating *Pink* to a form suitable for the floorplanner *Floyd* (Chapter 6).

The structure of the simulation software is such that each time a user creates a simulation of a new description a new executable image is generated by relinking the software modules. It is then quite possible to let the user supply software modules other than those created by translating the *Pink* description. This is made use of by supplying the user with access to two software facilities: one for the creation of translators from *Pink* to some other structural definition language, and another for adding user defined commands into the command interpreter syntax. The first is used for example in translating a *Pink* description into a form suitable for input to the floorplanner described in Chapter 6. The second facility is useful when users which to extend the range of available simulator commands, perhaps by tailoring commands to the particular description being modelled.

4.10 The Simulator Interface

This section describes the simulator facilities that enable the designer to actively model *Pink* system descriptions. This is not intended to be a comprehensive description of the simulator usage: the purpose is to describe the basis for the various design decisions made in the simulator construction. A complete description of simulator operations is made in the user manual [Dickinson, 1987].

The main objective in the design of the command interface was to make it as familiar and intuitive as possible. Familiarity has been achieved by where appropriate adopting concepts and commands from familiar sources such as operating system user interfaces ("shells") and debugging programs.

4.10.1 Translation and Compilation

The process of creating a simulation representation of a *Pink* description held in one or more files is carried out by one simple command. This first invokes a procedure to translate the *Pink* description into Modula-2. This is then compiled and the

resulting object code linked with a run time system to produce an executable image. This image may then be executed in the normal fashion and serves as a simulator of the original *Pink* description. Typically the only user interaction required prior to the simulation run itself is the correction of any syntactic errors in the *Pink* source that cause compilation errors.

4.10.2 The Command Interpreter

The structure of simulator commands is hierarchical: each is introduced by a single verb such as "show" or "set" followed by a number of qualifying verbs, nouns and identifiers. This approach makes it simpler for new users to become familiar with the commands as the structure resembles English:

```
clear break port CarryOut
```

("Clear [the] break [on the] port CarryOut") for example. For more experienced users the verbs and nouns may be shortened to unambiguous abbreviations:

```
cl b p CarryOut
```

for example. An interactive help facility provides information about the commands their arguments.

4.10.3 Browsing the Description Structure

In a heavily partitioned structural design the number of modules, the depth of the module hierarchy and the module interconnections result in a structure that may have considerable implicit complexity. It is important that this be reduced as much as possible by providing the user with a simple and consistent conceptual representation of these structural features. Such a representation and commands for manipulating make it simpler for the user to comprehend and "move around" in the hierarchical structure during a simulation session.

This problem of representing a design hierarchy is analogous (though not identical) to that of representing a hierarchical file system to the user of an operating system command interpreter (or "shell"). In the UNIX [Ritchie & Thompson, 1974] file system directories and files are presented as being similar entities: directories and files are uniquely identified by either a full "path name" including the root directory or a path name relative to a "current directory". The contents of a directory are listed with the `ls` command and the current directory is changed with the `cd` command.

To make use of the familiarity users generally have with these concepts the *Pink* simulator uses `ls` and `cd` in a similar manner. The `ls` command causes a listing of the structural information about an instance: firstly all of the subordinate instances are listed if the instance is topological, then a list of all of the ports on the instance are given. The `cd` command changes the current default instance. The combination of these two (often familiar) commands provides a simple and effective facility for dealing with a complex structural design.

4.10.4 Error Detection and Notification

The simulator has the capability to detect several classes of error that may exist in the description, or occur during the simulation. These include:

1. Errors in port connection: mismatched data types (`BIT` and `INT`), and mismatched signal types (`INP`, `OUTP`, `CLOCK`, `POWER`, `GND`, and `IOP`), and mismatched bus sizes.
2. Errors in leaving ports unconnected.
3. Errors caused by writing to a node more than once with different values in one instant.
4. Errors caused by reading a node after a preset amount of time has passed during which charge would have leaked off in a physical implementation.

Should one of these classes of error occur the simulation is halted, and and error message displayed. If the error is such that the simulation may be recovered and continued, the command interpreter is invoked to await further commands. If the error was fatal to the simulation, a special version of the interpreter is invoked that enables the user to examine the state of the simulation in the normal way, but not continue any further with actual simulation.

4.10.5 System Stimulation

In order to make use of the active model of the system provided by the simulator it is clearly necessary to stimulate the system or the portions of it that are of interest to the designer. The simulator provides a number of means of stimulation, each

applicable to particular styles of modelling. These methods and their applications are described in the following subsections.

4.10.5.1 Interactive

For informal debugging and "browsing" investigations of system behaviour, particularly when a description is being first simulated, an interactive style of stimulation is appropriate. The user may use interactive commands to attach input vectors to particular ports and buses, and then continue the simulation and observe the resulting activity.

The basic command for interactive stimulation has the syntax:

```
set vector <port> <vector>
```

The identifier <port> may refer to any port or entire bus. The vector is actually attached to the node that comprises all of the ports that connect to <port>, and all such ports will be supplied with values from <vector>.

The identifier <vector> has a variable syntax depending on the nature of the port specified by <port>: this syntax is described in the user manual [Dickinson, 1987].

There are several points to be noted with regard to the semantics of using the vectors described above for stimulation:

1. The first time that a node is read at a particular time it will return a new vector from the vector of values.
2. If a node that has a vector attached is read more than once at a particular time it will return the same value from the vector of values.
3. It is not valid to write to a node that has an attached input vector as this would lead to conflicts over which source to use for the nodes value.

4.10.5.2 File Based

In a number of modelling applications, particularly verification (Section 4.2.2) it is necessary to stimulate the system with a very large number of values. To this end

the simulator provides a facility for stimulating nodes with values obtained from data files.

The basic command for file based stimulation has the syntax:

```
set infile <port> <filename>
```

The file identified by <filename> contains data that is treated with the same semantics as the contents of the vectors described in the previous sections. The syntax is however slightly different and is described in the user manual [Dickinson, 1987].

4.10.5.3 Language Based

In some modelling situations the most convenient means of generating stimulation vectors is to build a "generator" module within the *Pink* description itself. This generator may then be connected up to the system under examination and the simulation run with the generator providing the stimulus. The most common example of this is the use of clock generators. Such a generator is described in Section 3.11 of the previous chapter where it is used to provide clocking stimulus to an adder.

4.10.6 System Observation

In the same way that it is necessary to stimulate the system being modelled it is clearly necessary to be able to observe and record its behaviour. The simulator provides a number of facilities for aiding the designer's observation of system behaviour, each intended to provide for different styles of modelling. These are described in the following subsections.

4.10.7 Interactive

As indicated previously, for casual "browsing" investigations of system behaviour an interactive style of stimulation is appropriate. To facilitate this style of modelling the simulator provides a number of methods of interactively observing system behaviour.

Commands are provided that allow the user to interrupt a simulation at any time and proceed to examine the state of the ports and nodes before continuing the simulation

run. Additionally however there are a number of facilities for examining a running system.

The first and most general of these is the *watch*. A watch can refer to either a single port, node or entire bus. After a watch has been set on a port or node, if a read or write activity takes place associated with the watched item during simulation, a message is displayed to the user indicating the name of the item accessed, the name of the accesser, the time of access and the items new value.

The second active observation method is via *breaks*. A break is similar to a watch except that as soon as an access occurs on an item with an attached break, the simulation is halted and control is returned to the user via the command interpreter. The commands for observing the state of the static may then be used.

The third active observation method is *plotting*. Any single BIT item or bus may observed using a graphical plotting facility. The plot may cover a number of different items over any reasonable amount of simulation time. A facility is available for obtaining hard copies of the plots.

4.10.7.1 File Based

In a number of modelling applications, particularly verification (Section 4.2.2) it is necessary to stimulate the system with a large number of test vectors. Typically this will produce a correspondingly large number of output vectors. The simulator provides a facility for recording these vectors in files analysis. This is particularly useful in situations where simulations are not run at all interactively but instead as background or batch jobs. Output data then needs to be recorded for later analysis.

4.10.7.2 Language Based

In the same way that *Pink* code may be written to stimulate a system description, code may also be written to observe system behaviour. A module may be created that reads the values that are of interest through its ports and compares them with expected values. The following is an example of a *Pink* module that both stimulates and observes the behaviour of an adder:

```
DEFINITION AdderTester;
  STATE
```

```
    TestA, TestB, Result, Carry : ARRAY [0..7] OF PortType;
    A, B, R : INTEGER;
PORTS
    DeclareBus(A,  OUTP);
    DeclareBus(B,  OUTP);
    DeclareBus(Result,  OUTP);
FUNCTION
    A := Random();
    B := Random();
    R := A + B;
    IF R < 255 THEN
        WRITEBUS(TestA, A);
        WRITEBUS(TestB, B);
        IF READBUS(Result) <> R THEN
           WriteString("Adder error");
           FatalError;
        END;
    END;
ENDFUNCTION
END AdderTester;
```

4.11 Limitations

Ideally the simulator should be able to model any *Pink* description. However the scheduling algorithm does place a basic limitation on the nature of the data dependencies within a structure: *it is not possible to simulate structures in which there are instantaneous multiway data dependencies.* Such a dependency occurs when for instance two connected modules rely on each others output in order to generate their own output. An example of one such self timed circuit is shown in Figure 4.6(a). Such *self timed* circuits are not common as most structured circuit design styles [Mead & Conway, 1980] utilize clocking strategies that avoid them. Note that the function of such a circuit may be simulated at a higher level in which the composition is represented by a single functional module as indicated in Figure 4.6(b). Simulating the actual decomposition requires iterative techniques that are far more computationally expensive than is acceptable in a functional simulator. An alternative solution is to move to a multilevel simulation in which the selftimed sections are modelled at the circuit level.

Limitations are also placed on simulations sizes by the nature of the implementation. A typical instance process requires several hundred bytes of data space, and if there

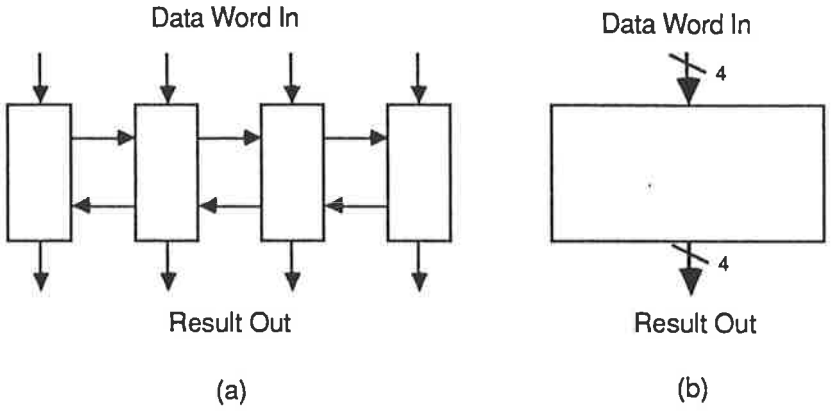

Figure 4.6: The self timed circuit in (a) finds the position of the least significant set bit in the input word. The expense of iterating to find a solution suggests the use of the abstraction in (b).

are more of these processes than may be placed in a machines main memory at a particular time, the simulation will cause system paging, thus reducing performance. This is a "soft" limitation as it causes performance degradation.

There are two basic "hard" limitations that could potentially cause a simulation to fail. The first is if the simulation is of such a side that it exceeds the operating system/machine virtual memory limit. The second is the exceeding of the basic integer size of the machine by by either the value placed on an `INT` port or by the total of simulation ticks exceeding that number. The first may be circumvented by using a `BIT` bus of sufficient width. The second is most unlikely owing to the simulation time that would have to pass to exceed this limit: if necessary it could be avoided by breaking the simulation into a number of shorter runs.

4.12 Summary

The fabrication of VLSI designs precludes the rapid feedback on design decisions that is typical of software design. This can be alleviated by the functional *modelling* of a VLSI design description in software. A functional simulator has been described that provides for:

1. The qualitative evaluation of the design by way of interactive observation of performance.

2. The quantitative evaluation of the design by way statistical *profiling* of data flow within the design.

3. The interactive "debugging" of design descriptions.

4. The verification of design decomposition and refinement by means of test vector validation.

5. The prototyping of software that is to be run on the VLSI device.

An internal simulator representation of the design has been described that features:

1. A process based representation of functional module instances.

2. A process communication technique based on Communicating Sequential Processes in which processes are suspended until required data is supplied.

3. A process scheduling algorithm that is guided by data flow and efficiently implements the *Pink instantaneous* and *delayed* communication semantics.

Having described a means of modelling the algorithmic *function* of a structural design in this chapter, the following two chapters examine the partitioning and modelling of a VLSI design in terms of its physical *form*.

Chapter 5

Partitioning for Physical Form

5.1 Introduction

The impact of structural partitioning on the *functional* design of VLSI systems has been examined in previous chapters. In this chapter the impact of such a partitioning on the physical *form* of the design will be presented.

Whilst it reduces the explicit complexity of a design, partitioning also has the effect of changing the nature of the design problems that occur. This is particularly so in the physical domain in which there is a great deal of constraint imposed on the design by the VLSI medium. This chapter will begin by describing these basic constraints and the cost functions that are used to evaluate physical design quality. An examination of the various physical design partitioning strategies will then be made. Of particular interest is the practice of *structured floorplanning.* Floorplans are the geometric plans that describe the physical form that may be used to realize the structural hierarchy on the silicon surface. *Structured* floorplans utilize a number of complexity management techniques to simplify this planning task.

The central objective of the chapter is to present a broad classification of the knowledge and techniques used in custom design to create these structured floorplans. This classification will serve as a prelude to Chapter 6 in which the evaluation of the physical partitioning implied by a structural design will be described. This evaluation is based around an automatic floorplanner embodying aspects of the floorplanning knowledge described in this chapter.

5.2 Constraints in Physical Design

There are two major sources of constraint introduced into the design process by moving to a physical design. The first is the planar nature of the medium in which all components are placed and connected in several planes resulting from different process layers. The second source of constraint results from the electrical properties of the components (devices) and their interconnect (wires). The wires in particular must be regarded as "imperfect" in that a long wire will introduce significant signal transfer delays into a design due to its RC properties.

The basic aims in producing a physical implementation of a system description are to minimize the silicon area and to maximize the circuit speed. Optimizing around these features minimizes cost and maximizes performance. These objectives must be traded off against design time and cost.

In creating a design to perform a particular specified function, it is clear that the total number of devices in the implementation is not likely to vary greatly as a result of design decisions. Some optimization is possible, particularly in a component such as a PLA, but in general area/speed optimization is not strongly affected.

The greatest possible source of area/speed optimizations is in device *interconnect* as this is governed by device placement, over which the designer may exert considerable control. In the remainder of this section the constraints that are placed on interconnect by the design medium are discussed as these guide the strategies adopted to optimize design. This discussion is carried out in terms of *components* and *connections*. A component may be as simple as a single device, or may be a complex module created from a number of devices. Even though the following discussion is focused on connection, it should be noted that the optimal placement of the components is also affected by their shape: the rectangular modules must be placed in order to minimize "white space" in the floorplan.

5.2.1 Connection Length

Connection length affects optimization in two ways: speed and area. The issue of area is treated in the next subsection, here only the speed factor will be considered. The speed of the circuit is not affected by the total, or even average length of interconnect. Rather it is affected by the length of specific *critical paths*. These paths are those that

govern the maximum speed of the circuit, for instance the carry chain in an adder circuit. Such paths are highly circuit dependent and it would be a false efficiency to attempt to optimize the length of all wire runs based on the assumption that they lay in a critical path. Instead it is necessary to use tools such as timing simulators [Jouppi, 1983] to locate these paths and then utilize techniques such as manual or automatic [Hedlund, 1987] transistor sizing to reduce the critical path delays. This may in general be performed as an optimization phase in the design process, much as software is often optimized in detail once it has been broadly designed.

5.2.2 Connection Area

Total connection area is clearly a critical issue in the optimization process as it directly consumes chip area. Total connection area is affected both by the length and number of connections. The number of connections is an invariant if a design is viewed as a simple net of wires connecting devices, however in a hierarchical design this is not the case: the function and interconnect of hierarchical modules is affected by the chosen partitioning. The total connection length in a design is primarily governed by the choice of placement position of the components in the plane. It should be noted that minimizing the number of connections is an aim shared with partitioning in the functional domain (Section 2.1), whereas connection *length* has no such functional analogue.

5.2.3 Connection Planarity

Often when a number of connected components are to be placed in the plane, the issue of *planarity* arises: can the components be placed in such a way that their connections do not cross? The problem is not one of finding a strictly planar solution. This is desirable, but not strictly necessary because of the presence of a number of available layers for creating connections on the chip surface. There are commonly two metal and one polysilicon layers available in current processes. The set of layers introduces the possibility of changing layers to avoid what shall be called planarity *faults* as illustrated in Figure 5.1(a).

Planarity faults are costly because:

1. Vias are required in order to change routing layers consuming area and adding

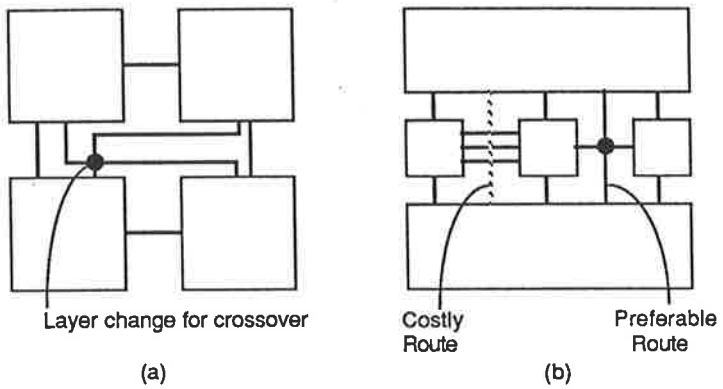

Figure 5.1: (a) Changing layers in order to resolve a planarity fault. (b) The use of a small rather than large crossover in resolving a planarity fault.

capacitance.

2. The cross-over layer may be in a less electrically desirable material than the routing run: polysilicon as compared to metal for instance.

3. The crossover layer may be better reserved for another purpose such as power or clock distribution.

Thus one of the design objectives is minimizing the number of planarity faults. In the case that they cannot be avoided, the expense incurred should be minimized. This may be achieved for instance by crossing smaller rather than larger routing runs as illustrated in Figure 5.1(b). Within hierarchical design, there are alternative methods for the resolution of planarity faults. These are discussed in Section 5.5.5.

5.3 Physical Partitioning Techniques

There are a number of different techniques that have been developed for the VLSI physical partitioning. They reflect the completely different partitioning philosophies outlined in Section 1.1 of *highly* and *functionally* partitioned designs. In this section the ramifications of these two different styles are analysed with respect to the physical layout domain. Of particular interest here is the style of physical design that corresponds to the functional partitioning approach.

5.3.1 Highly Partitioned Designs

As described in Section 1.1, highly partitioned designs are those in which there are a great number of fairly simple components. Even though the actual specification of design may be performed in a functionally partitioned manner in for instance a schematic editor, the circuit is flattened into a single group of interconnected gates before layout commences. In a *gate array* design, gates are already placed in rows on the silicon surface. Channels are made available for the interconnection of gates. Physical design is simplified to the task of selecting actual gates to implement logical gates in such a way that they may be connected (routed) together. There is no area advantage in minimizing routing as the channels have already been allocated as routing area. As long as the routing fits in the channels, and that critical timing paths have been checked, the design is regarded as complete.

Clearly the gate array design style has advantages in terms of simplicity of creating layout. Its major disadvantage is the area inefficiency. Unused gates and unused routing channel capacity absorb area. A great deal of the design is in fact taken up with routing channels because the patterns of communication are irregular and inefficient due to the limited options available in the relative positioning of layout gates.

Standard cells are a form of design that is less highly partitioned than gate array, but still has a number of advantages in design simplicity. Rather than using a fixed, predefined gate layout, standard cells are only a design style: the medium is still an unstructured silicon surface. The design task is kept simple however by allowing the user to only select from a predefined set of component standard cells, each implementing a function usually of the complexity of several complex gates. The layout of the cells is fixed, but their position in the layout is not. Typically the cells may be placed in any one of a number of rows separated again by routing channels. This system has a number of efficiency advantages over gate array designs:

1. Cells have greater individual complexity than gates, and so a higher order of function is contained within them, leading to less area being required to achieve those functions than would be the case with equivalent gate array implementations.

2. Cells may be moved between rows and within a row in order to minimize routing distance.

3. The rows may be moved in order to change the width of routing channels, thus taking advantage of the minimized route area by minimizing the channel widths.

In common with gate array design, the algorithms required to automatically perform standard cell placement are relatively simple [Hild & Piednoir, 1985]. The design style is still limited in an area efficiency sense however as the cells themselves must conform to common shape and interconnect standards for the process to work. There is no capability to shape the cell interfaces to optimize layout quality as described in the next section for custom design styles.

5.3.2 Functionally Partitioned Designs

Although simple to implement, the highly partitioned layout styles outlined in the previous section suffer from problems of performance and area inefficiency owing to excessive interconnect. One means of obtaining better layout densities is to adopt the *full custom* design style. This typically involves performing design almost completely manually, each device being individually placed and connected. The intricate nature of such a design has meant that often a single designer has performed the bulk of the work. This has led to a complexity problem with current large designs: layout must be performed by a single (or at best several) designers because of the complexity of the unstructured interactions between parts of the design. The use of so few designers obviously leads to strict limitations in the speed with which a new design may be produced.

A basic tool of the custom designer is the *floorplan:* a geometrical plan that describes one or more of the location, shape, and interconnect of the various parts of the chip design. There are a variety of styles of floorplans and methods of using them. These vary from informal abstract sketches such as that illustrated in Figure 5.2 to detailed geometrical layout. In the following section, a particular style of *structured* floorplan is describes is used in this thesis as the basis for the evaluation of the physical form of a design partitioning.

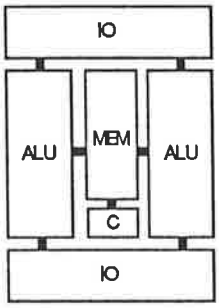

Figure 5.2: An informal floorplan.

5.4 Structured Floorplans

Many of the features of the Caltech Structured Design Style (Section 1.3) are embodied in the concept of *structured floorplanning*. Structured floorplans are abstract representations of the chip surface that delineate the general layout of the design. Such floorplans are not formally defined, however a number of research efforts have described there use. In this thesis the basic structured floorplan style is adapted from that described by Mead and Conway [Mead & Conway, 1980] and Mudge [Mudge et al., 1980b; Wardle et al., 1984]. The following sections provide an overview of structured floorplan and an examination of their structuring features.

Each module has an external interface and an internal construction in terms of the floorplan. The external interface consists of:

1. A rectangle that represents the outer boundary of the module. All of the components of the module lie within the proscribed area.
2. A set of *ports* that lie on the the rectangular boundary. This provide the only points to which connections may be made to the module geometry.

The external interface of a module is illustrated in Figure 5.3(a).

The internal implementation of a *composition* module consists of:

1. The external interface of the module that specifies the boundary and the externally connecting ports.
2. A set of submodule external interfaces. These are placed such that none overlap and there is no free space.

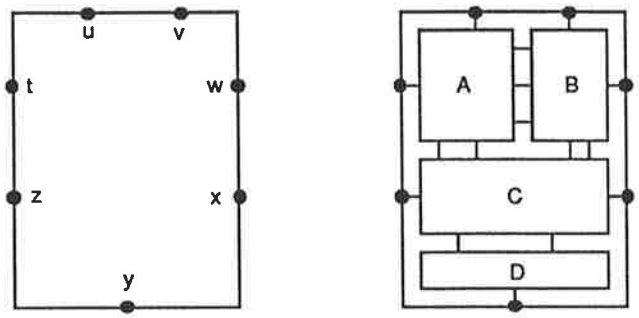

Figure 5.3: The external interface (a) and internal implementation (b) of a structured floorplan.

3. A set of connections. Connections between submodule ports, and between submodule ports and the module ports, may only occur over boundaries that abut one-another.

The internal implementation of a module is illustrated in Figure 5.3(b).

The internal implementation of a *leaf* module is a section of actual layout. Hence leaf modules are treated in floorplans as having only external interfaces: internal construction is carried out as a separate process.

There are a number of advantages to be gained from adopting a structured floorplanning design style:

1. *Functional Partitioning.* The designer is free to allocate functionality to the various elements of the partitioning. This freedom encourages the use of a common structural partitioning for function and physical form.

2. *Hierarchical Partitioning.* The designer can subdivide the design into sections that each contain a small number of modules. This is important as there is a low limit (perhaps as low as seven [Miller, 1956]) on the number of disparate objects with which the designer can successfully deal with simultaneously.

3. *Top-Down Design.* Design may proceed top-down because the predicted external interfaces of modules may be used in the composition. The modules may be implemented at the next lower level of design according to the constraints of the interfaces.

4. *Simple Abstract Interfaces.* The simple interfaces reduce the complexity of

interactions between modules. Rectangles are relatively simpler to pack than irregular polygons. Providing ports as the only access to the module contents simplifies design. Well defined interfaces for a module also facilitate design by separate groups. Interfaces simplify by filtering the amount of information supplied in a composition.

5. *Flexible Interfaces.* As designers have control over the design of the external interfaces of modules, they may be designed to minimize interconnect costs and maximize regularity.

6. *Short Interconnect.* The restriction to interconnect only being allowed across the boundaries of adjacent modules leads to shorter routing runs between modules.

7. *Simple Interconnect.* The restriction to interconnect only being allowed across the boundaries of adjacent modules leads to only simple channel routes being required between modules (Figure 5.4).

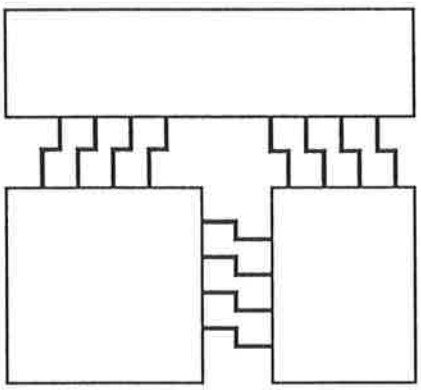

Figure 5.4: Edge to edge communication restrictions simplify construction routing requirements.

There are however a number of difficulties that arise in designing structured floorplans:

1. *Large Search Space.* The large number of degrees of freedom open to the designer in the design of a structured floorplan has the result of generating a large number of alternative design.

2. *Poorly Defined Cost Functions.* Large search spaces are best handled with simple and efficient methods for evaluating the quality of alternative designs. No such functions exist for structured floorplanning. The quality of a design is dependent on a large number of factors including the amount of interconnect, length of interconnect, packing of modules, suitability of module external interfaces to eventual implementation, and regularity.

3. *External Interface Design.* In a top-down design procedure, external interfaces are designed prior to the implementation of modules. This must be countered with an element of bottom-up design in order to ensure that the implementation is reasonably feasible.

4. *Planar Interconnect.* The designer is required to deal with the issue of producing a planar design by appropriate selection of module positions and structural changes to the partitioning.

5. *Technology Independence.* It is desirable that the floorplans being produced be technology independent so that a consistent floorplanning approach may be used across a set of technologies. Routing in particular is technology dependent, so it is necessary to relegate routing related functions to *construction* rather than *planning.*

Means for dealing with these problems are examined in the next section within the context of the floorplan design task.

5.5 Creating Structured Floorplans

The most direct method of modelling a structural design in the physical domain is to automatically create a structured floorplan for that design. As a prelude to designing a program that can provide the designer with this facility (Chapter 6), an examination of the techniques used by floorplan designers was undertaken. These techniques were derived from a variety of sources. Initially the literature provided a useful reference of simple structured floorplans. The bulk of the material however was gained from the author's involvement with a number of research projects and VLSI design groups. The primary sources and their influence are summarised below.

1. The theses of Rowson [Rowson, 1980] and Buchanan [Buchanan, 1980], together with the book by Mead and Conway provided a useful overview of floorplanning in the Caltech structured design style.

2. Case studies of VLSI chip designs that document floorplan design. Specifically a floating point processor by Digital Equipment Corporation [Mudge et al., 1980a] and a 32-bit processor by AT&T [Krambeck et al., 1982].

3. The author's involvement in the design of of a medium scale NMOS chip [Dickinson, 1982] provided useful initial first hand experience in floorplan design. A later review of the design by Carver Mead [Mead, 1982] provided a valuable critique of the design style adopted.

4. The author developed the hierarchical floorplan editor for the *Sprint* design system [Wardle et al., 1984]. This development took place in parallel to the design of a complex VLSI design [Mudge et al., 1984] providing an excellent opportunity to interact with floorplan designers and examine their methods.

5. The *Cadre* project [Ackland et al., 1985; Ackland et al., 1984] involved the coordination of multiple expert systems in order to carry out complete structural to physical design translations. A major part of the knowledge acquisition carried out for this project involved experienced designers performing system design under carefully controlled and monitored conditions. Although the primary objective of the exercise was to gain information about the processes used to direct the overall design process, the author was able to make additional use of the exercise to record and analyse floorplanning design decisions.

6. The VLSI design group at Symbolics Inc. are involved in the design of large VLSI processors. The author was able to hold discussions with experienced layout designers on the techniques they used in the design of large systems including the MIT *Scheme* chip [Shrobe, 1985].

The intent here is to not simply list the techniques obtained from the above sources, but rather to classify them into a small number of distinct classes of designer knowledge. This classification will serve as the basis for the design of the floorplanner described in the next chapter.

5.5.1 Design Procedures

As with any design task, structured floorplanning involves a class of knowledge associated with reasoning about the design task itself, the "meta-knowledge" [Lenat et al., 1983] required to apply the "domain knowledge" to the design.

As noted in the previous section, the large number of degrees in freedom involved in a structured floorplan design lead to a large search space. Much of floorplanning meta-knowledge consists of techniques for traversing this search space in a manner designed to find a good solution in the minimum time. Much of this knowledge consists of techniques that are applicable to design tasks in general and as such are not floorplanning domain specific. These will be introduced in the next chapter. However some aspects of the meta-knowledge required for floorplanning are specific to that domain and they are described here.

In a design task such as floorplanning that involves the assembly of a number of components (the modules) the sequential nature of the assembly task introduces several problems. Every time a new component is added into the ongoing design it affects those components already in place: these effects are usually quantifyable. However they also affect the in a far less easily quantifyable way the placement of later components. This effect in floorplan construction is illustrated in Figure 5.5. The most expensive means of avoiding this is to make the mistake, and then backtrack, undoing design work, and attempting the design again without error. This is inefficient as it requires destroying earlier work. A more efficient technique is to use *planning* and careful *sequencing* to minimize errors in the forward design path.

5.5.1.1 Planning

Designers create *plans* that will be used in guiding the floorplanning task. This is done at an early stage of the design. The most common form of plan is that formed by recognizing a regular structure in the input description. Once such a structure has been noted, constraints can be generated that ensure that the layout will occur in a particular regular fashion. The additional constraints will serve to remove a large number of possible configurations of the modules concerned, thus reducing the search space and making design more efficient. In addition an abstraction may be formed in which the clustered modules are treated as a single larger entity further increasing the efficiency of the design process. This is illustrated in Figure 5.6.

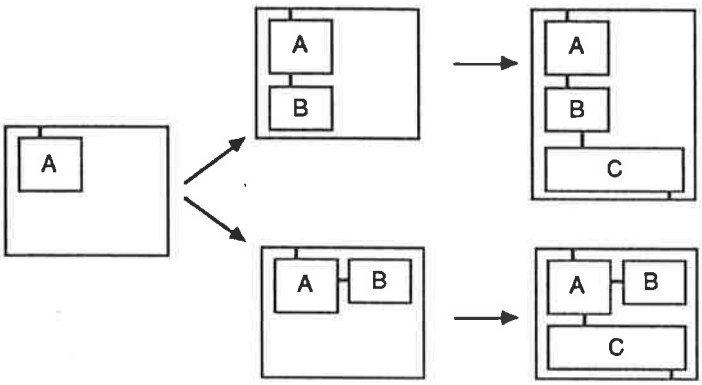

Figure 5.5: In the upper sequence the position selected for module B forces C into a position that increases the area of blank space in the floorplan. In the lower sequence the position of module B does not preclude a satisfactory position for C.

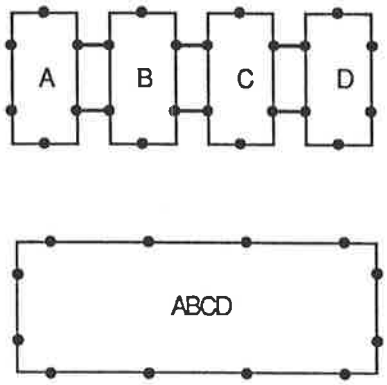

Figure 5.6: Combining a regular group of modules into an abstraction simplifies the search space.

5.5.1.2 Sequencing

The order in which components are selected to be inserted into the design can be critical in deciding the success or otherwise of a particular design exercise. As the design proceeds it becomes more constrained and there is more difficulty in meeting the connection requirements of modules as they are added. Designers minimize this problem by using a number of criteria in selecting which module shall be added next into the design:

1. Prefer a module that has a large number of connections to modules already in place. The later this module is inserted, the more difficult it will be to find it a position.

2. Prefer a large module because as the design progresses available large vacant spaces become more difficult to find.

3. Prefer a module that is part of any plan (for instance a regular structure being assembled) as it will have a relatively unambiguous target position.

4. Prefer a module that has a minimum number of possible positions at which it may be inserted into the design due to connection adjacency constraints. As the design progresses such positions will become even less available.

These issue of techniques for efficient guidance of the design procedure will be discussed further in the next chapter.

5.5.2 Designing with Rectangles

The most basic operations performed by floorplan designers are thosed concerned with the direct manipulation of the interconnected rectangles that are used to represent module instances. In order to generate floorplans of the structured style described in the previous sections, the rectangles must be placed according to a number of constraints. These constraints are based round the size, shape and connections of each rectangle. In placing another rectangle into the growing design, the designer must consider a number of factors:

1. What spaces are available that will accept a rectangle of this size and shape without significantly adding to the overall area of the floorplan?

2. What spaces are available that allow the new rectangle to be adjacent to all or most of the rectangles it must connect to?

3. What spaces are available that fulfill both the above criteria?

4. What spaces might be used without significantly hindering the incorporation of as yet unplaced rectangles?

The mechanism that designers use to perform the above analysis of the design are complex. They appear to rely on an innate ability to manipulate geometrical entities such as rectangles in complex systems of constraints. One common observation is that the designer uses visual aids such as a paper and pencil to record the current state of the design. This aids in the interpretation of the changing constraints on the rectangular components as the design progresses, and reduces the need of the designer to maintain and reason with the complex interplay of geometric constraints in short term memory.

In the initial stages the rectangles are represented as simple abstractions without size or shape, but still in general interacting with each other as rectangles. Positions are recorded as relative to other rectangles rather than as absolute coordinates. As more components are introduced into the design, the rectangles are attributed properties such as size, shape and orientation and eventually absolute position.

Although the specific methods used by designers to achieve these functions are obscure and difficult to represent, it is possible to imitate such behaviour once it has been isolated as a particular class of knowledge. The processes developed for doing so will be described in the next chapter.

5.5.3 Hierarchical Interactions

In a pure top down design, component interfaces are designed based on composition criteria: the components must interact in an optimal fashion. The implementation of the component as specified by the interface is treated as a separate design issue. This style of design is viable in for instance in software design. A procedural interface (name and parameters) may be specified prior to implementation, and will not in general overly constrain the possible implementations. The success of pure top down techniques is then dependent on the quality of the abstraction of the implementation that may be provided by the interface.

The application of pure top down design procedures to structured floorplanning suffers from strong interactions between module interfaces (size, shape and port positions) and implementations (internal layout). If a design is created under the assumption that module interfaces may be optimized for the composition level irrespective of component module implementation issues, the implementation of the components may become so difficult that the composition level gains are lost. There

are several alternate approaches to pure top down design that may be applicable to floorplanning:

1. *Bottom up design* involves constructing the implementations prior to the interfaces. This ensures that the implementation is efficient, but introduces the opposite problem of perhaps producing interfaces that are inappropriate at the composition level as they require large areas of routing. This is to be strongly avoided as routing presents one of the major costs structured floorplanning is intended to reduce.

2. *Specifying standard interfaces* ensures that there are neither major connection nor implementation problems. It is not then possible however to make optimizations of either connections or implementations that may be possible with less constrained interfaces. The resulting design suffers from the same inefficiencies as standard cells.

3. *Modified top down design* involves considering the implications of interface design decisions on possible module implementations during interface design at the composition level. This involves a complex trade-off between what is optimal at the composition level and what is optimal at the implementation level.

Experienced structured floorplan designers typically adopt the procedure outlined in (3) above. Whilst a floorplan is being created, as well as considering the composition issues of interconnect, size and shape optimization, the designer uses past design experience to predict the effect of interface design constraints on the eventual implementation of each module. This procedure is illustrated in Figure 5.7.

The knowledge used by designer to perform modified top down design has several basic features:

1. It is *expert* and *domain specific*: only used by experience practitioners in floorplanning.

2. It is *complex*: for each module there may be a number of alternative implementation styles, and the effect of interface constraints on the viability of each must be taken into consideration.

3. It is *imprecise*: the designer uses the knowledge of implementations only an approximate guide during interface design. This must be the case as the im-

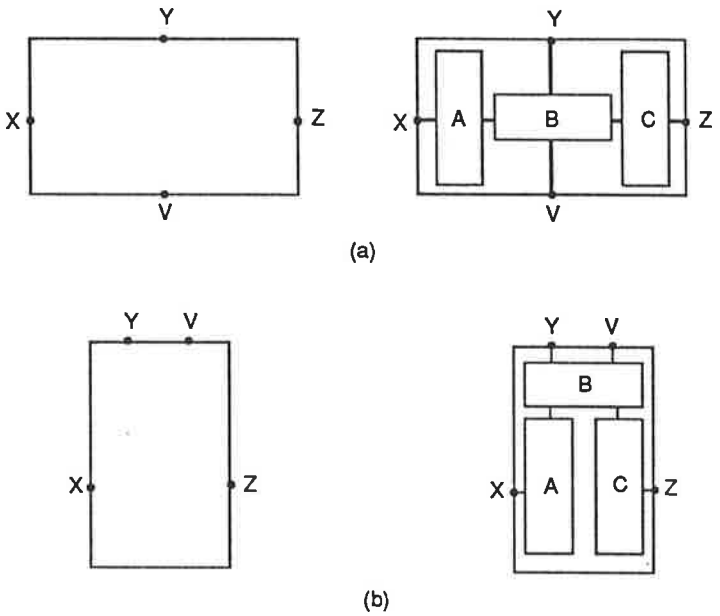

Figure 5.7: The interface in (a) results in a floorplan implementation that contains blank space. The interface in (b) results in a more efficient implementation.

plementation has not as yet been created, and hence only imprecise knowledge of its construction is available.

5.5.4 Structuring Techniques

There are two primary techniques used to impose structure on floorplan designs: regular instance creation and regular interconnection.

When a number of instances of the same module definition are connected to one another they resulting layout is often that of a linear array as illustrated in Figure 5.8. Linear array configurations have a two primary advantages: there only need be one layout of the instance which may then be repeatedly used, and the instances communicate by direct abutment saving routing area. Candidates for linear array structures can be recognized in the structural description and then careful interface design is used by the designer to ensure that the set may be implemented as an array structure.

Two dimensional array structures also occur, however these are commonly implemented using two linear array structures at different hierarchical levels as shown in

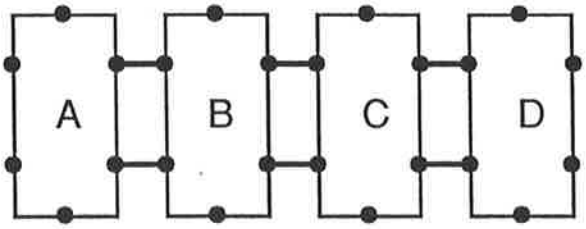

Figure 5.8: A linear array constructed from identical module instances.

Figure 5.9.

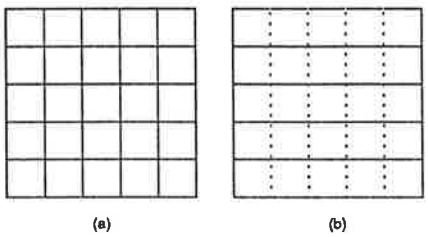

Figure 5.9: A two-dimensional module array (a) may be structured as a two one-dimensional arrays (b).

A common pattern of interconnect found in structural descriptions is that illustrated in Figure 5.10(a) in which the disparate module instances are connected via a common bus. It is possible to design the module interfaces in such a situation so as to ensure a regular pattern of interconnection between the instances, forming a data path structure. The presence of such buses is often a central constraining feature in a floorplan. Such regular interconnect have the potential of saving large areas of routing compared to an implementation in which the bus exist as a separate floorplan entity as illustrated in Figure 5.10(b). Designing at this level does however require interaction with implementation knowledge as the buses must be run through modules, possibly affecting their implementations.

5.5.5 Maintaining Planarity

Planarity faults occur when there are no positions remaining in a design into which a module may be placed such that it will be adjacent to all modules to which it must connect. One method of dealing with this situation is the *backtrack* through

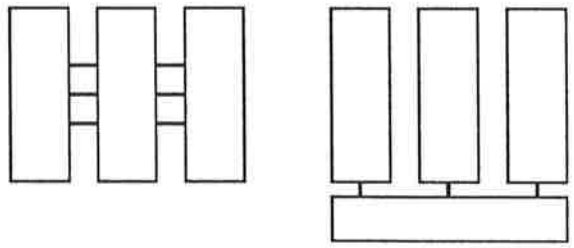

Figure 5.10: (a) Modules connected by an inserted bus. (b) Modules connected by an external bus module.

the design, investigating alternative positions for modules that may not lead to the planarity fault. This approach has a number of difficulties however as it is unclear how far to backtrack, and how to proceed again in order that neither the current fault nor a new fault occur. Additionally the backtracking process is inefficient as it involves discarding existing design work. The issue of backtracking in general is covered in the next chapter.

Although by appropriate selection of module positions the number of planarity faults that occur may be minimized, many partitionings are inherently non-planar and require further action in order to produce a planar embedding. Designers typically proceed by:

1. Temporarily repeatedly discarding small connections until a planar position may be found for the module.
2. Continuing with the design without the small connections.
3. When all modules are placed, locating shortest paths through the design along which the discarded connections may be inserted.
4. The routes may then be implemented by a mixture of two methods:
   (a) Inserting "route modules" that relocate the crossing over of interconnect into a leaf module, thus preserving the planarity of the composition module (Figure 5.11(a)).
   (b) Where appropriate, merging the connection into an existing module along the path. The construction and orientation of some modules is such that

an insertion of this nature is not disruptive to their implementation (Figure 5.11(b)).

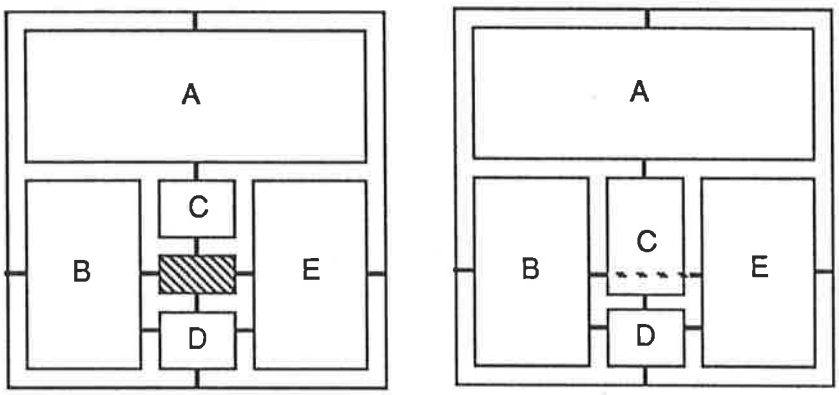

Figure 5.11: Use of a route module (a) and an insertion block (b) to resolve a planarity fault.

5.5.6 Technology Independence

Floorplanning is largely technology dependent. The rectangular module abstractions may have their sizes and position may be given in dimensionless quantities as they really only imply relative sizes and shapes in a top-down design style. Actual sizes and positions need only be considered during the constructions phase.

As noted earlier however, the primary aspect of technology dependence that impinges on floorplanning results from routing related issues. In particular, the nature and number of the layout levels available for routing affect the connection of the modules. The basic objectives of planarity and abutting connection remain applicable however regardless of the routing layers available. Connection lengths affect performance, and vias use area, so locality of connection remains important.

A more complex issue is raised by the alternatives that arise when a designer must planarize a design. Additional routing layers make it simpler to maintain a planar design, and affect the use of the route and insertion module techniques described in the previous subsection.

In order to preserve the technology independence of the floorplanning task, it is sufficient to define the floorplan as a placement of all of the modules such that

the majority of connections may be made edge-to-edge. Any other connections will have abstract paths specified for their routing through the floorplan, but the actual method of implementing the path shall be transferred to the bottom-up construction process which contains technology dependent routing information by definition as it is required in order to connect the modules. For the objective of this research into the evaluation of the structural design, these abstract routing paths are sufficient in themselves to provide feedback to the designer on floorplan planarity.

5.6 Evaluating Form with Structured Floorplans

The connection between floorplanning and the evaluation of physical form is direct: the floorplan is simply an abstract representation of the physical form, and as such provides an excellent basis for the qualitative and quantitative evaluation of the partitioning. Figure 5.12 illustrates how feedback from the floorplan may be used in order to optimize a structural design for layout.

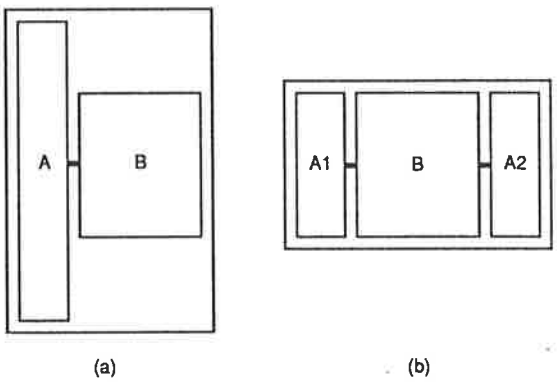

Figure 5.12: The structure proposed in (a) leads to an inefficient floorplan implementation. Dividing the module A into two modules results in a structure that can be more successfully mapped into a floorplan (b).

The form of structured floorplanning outlined in this chapter is particularly appropriate to the floorplanning task because it may be performed in a top down manner. This is well matched to the *structural* design philosophy outlined in Chapter 2. As a design is being developed according to a particular structural methodology (data flow, for example) the physical form of the structure may be examined in parallel with the functional performance.

Current research in the area of the evaluation of physical layout is in general based on statistical analysis of existing designs applied to the design in progress. In the work of Anceau for instance, the statistical properties of past designs are used to predict the size and shape of the modules being designed [Anceau & Reis, 1982]. This approach appears to have the potential disadvantages that it is technology dependent and only provides a very abstract evaluation that is not valid of the current design style varies greatly from previous techniques. Sparta [Resnik, 1986] is basically a spread sheet program for evaluating partitionings. The quantitative predictions are based on algorithms that are intended to fulfill the same purpose as the statistical data of Anceau's work.

The major problem with the use of structured floorplans as an aid to evaluation is the difficulties involved in automating the design task. This problem is addressed in the next chapter.

5.7 Construction with Structured Floorplans

Although not the specific objective of this research, structured floorplans may of course also be used as the actual plans that guide the assembly of the modules into a layout in the construction phase.

The external interfaces that are created for leaf modules in the floorplanning process may be used as an "environment" for the implementation of active circuitry by manual of automatic [Kollaritsch & Weste, 1984; Kollaritsch & Weste, 1985; Watson, 1987] means.

An appropriate method for the construction of composition modules is based on creating a "slicing" structure for the floorplan and using this to guide the assembly process [Wardle et al., 1984; Watson, 1985; Van Ginneken & Otten, 1984]. Each module floorplan is sliced as illustrated in Figure 5.13(a). The result is a slicing hierarchy (Figure 5.13(b)) in which each node represents a binary composition which may be achieved with a simple river or channel route. The slicing hierarchy is traversed upwards from the leaves, resulting in a complete layout as illustrated in Figure 5.13(c). Power, ground and possibly clock signals may be ignored in the floorplan and simply automatically routed to systematically labeled ports on the modules. The routing is simplified by the use of regular routing patterns on specifically allocated layers for

the interconnect.

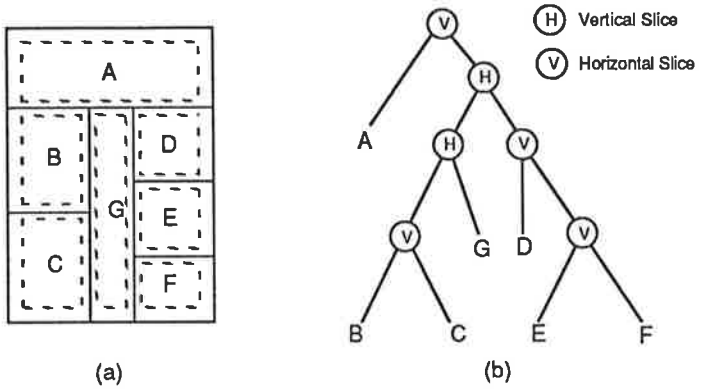

Figure 5.13: A floorplan may be sliced as shown in (a) to produce a hierarchical slicing representation (b) that may be used as a guide to chip assembly.

5.8 Summary

The implementation of a structural design as a physical form involves additional constraint being placed on the design. The components must be placed and interconnected in a plane. Total area and interconnect must be minimized in order that adequate performance and yield be achieved.

In gate array and standard cell styles used to implement highly partitioned designs, interconnect dominates and restricts the achievable level of integration. In the custom design styles used to implement functionally partitioned designs the additional degrees of freedom in design allow for a reduction in interconnect area at the price of additional complexity of design.

Structured floorplans incorporate the complexity management techniques of hierarchy, regularity, and abstraction to reduce the complexity of custom layout. Thus they present a useful technique for the layout of the functionally partitioned structural designs discussed in earlier chapters.

The creation of structured floorplans is complicated by:

1. Many degrees of freedom leading to a large search space in design.
2. Poorly defined cost functions.

Designers manage the complexity of the floorplanning task by applying diverse heuristics. These allow the designer to reduce the search space and optimize the global design based on local observations. The knowledge used by designers may be categorised into several classes:

1. *Meta knowledge* that defines how plans are created and the design process proceeds.
2. *Geometric knowledge* that assists reasoning with interconnected rectangular objects in the plane.
3. *Implementation knowledge* that uses past experience to guide the design of module interfaces such that a reasonable implementation of the modules will eventuate.

Structured floorplans provide an excellent basis for the evaluation of the embedding of the structural design into the plane. They can be produced in a top-down manner that complements the structural design style introduced in Chapter 2. In addition they may be used as a guide to the final assembly of the chip layout in the construction phase of the design.

The problem of automating the top-down design of structured floorplans in order to assist in the evaluation task is described in the next chapter.

Chapter 6

Modelling Form: Floorplanning

6.1 Introduction

The structured floorplanning methodology described in the previous chapter offers the possibility of using the same structural design across both function and form domains. In order to facilitate this it is useful if the designer may model the behaviour of the *form* of the structure in addition to modelling the *function* as described in Chapter 4. What then does *modelling* form entail? Modelling function involved observing the behaviour of a specified function: once the structural design had been created, the designer specified the function for each module explicitly. The direct analogy with modelling form would be to insist that the designer specify the form by designing a floorplan. The floorplan is however the desired *result*: once formed it is in itself an evaluation of the partitioning. If the designer is to be assisted then it is by *creating* the floorplan automatically.

In this chapter an examination of existing automatic floorplanning techniques is made. In the light of their respective limitations, and the structured floorplanning knowledge described in the previous chapter, the structure of an automatic knowledge-based floorplanner is described. This is based on several knowledge representations designed specifically to match the requirements of the floorplanning domain.

An implementation of the floorplanner is described that provides a sound basis for the continued expansion of its base of floorplanning knowledge. Although the current implementation contains only vestigial knowledge in comparison to an experienced

human floorplanner, it's results provide sufficient information to enable a designer to rapidly evaluate the quality of a structural design in its mapping to form. It assists this evaluation by not only creating a floorplan, but also providing feedback on what layout implementations of modules in the floorplan are most appropriate, and what use it was able to make of structuring methods in creating the floorplan.

6.2 Approaches to Automatic Floorplanning

The term "floorplanning" is used in a number of contexts in computer aided design research and there is hence considerable ambiguity in the term. In this thesis there will be a distinction made between "placement" and "floorplanning" to avoid confusion.

Placement is the task of placing a set of *fixed sized* connected items in an arrangement that minimizes some cost function, usually related to total connection length/area. There are two basic sub-classifications of placement. The first is the gate array placement problem in which the items are homogeneous and interchangeable. The second is the standard cell placement problem in which the items are non-homogeneous rectangular blocks.

Both classes of placement problem are distinguished by the large number of items that are to be placed in the design, and the fixed specification of each of the items. The problem set size ranges up to about three thousand items for large gate arrays [Sechen & Sangiovanni-Vincentelli, 1985].

There are a number of elegant and efficient algorithms that have been successfully developed for, or adopted to, the placement task. The best known of these include *simulated annealing* [Jepsen & Gelatt, 1983; Sechen & Sangiovanni-Vincentelli, 1985], min-cut [Lauther, 1979], force directed [Quin, 1975] and Kernighan-Lin [Kernighan & Lin, 1970]. The diversity of these techniques is indicative of the simple and general way in which the placement problem may be stated owing to the simplicity of the placed items and the cost function.

After placement has been carried out on a design, a *routing* algorithm is used to connect up the items in the placement.

Floorplanning may be distinguished from placement primarily by the complexity of the items being placed. Floorplanning may degenerate to placement in the case

where the items being planned with are fixed in size. In the more general case however floorplanning is carried out in a top-down style in which only estimates and predictions about the nature of the items exists. As part of the floorplanning process the sizes, shapes and connection points of the items must be selected. Floorplanning is generally hierarchical, with each floorplan containing only a small number of items, and each item in general being comprised of a further floorplan (typically for the sake of simplicity of design the items are constrained to be rectangles). In this context the cost function becomes complex as what appears to be optimal according to a simple connection length criteria at one level may be found to be inappropriate at another level higher or lower in the floorplan hierarchy. Rather than evaluating design according to simple cost functions, human floorplan designers tend to use what is regarded as good design practice (such as structured floorplanning as described in the previous chapter) in order to minimize global area.

In the remainder of this section the an examination shall be made of the various approaches to automatic *floorplanning*, not simple placement. Unlike the placement problem there has been relatively little work in the are of automatic floorplanning, the more common approach being interactive manual floorplanners such as that written by the author for the *Sprint* system [Wardle et al., 1984]. There has however been considerable research in automatic floorplanning in the field of building design in which it is more commonly known as *space-planning.*

The space-planning problem may be defined as:

> ...a problem which has the goal of the placement of a set of subspaces in a particular larger space, subject to both a class of location requirements and and to the constraint that the subspaces must entirely fill the larger space.
>
> [Grason, 1970]

The basic constraints and location requirements may include:

1. The spaces must all be rectangular.
2. Some spaces will be *contiguous*, that is they share a wall.
3. Some spaces *communicate*, that is there is a door between them.
4. There are often physical size constraints on the wall lengths.

5. There are often constraints on the positioning of doors.

There are a number of similarities with structured VLSI floorplanning. Firstly rooms, like modules, must be contiguous (adjacent) in order to communicate (share a door or a route path respectively). Secondly the designer must take into account the internal structure of rooms when considering the placement of doors, as must the VLSI designer consider the internal structure of modules when allocating port positions.

The space-planning problem tends to differ from the floorplanning problem in that the global objective is to *fill* the given space, whereas the floorplanning global objective is to minimize the area of the space. As noted previously however, in structured floorplanning the objective is not always the local minimization of space as this does not necessarily relate to a global minimum.

In this section a number of automatic floorplanning techniques will be surveyed. Each has particular strengths and weaknesses with regard to its ability to create floorplans. The ambiguity in the definition of the floorplanning task makes it difficult to compare programs that implement the techniques in an absolute sense. Our interest here however is the creation of structured floorplans of the style described in the previous section, hence the ability of each approach to create such floorplans will be examined. This is not a general critique of the programs: more an analysis of the suitability of their basic methods to creating structured floorplans.

6.2..1 Planar Graph Techniques

An general method of representation for a structural design is that of a simple linear graph in which the nodes represent modules and arcs represent the connections between them. The arcs my in fact be weighted in order to represent the relative widths of the communication paths. This form of representation is often referred to as an *adjacency graph* and is illustrated in Figure 6.1. Adjacency constraints to the perimeter of the floorplan may be represented by nodes representing the four external sides of the design as shown in Figure 6.1.

Given a graph representation it may be possible to create a *planar embedding* of the graph such that none of the arcs cross. If the graph is non-planar it is possible to planarize it by suitable addition of nodes at the crossover points as illustrated in Figure 6.2. Clearly there may be a number of embeddings of any such planar graph.

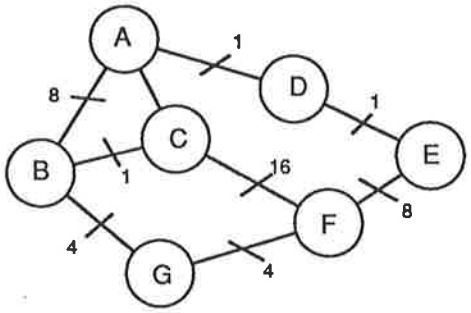

Figure 6.1: An example of an adjacency graph.

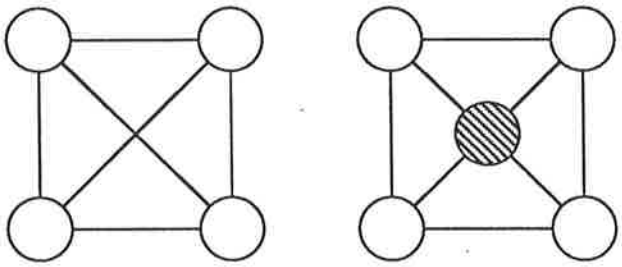

Figure 6.2: A planar embedding may be created by the insertion of nodes at crossovers.

The planar graph representation is useful in floorplanning because of the properties of its *dual graph*. The dual is formed by translating each face in the graph to a node, and each node to a face as illustrated in Figure 6.3. Given a dual graph that is appropriately constrained (a rectangular dual graph) it may be translated into a floorplan as shown in Figure 6.3. The constraints required on the graph are quite restrictive [Grason, 1970] however there may be any number of floorplans achievable from a particular dual graph.

The floorplan that result from a rectangular dual graph may be used as a basis for the creation of a set of linear constraints that define the floorplan topology. These may be combined with other constraints on module size and interconnect width requirements to produce a set of linear equations that when solved supply numerical values for the edge lengths of the floorplan. This process is well described in [Heller et al., 1982; Maling et al., 1982].

The use of planar graph techniques was first taken up in space-planning research

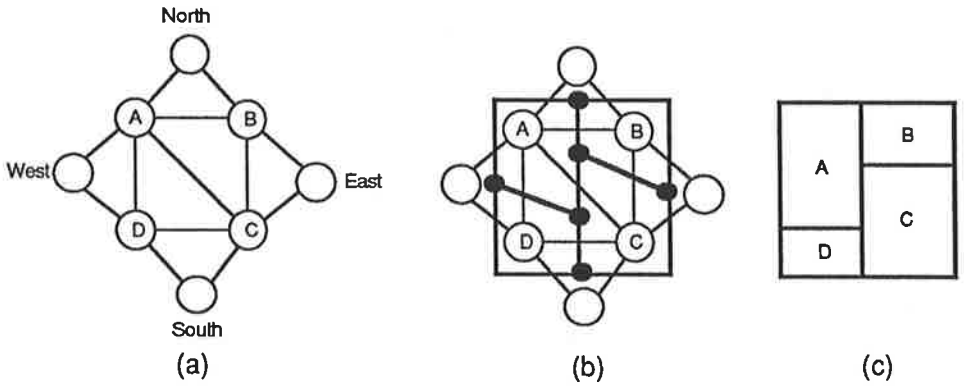

Figure 6.3: (a) A planar embedding of a graph. (b) A rectangular dual of the planar embedding. (c) The resulting floorplan.

Number of rectangles	1	2	3	4	5	6	7	8
Number of floorplans	1	1	2	7	23	116	683	4866

Table 6.1: Growth of complexity in floorplan enumeration.

[Grason, 1970; Mitchell et al., 1976; Earl, 1980; Baybars & Eastman, 1980; Baybars, 1982; Gilleard, 1978; Korf, 1977]. The methods were later adapted to integrated circuit floorplanning by Heller [Heller et al., 1982; Maling et al., 1982].

These methods rely in general in enumerating a large number (if not all) of the floorplans that may be derived from and initial adjacency graph. This is done because there are few hueristics applied in the process: rather each floorplan is proposed and the constraints solved in order to find the optimal of near optimal configuration. Given that there may be a number of embeddings of the adjacency graph, a number of duals resulting from each embedding, and a number of rectangular duals resulting from each dual, the possibility of a combinatorial explosion is present. The actual growth of complexity is problem dependent, but Mitchell provides the results shown in Table 6.1 and estimates that there are over 250, 000 possibilities for a 9 component system [Mitchell et al., 1976]. Note that each of these must be solved for its constraints and evaluated in terms of some cost function.

The computational complexity is also increased because of the difficulty of generating all of the rectangular duals of a planar graph. The algorithms developed for this

task have been exponential in growth except those developed for certain constrained cases[Kozminski & Kinnen, 1984].

In order to prune the search space of possible floorplan configurations, most graph based system rely on an interactive interface with which the designer specifies various initial configurations. In Heller's system for instance [Maling et al., 1982] the adjacency graph is planarized and embedded manually by the designer. The system then has a greatly reduced number of possible floorplans to evaluate. The interactive nature of the tool is also important as a method of applying designer expertise to the problem:

> ...although the system accelerates the process of developing and a compact and well structured layout significantly, the design process is not at all automatic. Engineering choice and judgement are required to create a hierarchy of planar, original graphs that represent the design and have RDG [Rectangular Dual Graph] solutions. An understanding of chip design is essential because the POGs [Planar Original Graphs] must include reasonable wiring macros [modules] for pads, for busing and for rearrangement of wire order at functional macro interfaces.
>
> [Maling et al., 1982]

An analysis of the success of graph based space-planning techniques by Henrion [Henrion, 1978] resulted in similar conclusions. Analysis of the performance of a number of systems suggested that they lacked utility by not containing strong floorplanning domain knowledge . The algorithms used simply did not take into account issues dealt with in planning by human designers. It was suggested that the programs should be structured in such a way that the knowledge required by designers be represented explicitly in them, and that this should be readily examinable and alterable by the designers themselves to facilitate improved program performance.

6.2..2 Slicing Techniques

In Section 5.7 *slicing* was introduced as a tool for assembly. There are several examples of systems in which it has been used as a means of constructing floorplans [Szepieniec & Otten, 1980; LaPotin & Director, 1985]. The general procedure involves starting with an adjacency graph representation of the design. This is then

bipartitioned using one of a number of algorithms such as min-cut [Kernighan & Lin, 1970], Fidducia-Mattheyes [Fidducia & Mattheyses, 1982] or Schoenberg [Otten, 1982]. The criteria for bipartitioning is usually based on minimizing the connections between the partitions and balancing the areas between the two. The bipartitioning proceeds recursively and the results are used to build a *slicing tree* as was illustrated in Figure 5.13.

Once the slicing tree has been completed, the rectangular blocks that form the actual floorplan must be created and oriented. Each of these forms a leaf in the slicing tree. In *Mason* [LaPotin & Director, 1985] the designer suggests an aspect ratio for the overall floorplan, and this is recursively subdivided and passed to child slices until the leaves are reached. The modules are then oriented so as to best achieve the required aspect ratios of each slice.

Slicing methods have the advantage of being fast: the partitioning step is rapid and the clean structure of the slicing tree makes assembly a simple task. There are a number of disadvantages however:

1. Only the area and connectivity of the modules as a basis for the partitioning and hence slicing structure. Shape is not considered until the slicing is completed.

2. The slicing structure places a rigid sequencing constraint of the floorplanning process: once a module has been allocated into a slice, it must always lie in that slice or one of its child slices. This means that it is not possible to adopt a strategy of *deferred commitment* in which a decision on placement is not made until there is sufficient constraint information available.

3. It is clear that the approach is a so called *weak method* [Newell, 1969] in which there is little domain specific knowledge. This reduces the ability to apply structuring techniques to floorplan designs, thus working towards global rather than local minima in a hierarchical design. Additionally it is difficult to provide the designer with specific feedback on changes to the partitioning that might lead to an improved design.

4. The method has no knowledge of module implementation. Even though it is possible to work with flexible modules defined by various constraints on size and port positions, these must be specified in detail by the designer.

6.2..3 Knowledge Based Techniques

A number of automated VLSI floorplanning programs may be classified as *knowledge based.* The methods used in these programs involve specifically mimicking some of the methods used by experienced human floorplan designers. Clearly in order to operate in this manner they must in some way internally represent this designer knowledge such that it may be applied to a range of floorplanning problems.

The *IF* program [Nixon, 1984] is based on the observation that designers often use "standard floorplans" or "idioms" in order to realize particular structural entities as layout. The program first performs a "clustering" phase in which regularly interconnected modules are grouped into single entities. A pattern recognition phase is then used to classify particular idioms. For each class of idioms the exists an "expert floorplanner" that contains knowledge on how to lay out a particular idiom.

The idiomatic strategy has a number of advantages. Firstly, it is time efficient: the various algorithms used for clustering, classification and placement are not expensive. Secondly, each "expert" only needs to have very specific knowledge on how to lay out a single floorplan structure, hence this may be done in a compact and structured fashion.

The strategy does have a number of weaknesses however:

1. The clustering stage is critical: if this fails to separate various idioms in a single floorplan, they will not be later recognized for layout.

2. There are only a small number of idioms, and adding additional idioms does not seem to be a simple task. If a floorplan does not contain any known idioms, a poor layout will most likely result.

3. There seems to be difficulty in constraining the form of the floorplan that is laid out: the external shape and port positions are not constrainable and floorplans are created purely on the basis of what is locally optimal for the idiom. Fourthly, no allowance is made for the implementation of modules in the floorplan.

The *Class* program [Birmingham et al., 1985] is a design *assistant* that coordinates the operation of a placer [LaPotin & Director, 1985], a router [Joobani & Siewiorek, 1985] and a cell layout generator [Kimm et al., 1984]. It appears to be based on an idiomatic model of floorplanning with an additional ability to manage design

constraints specified interactively by the user.

The automatic floorplaner described by Jabri [Jabri & Skellern, 1986] is an extension of the work of Heller [Heller et al., 1982; Maling et al., 1982]. The system uses the graph algorithms described by Heller to enumerate all possible rectangular floorplans that may be generated from a graph representing the functional connectivity of the design. These are then exhaustively examined by a rule base in order to select a particular optimal floorplan. This scheme progresses further along the line of completely automating floorplan design than the system described by Heller, but is accompanied by a number of disadvantages:

1. In the initial phase, a single planar embedding of the graph is chosen without consideration of rectangular shapes and sizes. This occurs because the embedding is performed on a simple graph which describes only connectivity. This approach was considered in the early design of *Floyd* [Dickinson, 1985] but later abandoned in favour of the rectangular graph approach described in Section 6.5.

2. In the floorplan enumeration phase, all rectangular implementations of the planar embedding are generated. This leads to problems of a combinatorial growth in the number of alternative floorplans as described in Section 6.2..1.

3. The rule based system is required to examine each of the potentially large numbers of floorplans in order to select an optimal case. This methodology does not model the behaviour of expert floorplan designers. They apply expert knowledge *during the development* of a floorplan, not as a post-design filtering stage. This mismatch between floorplanner and designer methodologies suggests that incorporating designer knowledge into the floorplanner would be quite difficult: procedural expert knowledge have to be converted to judgemental knowledge.

The *Flute* floorplanner [Watanabe & Ackland, 1986; Watanabe & Ackland, 1987] performs the floorplanning task by applying expert knowledge to design *procedure*, not evaluation. This approach results in a modelling designer behaviour, thus easing the problem of knowledge acquisition. In Flute modules are placed on a cartesian grid, their position being selected by a set of rule bases invoked in step-by-step process. Module interconnection is used as the basic criteria for placement, but this is influenced by the size and shape of the module rectangles. Although it models designer behaviour more closely than the previously described floorplanner by Jabri, this approach appears to have several limitations:

1. The use of a grid for the placement of modules implies a degree of arbitrary constraint on the placement: relative module placements on the grid implied constraints that were not meaningful in the context of the design. This is a similar problem to that found in virtual grid compaction [Weste & Ackland, 1981] where components placed common grid lines are arbitrarily constrained together in one dimension.

2. The rule bases do not appear to contain explicit knowledge on the *implementations* of modules. Design is carried out in the context of a single level of hierarchy, and the feasibility of implementing the next lower level of modules floorplans is not regarded as significant.

3. The basic control of the system is algorithmic: rule bases and procedures are invoked according to a set procedure. This may discourage integration of control knowledge with the other rule bases, removing access to the detailed state of the design that may be relevant to the control flow. In addition, algorithmic representations are most appropriate for well structured, well understood and complete domains (Section 6.3.1.3). There is little evidence to suggest that the knowledge required in the control of the floorplanning process meets any of these criteria.

6.2.1 Discussion

The *algorithmic* methods of planar duals and slicing have the following advantages as approaches to floorplanning:

1. Floorplan representations that allow for the application of well understood results from graph theory and general algorithmic techniques such as linear programming and Kernighan-Lin.

2. Well defined, though often high, time complexities.

These advantages are countered by a number of disadvantages:

1. Optimization occurs based on overly simple cost functions that may not reflect global design cost.

2. Lack of strong domain knowledge hinders algorithms from performing "pruning" of the design search space: large numbers of floorplan possibilities are generated and must be filtered by further evaluation techniques.

3. An inability to recognize structures that are most efficiently laid out as structured floorplans.

4. An inability to explain their actions and thus assist the designer improve the design.

5. Have no knowledge of how implementations of modules will be created, and thus may place poor constraints on modules unless these have been specifically stated by the designer.

The knowledge based approaches described have had a number of positive aspects:

1. They can recognize important structuring entities such as buses and create structured floorplans based on their presence in the design.

2. They *may* be able to provide the designer with more information as to the success or failure of the partitioning owing to its structuring potential. No such facility has been reported however.

There are however a number of problems with these programs stemming from both their incomplete floorplanning domain knowledge and their structuring not meeting the design criteria considered desirable in artificial intelligence systems:

1. Although they can incorporate structuring techniques into designs, these systems do not have any knowledge of module implementations and as such are poor at top-down design.

2. Typically only a narrow portion of designer expertise is used, that dealing with idioms and structuring techniques. The meta-knowledge used by designers to control the design's progress is not explicitly represented.

3. The knowledge representations are *ad hoc*: distributed into various data structures and algorithms in a non-modular fashion. This makes it difficult to understand the program, difficult for explanations on design decisions to be generated, and difficult for the knowledge base to be expanded by floorplan design experts. This is a well known problem: Feigenbaum [Feigenbaum, 1977a]

has suggested that these problems may only be avoided by using knowledge representations that are compatible with the human expert.

From the outlines of the various approaches given here, and the description of structured floorplanning given in the previous chapter, it is clear that the automatic generation of such floorplans requires considerable domain knowledge. This knowledge however has many aspects, requires explicit and well designed representations, and must include design task meta-knowledge.

Graph based algorithmic techniques do however provide considerable efficiency in dealing with complex geometries by virtue of the well defined algorithms that may be formulated to operate on them.

6.3 Knowledge Based Design

The concept of an *ill-structured* problem was defined by Newell [Newell, 1969] as one which was not *structured* where a structured problem exhibited the following properties:

1. It can be described in terms of numerical variables, scalar and vector quantities.
2. The goals to be attained can be specified in terms of a well-defined objective function.
3. There exist computational routines (algorithms) that permit the solution to be found and stated in numerical terms.

The problem of creating structured floorplans has been defined in such a way in this thesis that it meets none of the above criteria, and is as a result ill-structured. In fact, the lack of success of many floorplanning techniques may be attributed to the attempted transformation of the problem into a *structured* form. The resulting algorithmic methods produce "good" solutions, but to the wrong (structured) problem.

Newell suggested that the only approach to solving ill-structured problems with a computer required *heuristic* programming (artificial intelligence) techniques. These are programming techniques that use processes based on human problem solving techniques in order to arrive at a solution. These techniques were classified as either *weak* or *strong*. Weak techniques are general, that is they are applicable to reasoning in a

wide range of domains. Strong techniques are specific, they incorporate knowledge about a particular domain and only operate successfully in that domain.

After an early interest programs such as GPS [Winston, 1984], the success of weak methods fell into question as they were poor at solving real problems. Far more success has been achieved with strong methods in the field of *knowledge based* or *expert* systems. These incorporate specific domain knowledge in order to solve narrow ranges of ill-structured problems.

The automatic floorplanner described in this thesis relies on specific (strong) domain knowledge of the type described in Chapter 5 to generate structured floorplans.

In this section the various knowledge representations and reasoning techniques that have been successfully applied in working knowledge based systems will be described. This will serve as a foundation for describing the structure and representations that are used in *Floyd*.

6.3.1 Knowledge Representation

One of the most critical issues in the creation of a knowledge based system is the design of the representations that are required in order to express the various forms of domain knowledge. Here we shall consider three broad classes of representation are either used or have influenced the knowledge representations in *Floyd*.

6.3.1.1 Production Rules

The most common form of representation in use in knowledge based systems is the *production rule* [Hayes-Roth et al., 1983]. In its most general form, a production rule consists of a left hand side predicate and a right hand side action in an **IF** $<$ *situation* $>$ **THEN** $<$ *action* $>$ form.

A typical *forward chaining*[1] *production system* is illustrated in Figure 6.4.

The *rule base* is comprised of a number of rules, each representing some independent fragment of knowledge. The *working memory* contains items that represent the state of the system: rules may insert and delete items from working memory with their

[1]Backward chaining strategies are uncommon in design systems, being more amenable to analysis and diagnosis.

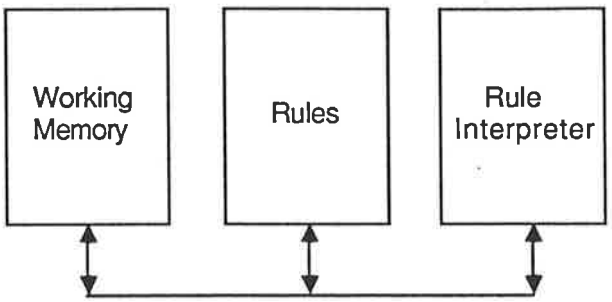

Figure 6.4: The components of a production system.

< *action* > parts. The *rule interpreter* test all of the rule < *situation* > parts against the contents of the working memory. If one of the rule predicates are satisfied by the contents of the working memory, the rule is "fired" allowing its action part to alter the memory. If more than one rule is applicable, a *conflict resolution strategy* [Forgy, 1982] is used to select the rule to "fire" from that set.

Production rule representations may be appropriate in the following situations:

1. Where the domain knowledge may be expressed in small, discrete units.
2. Where the domain knowledge is likely to change: the addition of knowledge by the addition of rules should be relatively straightforward if the rules are independent.
3. Where there may be a requirement for the system to explain its behaviour. This may be done by tracing the execution order of the rules.
4. Where the homogeneous representation of problem state provided by the working memory does not lead to extreme inefficiency.

The majority of knowledge based systems utilize rules as their primary means of knowledge representation. These are mostly *classification* and *diagnosis* systems [Davis et al., 1977; Lindsay et al., 1980; Duda et al., 1976; Feigenbaum, 1977b] but a number of *design* systems have also been produced:

1. *R1/Xcon:* configures computer systems [McDermott, 1980].
2. *DAA:* produces digital system structural architectures from instruction set processor specifications [Kowalski, 1986].

3. *Talib:* designs integrated circuit leaf cell layouts [Kimm et al., 1984].

4. *Weaver:* performs integrated circuit routing [Joobani & Siewiorek, 1985].

6.3.1.2 Frames

The *frames* knowledge representation proposed by Minsky [Minsky, 1975] as a general representation model for human intelligence, though the particular application he describes is in visual scene analysis.

In Minsky's model a frame is a structure that provides a framework for representing reality: when a new situation is encountered, the remembered frame that best fits that situation is recalled from memory. Every frame consists of terminals (or slots). Attached to some terminals are invariant data that holds true for all situations that relate to that frame. Other terminals have attached *default* information that may change in order to tailor the frame to a specific situation. *Frame systems* are created when terminals have attached *subframes* thus generating a hierarchy of frames.

Since the original conception of frames, the representation has received wide acceptance. It has also become entwined with the concept of *object oriented programming* [Weinreb & Moon, 1981b; Goldberg & Robson, 1983]. In particular the frame representation systems incorporated in development systems such as KEE [Intellicorp, 1985] include the concept of inheritance: a new frame may inherit properties (slot defaults) from an existing frame.

Frames are applicable to knowledge representation in the following situations:

1. Where the knowledge is structured in a hierarchical manner.

2. Where the knowledge is liable to change.

A number of knowledge based system that make use of hierarchical knowledge structures have been built around frame languages [Genesereth, 1982; Giambiasi, 1985].

6.3.1.3 Algorithms and Data Structures

There is often confusion over what defines a knowledge based system. In this thesis it is defined simply as a program that solves ill-structured problems as suggested by Newell. The *representations* chosen to describe the required knowledge do not affect

this definition, they are simply selected based on their applicability to the specific knowledge representation task.

This suggests that the most well known of programming techniques, algorithms and data structures, form a valid knowledge representation. This is particularly so in cases where the ill-structured problem may be partitioned into subproblems, some of which may then appear to have considerable structure. For instance the task of finding a plumber to fix a leaking tap may be regarded as quite ill-structured overall, but the subtask of searching for plumbers in a telephone listing may be performed quite well by a suitable algorithm and data structure that in effect performs the same task as the human. Algorithmic representations may be applicable in the following situations:

1. The knowledge is *complete*, that is there is not likely to arise the need to alter the body of knowledge.
2. The knowledge contains a large number of *associations* between its components. A suitable data structure may represent these associations explicitly, making the traversal of such associations more efficient with suitable algorithms.
3. The representation is not required to explain its actions.

An example of algorithmic techniques in use in a knowledge based system occurs in the Design Automation Assistant (DAA) [Kowalski & Thomas, 1983] in which the *estimators* are implemented as algorithms, not rules. These estimators are responsible for the "back of the envelope" calculations used by designers to assist in decision making.

6.3.2 Reasoning Techniques

Related to the issue to knowledge representation is that of *reasoning techniques*. This is a general term to describe a variety of methods that have been developed for reasoning within and with the domain knowledge.

Design is often characterized in artificial intelligence as a *search* through an often large space of possible solutions. Domain knowledge plays a part in reducing the complexity of this search by guiding the search down some pathways and ignoring others. (This is generally known as heuristic search [Stefik et al., 1983]). This section

briefly describes some of the concepts that may be applied in order to apply domain knowledge to the problem of limiting such a search.

6.3.2.1 Abstraction

The complexity of the search through a solution space may be reduced by using abstractions of the problem space. A well known example is the task of finding a road route between two cities [Stefik et al., 1983]. Searching out through all possible roads leading from one city looking for a path to the next is usually impractical. Instead a map that simply shows the main highways between the cities is used: the details of getting onto and off the highways may be solved later.

By finding similar abstractions in design problems it is possible to greatly reduce the combinatorics involved in search. The basic strategy is to search for broad solutions under the assumption that particular details will not greatly effect the optimality of the solution and may be filled in later.

6.3.2.2 Planning

Plans are basically sequences of instructions that suggest a method that may be used to solve a particular problem. Plans are in a sense *abstractions* of the actual detailed methods that will be required in order complete the task. Once a plan has been put forward, other reasoning facilities must be available to follow the plan and fill in the details.

There are two classes of planning that appear in knowledge based systems. The first is that which is not specific to a particular problem, but is sufficiently general that it may be used as a broad approach to the problems encountered by the system. This may also be classified as metaknowledge and will be described in Section 6.3.2.5. This form of planning knowledge is essential in giving a system some direction in solving a problem.

The second form of planning information is that which is formulated based on the particular problem instance. In the road example this would be a plan that described which highways form the best route. In order to be produce a detailed solution to the system, details on highway entrances, exits and fueling availability must be filled in based round the outline provided by the plan. This form of planning knowledge

serves to direct the system activities down particular paths thus speeding the search. In addition, in systems that have only poor estimates of the quality of the incomplete design, following a plan suggests that the design is proceeding in a fashion that may lead to a good result.

6.3.2.3 Constraints and Least Commitment

In a typical design problem, a number of subproblems are often being solved at one time. Typically these are not independent: decisions made in the solution of one subproblem may impact the solution processes taking place in other subproblems. The most common means of representing these interactions are entities known as *constraints* [Sussman & Steele, Jr., 1980; Stefik, 1980; Stefik, 1981; Sarcerdoti, 1974; Kimm et al., 1984]. As the design progresses in a particular subproblem, constraints that represent the changes taking place in that design are generated. A *constraint propagation* mechanism then passes these constraints on to subproblems that may be affected.

Stefik's program, MOLGEN, creates plans for genetic experiments. The steps in the experiment are designed as subproblems. These subproblems may be put into a state of suspension when there is insufficient information present to continue design of the step. When another subproblem proceeds, constraints are generated and propagate out to the other subproblems. Typically this will provide enough constraint on another subproblem for it to continue in turn. This strategy is known as *least commitment*: progress on an insufficiently constrained problem is halted until enough constraints arrive for progress to be resumed.

One problem that arises in a least commitment strategy is that at times there will be an overall lack of constraint, and all progress in all subproblems will cease. At this stage MOLGEN takes a *guess* in order to resume progress. If this guess should later prove to have been incorrect, the design carried out since the guess is undone in a *backtracking* procedure and resumed with an alternative guess. Backtracking is discussed further in the next section.

6.3.2.4 Backtracking

As mentioned in the previous section, backtracking is a procedure that is adopted in order to undo the results of a poor decision. There are two classes of backtracking.

Chronological backtracking involves simply removing *all* design carried out since the poor decision and then proceeding with an alternative. *Dependency directed* backtracking involves only undoing that design work that was the direct result of the poor decision. This is clearly more efficient but also more complex to implement as all design activity must be tracable back to specific decisions.

Backtrack is always inefficient as it requires undoing work, and is hence to be avoided. R1 [McDermott, 1980] and DAA [Kowalski & Thomas, 1983] use the *match* reasoning strategy that is based on the assumption that sufficient knowledge is always available to make correct decisions: the implications of each design decision may be evaluated fully prior to its instigation.

When used as a cure for poor decision making, backtracking has a number of problems:

1. Undoing design work is inherently inefficient. Backtracking may lead to the depth first traversal of the solution space.
2. A great deal of state must be preserved in order to allow undoing.
3. Design elements must be labeled with their causative decisions.
4. The point at which backtracking should be instigated, and to where it should be continued back too, is often unclear.

6.3.2.5 Metaknowledge

Metaknowledge (knowledge *about* knowledge) is that control knowledge used in a system to reason with the remaining knowledge embodied in the system [Lenat et al., 1983]. The overall strategy used by the system in order to solve a problem is controlled by guiding the selection of what bodies of knowledge in the system are to be applied to what subproblems at what time. This guidance is the result of the system selecting a particular design phase or strategy: constraint propagation, guessing or backtracking for example.

Some systems have represented metaknowledge in forms that are explicitly different from other knowledge: for instance predicate calculus [Genesereth, 1983]. More commonly however the metaknowledge is expressed in the same form as the base level knowledge, usually in rules [Kimm et al., 1984].

A common organization of knowledge in a system is based around its division into a number of *knowledge sources* [Erman et al., 1980]. The various knowledge sources are activated and deactivated according to design strategies embodied in the system metaknowledge.

6.3.2.6 Inexact Reasoning

Newall observed that: *Typically, an ill-structured problem is full of vague information* [Newell, 1969].

Even given the generality of the problem , there has not been a great deal of success in developing representations and reasoning techniques for vague information (now more commonly referred to as *uncertain knowledge*).

When reasoning with uncertain knowledge, the production rule structure of **IF** < *situation* > **THEN** < *action* > becomes **IF** < *evidence* > **THEN** < *hypothesis* > where the evidence suggests the hypothesis with some strength. The difficulty lies in deciding on an appropriate means of expressing the uncertainty in the evidence, the uncertainty in the inference, and the total resulting uncertainty in the hypothesis. A variety of methods have been suggested, the best known being:

1. *Measures of belief and disbelief* [Davis & Buchanan, 1977]
2. *Subjective Bayesian reasoning* [Duda et al., Proceedings AFIPS 1976 NCC]
3. *The Dempster-Shafer theory of evidence* [Gordon & Shortliffe, 1984].
4. *Fuzzy logic* [Zadeh, 1979].

In this section the Mycin measures of belief and disbelief will be presented as an appropriate model for inexact reasoning. This model, although lacking a rigorous mathematical basis, is simple and well characterized by its incorporation in the Mycin system [Buchanan & Shortliffe, 1984]. Subjective Bayesian reasoning is inappropriate because of the need to have a wide statistical base in order to estimate the *a priori* probabilities required [Duda et al., Proceedings AFIPS 1976 NCC]. Fuzzy sets are as yet poorly understood and the quantification and combination of fuzzy variables are ill defined [Shortliffe & Buchanan, 1984, p. 246]. The Dempster-Shafer theory of evidence shows considerable promise [O'Neill, 1987], however there is a lack of pub-

lished results from working systems that incorporate the model, and experimenting with basic models of inexact reasoning is not an intent of this thesis.

In the Mycin diagnosis system, hypothesis are proposed and then the likelihood of their being true is estimated based on expert knowledge encoded into rules. Probabilistic techniques were judged inappropriate as experts seemed unable to estimate *a priori* probabilities and the statistical distribution of diseases undergoes constant change. The model developed has a number of informal associations with Bayesian probabilities. In this section a subset of the model that is relevant to the problem at hand will be described. A more detailed description of the model itself and its empirical and theoretical justifications may be found in [Shortliffe & Buchanan, 1984].

The basic means of expressing uncertainty in Mycin is the *Certainty Factor* (CF). Each rule has a CF associated with it that is intended to quantify the expert's certainty that if the evidence is all true than the CF reflects the degree of certainty with which the hypothesis may be attributed. For example:

```
IF: 1) The stain of the organism is gram positive, and
    2) The morphology of the organism is coccus, and
    3) The growth confirmation of the organism is chains
THEN: There is suggestive evidence (0.7) that the identity of the
          organism is streptococcus
```

The CF in this case is 0.7. CFs are allowed in the range $[-1, +1]$ where positive CFs imply belief (confirming evidence) and negative CFs imply disbelief (disconfirming evidence).

The accumulation of CFs as more rules fire that apply to a single hypothesis is carried out by the assignment of *belief measures* to the hypothesis. In a Bayesian model, the lack of confirming evidence for a hypothesis suggests that it is less likely. This may however not be the case in the area of expert opinions and so two measures, one of *belief*(MB) and the other of *disbelief*(MD) are introduced. For each hypothesis under examination, the confirming evidence is recorded separately to the disconfirming evidence as MB and MD respectively. Both are constrained to lie in the range $[0, 1]$ in order to emphasise their relationship to probabilities [O'Neill, 1987].

The CF implied by each rule is combined with the current belief measures (MB and MD, initially 0) to produce the new belief measures (MB' and MD') with a

combining function which may phrased as:

$$MB' = \begin{cases} 0 & if MD' = 1 \\ MB + CF \times (1 - MB) & otherwise \end{cases}$$

$$MD' = \begin{cases} 0 & if MB' = 1 \\ MD + CF \times (1 - MD) & otherwise \end{cases}$$

Once all applicable rules have fired, the total CF for a hypothesis is calculated by: $CF_{total} = MB - MD$ The total CF for each hypothesis is then used in order to rank the their likelihood, those with the highest CFs being the most probable correct hypothesis.

The overall performance of this model of inexact reasoning in the domain of medical diagnosis was found to be excellent [Buchanan & Shortliffe, 1984, p.214].

6.4 System Overview

The design of *Floyd* is based on a *representational* approach in which three distinct knowledge representations have been adopted to describe specific aspects of floor-planning expertise:

1. The *rectangular graph* is a data structure and set of related operations that assist in 'common sense' geometrical reasoning about two dimensional spatial relationships.
2. The *classes* representation is an extensible mechanism for encapsulating knowledge about the abstract implementations of a variety of module layouts.
3. The most general representation is the *production system*, used in *Floyd* to represent metaknowledge (design control strategies and planning) and to perform pattern recognition.

The overall structure of the system is illustrated in Figure 6.5. There are a total of six *subsystems*, one for each of the three knowledge representations and a further three ancillary subsystems.

The central *Design Manager* is a *production system* consisting of a rule set, working memory and rule interpreter. The design manager controls the design process, interacting with each of the subsystems when it requires the facilities each provides.

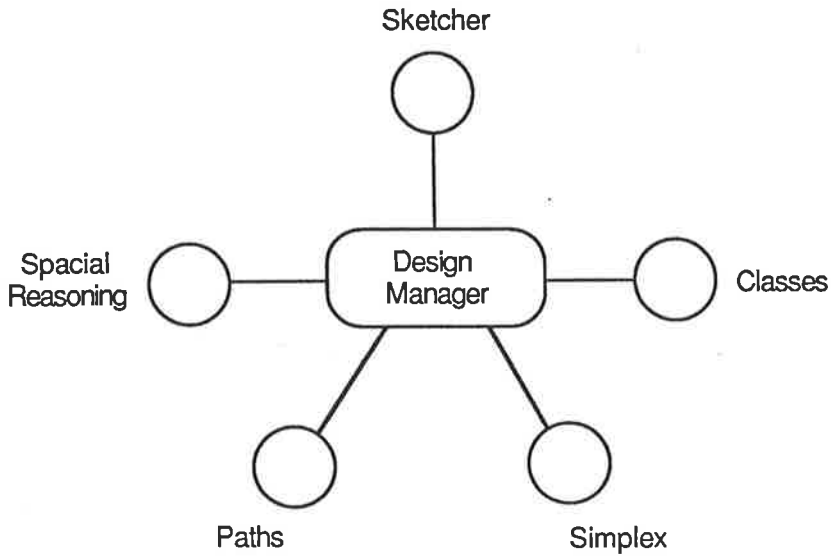

Figure 6.5: An overview of the components of the system.

The *Spatial Reasoning* subsystem is responsible for providing information about the spatial relationships between the rectangles that represent modules in the ongoing design. The subsystem is based around the *rectangular graph* representation.

The *Classes* subsystem is actually a data base of information about the possible implementations of modules in a floorplan and their layout properties represented by *class* structures.

The *Paths* subsystem is responsible for traversing the rectangular graph and generating a set of linear inequalities that form part of the set that may be solved in order to create the floorplan.

The *Simplex* subsystem is an implementation of the simplex [Dantzig, 1963] algorithm that is used to solve the set of linear inequalities that describe the floorplan. These are derived from those constraints derived from the rectangular graph and those generated by the design manager.

The *Sketcher* subsystem provides window facilities for sketching planar graphs and floorplans to aid in both debugging newly acquired design knowledge and to provide graphical feedback of floorplans to the user.

These subsystems are described in detail in the following sections, however discussion of their software implementation is deferred until Section 6.11.

The sequence of the following descriptions has been chosen for maximum clarity of presentation and does not in general relate to the relative significance of the subsystems.

6.5 The Spatial Reasoning Subsystem

In discussing the problem of constructing knowledge based systems to perform design, Stefik et. al. note:

> Many design problems require reasoning about spatial relationships. Reasoning about distances, shapes, and contours demands considerable computational resources. We do not yet have good ways to reason approximately or qualitatively about shape and spatial relationships.
>
> [Stefik et al., 1981a, p. 25]

In the previous chapter it was noted how designer use pencil and paper in order to represent and reason with the spatial relationships used in floorplanning. This assists them to record and observe the constraints that occur in a spatial structure. This knowledge was identified as being quite different in nature to the "expert" knowledge used for other purposes in the floorplanning problem. It appears to make use of "common-sense", innate abilities of the designer. Rather then than attempt to represent this aspect of knowledge in a general rule form, a special purpose data structure and related operations have been designed for the task. They interact with the remainder of the system through a well defined interface that hides the implementation such that it appears simply as an competent manipulator of rectangular floorplan elements.

The data structure has been named the *rectangular graph* (RG). It is a highly constrained planar embedding of an undirected graph in which the nodes represent layout modules and the arcs imply connection constraints between modules. The representation has a number of useful properties:

1. It can be used to record the current state of the floorplan topology.
2. It provides an abstraction that allows for the incremental inclusion of detailed size and shape information into the floorplan.

3. It can be used to efficiently generate the possible positions into which a connected module may be inserted into the design.

4. It can be used to estimate the size and shape of the space available in a floorplan for the placement of a new module.

5. It efficiently records the entire design history in a tree structure that facilitates backtracking.

6. It may be used for the generation of linear equations that define the final floorplan.

In the remainder of this section the data structure itself is presented, followed by the various procedures related to its use.

6.5.1 The Rectangular Graph

The original spatial representation developed for *Floyd* was based on a grid onto which the various modules were placed. As noted earlier, this structure, similar to that later used in Flute [Watanabe & Ackland, 1986] has the disadvantage that it places a degree of arbitrary constraint on the placement: relative module placements on the grid implied constraints that were not meaningful.

The rectangular graph avoids this effect by representing module placements only by adjacency constraints. Modules that are not required to be adjacent as a result of a connection are free to move as necessary. This allows a natural implementation of *least commitment*: only those constraints that are strictly required are expressed.

The basic entities of the graph are named *nodes, sides, arcs* and faces. These are illustrated in Figure 6.6 and described below.

Nodes: The nodes in the RG represent modules in the design. Nodes in an RG are denoted N_i where $i : 1..n$ for a graph with n nodes.

Sides: Each node has 4 "sides", each corresponding to the side of a square. These are referred to informally as the the *north, east, south* and *west* sides. Formally a node N_i has sides denoted $S_{i,j}$ where j may take on a value from the ordered sequence $\{n, e, s, w\}$ corresponding to the *north, east, south* and *west* sides respectively. The four sides of a node correspond to the four sides of a rectangular module.

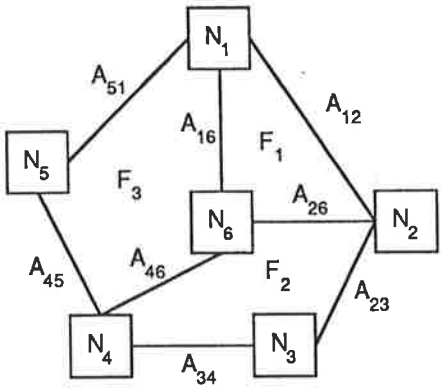

Figure 6.6: An example of a rectangular graph.

Arcs: A requirement that two nodes be adjacent to one another may be indicated by and *arc* joining the two. The arc associating the nodes N_i and N_j is denoted by $A_{i,j}$. An arc may be further constrained to associate pairs of sides of nodes. The arc associating the sides $S_{i,m}$ and $S_{j,n}$ is denoted A_{i_m,j_n}.

Faces: As will be described, a valid RG may be represented by a planar embedding of the graph. The closed regions bounded by the arcs and nodes are called *faces*. The faces in a graph are denoted F_i where $i : 1..n$ for a graph with n faces. A face may be described by a clockwise sequence of the nodes N_i that impinge upon it or a clockwise sequence of the arcs A_{i_m,j_n} that bound it. When used to describe a face, the arcs are *directed* such that traversing the arcs in their nominated direction causes a clockwise traversal of the face (Figure 6.7).

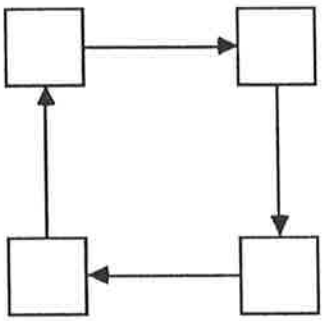

Figure 6.7: An RG face described by nodes and directed arcs.

Type	Turns	Arc Description
L	0	Non turning opposite sides
A_+	+1	Right turning adjacent sides
A_-	−1	Left turning adjacent sides
S_+	+2	Right turning same sides
S_-	−2	Left turning same sides

Table 6.2: Node and arc side configurations.

6.5.2 Turning

The most useful properties of the RG arise from the nature of the faces that comprise it.

The nodes in a face each of one incoming and one outgoing arc. There are only a limited number of arrangements of these two arcs with respect to each other:

1. The arcs are on opposite sides of the node.
2. The arcs are on adjacent sides of the node.
3. The arcs are on the same side of the node.

The possible configurations are shown in Figure 6.8 and described in Table 6.2. Each corresponds to a different number of *turns* in the face, where a turn is a right angle change in direction. Turns are *positive* if they imply a righthanded change in direction, and negative if they imply a lefthand change of direction.

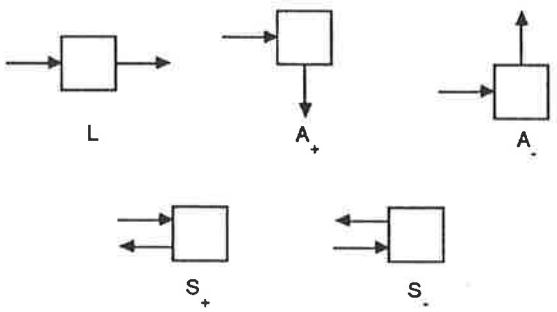

Figure 6.8: Configurations of arcs and nodes.

Theorem 6.1 *The total sum of all the turns in a face is always 4.*

Proof 6.1 *A face is a closed polygon. All of the angles in a polygon must sum to* 360°. *Hence the sum of the angles of each node in a face must sum to* 360° = 4 *turns.*

6.5.3 Properties of the Embedded RG

The objective of this representation is to assist in the manipulation of rectangular floorplans. In order that an RG be realizable as such a floorplan, a number of constraints must be placed on its construction. In this thesis a *feasible* RG will be regarded as one that may be translated to a rectangular floorplan. The following subsections introduce the various constraints and their significance.

6.5.3.1 Planar Embeddability

Each of the nodes in an RG represent a module rectangle. Clearly if the graph is to be used to represent a rectangular floorplan, which exists in the plane, the RG must also be presented as embedded in the plane. Additionally, since the arcs in an RG imply adjacency between connected nodes, they cannot cross one another. A crossing would imply that more than the two pairs of rectangles could share edges as illustrated in Figure 6.9 which is not physically possible as the adjacency of any two required edges directly excludes the adjacency of the other two.

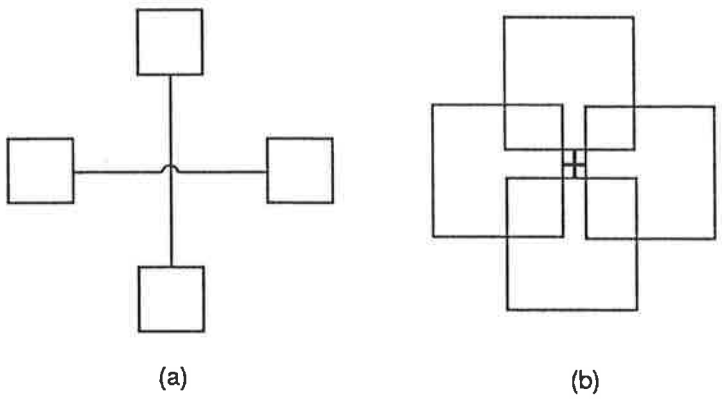

Figure 6.9: The crossing of two arcs implies the unrealizable adjacency of two pairs of rectangles.

6.5.3.2 Side Arcs

The connections between modules in a structured floorplan can only occur between parallel edges of the modules as shown in Figure 6.10(a). As the arcs in the RG represent adjacency hence possible connection, the arcs must be constrained to only connect opposing sides on the nodes. Thus an arc may only connect the sides $S_{i,n}$ to $S_{i,s}$ and $S_{i,e}$ to $S_{i,w}$ as illustrated in Figure 6.10(b).

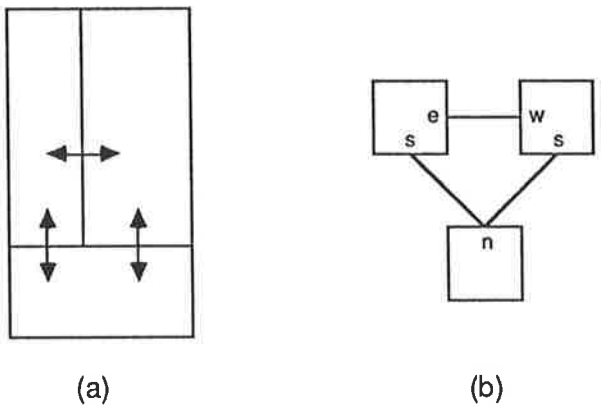

Figure 6.10: (a) Parallel edge routing in a structured floorplan. (b) The equivalent rectangular graph.

6.5.3.3 Triangulation

In general a planar graph will be comprised of faces of a number of different *orders*, and order being the number of nodes in a face. Note that the number of nodes and arcs in a face will always be equal as each node has one outgoing arc connecting to the next node. In general there will be a number of ways of realizing the adjacency constraints implied by the arcs in a face. For instance in Figure 6.11 a face of order 4 has a number of possible implementations. We shall postulate that this ambiguity may be removed if *all faces in the graph are of order 3.*

Lemma 6.1 *The lowest possible order face in a rectangular graph is 3.*

Proof 6.1 *Faces must be comprised of nodes and arcs that form a* closed *region and hence there must be at least three arcs as this (trivialy) is the fewest number of straight lines required to enclose a region.*

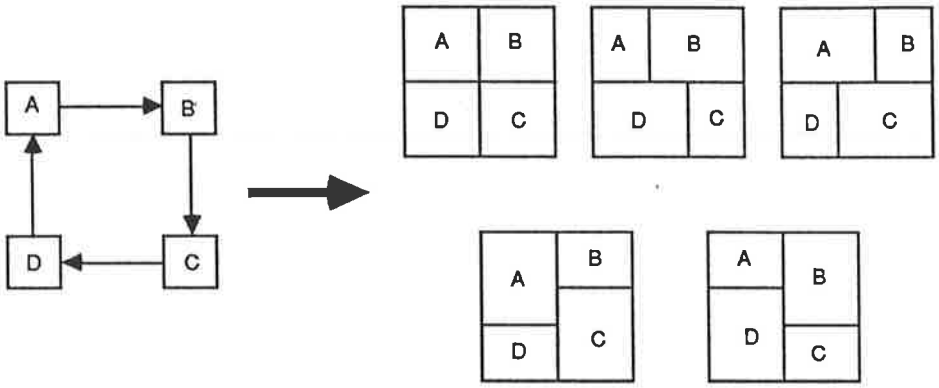

Figure 6.11: A face of order 4 has a number of possible implementations as a floorplan.

Lemma 6.2 *In order that there be no ambiguity in the adjacency constraints between nodes in a face, the face must be of order 3.*

Proof 6.2 *By Lemma 6.1 all faces are of order* ≥ 3.

In a face of order 3, each node has an arc to every other node in the face, thus unambiguously *defining its adjacency to every other node in that face as required.*

As a node in a face can only connect to 2 other nodes, then in a face of order greater than 3, any node will have at least *one other node it is not connected to via an arc, thus generating an ambiguity in adjacency.*

Theorem 6.2 *In order that there be no ambiguity in the adjacency constraints between sides of nodes in a face, the face must be of comprised only of nodes in the* A_+, A_+, S_+ *configuration (Figure 6.12).*

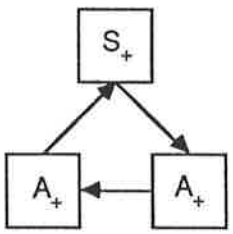

Figure 6.12: The A_+, A_+, S_+ configuration of a face.

Proof 6.2 *By Lemma 6.2 the face must be of order 3 to remove ambiguity in the constraints between nodes. If the ambiguity in sides is to be removed, each side must*

have an arc connecting it to every other side in the face. Thus we require a face of order three that has three sides, one from each node, connected. Given the limited number of node connection configurations, there are only two possible faces of order 3: A_+, A_+, S_+ and S_+, S_+, L. These are illustrated in Figure 6.13. Clearly only A_+, A_+, S_+ satisfies the criteria of having only three sides in the face, one from each node, so it is this face only that removes ambiguity.

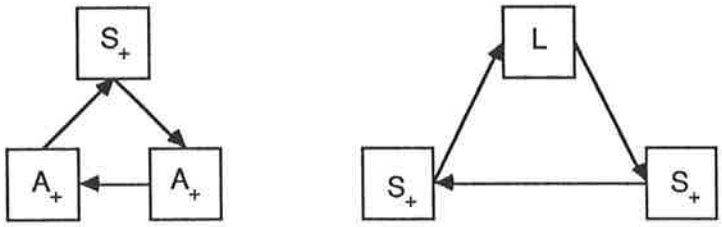

Figure 6.13: Of the A_+, A_+, S_+ and S_+, S_+, L faces, only the first has one connected side from each node in the face.

Thus it can be seen that in order to remove the ambiguity in the adjacencies in an RG, that RG must be *triangulated* have all its faces of the form A_+, A_+, S_+. We now proceed to show that any rectangular graph may be triangulated if it meets certain conditions.

First we introduce three concepts, *exposure*, *sidedness* and *reachability*.

A side of a node is *exposed* in a particular face if when drawn the side touches the inside of the face. This is illustrated in Figure 6.14.

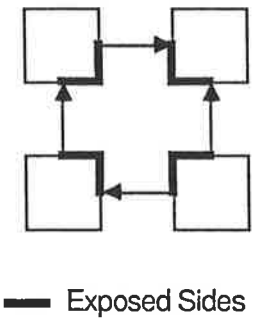

Figure 6.14: Sides exposed to a face.

A face is *concave* if all of the nodes in the face produce a non-negative turning. This is illustrated in Figure 6.15(a).

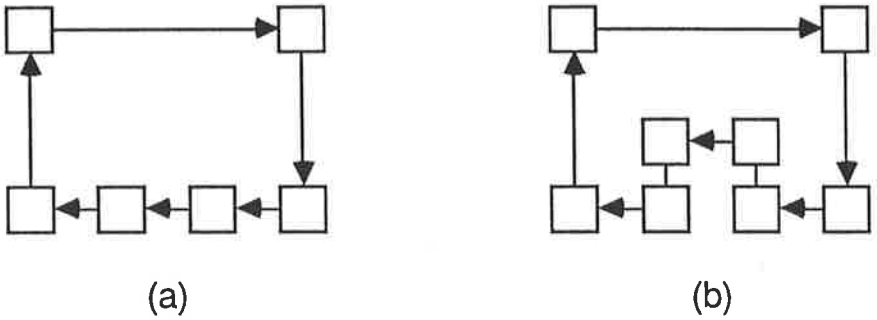

Figure 6.15: A concave face (a) and a convex face (b).

A face is *convex* if it is not *concave*. That is, it contains at least one negative turning node. This is illustrated in Figure 6.15(b).

A face has *sidedness* if for each node at a time, arcs may be drawn from each of that node's exposed sides to other exposed sides of other nodes in the graph such that none of the arcs overlap. This is illustrated in Figure 6.16.

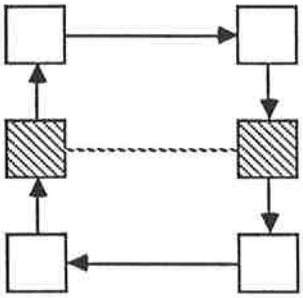

Figure 6.16: A sided face.

Theorem 6.3 (Triangulation) *If a face has* sidedness *then the face may be triangulated. Clearly if all the faces in an RG meet this criteria, then the RG is feasible.*

Proof 6.3 *If a face has* sidedness *then arcs may be drawn from each node that has a negative turning coefficient. This may be repeated till each of the faces thus generated contain only faces of non-negative turning coefficient (i.e. the face is concave). If we take any node in a concave face, then that node and the two connected to it may take any of the configurations indicated in Figure 6.17. As illustrated, any of these may be used as a basis for triangulation, except that in which the three are linearly arranged. If however the face is* concave *then there will exist another node in the face that may*

be used to break up the linear arrangement, thus ensuring that the triangulation of the faces thus produced may proceed until completed.

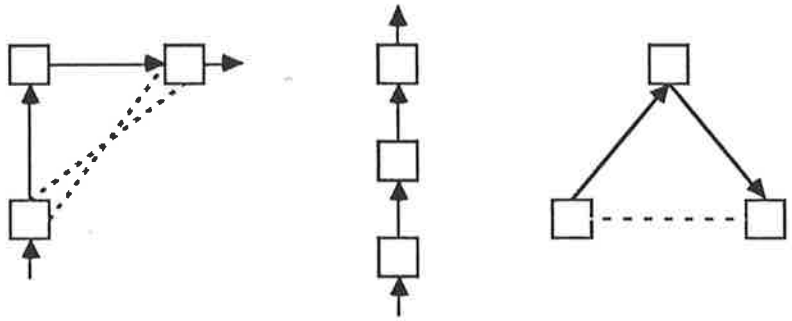

Figure 6.17: Possible configurations for three nodes in a concave face. All but the linear configuration may be used as the basis for triangulation as illustrated by the dashed lines.

In summary, if an RG is constructed in such a way that each face has *sidedness* then it will be possible to triangulate all of the faces in the RG. Hence the RG will be feasible, that is it may be realized as a rectangular floorplan.

In the following sections we will examine methods of RG construction that guarantee feasibility.

6.5.4 Ensuring Feasible Placements

We are working towards a method for constructing an RG from a set of node and adjacencies such that the RG is feasible. The basic method involves adding new nodes into existing faces in a feasible RG. In this section we describe a method of checking that such a proposed *placement* will result in feasible faces being generated. A placement is simply a description of a position that a node may be placed into an RG. It comprises the name of the node, N_i and the arcs which connect it into the RG: A_{i_m,j_n}. A typical placement is illustrated in Figure 6.18.

The problem may be formulated as follows. *Given a placement that describes the arcs A_{i_m,j_n} that define the place of a node N_i in a face F_j, check that all the faces that would be generated are feasible.*

Rather than actually inserting the node into the graph according to the placement, it

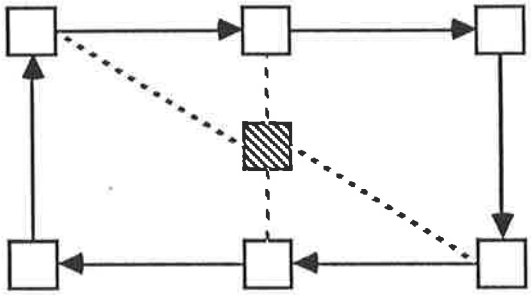

Figure 6.18: A typical placement in an RG face.

is clearly preferable to have a method that efficiently allows us to check the validity of the placement without creating new faces.

As a first step to developing such a method, we divide the problem into two parts: checking that the node being inserted would be sided, and checking that all of the already placed nodes in the face will remain sided. The first is relatively simple as will be described, but the second is more complex and requires the introduction of a new property, *reachability*.

Two connected nodes in a face are *reachable* to each other if the the arc that joins them does not imply an adjacency that cannot be realized because of the imposition of other constraints in the face. This is illustrated in Figure 6.19.

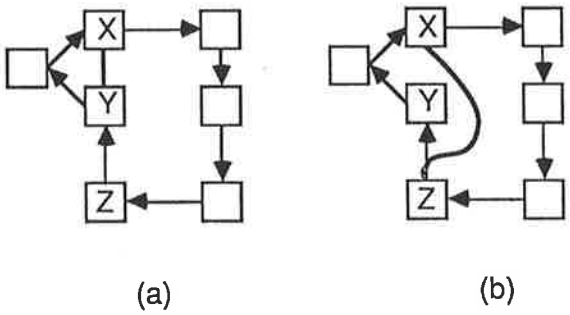

Figure 6.19: (a) The two nodes are reachable. (b) The nodes are not reachable.

If a new node is being added into a face, all of its arcs must be reachable, otherwise the faces generated by the placement would not have sidedness and thus would be infeasible.

Both the sidedness and reachability problems may be solved efficiently by *turn counting*. Consider the situation shown in Figure 6.20. One of the arcs in the placement is arbitrarily chosen as a starting point. The quantities $TurnCount$ and $SideCount$ are initialized to 0. The face is then traversed in a clockwise direction. $TurnCount$ is incremented or decremented so as to maintain a count of the turns the face has taken towards or away from the starting side. $SideCount$ is incremented to every time a new side of N_i is reached by a connection such that it represents the number of sides of N_i that have been traversed since the first. This proceeds until an in-place node is reached that has a connection to N_i. The feasibility of the adjacency implied by that connection is then determined by the values of $TurnCount$. The significance of these values are described below and illustrated in Figure 6.21.

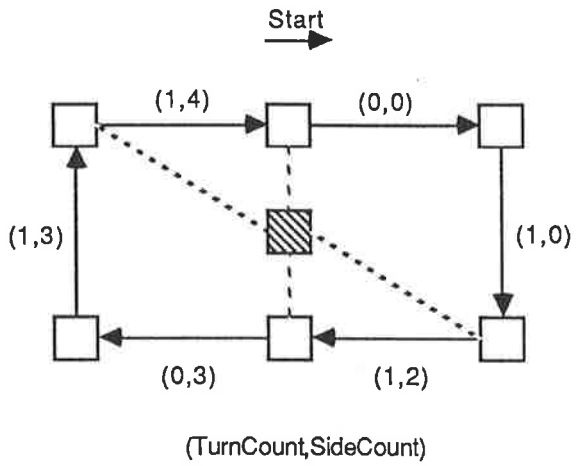

Figure 6.20: Examples of the use of turncounting to accept a placement (a) and to reject a placement (b).

$TurnCount < 0$ The face has turned away from the last connected face of N_i and the adjacency is infeasible, making the placement infeasible.

$TurnCount + SideCount > 4$ The face has "spiraled in" and cut off another connection. The adjacency and hence the placement is infeasible.

$TurnCount = 0$ The face is parallel to the last connected side of N_i and thus the connection is feasible.

$TurnCount = 1$ The face has turned once in towards N_i. Increment $SideCount$ by 1 to indicate the move to a new side of N_i. For example, if the last connection was to $N_{i,e}$ then this one must be to $N_{i,s}$.

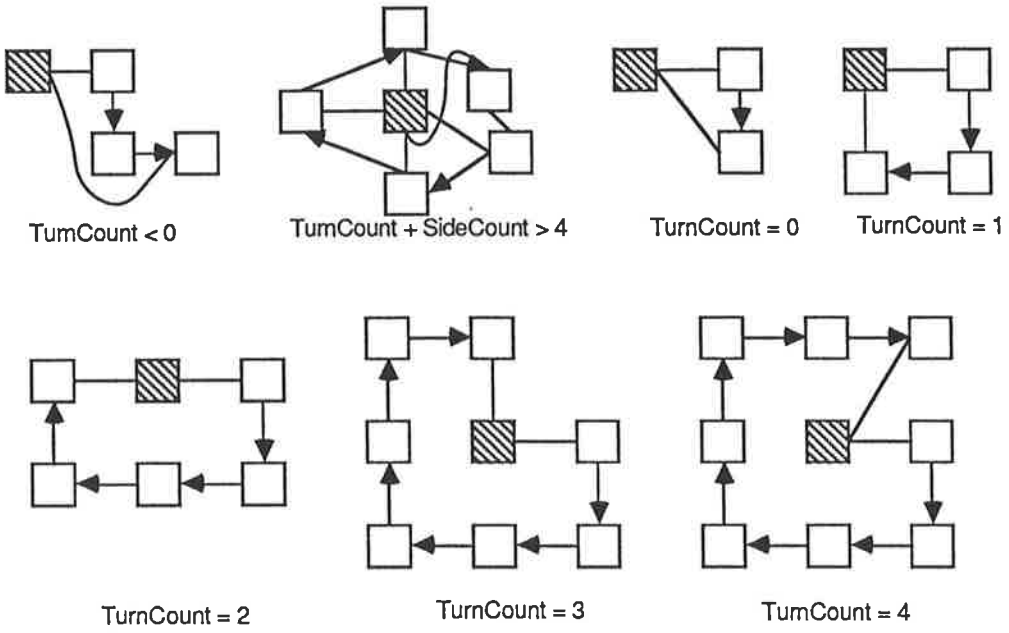

Figure 6.21: The implications of various values of the $TurnCount$ and $SideCount$ variables.

$TurnCount = 2$ The face has turned twice in towards N_i. Increment $SideCount$ by 2 to indicate the move to a new side of N_i. For example, if the last connection was to $N_{i,n}$ then this one must be to $N_{i,s}$.

$TurnCount = 3$ The face has turned thrice in towards N_i. Increment $SideCount$ by 3 to indicate the move to a new side of N_i. For example, if the last connection was to $N_{i,n}$ then this one must be to $N_{i,w}$.

$TurnCount = 4$ The face has turned four times in towards N_i. Increment $SideCount$ by 4 to indicate the move to a new side of N_i. For example, if the last connection was to $N_{i,n}$ then this one must also be to $N_{i,n}$.

When a new connection is reached, and found to be feasible, the $TurnCount$ is reset to 0. If the connection is infeasible, the placement may be rejected, thus saving a full traverse of the face. The procedure has been completed when either the placement is found to be infeasible or the original starting arc is reached.

The above procedure ensures reachability and hence the sidedness of the nodes already in place. The sidedness of the node *being* placed may be checked at the same time. As the face is traversed $SideCount$ is incremented as each new side of the node being placed is reached. To check sidedness of the node, all that need be done is that

for each side of the node traversed that has no arc, a node is traversed in the existing face that could potentially provide a side to which an arc might be attached. If no such node is passed, then the placement is infeasible.

This procedure is quite efficient, requiring a maximum of $O(n)$ operations where n is the number of nodes in the face being examined. Less time is required if the placement is found to be infeasible.

6.5.5 Placement Enumeration

In the previous section we described a method for validating the feasibility of a particular placement. In this section we examine a means of generating a set of *possible* placements that may be checked for feasibility.

We assume that there is a feasible RG. Into this RG we wish to find all the feasible placements of a node N_a that requires adjacency with a subset M of the nodes already in the RG.

As a first step, all of the faces in the RG are examined to find those that contain all the nodes in M. Only such faces can offer possible placements. If there are no such faces then a *planarity fault* has occurred and the invoker of the procedure is informed that remedial action is required (Section 6.10.11).

For each of the faces found, the following procedure is adopted to generate possible placements for N_a into the face F_k.

1. For each of the nodes in M, note their exposed sides in the face F_k.
2. The number of useful exposed sides may be reduced somewhat by excluding sides that would disallow other required connections. Basically if a connection is made to the side of a node, and that node has a connection to another required node on the opposite side, the original side is of no use (Figure 6.22(a)).
3. The number of possible sides may be further reduced by noting any S_+ nodes that are required for connection. If any exist, then it must be the *only* node that connects to that side of N_a as illustrated in Figure 6.22(b).
4. Generate all of the permutations of the remaining exposed sides, each permutation having one side from each of the nodes in M. Each of these permutations

may be a possible placement of N_a into the face F_k, but the feasibility of each must first be checked with the procedure outlined in the previous section.

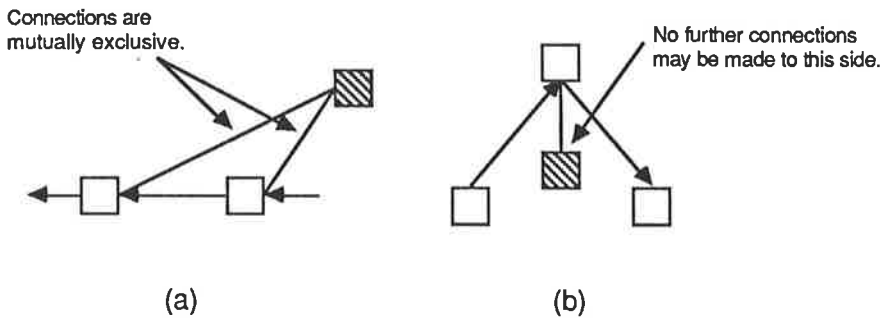

Figure 6.22: (a) The placement may be discarded as one node will always be interposed. (b) The placement may be discarded as the S_+ node must occupy the entire side of the inserted node.

Using the above procedure it is possible to generate all of the valid placements of a node into an RG.

6.5.6 The RG Data Structure

In previous section the basic procedures for checking the validity of placements have been outlined. These possible placements are supplied upon demand to the *design manager*. Typically at a later time the design manager will request that one of the placements for a node be *accepted*. That is, incorporated into the current rectangular graph. In this section the data structure used for representing the rectangular graph, and procedures for its construction are described.

The initial feasible graph is represented by four nodes, each representing the outside perimeter of the floorplan as illustrated in Figure 6.23. As placements are accepted into the graph, the face that they are inserted into is split into a number of subsidiary faces as illustrated in Figure 6.24. The original face is no longer actually a face in the graph as it has been split. The actual faces of the RG are the faces that exist as leaves of the tree.

Given this structure it is now possible to describe a complete procedure for placement generation and placement insertion.

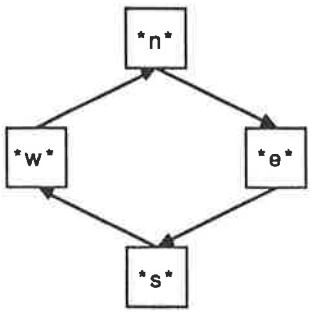

Figure 6.23: The four nodes that represent the perimeter of the floorplan.

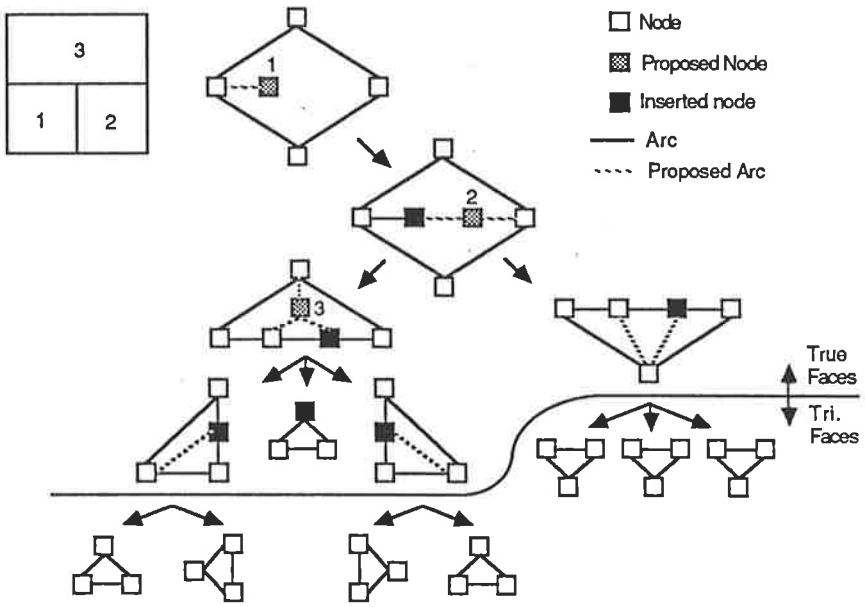

Figure 6.24: The splitting of an RG into a tree of faces.

6.5.6.1 Placement Generation

Each placement acceptance causes a *splitting* of a face into a number of child faces. When looking at generating new placements, we need only look at the new child faces as these are the only faces that have not been in the graph previously. The procedure is as follows:

1. Any proposed placement for a node that used the recently split face is invalidated as that face no longer exists.

2. Any proposed placement that would have included the just placed node had it been in place previously is invalidated.

3. From the set of new faces, note all those that might contain placements of all as yet unplaced nodes.

4. Use the placement generation procedure outlined previously to generate all placements in the new faces.

This procedure has a two of useful properties:

1. It is complete: all placements for all nodes are generated.

2. It is efficient: new placements are only generated from the new child faces. Also, with each new placement acceptance, only some of the existing placements are invalidated, and some new placements are generated.

The data transfer bandwidth of the link to the *design manager* may be kept quite low. The manager may keep a basic set of placements, and this simply has some placements incrementally added and invalidated as the design proceeds.

6.5.6.2 Placement Acceptance

A placement is recorded as a node name, a destination face, and a list of in place node and sides to which they should be connected to. Inserting a placement in the RG simply involves finding the face, traversing from one connected node to another, creating a new child face each time.

An interesting representational issue arises in the generation of some faces. In the example shown in Figure 6.25(a), the face appears to require that a node be created that has more than one incoming and one outgoing arc. Creation of such a node would of course invalidate many of the previously described theorems and procedures. This can however be avoided by representing the face in the form shown in Figure 6.25(b). The nodes are split into "doubles" that appear twice in the face and each have one incoming and outgoing arcs. Such a representation does introduce some further "book-keeping" in the management of the face data structure, but does leave the basic methods unchanged.

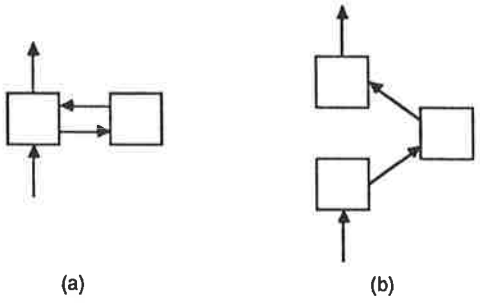

Figure 6.25: (a) This configuration requires complex nodes with more than one incoming and one outgoing arc. (b) The use of "double nodes" to allow the use of simple nodes.

6.5.7 Triangulation

The rectangular graph will in general not have triangular faces. Such a graph is limited in usefulness because as pointed out previously, there is a lack of constraint in each non-triangular face. Figure 6.11 illustrates the possible interpretations of a non-triangular face in a floorplan. Hence if the graph is to be useful we require a means of triangulating it. This is a non-trivial task: the additional arcs that must be added into the face imply constraints that may strongly affect the quality of the resulting floorplan. Figure 6.26 illustrates how a poorly chosen triangulation can lead to bad design. As well as producing high quality triangulations, we require that the algorithm be efficient: as shall be seen in the next section, faces are required to be triangulated whenever a placement is being examined for *size*.

There appears to be no reliable method for optimally triangulating an RG face except exhaustive enumeration of the triangulation possibilities, followed by transformation to, and evaluation as, a final floorplan. Observation of designers suggested a heuristic method for triangulation. As described previously, designers typically work with pencil and paper, gradually refining the representation into a floorplan. Although triangulation is not an explicit phase of this refinement, at some stage the designer must move from loose abstract representations to actual rectangles. As apart of this the constraints between the various sides of the rectangles must be specified: an implicit triangulation phase.

The heuristic may be characterised as one of finding the longest sides of rectangles that front onto the face, and applying constraints such that the other sides in the face

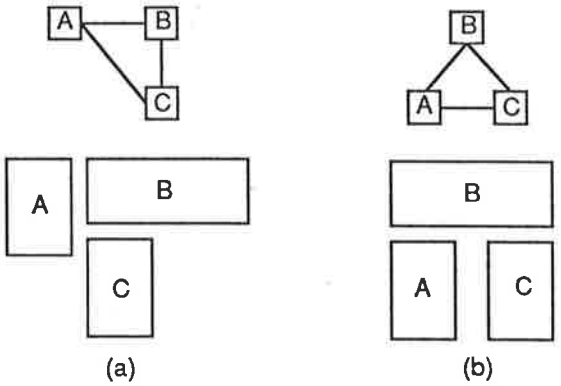

Figure 6.26: Poor selection of the triangulation (a) leads to an inefficient floorplan realization compared to the (b).

abut to the longer side. The method used to apply this is based on the observation that a triangular face always has one "apex" (or S_+) node and two "corner" (or A_+) nodes as illustrated in Figure 6.12.

The application of the heuristic to a face may be described as follows:

1. Traverse the nodes in the face till a corner node is found. Propose this as one corner of the new triangular face.
2. Assume that the nodes on either side of the corner node will form the triangular face. One will be another corner node and the other will be the apex node.
3. The apex node is the node that has the longest inward facing side. The other node is the corner node.
4. Split the face into two: the new triangular face and the remaining face which may or may not be triangular.
5. If the remaining face is not triangular, recursively apply this heuristic to it.

This process is illustrated in Figure 6.27.

The use of the above procedure does sometimes result in an illegal face: one that is not valid because it contains one or more of the structures illustrated in Figure 6.28. Thus after each application of the heuristic, the faces are checked for their validity. If invalid, there new faces is disregarded and a new corner node is selected. If there are no further corner nodes available, the following procedure is adopted:

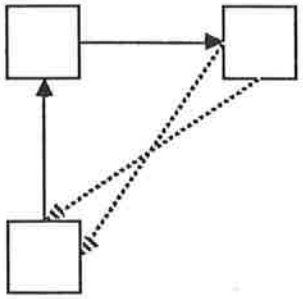

Figure 6.27: Selection of nodes to produce a triangular face.

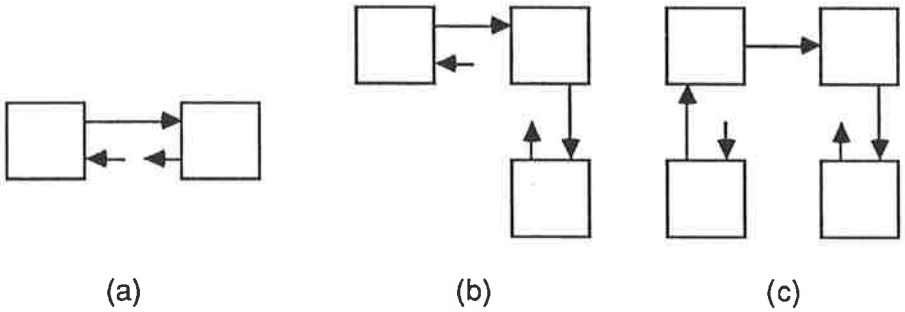

Figure 6.28: Structures that may occur in a face after triangularization that invalidate the face.

1. Select an apex node in the face.
2. Use the two nodes connected to this apex node as the corner nodes in the new triangular face.

This process is illustrated in Figure 6.29.

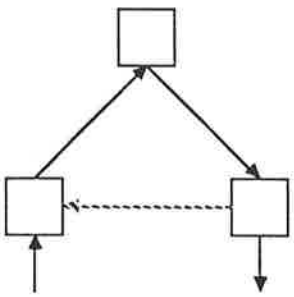

Figure 6.29: Alternative method of triangulating a face.

The combination of the above heuristics for has been found empirically to be capable of generating of reasonable quality (as judged qualitatively against human performance) triangulations of rectangular graphs.

The method has the advantage that it is computationally efficient, requiring only linear searches of the face being split.

At this point it is necessary to describe how the data structure used to represent the RG may be expanded to include the triangulated faces of the graph. It will be found useful in the size calculations described in the next section to keep the graph triangulated at all times. Thus whenever a new node is added into the graph, the faces thus created by splitting the placement face are in turn split into triangular subfaces. This is illustrated in Figure 6.24. The leaves of the data structure tree are now the triangular faces, and these must be distinguished from the "virtual leaves" that represent faces which may be used for actual placement generation.

6.5.8 Size Estimation

As part of the process of evaluating the quality of a placement, we naturally require information on the amount of space available in the design for placement of the rectangle being considered.

Given that the entire graph is kept triangulated, estimation of the size of the space is fairly straight-forward. The procedure is illustrated in Figure 6.30 and described below:

1. Temporarily place the node that is the target of the placement being sized into the RG.
2. Complete the triangulation of the RG by triangulating the new faces created by the temporary placement.
3. Starting at each side of the newly placed node, search out the longest (in terms of rectangle sizes) path to the outer boundary of the floorplan.
4. Subtract each of these path lengths from the current total size of the floorplan in order to calculate the total size and shape of the space in question.

The procedure may be made more efficient by labeling each node in the RG with

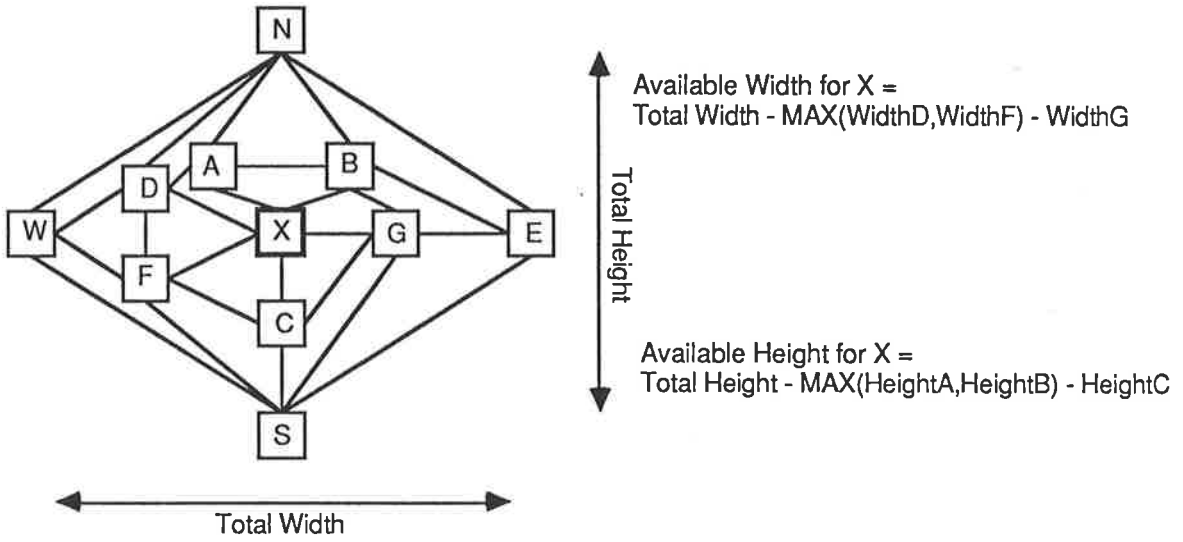

Figure 6.30: Estimation of the space available for a rectangle to be placed into an RG.

the respective distances of its sides from the sides of the complete floorplan, thus eliminating the need to traverse paths more than once in the search.

Once this procedure has been completed, the temporary placement is stored away: if that placement is actually accepted into the design, the triangulation will still be valid and need not be recreated. In addition, when a placement is accepted into the design, the sizing information is used to update the total size of the floorplan if the new placement has used up more space than was available.

6.5.9 Planarization Support

There are three basic facilities that are provided by the subsystem to support the planarization of the floorplan:

1. Identification of connections that may be temporarily disregarded in order that a face be found that may be used to generate placements for a particular module.

2. Search of the RG in order to identify possible paths that may be used to route in connections that have been previously discarded in order to maintain planarity.

These facilities are all provided by straightforward graph search techniques.

6.5.10 Interface Definition

At this point a useful summary of the operation of the spatial reasoning subsystem may be provided by listing the functions that are provided by the interface of the subsystem with the design manager. Details on the method of communication between the subsystems is discussed in the Section 6.11.

1. *Initialization.* Called only once to initialize the subsystem for a new design.
2. *Declare Connection.* Declare a connection between two modules in the design. Implicitly declares the modules as well.
3. *Declare Alternate Connection.* Declare an alternate connection between two modules. This is used when a module may requires a connection to any *one* of a number of other modules.
4. *Start Generation.* Generates the first batch of placements.
5. *Accept Placement.* Merge a particular module placement into the current design. Also update the set of active placements.
6. *Orientate Module.* Add size and shape information into a module in the design. This can then be used for size calculations by the subsystem.
7. *Get Placement Size.* Find out how much space is available for the placement of a particular module into the design.
8. *Find Route Paths.* Supply the shortest paths between two modules that must be connected.

6.6 The Classes Subsystem

In the previous chapter one of the major areas of floorplanning domain knowledge that was identified was that concerned with the *implementation* of modules. Although some of the modules in a floorplan may have specific constraints on them, top-down design will in general mean that the modules have no specific structure at the time the floorplan is created. Thus the designer draws on experience to postulate various implementations of the module, and takes the constraints implied into account in designing the floorplan.

In this section a representation for this form of knowledge is introduced. Requirements on such a representation include:

1. It must be capable of representing *uncertain* or *advisory* knowledge. Designer knowledge of implementations is by definition somewhat vague as the modules have yet to be constructed.

2. It must be *extensible.* As the floorplanner is run on more designs, it should be possible to add the resulting experience into the system in a simple way. In particular it should be possible to build on top of the existing knowledge base.

3. It must be *discrete.* Rules often interact in a complex fashion and thus any changes must be made only with a knowledge of the wider context of the rule. Ideally the representation should be comprised of independent pieces of knowledge.

4. It should be *intuitive.* The complex rule bases of systems such as R1 [McDermott, 1980] make it difficult for users to add in new information required to cope with previously unforeseen situations. A simple form of representation encourages users to expand the knowledge base.

In order to fulfill these requirements, the *classes* knowledge representation has been developed. Similar to a frame (Section 6.3.1.2) in structure, a class represents the abstract physical properties of a particular type of module. Typical classes are `generic`, `linear-array`, `register`, and `serial-adder`.

A class is comprised of one or more *implementations*, each representing alternate physical implementations of a module of that class. For example the `register` class may have `vertical-clock` and `horizontal-clock` implementations.

An implementation is primarily comprised of a set of *slots* In each slot is a pair of the form `(property value)`. A `property` is the name of a physical property of the module implementation that might affect the floorplan being developed. The associated `value` is a measure associated with the property. Properties only refer to those physical aspects of the module that are apparent in its interface to other modules in the floorplan: its width, height and port positions. In some cases it is desirable to be able to express the property with respect to a specific direction relative to the normal orientation of the module. In such a case the slot becomes a

triple: `(property direction value)` where the direction may be either `horizontal` of `vertical`.

Examples of properties include:

1. `height` The height of the module in relative units.
2. `width` The width of the module in relative units.
3. `io-port-opposition` The desirability of having input and output ports opposite one-another.
4. `io-port-adjacency` The desirability of having input and output ports on adjacent sides.
5. `wireability` The ease with which additional connections may be routed through the module.

In addition to the slots, there are several other components of an implementation. These are: its name; the name of the class the implementation belongs to; and a list of ancestor implementations. The ancestors are implementations from which properties are inherited. Thus a new implementation may be constructed by first inheriting properties of existing ones and adding or altering some properties in order to create the new implementation. Ancestor implementations are referred to in the form `class-name:implementation-name` as the names of implementations need only be unique within a class. Many classes have for instance implementations named `simple` or `basic`.

There are two types of values associated with properties. Dimensional properties have values that are simple integers representing approximate lengths. The symbol `N` may be appended if the dimension is dependent on some multiple of a basic unit. The width of an adder with each bitslice being 10 units wide would be specified as `10N`.

The second type of value is the *configuration factor* (CF). Like Mycin's *certainty factors* (Section 6.3.2.6), CF's provide a means of representing uncertain domain knowledge. For each physical property (other than width and height) a value is selected by designer acting as a knowledge source that reflects the desirability or otherwise of the configuration suggested by that property. Each CF must lie between -1 (very undesirable) to +1 (very desirable). The use of CF's in the evaluation of placements will be described In Section 6.10.7.

```
(define-implementation simple register
   (linear-array:simple)
   (io-port-opposition vertical 0.6)
   (io-port-opposition horizontal -0.4)
   (io-port-adjacency -0.4)
   (height 10)
   (width 20N)
   (wireability horizontal 0.6)
   (wireability vertical 0.4))
```

Figure 6.31: An example of an implementation of the `register` class

An example of the definition of an implementation is illustrated in Figure 6.6. The example shows the `simple` implementation of the class `register`. The definition inherits the slots of the more basic `simple` implementation of the `register` class.

The classes subsystem is basically a knowledge base of implementations that may be easily extended to cover new classes as required. The knowledge contained in the subsystem is utilized by the design manager in evaluating competing placements.

6.7 The Paths Subsystem

Whilst being a useful intermediate design representation, the completed rectangular graph is not a floorplan. It does however provide a network of constraints that may be combined with other data and then optimized to produce a complete rectangular floorplan solution.

The *paths* subsystem is invoked by the design manager to construct a set of linear inequalities that when solved will give the sizes and positions of all the modules in the floorplan.

There are four sources that are referred to in constructing the inequalities. The variables in the inequalities are chosen to be the position of the sides of each of the rectangles that represent the module boundaries. For instance the north side of a module named A has a y coordinate denoted by the variable name A_{north}. Only one coordinate is required as the x position of the side will be described by the variables A_{east} and A_{west}. The boundary of the floorplan is treated as being made up of four

modules: $*n*, *e*, *s*$ and $*w*$ as illustrated in Figure 6.23.

The first set of inequalities is made up from the minimum size requirements of the individual modules. For each module A these take the form:

$$A_{minwidth} \leq A_{east} - A_{west}$$

and

$$A_{minheight} \leq A_{north} - A_{south}$$

The second set of inequalities is made up from the coincidence of various sides that occur where ever there is a link between modules. For instance two modules A and B that are joined along a vertical edge (A to the right of B) will result in an equation of the form:

$$0 = A_{west} - B_{east}$$

The third set of inequalities arise from the fact that the individual widths and heights of the modules must sum to the width and height of the overall floorplan. The modules and links may be traversed to find all of the possible paths from one side of the floorplan to the other. Two new variables are introduced. They are defined by:

$$TotalWidth = *e*_{west} - *w*_{east}$$

and

$$TotalHeight = *n*_{south} - *s*_{north}$$

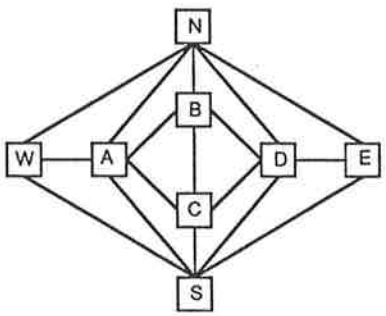

Figure 6.32: Example of traversing an RG in order to produce a set of linear inequalities.

Figure 6.32 illustrates an interconnection pattern that would give rise to the following sets of equations for the horizontal direction:

$$TotalWidth = (A_{east} - A_{west}) + (B_{east} - B_{west}) + (D_{east} - D_{west})$$

$$TotalWidth = (A_{east} - A_{west}) + (C_{east} - C_{west}) + (D_{east} - D_{west})$$

and for the vertical direction:

$$\begin{aligned} TotalHeight &= (A_{north} - A_{south}) \\ TotalHeight &= (B_{north} - B_{south}) + (C_{north} - C_{south}) \\ TotalHeight &= (D_{north} - D_{south}) \end{aligned}$$

The fourth set of inequalities is derived from from the minimum size of the connections required between modules. Sufficient overlap must occur between modules to allow for the wires connecting them. For example the connections from A to B and C in Figure 6.32 give rise to the following equations:

$$\begin{aligned} AB_{minwidth} &\leq A_{north} - B_{south} \\ AC_{minwidth} &\leq C_{north} - A_{south} \\ AB_{minwidth} &+ AC_{minwidth} \leq A_{north} - A_{south} \\ AB_{minwidth} &\leq B_{north} - B_{south} \\ AC_{minwidth} &\leq C_{north} - C_{south} \end{aligned}$$

Inequalities from the above sources are separately derived for the x and y direction and formated in preparation for solution by the *simplex* subsystem.

6.8 The Simplex Subsystem

The *path* subsystem generates two sets of linear inequalities: one for the horizontal direction and one for the vertical direction. When solved, the values allocated to the variables in these inequalities may be used to construct a rectangular floorplan as they provide locations for all of the sides of the various modules.

Ideally the total area of the floorplan should be used as the evaluation function to be minimized in the solution of the inequalities. This would however require the solution of a set of quadratic inequalities produced by the combination of both horizontal and vertical constraints. The solution of such a system is more difficult that the solution of a linear system of inequalities. However, in the general case the size constraints in a floorplan are approximate, and the floorplans are primarily intended as a tool for examining structural topologies. Thus it is reasonable to

approximate the minimization of the quadratic system by keeping the vertical and horizontal inequalities separate, and minimizing the total height and width of the floorplan.

Given that the task is to solve a set of linear equations such that a linear evaluation function is minimized, the Simplex [Dantzig, 1963] algorithm has been adopted as it is widely accepted as an efficient solution technique for this class of problem.

The *simplex* subsystem is an implementation of the Simplex algorithm. It accepts a system of linear inequalities, and a single objective function for minimization. In solving inequalities for a floorplan, the algorithm is run twice: once on the vertical inequalities with an objective function that is simply $total - height$ and once with horizontal inequalities with an objective function that is simply $total - width$.

In the first phase of the process, symbolic manipulation of the inequalities is carried out to reduce the number of variables and equations. This is done by noting equations of the form:

$$0 = variable1 - variable2$$

that imply that the two are equal. All instances in the system of $variable2$ are replaced by $variable1$ and the substitution is recorded for later expansion on solution of the system.

In the second phase, "slack" variables are created and included with appropriate coefficients into the system as required by the algorithm. Next a "tableau" is assembled from the system of equations and the evaluation function. The steps of the algorithm are then applied to the tableau in order to arrive at a solution. Finally the tableau is examined and all variables are allocated their solution values.

The result of a vertical and horizontal run of the algorithm is a set of variables that define the positions of all of the sides in the floorplan. The result may either be written to a file or graphically displayed to the user as described in the next section.

6.9 The Sketcher Subsystem

As much of the information dealt with in the floorplanner is geometrical, it is useful to have a flexible means of displaying different types of geometric data. There is a layered facility provided by the *sketcher* subsystem for this purpose.

The lowest layer of the graphics system is provided by a *window facility*. Based on the Lisp Machine [Weinreb & Moon, 1981a] Windows Package, it provides an object based implementation of a window system. Windows may be moved, overlapped and revealed, allowing the user to focus attention on particular aspects of the floorplan design in progress. An application may create a window and then send it messages in order to create graphics.

The upper layer of the sketcher subsystem consists of applications built on the window system for drawing graphs and floorplans from data supplied by the design manager.

Graphs are in general quite difficult to draw in a manner that is informative to the user. The method developed for the display of planar rectangular graphs has proven to be quite effective. Firstly the outer four nodes of the graph (the north, east, south and west) are placed equidistantly round a circle that fits within the window in use (this placement is notional, the graph is not actually drawn till the process is complete). The other nodes are then placed in the centre of the circle. An iterative process is then used to place the nodes in appropriate positions. For each node, the links to other nodes are treated as "springs", applying a "force" in the direction of the link proportional to the cartesian length of the link. These forces are vector summed for each node, and then the at each iteration the node is permitted to move in the direction of the "force" along a distance proportional to the magnitude of the force. The iterations are continued till there is no more movement (the system of "springs" is in equilibrium) or until a certain number of iterations has passed (in case the system is cycling due to too large iterations). The resulting nodes and links are drawn on the window in the resulting positions. An example of a graph drawn in this way is illustrated in Figure 6.33. As may be seen, the equilibrium state corresponds to a state in which the graph is comprehendable. In some cases however nodes will tend to overlap. To overcome this an addition to the algorithm adds in a "resisting" spring that forces nodes apart when they move too close.

The second sketching application involves drawing floorplans. This is a simple application that takes the floorplan side position variables such as `A-west` and their corresponding solution values. It parses the variable names and then uses the value of each to draw the complete floorplan, complete with module names and positions on a graphics window. This is illustrated in Figure 6.34.

The window based graph and floorplan sketching facilities provided by the subsystem are valuable for debugging, knowledge acquisition/tuning and designer interaction.

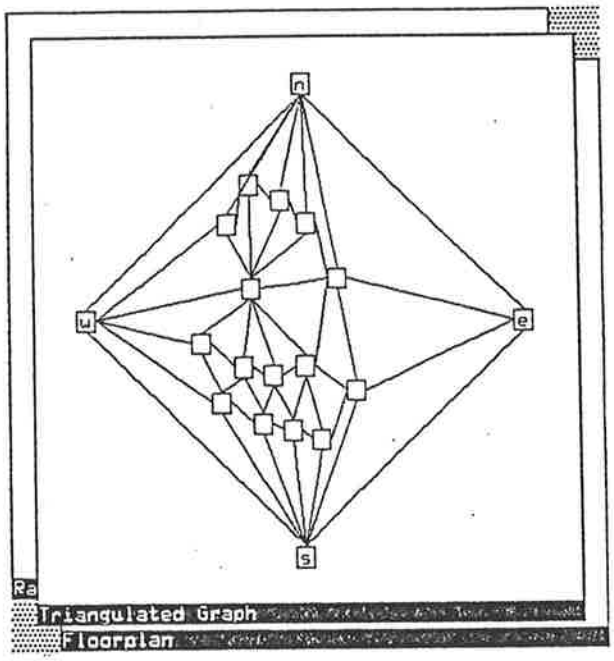

Figure 6.33: Display of a planar graph of a Booth multiplier design.

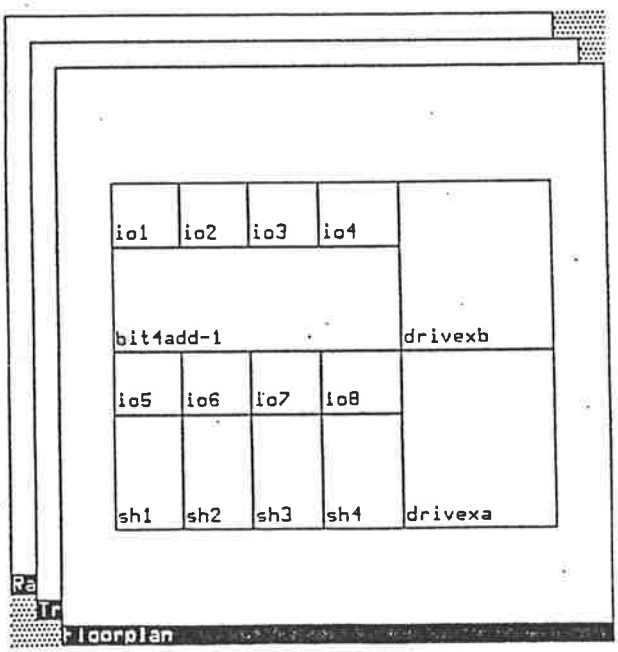

Figure 6.34: Display of a completed floorplan for a Booth multiplier design.

6.10 The Design Manager

The rectangular graph and classes representations are *domain dependent* — they are useful for representing narrow aspects of *floor planning* design knowledge. In addition to these, another more general representation is required to cover the remaining major knowledge representation requirements. These include the two meta-level functions

of *planning* and *control*, and the additional function of pattern recognition.

A *production system* (Section 6.3.1.1) has been chosen as the most appropriate structure for these purposes. It has the advantage of being quite general and incrementally expandable. Additionally the overall operation of recognizing a pattern in the working memory and carrying out an operation on it is closely analogous to the design problem. The presence of specific patterns in the structural and physical design trigger further design operations.

6.10.1 Control Strategies

Control is primarily the selection what activities should be taking place at a particular time. In *Floyd* rules are grouped into *tasks* that perform specific functions. Tasks may be activate or deactivated. In their activated state the rules that comprise the task have there antecedents checked to see if they match any elements of the working memory. In the deactivated state the rules within the task are never matched. This strategy has a several aims:

1. The rules are partitioned by tasks, thus making comprehension of the system operation simpler.
2. Only the antecedents of rules in active tasks need be matched against working memory elements, increasing efficiency.
3. Meta-rules may be used to activate and deactivate tasks. These meta-rules can thus direct design by activating tasks when appropriate data is available, and deactivating them when there is a invalid data present.

The tasks in *Floyd* are initialized once at the start of the design, and persist throughout the design process, being activated and deactivated by meta-rules. The tasks may be regarded as prioritized: they are given the opportunity to run by a set of scheduling rules in a strict order. The higher priority tasks are those concerned with maintaining the consistency of the design state: initialization, design fault handling and constraint propagation. The next priority tasks are those concerned with planning: if any situation has arisen that allows the formation of a plan it must be treated before further design takes place. The lower priority tasks are those actually concerned with moving the design forward by the placement of modules into the floorplan. The main tasks and their priorities are listed in Tablereftable:tasks.

Priority	Task	Description
1	Initialization	Initialize manager and subsystems
2	Wait for Placements	Wait for RG placements
3	Planarization Fault	Handle planarization faults
4	Placement Filtering	Remove contradictory placements
5	Constraint Propagation	Propagate effects of constraints
6	Run Deferred Tasks	Run constrained deferred tasks
7	Planning	Create plans
8	Module Selection	Select an unplaced module
9	Placement Selection	Evaluate alternative placements
10	Placement Acceptance	Accept a module into the design
11	Route Completion	Search out paths for the route insertion
12	Solve	Construct and solve the floorplan inequalities

Table 6.3: Tasks and their priorities.

Related to *Floyd* tasks are *subtasks*. Like a task, a subtask is a control structure comprising a set of rules. Unlike a task however, a subtask may only exist for a short period of design before it ceases to exist. Subtasks are initiated by task or other subtasks to perform some small operation and then delete all traces of themselves. Subtasks may be thought of as analogous to subroutines in a more conventional control structure as they allow they may be activated in a particular sequence, allowing sequential operations to be implemented. Subtasks are also implemented with a mechanism for receiving "parameters" from their creator.

One of the primary aims of the control strategy is to eliminate the need for backtracking within the design. As pointed out previously (Section 6.3.2.4) there are a number of disadvantages to backtracking based on efficiency. There is also the issue of implementation difficulty: previous state must be preserved in case backtracking requires a return to that point. Although an earlier version of the program made use of a limited backtracking facility [Dickinson, 1986b], it had a number of difficulties. The primary problem was one of ambiguity in deciding when backtracking should be initiated and when it should be stopped. Backtracking may be initiated when:

1. There are no placements available for a module because of a planarity fault. It is possible that by initiating backtracking and redesigning, a planar solution *may* be found. On the other hand, the new solution may cause problems in placing other modules.

2. The size, shape and implementation constraints lead to a situation in which all placements for a module are highly undesirable. In this situation backtracking may lead to a better result, however a worse result for this or other modules is also possible.

These same problems lead to a difficulty in deciding how far to backtrack as the point at which an incorrect decision was made is difficult to determine. Overall, the disadvantages of backtracking were found to be considerable. Rather than focus effort on improving design by backtracking, it was decided to take an approach similar to that of the Design Automation Assistant (DAA) [Kowalski, 1986] and R1 [McDermott, 1980] expert systems and eliminate backtracking altogether. Newell has suggested that this heuristic strategy of *match* is sufficient for solving ill-structured programs provided there is sufficient domain knowledge available. To this end further effort was put into the *planning* and *least commitment* aspects of the control strategy.

The control strategy is distributed between the various tasks of the production system, each performing functions as they are activated according to their respective priority.

In the remainder of this section each of the major tasks is described in more detail. The order in which they are described has been chosen for the sake of clarity of description.

6.10.2 Initialization

The structural description of the module to be floorplanned is expressed in Lisp code as described in Section 6.11. This code is interpreted and results in a set of elements appearing in the working memory that represent the input specification. Typically these elements are *constraints* (Section 6.3.2.3). Examples of such constraints include:

1. `Port A on Module X must connect to Port B on module Y`

2. `Port A on Module X must lie on the east side`

3. `Module A must have a height of at least 200 units`

Most constraints are a product of the structural input description. Some modules, such as PLA's are highly constrained and this may be expressed in terms of constraints on port positions and sizes. In addition the user may simply add further constraints to the input in order to influence the resulting design in a particular way.

Given a structural description in working memory, the initialization process proceeds to perform a number of operations:

1. Check the consistency of the initial constraints: only proceed if there is sufficient data and no obvious contradictions.

2. Make up the constants in the working memory. These are used for reasoning at a later stage. For instance the fact that `north` and `south` sides are `opposite`.

3. Propagate initial constraints. For instance if two ports are connected, and one of the ports must lie on a particular side, then a constraint on the other port may be created.

4. Gather numerical measures that will be used later in design. For instance, count and record the number of connections to each module.

5. Extract relevant data from the classes data base. Each module is assumed to have an associated class, otherwise it is assumed to lie in the `generic` class. An set of `implementations` is made up for each module. Each implementation is a permutation of the name of the module, one of the possible implementation types of the module, and one of the possible orientations of the module. For example a module `Xreg` with possible implementation types `basic` and `stretched` would result in the following implementations:

   (a) Xreg basic horizontal

   (b) Xreg basic vertical

   (c) Xreg stretched horizontal

   (d) Xreg stretched vertical

   In the current version of *Floyd* only the orientations `horizontal` and `vertical` are supported. The system could be further extended to cover more complex reflections and rotations.

6. Initialize the various subsystems. In particular pass all of the relevant connection data to the spatial reasoning subsystem.

7. Request that the spatial reasoning subsystem generate the first set of placement alternatives and place them into the working memory.

6.10.3 Planning

In order to perform a design task a plan that describes how the task is to be achieved must first be constructed. A strategy of *constraint based planning* [Stefik, 1980] has been adopted in which plans are made by building constraints that apply to objects and activities within the system. For instance if a linear array is recognised, an ordering constraint is generated that specifies the next module to be placed. Array elements are usually placed in sequence before other less regular elements are inserted into the design. Similarly it is possible to recognise other commonly occurring structures (data paths, arrays) and formulate plans that place them in an appropriate manner.

The purpose of the planning task is to reduce the amount of searching that must be performed by the system to design a floor plan. The planning task rules recognize particular structures. Special purpose plans are then used guide the realization of these structures in the floorplan. This results in higher quality designs and a reduced amount of search through alternate designs. Note that unlike IF [Nixon, 1984] the program does not *necessarily* require prior knowledge of how to floorplan a particular structure: such a plan does however have the effect of improving the efficiency of the design process and result.

6.10.4 Wait for Placements

A simple task that simply waits until the spatial reasoning subsystem has completed the generation of a new set of placements following the incorporation of another module into the design.

6.10.5 Placement Filtering

Whenever a new module is accepted into the design, a number of new placements are generated, and a number of existing placements are invalidated. This initial operation performed by this task is to remove the invalidated placements. It then proceeds to use a number of other criteria to invalidate further placements. This form of "pruning" reduces at an early stage the number of placements that need to be considered in the design process. The criteria used to perform this pruning are:

1. If a placement requires that a port lie on a particular side, and the other end of the connection to that port is to a port on an incompatible side owing to other constraints then invalidate the placement.
2. If a placement requires that a connection be made to a side that is being used to build an array (Figure 6.35(a)), then invalidate the placement.
3. If a placement requires that port that connects outside an array, and a port that connects inside and array, both lie on the same side of a module (Figure 6.35(b)), then invalidate the placement.

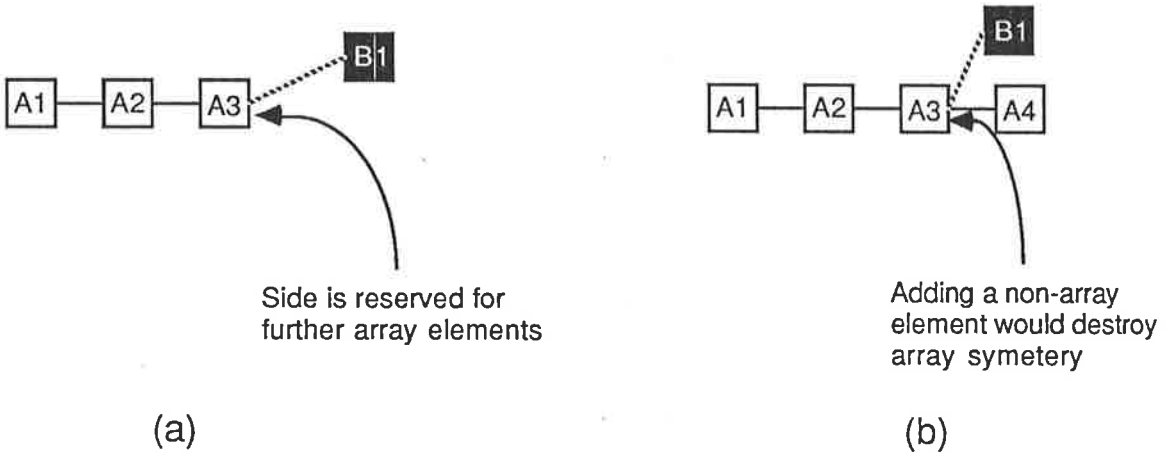

Figure 6.35: Pruning of placements based on (a) previous constraints (b) array under constructions and (c) array already constructed.

6.10.6 Module Selection

In an earlier version of the program [Dickinson, 1986a] all of the valid placements for each of the unplaced modules were evaluated. The best placement was then accepted into the design. This process was found to be inefficient as there are of course many placements to be evaluated. Also, further discussions with designers suggested that it was possible to select a particular module for placement consideration rather than consider all the modules all of the time. The following criteria for deciding which module should be considered for placement were developed. Note that they do require that all of the placements for all of the modules be known, however the spatial reasoning subsystem provides this efficiently. Each of the criteria is applied in turn. If at any stage a single module meets the criteria presented so far, it is selected as the next module. If after any stage there are no candidate modules remaining then an arbitrary choice is made from those remaining after the previous stage. Similarly an arbitrary choice is made if more than one candidate remains after application of all the criteria.

1. Select any module with the fewest available placements. Such a module is the currently most constrained, and should be placed as early as possible.

2. Select any module that is connected to the exterior of the floorplan. This encourages the meeting of external global design constraints.

3. Select any module that is being used in a current plan. This reduces the difficulty in fulfilling the plan.

4. Select the module with the largest number of connections to other modules. This module is heavily constrained, and the opportunity presented by any current placements should be used .

5. Select the module with the largest area. Early placement of such a module is less likely to disrupt area planning.

6.10.7 Placement Selection

Once the module placement task has selected a module as the next to be placed, the placement selection task is activated.

Firstly each placement for a module is next paired with an alternative implementation (Section 6.6) to form a *configuration*. Each configuration then represents a way in which the module may be inserted into the existing design.

The purpose of the placement selection task is to select a particular configuration (and hence placement) to act as a plan for inserting the nominated module into the design. The chosen configuration will describe the position, orientation and implementation of the module.

The task of evaluating the various configurations in order to rank them in terms of desirability is not a simple one. In particular the use of some linear numerical measure of "goodness" is inappropriate. The configuration factors that are associated with particular properties of implementations (Section 6.6) are inherently *uncertain* and it is thus appropriate to treat the evaluation problem as one of processing *uncertainties* rather than simple "scores". To this end the configuration evaluation task is set up in a form that allows the application of techniques similar to those developed for Mycin [Shortcliffe, 1976].

Each configuration is treated as a *hypothesis* that it represents the best plan for the incorporation of the module into the design. Thus the set of configurations may be regarded as a set of competing hypotheses. This is analogous to Mycin's model of competing hypotheses, each representing a diagnosis. The task of selecting the best configuration may then proceed by noting pieces of evidence that confirm or disconfirm each associated hypothesis.

In the case of a configuration, such evidence is obtained by noting the presence of particular floorplan structures and relating these to the properties contained in the implementation type of the configuration. Each such property has associated with it a configuration factor that describes its desirability or otherwise for that particular implementation type. Thus the CF in fact may be regarded as representing an amount of confirming or disconfirming evidence for the hypothesis represented by the configuration.

Typically for a particular configuration a number of structures will be recognized and the associated CF's must be combined into a total CF for the configuration. This is analogous to the situation in Mycin where the various confidence factors associated with evidence are combined to form a total confidence factor for the hypothesis. Once all of the available evidence has been collected, the hypotheses with the highest

total confidence factor is nominated as that most likely to be the correct diagnosis. Similarly in *Floyd* the configuration (hypothesis) with the highest total configuration factor is selected as that to be used in the design. The similar nature of the two situations suggests that it is appropriate to use the same combining functions as Mycin for the combination of configuration factors (Section 6.3.2.6).

The mechanism for recognizing relevant structures and applying the appropriate configuration factors is well matched to the rule based implementation of the design manager. When the placement selection task is initially activated, the proposed configurations for the module are used to generate a set of *provisional* constraints labeled with the name of the configuration who's acceptance would cause their realization. These constraints are propagated throughout the design in order that the global effects of the configuration be visible during the evaluation process.

Once all provisional constraints have been propagated, the rules responsible for recognition of the various structures listed in the class subsystem become active. These examine the design globally for any effects of the configuration. Not only do they evaluate the effect of the configuration on the module being placed, they also examine other modules in the design to evaluate how they will be effected by the adoption of the configuration. As each of these rules fires a CF is recovered from the class subsystem and combined with the current total CF for the configuration.

The issue of the suitability of the space available in the placement for the particular configuration may be presented in such a way as to fit the configuration factor scheme. The spatial reasoning subsystem is requested to supply an estimation of the height and width of the space available for the configuration. If either the height and width required by the configuration exceed the available amount then it is desirable to discourage the use of that implementation. This may be done by using the following equations to generate a type of CF that represents the desirability of the fit. These CF's may be combined in the usual way to contribute to the total CF for a configuration.

$$CF_{horizontal} = MIN\left(\frac{Width_{available} - Width_{required}}{Width_{available} + Width_{required}}, 0\right)$$

$$CF_{vertical} = MIN\left(\frac{Height_{available} - Height_{required}}{Height_{available} + Height_{required}}, 0\right)$$

Note that the functions are asymptotic to -1 (undesirable) as the fit grows worse, but can grow no greater than 0 if the fit is adequate or even generous. Thus sufficient

space for a module does not encourage its acceptance. Rather the converse situation, that there is insufficient space, is used to discourage the selection of the configuration. This is because (particularly early in design) often plenty of space, but this is an artifact of the incompleteness of the design, rather than the quality of the particular configuration. Lack of space however is not ambiguous.

Once all of the relevant rules in the task have fired, the configurations are sorted by total CF. The configuration with the highest CF is then recommended as the most suitable. If there is more than one, an arbitrary choice is made unless the configurations share the same placement but have different orientations. In this case there is insufficient information available to orient the module in placement, placement is recommended but the selection of an orientation is deferred.

6.10.8 Placement Acceptance

The placement acceptance task is activated when a configuration for a module has been recommended. It firstly changes the status of the constraints related to the configuration from *provisional* to *accepted.* It then removes all of the remaining provisional constraints as they are now associated with invalid configurations. The task then passes the placement name and the width and height of the configuration to the spatial reasoning subsystem which incorporates the module into the rectangular graph and updates the set of available placements to reflect the changed design. If the configuration was recommended without an orientation, the rectangular graph representation is led to believe that the module has $width = height = MAX(width, height)$. This means that size calculations in the RG will result in the worst case available space results.

6.10.9 Constraint Propagation

The constraint propagation task is comprised of a number rules that note the presence of a particular constraint and proceed to generate new constraints that must come into effect. For instance (Figure 6.36):

```
IF there exists a constraint on a port to lie on a particular side
THEN generate another constraint on a connected port to lie on
an opposite side.
```

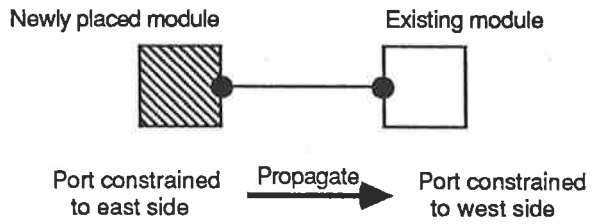

Figure 6.36: Propagation of a port side assignment constraint from a newly placed module to an existing module.

6.10.10 Deferred Tasks

This task examines previously deferred tasks to see if they have become sufficiently constrained to execute. For example of the spatial reasoning subsystem may note during the sizing of a newly placement that the path limiting the available size of the floorplan contains an unoriented module. It may then recommend to the design manager that the module be oriented such that it minimizes the size in the limiting direction. This additional degree of constraint removes the ambiguity that led to the deferring of the orientation, and the task proceeds to accept the designated orientation and remove any invalidated constraints.

6.10.11 Planarization Fault Handling

If at any time during the design the spatial reasoning subsystem does not generate any placements for an as yet unplaced module, a planarization fault has occurred. This is a result of there being no face in the RG in all of the required connections for a module are available. In this case this task is activated in order to consult with the spatial reasoning system to decide on which connection to the module should be temporarily dropped to enable design to continue. The task basically removes the smallest connections first, continuing till the subsystem reports that it has found a placement. The insertion of the removed connections is set up as a deferred route completion to be activated once the all of the modules have been placed in the design.

6.10.12 Route Completion

The connections dropped as a result of the planarization fault handling are required to be inserted in such a way as the minimize the amount of additional area and connection length required. To this end the spatial reasoning subsystem is required to search out possible shortest paths for the insertion of the route. This information is returned to the design manager. Here it is combined with classes knowledge of the difficulty of routing through floorplan module implementations in order to suggest a set of possible final paths.

At present the user is simply presented with a series of alternative possible paths as part of the final design. The path is not used to actually place the route into the floorplan for several reasons:

1. There are typically a number of apparently suitable paths for the route. The routes are of course essential in the final design, but they do not form a major part of the topology of the floorplan. It is often more appropriate to delay implementing the routes till more is known about the *actual* rather than predicted implementations of the modules.

2. Providing the user with suggested paths is sufficient to guide design and perform floorplan evaluation. If the non-planarity implied by the use of route paths is a problem, this will be made apparent to the user by the recommended route paths.

3. The method used for completion of such routes depends somewhat on the exact style of assembly that is to be used in the bottom up composition process. It may well be more appropriate to complete the routing according to the specified paths at that time.

As suggested in Section 5.5.6 the insertion of these paths is best deferred till the construction phases, thus maintaining technology independence in the planning phase.

6.10.13 Solve

The solve task simply carries out the following functions in sequence:

1. Interacts with the paths subsystem and the spatial reasoning subsystem to complete the processing of the RG and build up a set of linear inequalities describing the floorplan.

2. Invokes the simplex subsystem on the linear inequalities in order to obtain values for the floorplan variables.

3. Records the details of the floorplan design in an output file.

4. Passes the floorplan variables along to the sketching subsystem for display to the user.

6.11 Implementation

The variety in the knowledge representations and reasoning methods employed in *Floyd* imply that the implementation language or languages must meet a number of requirements:

1. There should be a common and simple means of interfacing the various modules so that data and control may be efficiently passed between them.

2. There must be support for a production system. This consists of a rule specification language, working memory and rule interpreter.

3. There should be support for general symbolic computation. This is based around the central place of symbols (names) in the language. Dynamic symbolic manipulation facilities include:

   (a) Creation of symbols.

   (b) Creation of groupings of symbols that represent "chunks" of information.

   (c) Creation of associations between information and symbols that may be used as labels for referring to that information.

   (d) Automatic removal of discarded symbols.

4. The languages and environments should be widely available for ease of development and portability.

The justification for these requirements will become apparent in the remainder of this section. In addition of course it is preferable that the language or languages be commonly available for reasons of portability.

The intersection of the requirements for symbolic processing, portability and generality suggest the use of the language Lisp [Winston, 1984]. Lisp is one of the earliest and widely implemented symbolic languages. It provides all of the necessary basic facilities and was available to the author. In addition its "macro" construct (Section 6.11.2.4) has encouraged the development of higher level, more task specific facilities that fulfill the other requirements. Use of these packages has the advantage of ensuring simple interfacing to code written in basic Lisp as the packages run in the same environment and are themselves written in Lisp. The Franz Lisp [Foderaro & Sklower, 1983] implementation was chosen because of its wide availability on machines running the Unix [Ritchie & Thompson, 1974] operating system.

A number of Lisp based packages have developed that include production system facilities including KEE [Intellicorp, 1985], LOOPS [Bobrow & Stefik, 1983], Prudence/Describe [Dickinson, 1984a; Joseph et al., 1986] and OPS5 [Forgy, 1981]. KEE and Prudence/Describe are not easily available. LOOPS and OPS5 are quite widely distributed, however LOOPS is a large package of which production system support is only a portion. OPS5 on the other hand provides only the support required for production systems and has a sophisticated rule compilation and conflict resolution implementation [Forgy, 1982]. On this basis OPS5 was selected as the most appropriate language for the construction of the rule based portion of *Floyd*.

Lisp has been used in *Floyd* for the bulk of the procedural code. The only exceptions are portions of the Simplex and Sketcher subsystems. Most implementations of Lisp are optimized for numerical computation due to the large amount of run-time checking that must be made on the typically untyped "variables". This became apparent in an early version of *Floyd* and the tableau manipulation code of the simplex subsystem was re-written in the "C" language [Ritchie & Kernighan, 1978]. This resulted in a factor of ten decrease in the time taken for solution of the floorplan inequalities.

Early research on *Floyd* was carried out on Lisp Machine workstations. [Weinreb & Moon, 1981a]. This code relied on the "windows" package written in the "Flavors" object oriented programming package. Given the graphic nature of the task, it was desirable to preserve some of the benefits of the versatile Lisp Machine user interface under the more widely used Franz Lisp environment. To this end the Sketcher sub-

system incorporates an basic version of the Lisp Machine Windows package written in Flavors running on Franz Lisp. This package is not host dependent and supports several terminal types.

The remainder of this section describes some of the more interesting aspects of the implementations of the various sections of the system and illustrates the matching of the various languages with the application.

6.11.1 Design Manager Implementation

The design manager is completely written in the OPS5 [Forgy, 1981] production system language. The rules are executed by an OPS5 interpreter written in Lisp. In the following subsections the various components of an OPS5 production system are described in terms of their use in the implementation of the design manager.

6.11.1.1 The Working Memory

The *working memory* (WM) is a data storage area comprised of slots that may each hold a *working memory element* (WME). Any rule that fires may create, alter, or remove these elements in order to change the design state. In this application a WME may be regarded as having a *type* and a number of named *fields*. The type of a WME may not be changed, but the contents of each of its fields may be empty or associated with some symbol.

Each type of WME is used to represent some portion of the design state and may be classified into one of the following groups:

1. *Component Types.* These are used represent actual items in the design such as modules, sides, connections and ports.

2. *Concept Types.* These are used to represent items in the design that have no direct physical realization in the floorplan. Examples include placements, implementations and similar types that represent design concepts rather than components.

3. *Control Types.* These are used to exert control over the execution of rules. There are only two types, tasks and subtasks.

4. *Constraint Types.* These may represent relationships between elements of types listed in any of the above groups. A constraint might fix a port to the side of a module. Another might constrain two tasks to be activated in a particular sequence.

6.11.1.2 The Rule Set

An OPS5 rule is comprised of a left hand side (LHS) which is the antecedent, and a right hand side (RHS) which is the action. Basically the LHS is a list of patterns that are matched against the elements in the WM. OPS5 provides a sophisticated pattern matching facility in which variables in the patterns are matched against the symbols in the fields of the matched working memory elements.

The RHS is basically a list of sequential actions. Most actions either add, remove or alter working memory elements. The working memory elements and the variables matched in the LHS may be accessed in the RHS Additionally however a RHS may call other Lisp procedures to perform external functions and input/output.

Figure 6.11.1.2 illustrates a simple design manager rule. The rule is responsible for incorporating the quality of the fit of the configuration in the floorplan into the total CF for the configuration. The name of the rule is `placement-size-fit` and it is contained in the `evaluate-configuration` subtask (the name of the task is prepended to the rule name by convention).

The first LHS pattern only matches if there is an active subtask with the name `evaluate-configuration`. This ensures that the rule will only fire if the subtask is indeed in existence. If so, variable denoted by `<cn>` is bound to the value of the `arg1` field of the subtask. This field is a subtask parameter that in this case is used as the name of a configuration that the subtask has been set up to evaluate.

The second LHS pattern matches the WME that records the details of the configuration which has a name `<cn>`. The variables `<place1>` and `<imp>` are respectively bound to the name of the placement and implementation associated with the configuration named `<cn>`.

The third LHS pattern matches the WME that records the details of the implementation names `<imp>`. The variable `<impt>` is bound to the implementation type of the implementation. The `<w>` and `<h>` variables are bound to the width and height

```
(p evaluate-configuration::placement-size-fit

   (subtask ^name evaluate-configuration ^arg1 <cn>)
   (configuration ^name <cn>
                  ^placement-name <place1>
                  ^implementation-name <imp>)
   (implementation ^name <imp>
                   ^type-name <impt>
                   ^width <w>
                   ^height <h>
                   ^orientation <or>)
   (constraint ^type placement-size-constraint
               ^width <p-w> ^height <p-h>)

   -->

   (call CF-calculate-size-fit <cn> <p-w> <w> <p-h> <h>))
```

Figure 6.37: A simple design manager rule for accounting for placement size.

estimated to be required for the implementation of the module.

The fourth LHS pattern any constraint produced by the spatial reasoning subsystem that describes the limits of the available space for the module in the floorplan. The `<p-w>` and `<p-h>` variables are bound to the width and height estimated to be available for the implementation of the module. Note that any such constraint named `placement-size-constraint` will be matched. This is allowed because there is only one such constraint ever placed in WM at a time: that for the current placement being examined.

If all four patterns are matched, and the rule is selected to fire (Section 6.11.1.3) then the RHS `call` action is carried out. This has the effect of calling an external (not in the design manager) procedure for calculating the new total CF given the fit parameters as described in Section 6.10.7. The passed parameters are of course the values bound onto the variables `<w>`, `<h>`, `<p-w>` and `<p-h>`.

6.11.1.3 The Rule Interpreter

The interpretation of the OPS5 rules is carried out by an *inference engine.* This is responsible for matching, selecting and executing the rules: the *recognize-act cycle.* This is described in detail in [Brownston et al., 1985] and will only be summarised here.

For each cycle of rule firing, the inference engine collects all of the rules that successfully match elements in working memory. Each match is termed an *instanciation* as the variables are instanciated with the matched values. If this set is empty, then the system halts as it can proceed no further. If there is only one match, then that rule may be executed. If however more than one rule, a *conflict resolution* strategy is used to select a single rule for execution from the *conflict set.* Each step of the strategy is used to reduce the number of instanciations in the conflict set, until one dominates or the last step is reached. The steps are:

1. *Refraction.* All previous instanciations of a rule are deleted from the conflict set. Thus is a rule has been fired with particular data elements matching the LHS, it cannot fire again unless they have been altered. Clearly this is useful in ensuring that the system does not cycle on one rule/data match.

2. *MEA.* If the OPS5 MEA (means-ends analysis) strategy is being used, this step examines the *first* condition of each rule. Only those rules with the most recent first condition element are retained in the conflict set. If, as in *Floyd*, the first condition of most rules is a task or subtask element, then this ensures that only rules in the most recently created task or subtask are executed. This provides a useful focus of control in rule firing.

3. *Recency.* The instanciations are ordered based on the recency of the working memory elements matched. Only those rules with the most recent elements are retained in the set. This ensures that the system work on the most recent data.

4. *Specificity.* The rules are instanciations are ordered based on the number of tests in the LHS of the rules. Only those rules with most tests are retained in the conflict set. Thus only the most specific rules are retained.

5. *Arbitrary.* If by this stage no single instanciation has dominated, and arbitrary decision is made as to which instanciation in the conflict set should be fired.

In *Floyd* the properties of the conflict resolution strategy is used to provide task control in addition to the simple firing of rules. Firstly, the MEA strategy is used to ensure that all of the rules ready to fire in a task do in fact fire before another task is undertaken. Secondly, the recency criteria may be used to provide sequencing of subtasks by creating them in the reverse order to that desired for execution. Thirdly, rules with a single subtask condition element and no other may be used as "cleanup" rules. These will only fire when there are no more rules in the subtask ready to fire, and may initiate an overall cleanup of the subtask and deletion of the subtask element itself.

6.11.2 Subsystem Implementations

Other then the design manager, the bulk of the system code is written in Lisp. The symbolic manipulation facilities of the language are valuable in applications required for a number of reasons covered in the remainder of this section.

6.11.2.1 Symbols

A Lisp *symbol* may be regarded as a unique "name" (such as `mike`, `53`, `apple` or `shift-register`) with which three things may be concurrently associated. Firstly, a symbol may have a *value* that is accessed when it is evaluated. This may for instance be another symbol. A number is a special case of a symbol. Hence a "name" may be associated with a number and thus symbols may be used in a similar fashion to a variable in a conventional language: to hold a number value. However, since a symbol's value may also be another symbol, or even a list of symbols (described in the following section) then the structure is more general.

In addition to a *value*, a symbol may have a *function* associated with it. Thus the symbol may also be used as a "name" for invoking a procedural entity.

The third entity that is associated with a symbol is it's property list ("plist"). This is a list that may be used to describe various properties of the symbol. It is arranges as a series of pairs: the first of each pair is the name of the property, the second the value associated with that property. For example a symbol `george` might have a property `age` with a value `32`.

All three items associated with a symbol are used in *Floyd*. Firstly, symbols are often

used as general variables to hold values during calculations. For instance the code fragment:

```
(loop for module in all-modules doing (print module))
```

has the effect of setting the value of the symbol `module` in turn to each of the elements of the list that is the value of the symbol `all-modules`. As each new value is set, it is printed out. This fragment also illustrates the use of symbols as the names of functions: *loop* and *print* may be regarded as functions (though in fact they may also be "macros" as described in the next section). Note that `tt` for, `in` and `doing` are all simply symbols that serve to make up a syntax—even if they have values they are not used here.

Property lists are used extensively in the implementation as they provide a useful method of associating information with a particular named item. For instance each time that the spatial reasoning subsystem creates a new placement for a module, it generates a unique symbol to act as a name for it: `p4764` for instance. It then places all information relevant to that placement on the property list associated with that symbol:

1. The name of the node that is the target of the placement.
2. The list of all the nodes that node must eventually connect to.
3. The list of nodes and their sides that comprise the face of the the placement.
4. The space available in the placement.
5. The set of triangular RG faces that would result from accepting this placement into the RG.

The name of the placement may be passed to the design manager for processing. The design manager does not require the property list, but should it request that the placement be accepted, it simply informs the spatial reasoning subsystem of the name of the placement. The subsystem may then simply examine the property list associated with that name in order to access all of the relevant information. This style of programming ensures that all of the data associated with a named item such as a placement or node is always directly attached to the item and easily accessible.

6.11.2.2 Lists

Lisp provides the *list* as the basic data structure for sets of symbols. A list is simply an ordered sequence of symbols or other lists. Examples include:

```
(mike greg alex 234 mike)
(trees (blue green) turkeys)
```

Tree structures may be created by nesting lists as shown in the second example. *Circular* lists may be created that have neither beginning or end—the last element is followed by a link to the first. A number of basic operations are provided in the language for creating, joining, examining and dismembering lists.

Lists are used in virtually all of the code. Many of the procedures involved with manipulating the RG involve traversing circular lists of nodes that are used to represent faces. Concurrently the procedures build further lists that represent all of the nodes in the face with some common feature: sides available for connection for instance. The circular lists that are used to represent faces are themselves members of trees and subtrees also represented as lists.

6.11.2.3 Automatic Memory Management

Unlike most languages conventional languages (such as Pascal, Fortran and C), Lisp provides for the "transparent" allocation and deallocation of memory. Whenever a list is created or added to, the memory required is automatically allocated. In turn, whenever a list is no longer in a position to be referenced (there are no paths from any symbols to the list) then the memory is automatically reclaimed. This is particularly useful in *Floyd*as objects such as placements are being constantly created and abandoned.

6.11.2.4 Macros

In order to facilitate the construction of application specific languages on top of Lisp, the language provides a *macro* facility. This is provides a means of taking a piece of lisp in one form, and processing it with Lisp code to produce another form for final evaluation.

The macro facility is used in *Floyd* in order to perform design description input and to implement the classes subsystem.

The input format to the floorplanner is of the form is comprised of a number of macro statements such as:

```
(connection in1 bit4add-1 ioout io1)
```

which defines a connection between the `in1` port on module instance `bit4add-1` and the `ioout` port on module instance `io1`. The `connection` macro simply rewrites this into the form:

```
(make connection ^port1 in1 ^module1 bit4add-1
                 ^port2 ioout ^module2 io1)
```

which is then executed, making a placing an element in the working memory that represent the connection and may be accessed by the rule base.

The classes' `define-implementation-type` statement (Section 6.6) is actually a complex macro. The macro is invoked with the name, inheritance list and collection of properties all as parameters. These are then processed into OPS5 actions for inserting three working memory elements into the data base: one for undirected properties, and one each for horizontal and vertical properties. For instance the definition:

```
(define-implementation-type simple linear-array
  ()
  (input-output-adjacency vertical -0.5)
  (input-output-adjacency horizontal -0.7)
```

would result in elements:

```
(implementation-type ^name simple
                     ^class-membership linear-array
                     ^direction-of-interest nil)

(implementation-type ^name simple
                     ^class-membership linear-array
                     ^direction-of-interest vertical
                     ^input-output-adjacency -0.5)

(implementation-type ^name simple
                     ^class-membership linear-array
                     ^direction-of-interest horizontal
                     ^input-output-adjacency -0.7)
```

Once initialization is complete, these working memory elements serve base of implementation type knowledge that may be accessed by the relevant rules.

6.11.3 Subsystem Interface Mechanisms

The basic form of intermodule communication in the system is between the design manager and the other subsystems. The design manager passes data and control to the subsystems by means of simple lisp function calls augmented by the OPS5 parameter passing mechanism [Forgy, 1981]. The subsystems communicate data to the design manager by constructing working memory elements containing the data and calling utility routines that insert them into the working memory. They may then at will pass control back to the design manager by simplying allowing the initiating function call to complete.

Another form of subsystem communication uses the binding of values to global symbols. The lists representing the floorplan inequalities are accessed in this manner for instance.

A third method has been described previously: the attachment of data to the property list of a symbol being used as the name of an object. The name is passed between subsystems, and the data accessed by simply consulting the relevant property.

6.12 Limitations

The objective of the prototype floorplanner described in this chapter was to demonstrate the feasibility of partitioning floorplanning knowledge into a set of suitable representations, and apply that knowledge successfully. As will be shown in examples in the next chapter, this purpose was in fact achieved. Due to the great deal of knowledge involved in the floorplanning task however, practical limitations on the period of this research has meant that only a portion of the total knowledge required for expert floorplanning has been put in place. In particular, the classes and planning aspects of the prototype could be extended considerably. There are no inherent limitations placed on such extension by the structure that has been outlined for *Floyd*. It has been proposed that the prototype be extended into a fully viable design aid in the commercial environment as described elsewhere [Dickinson et al., 1987].

One limitation of the OPS5 environment is that it does not provide a mechanism for *explaining* the actions of the expert system it implements. The facilities for tracing lines of reasoning are primitive: simply the listing of the rule firing sequence which is primarily useful for debugging. Although in other ways OPS5 provided an excellent facility for the expression of the rule base, a language with an explanation capability such as that of EMYCIN [Buchanan & Shortcliffe, 1984] would be useful. This would simplify understanding of the operation of the floorplanner in a particular situation, and hence aid the addition of new knowledge into the system.

The problems that arose in backtracking led to the abandoning of that approach (Section 6.10.1). There can be no doubt however that designers do in fact backtrack at times in the development of an optimal design. The simple approach of purely *chronological* backtracking [Winston, 1984] was found to be inappropriate as it may lead to another problem similar to that being backtracked from. Considerable further work would be required in order to obtain an understanding of how designers use the knowledge gained in the forward design path to guide backtracking and thus efficiently achieve an improved solution. As noted earlier however, intelligent backtracking is a desirable, but not necessary component of the design strategy.

6.13 Summary

Modelling the form of a structural VLSI design requires that the designer be provided with feedback that describes the realization of the structure in the silicon plane. This may be performed by an automatic floorplanner.

Existing approaches to automatic floorplanning generally involve the algorithmic enumeration of a large number of alternative possible floorplans. The large search space implied is then pruned by heuristic methods in order that the designer be able to select an optimal design. These methods are inefficient as they delay application of domain specific knowledge till *after* the very large number of designs have been created, rather than using it to intelligently guide the creation of a single good result.

Expert systems approaches to floorplanning have either shared the enumeration problems of algorithmic floorplanners, or only represented and applied domain knowledge in an unstructured manner. There has been little attention paid to the representation of uncertain design knowledge such as that used in applying bottom-up influence in

top-down design. The structural evaluation task being examined here requires this as floorplans must be generated in a top down manner in parallel the development of the structural description.

The floorplanner that has been described here is based around the partitioning of designer knowledge into distinct categories, and the use of both special purpose and general knowledge representations for their explicit representation. The three knowledge representations are:

1. *The Rectangular Graph.* A set of data structures and algorithms that as a whole emulate the designer's ability to reason with interconnected rectangles in a plane. This category of knowledge may be regarded as an innate, visual ability of the designer.

2. *Classes.* A store of inexact knowledge about the alternate implementations of floorplan modules. Access to this knowledge assists the floorplanner in designing hierarchical floorplans in a top-down style. The use of configuration factors (CF's) allows alternate design options to be treated as competing hypotheses. The simple hierarchical structure of the classes representation encourages designers to incrementally add further knowledge to the system.

3. *The Production System.* The meta-knowledge required to apply domain specific knowledge is may be expressed as production rules. The "recognize-act" structure of production rules is well matched to recognition of occurrences in the design and the resulting execution of an appropriate activity.

Direct comparison with existing floorplanners has been omitted as there are no distinct parameters with which such a comparison may be made. However, comparison of the catagories of knowledge and reasoning techniques used suggest that the floorplanner described in this chapter has the potential to generate floorplans with a greater range of designer expertise than those described in Section 6.2..3.

The Lisp/OPS5 implementation of a prototype floorplanner has been described. The evaluation of the success of such a design aid is complex as it must be viewed in the context of the entire planning phase of a design. In the following chapter the success of the floorplanner is evaluated in the context of the design procedure adopted in the creation of a large VLSI system.

Chapter 7

A Structural Design Case Study

7.1 Introduction

In this chapter the results of the previously described research are demonstrated by way of a design case study. The nature of the problem, and the lack of comparable research, requires that the effectiveness of the methods and design tools be illustrated by such a demonstration.

This case study is directed at demonstrating the utility of the following techniques:

1. The specification of structure and function at a high level of of abstraction.
2. The used of software structural design techniques as an aid to partitioning a VLSI system.
3. The successive refinement of intermodule communication into detailed timing specifications.
4. The use of functional modelling in the verification of the decomposition.
5. The use of modelling in evaluating alternative designs by:
   (a) The profiling of the functional description to aid in the evaluation of the coupling and cohesion of the partitioning.
   (b) The generation of structured floorplans to aid in the evaluation of the geometric suitability of the partitioning.

7.2 A Design Example

The basis for the system design chosen for this exercise is a floating point adder (FPA) chip designed by Zyner [Zyner, 1988]. The selection of this project was based on the following criteria:

1. The design is of a reasonable complexity ($\approx 10,000$ devices). Although of not true VLSI scale, this was regarded as sufficiently complex for the demonstration purposes.
2. The design was expected to offer a challenging combination of irregular and regular structures. Floating point operations tend to be irregular at the higher levels owing to the asymmetric nature of the algorithms used. They do however exhibit regularity in the lower levels as the basic sub-operations include simple adding and shifting.
3. The functional architecture incorporates some novel algorithmic techniques, illustrating the need for structural specification of architectures as opposed to the fixed architecture paradigms of typical silicon compilers.
4. The system was still in the functional specification phase.
5. The designer was available for consultation and had a requirement for more advanced specification and modelling aids.

Although the integration of the structural design aids into a system based on automatic assembly [Dickinson et al., 1987] has not yet been completed, it was felt that the partitioning exercise would still be beneficial as a basis for manual design. The design procedure for the 32-bit FPA described in this chapter is an extension of an the original 8-bit specification created by Zyner [Zyner, 1988]. The full 32 bit system was later implemented in symbolic layout using the *Pink* specification and modelling results as an informal guide.

Rather than attempt to describe the approach in terms of the entire FPA design, a single branch of the hierarchy was chosen as being representative of the techniques involved, and will be described in the remainder of this chapter. This description of the design example concentrates on structural issues in the design of the upper levels of the hierarchical partitioning, and functional issues farther down the hierarchy.

Further details of the FPA algorithm, specification and layout not directly relevant to the partitioning may be found elsewhere [Zyner, 1988].

The next section provides a functional overview of the FPA system.

7.3 Functional Overview

The FPA adds or subtracts two floating point numbers. Each is represented by an 8 bit exponent, a 23 bit mantissa and a sign bit. The two numbers are added or subtracted as required, the result being represented in the same format as the arguments. Overflow and underflow are indicated by way of two additional signal lines.

An overview of the basic algorithm implemented by the FPA is illustrated by the flow chart in Figure 7.1. More detailed descriptions of the various steps will be introduced as they are required. The design presented here is interesting in its use of parallelism in the implementation of the basic algorithm. In particular, at the same time that the aligned mantissas are added together, the most significant bit position is being calculated in order that it be available for mantissa normalization at the same time as the result of the addition stabilizes. Additional parallelism is achieved by simultaneously adjusting the result exponent and performing rounding and complementing of the resultant mantissa.

7.4 An Initial Partitioning

A simplistic view of the process of refining a design description would suggest that the initial description would be of a single functional module that would be successively refined into simpler modules. The composition of these modules would then be verified as being the same as the initial description. In practice however this is often not the case. Such a simple top level functional module does not provide a significant amount of design information, and may well suffer from being overly complex. Typically in developing the algorithm to be implemented, the designer has already partly decomposed the design into a set of interconnected modules. Hence it is appropriate to represent the highest level of the design as a composition of submodules. These submodules in turn may be implemented as structural or functional modules. Thus

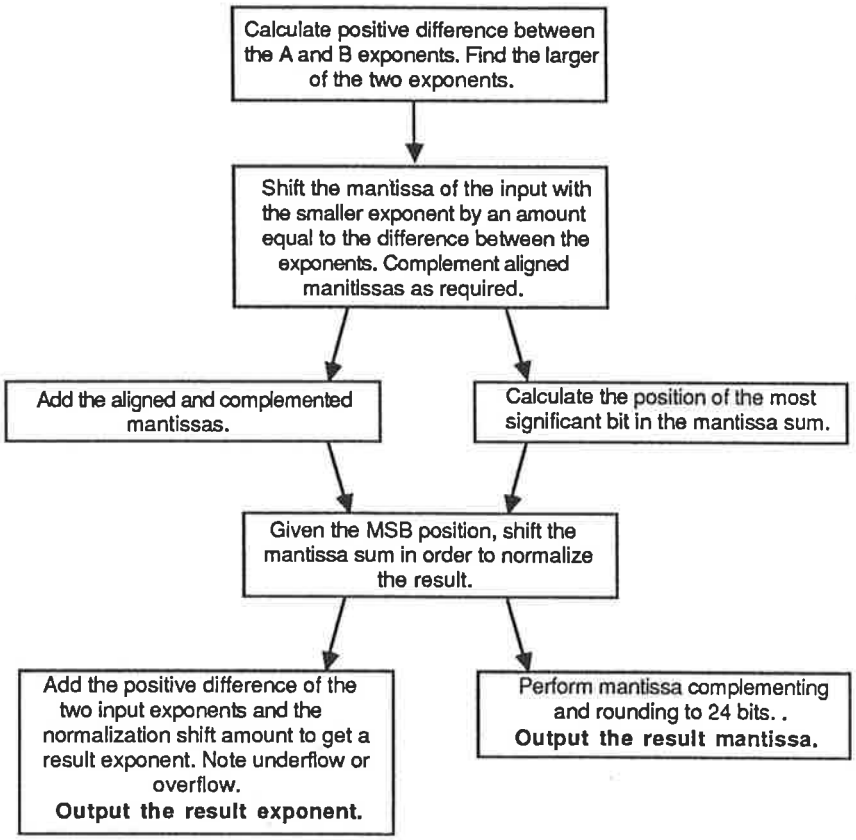

Figure 7.1: Flowchart of the basic FPA algorithm.

in general the designer requires considerable freedom in selecting the initial mixture of structural and functional implementations of modules.

Such a procedure may be easily justified in terms of software design techniques. Programmers do not write a single block of code at the start of a design that will implement the total function, and then refine it by splitting it into smaller sections. Stepwise software refinement dictates that first the problem is decomposed into a set of interacting subproblems. Each of these is treated as a procedure call to an as yet unwritten procedure, and interactions are planned by the specification of parameters to those procedures. This is simply functional decomposition. Similarly it is unreasonable to expect VLSI designers to specify the algorithm as a monolithic entity in the initial specification.

7.4.1 Specification

The initial top level partitioning was based on an informal data flow (Section 2.2.2) approach with afferent, central transform and efferent elements. This partitioning is illustrated by the data flow graph in Figure 7.2. The resultant module hierarchy is shown in Figure 7.3. Note that one of the top level modules (ManAdder) has been specified simply as a functional element, whilst the other two (AlignUnit and Adjuster) have been specified as compositions of functional elements at one level lower.

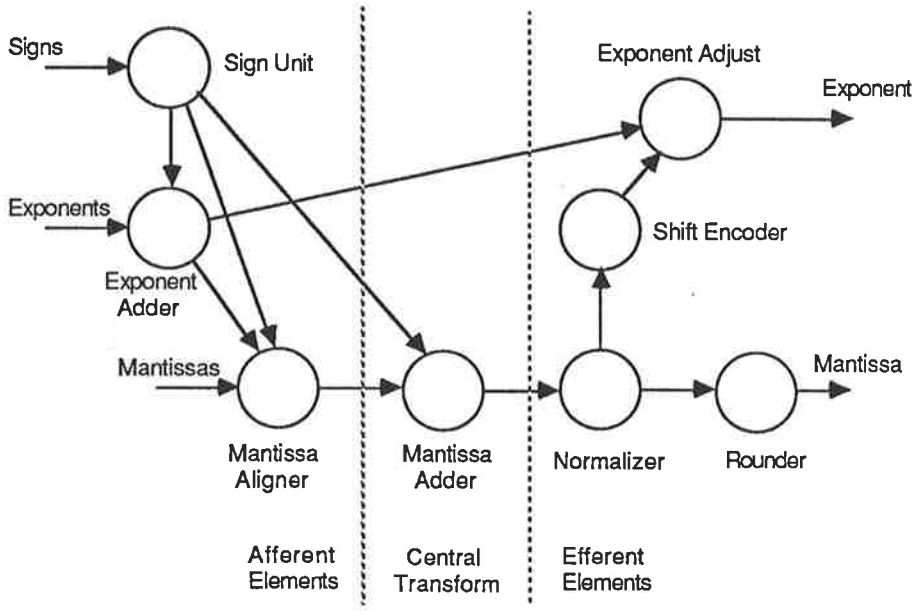

Figure 7.2: Data flow graph of the FPA.

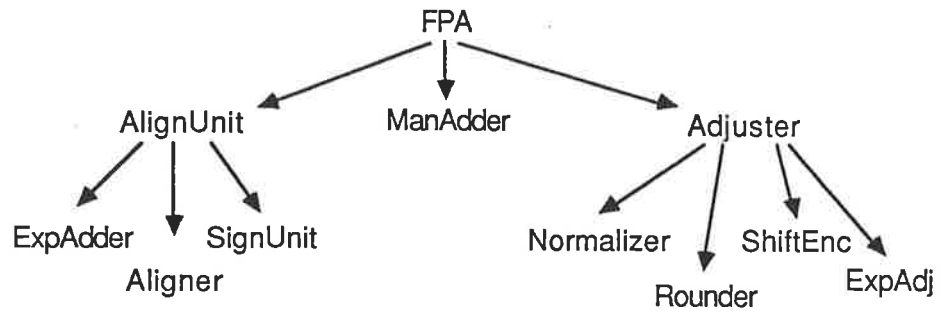

Figure 7.3: Module hierarchy (initial partitioning).

The three top level modules and their compositions/functions are described in the following subsections.

7.4.1.1 The Alignment Unit

The afferent (input) module of the system is the Alignment Unit. This in turn is specified in the initial partitioning as the composition of three submodules:

1. *The Exponent Adder* inputs the two exponents and generates the positive difference between the two. This difference is used as the shift amount for alignment. The module also uses the result of the addition to select the larger of the two exponents to be passed through as the unadjusted result exponent. In the initial specification it is represented by a functional definition.

2. *The Sign Unit* inputs the add/subtract opcode, the carry output of the exponent adder, and the signs of the two floating point operands and generates a number of complementing control signals. In the initial specification it is represented by a functional definition.

3. *The Aligner* uses control signals from the exponent adder to select one of the mantissas for alignment (shifting). The two mantissas are then complemented as directed by the sign unit. In the initial specification it is represented by a functional definition.

The complete Alignment Unit has the *Pink* description:

```
DEFINITION AlignUnit;
STATE
  -- Ports:
  SignA, SignB, Sign, Eop, Opcode, Cout : PortType;
  ExpA, ExpB, Exp : ARRAY [0..7] OF PortType;
  ManA, ManB, ManX, ManY : ARRAY [0..22] OF PortType;
PORTS
  DeclarePort(SignA, BIT, INP); -- The sign of A.
  DeclarePort(SignB, BIT, INP); -- The sign of B.
  DeclarePort(Opcode, BIT, INP); -- The add/subtract opcode.
  DeclarePort(Sign, BIT, OUTP); -- Sign control.
  DeclarePort(Eop, BIT, OUTP); -- Sign control.
  DeclarePort(Cout, BIT, OUTP); -- Complementor control output.
  DeclareBus(ExpA, INP); -- The exponent of A.
  DeclareBus(ExpB, INP); -- The exponent of B.
  DeclareBus(Exp, OUTP); -- The intermediate result exponent.
  DeclareBus(ManA, INP); -- The mantissa of A.
  DeclareBus(ManB, INP); -- The mantissa of B.
```

```
  DeclareBus(ManX, OUTP); -- The aligned X mantissa.
  DeclareBus(ManY, OUTP); -- The aligned Y mantissa.
TOPOLOGY
  MakeInstance(ExpAdder, ExpAdder);
  MakeInstance(SignUnit, SignUnit);
  MakeInstance(Aligner, Aligner);
  --  connection to the AlignUnit input ports:
  Connect(SignA & SignUnit/SignA);
  Connect(SignB & SignUnit/SignB);
  Connect(Opcode & SignUnit/Opcode);
  Connect(ExpA & ExpAdder/ExpA);
  Connect(ExpB & ExpAdder/ExpB);
  Connect(ManA & Aligner/ManA);
  Connect(ManB & Aligner/ManB);
  --  Connection to the AlignUnit output ports:
  Connect(Sign & SignUnit/Sign);
  Connect(Eop & SignUnit/Eop);
  Connect(ManX & Aligner/ManX);
  Connect(ManY & Aligner/ManY);
  Connect(Cout & Aligner/Cout);
  -- Submodule interconnect:
  Connect(SignUnit/Cin & ExpAdder/Cout);
  Connect(SignUnit/X & Aligner/X);
  Connect(SignUnit/Y & Aligner/Y);
  Connect(ExpAdder/Switch & Aligner/Switch);
  Connect(ExpAdder/Shift & Aligner/Shift);
ENDTOPOLOGY
END AlignUnit;
```

7.4.1.2 The Mantissa Adder

The mantissa adder adds the two mantissas. Closely connected to the adder is the most significant bit position finder (MSBPF). This inputs the two mantissas to estimate the position of the most significant bit of the sum in parallel to the add operation. The estimate is refined to an exact position on arrival of the carry out signal from the adder. In the initial specification this module appears as a single functional module rather than a composition.

7.4.1.3 The Adjuster

The efferent (output) module is the Adjuster. This is functionally responsible for adjusting the resulting mantissa and exponent and is specified as a composition of:

1. *The Normalizer* uses the position located by the MSBPF to normalize (shift) the result mantissa. In the initial specification it is represented by a functional definition.

2. *The Rounder* adjusts the result mantissa to allow for overflow conditions. In the initial specification it is represented by a functional definition.

3. *The Shift Encoder* translates the shift distance that was required for normalization into an integer. In the initial specification it is represented by a functional definition.

4. *The Exponent Adjuster* takes the difference between the shift amount and the result exponent from the Exponent Adder in order to generate a final result exponent together with over and underflow indication. In the initial specification it is represented by a functional definition.

The complete Adjuster has the *Pink* description:

```
DEFINITION Adjuster;
STATE
  ManIn : ARRAY [0..22] OF PortType;
  ManR : ARRAY [0..22] OF PortType;
  ExpR, Exp : ARRAY [0..7] OF PortType;
  Sign, Cin, Eop, Overflow, Underflow : PortType;
PORTS
  DeclareBus(ManIn, INP); -- Input mantissa.
  DeclareBus(ManR, OUTP); -- Result mantissa.
  DeclareBus(Exp, INP); -- Input exponent.
  DeclareBus(ExpR, OUTP); -- Result exponent.
  DeclarePort(Sign, BIT, INP); -- Sign control.
  DeclarePort(Eop, BIT, INP); -- Sign control.
  DeclarePort(Cin, BIT, INP); -- Carry input.
  DeclarePort(Overflow, BIT, OUTP); -- Overflow output.
  DeclarePort(Underflow, BIT, OUTP); -- Underflow output.
TOPOLOGY
  MakeInstance(Normalizer, Normalizer);
  MakeInstance(Rounder, Rounder);
  MakeInstance(ShiftEnc, ShiftEnc);
  MakeInstance(ExpAdj, ExpAdj);
  -- Input port connections:
  Connect(Exp & ExpAdj/Exp);
  Connect(ManIn & Normalizer/ManIn);
  Connect(MSBP & Normalizer/MSBP);
```

```
  Connect(Cin & Normalizer/Cin);
  -- Output port connections:
  Connect(ExpR & ExpAdj/ExpOut);
  Connect(Overflow & ExpAdj/Overflow);
  Connect(Underflow & ExpAdj/Underflow);
  Connect(ManR & Rounder/ManR);
  -- Submodule interconnections:
  Connect(Normalize/ManOut & Rounder/ManIn);
  Connect(ShiftEnc/EncAmount & ExpAdj/Shift);
  Connect(Normalize/ShiftAmount & ShiftEnc/ShiftAmount);
ENDTOPOLOGY
END Adjuster;
```

7.4.1.4 The FPA Composition

The Alignment Unit, the Mantissa Adder and the Adjuster are combined to form the top level *Pink* specification of the FPA:

```
DEFINITION FPA;
STATE
  -- Ports:
  SignA, SignB, Opcode, Overflow, Underflow : PortType;
  ExpA, ExpB, ExpR : ARRAY [0..7] OF PortType;
  ManA, ManB, ManR : ARRAY [0..22] OF PortType;
PORTS
  DeclarePort(SignA, BIT, INP); -- The sign of A.
  DeclarePort(SignB, BIT, INP); -- The sign of B.
  DeclarePort(Opcode, BIT, INP); -- The add/subtract opcode.
  DeclarePort(Overflow, BIT, OUTP); -- HI if overflow.
  DeclarePort(Underflow, BIT, OUTP); -- HI if underflow.
  DeclareBus(ExpA, INP); -- The exponent of A.
  DeclareBus(ExpB, INP); -- The exponent of B.
  DeclareBus(ExpR, OUTP); -- The result exponent.
  DeclareBus(ManA, INP); -- The mantissa of A.
  DeclareBus(ManB, INP); -- The mantissa of B.
  DeclareBus(ManR, OUTP);  -- The result mantissa.
TOPOLOGY
  MakeInstance(AlignUnit, AlignUnit);
  MakeInstance(ManAdder, ManAdder);
  MakeInstance(Adjuster, Adjuster);
  -- Connections to FPA input ports:
  Connect(SignA & AlignUnit/SignA);
  Connect(SignB & AlignUnit/SignB);
  Connect(Opcode & AlignUnit/Opcode);
```

```
  Connect(ExpA & AlignUnit/ExpA);
  Connect(ExpB & AlignUnit/ExpB);
  Connect(ManA & AlignUnit/ManA);
  Connect(ManB & AlignUnit/ManB);
  -- Connections to FPA output ports:
  Connect(Overflow & Adjuster/Overflow);
  Connect(Underflow & Adjuster/Underflow);
  Connect(ExpR & Adjuster/ExpR);
  Connect(ManR & Adjuster/ManR);
  -- Submodule interconnect:
  Connect(AlignUnit/Sign & Adjuster/Sign);
  Connect(AlignUnit/Eop & ManAdder/Eop);
  Connect(AlignUnit/Eop & Adjuster/Eop);
  Connect(AlignUnit/Cout & ManAdder/Cin);
  Connect(AlignUnit/ManX & ManAdder/ManX);
  Connect(AlignUnit/ManY & ManAdder/ManY);
  Connect(AlignUnit/Exp & Adjuster/Exp);
  Connect(ManAdder/ManOut & Adjuster/ManIn);
  Connect(ManAdder/MSBP & Adjuster/MSBP);
  Connect(ManAdder/Cout & Adjuster/Cin);
ENDTOPOLOGY
END FPA;
```

7.4.2 Modelling Function

Functional modelling was assisted by the construction of a testing module that could be connected to the completed system. This module wrote a combination of general and specifically chosen (e.g. overflow and underflow generating) vectors to the FPA model. The same module also read the outputs of the FPA model and tested them against those expected.

This exercise resulted in the location of one data synchronization error which involved an unexpected one-cycle delay of data delivery from the exponent adder. If undetected this error would have been fatal to the operation of the FPA.

During a 1024 cycle validation test, data flow and module activity statistics were collected. The significant links in the design all had a saturation approaching 100% as they transfer data on virtually every simulation cycle. The module activation statistics all approached 100% as all modules are activated on each cycle in the design. Variations in both link saturation and module activity may be expected to occur in less parallel designs in which some links/modules are active whilst others

are idly waiting. Thus these figures may be regarded as indicating a high degree of parallelism in the FPA design.

The remaining profiling statistic, volume, showed considerable variation across the design. A summary of relevant statistics are presented as annotations to the data flow graph shown in Figure 7.4. The following inferences may be drawn from the data displayed:

1. The Alignment Unit was comprised of only lightly coupled components with the traffic being of relatively low volume. Much of the traffic simply passes out of the module rather than circulating inside it. This suggests that as a whole the Alignment Unit has low cohesion.
2. The Mantissa Aligner exhibited strong coupling (high link volume) with the Mantissa Adder.
3. The Exponent Adder exhibited strong coupling (high link volume) with the Exponent Adjuster.
4. The Mantissa Adder exhibited strong coupling (high link volume) with the Normalizer.

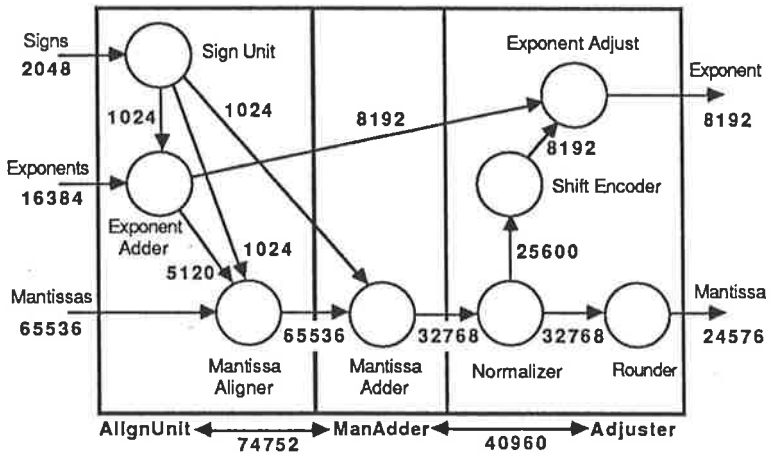

Figure 7.4: Profiling data derived from the FPA model (initial partitioning).

7.4.3 Modelling Form

The result of floorplanning the initial partitioning is shown in Figure 7.5. As a descriptive aid the floorplans have been redrawn with the designer estimated minimum

areas for module implementation superimposed as shaded rectangles. There are a number of points of note:

1. Due to the imposition of the Mantissa Adder, a number of Alignment Unit to Adjuster connections must be routed through the Mantissa Adder or a route module created.
2. The oversized space allocated to the Mantissa Adder is also caused by its position in the centre of the design.
3. The height of the Align Unit is imposed by the Sign Unit: the Exponent Adder and Aligner are not as high. This results in unused space.
4. The imposition of the Shift Encoder between the north side of the Adjuster and the Exponent Adjuster prevents the exponent bus being easily connected to the Exponent Adjuster: completion requires the addition of a route of the exponent bus through the Shift Encoder or the creation of a route module.

A considerable area of the floorplan (35%) consists of unused space.

7.5 Discussion and Modified Partitioning

The results of the modelling of the top level specification lead suggest an alternative partitioning that may be more appropriate in this case:

1. The strong coupling between the Mantissa Aligner and the Mantissa Adder suggest that the two should be migrated into the same partition.
2. The strong coupling between the Exponent Adder and the Exponent Adjuster suggest that the two should be migrated into the same partition.
3. The poor packing and low cohesion of the Alignment Unit may be alleviated by migrating the Sign Unit out of the Alignment Unit and merging it with the exponent data bus to form a data path. This also allows the direct completion of connections to the Sign Unit.
4. The implication of (1) and (2) above is that the top level should be repartitioned into two modules: one a mantissa data path and the other an exponent data

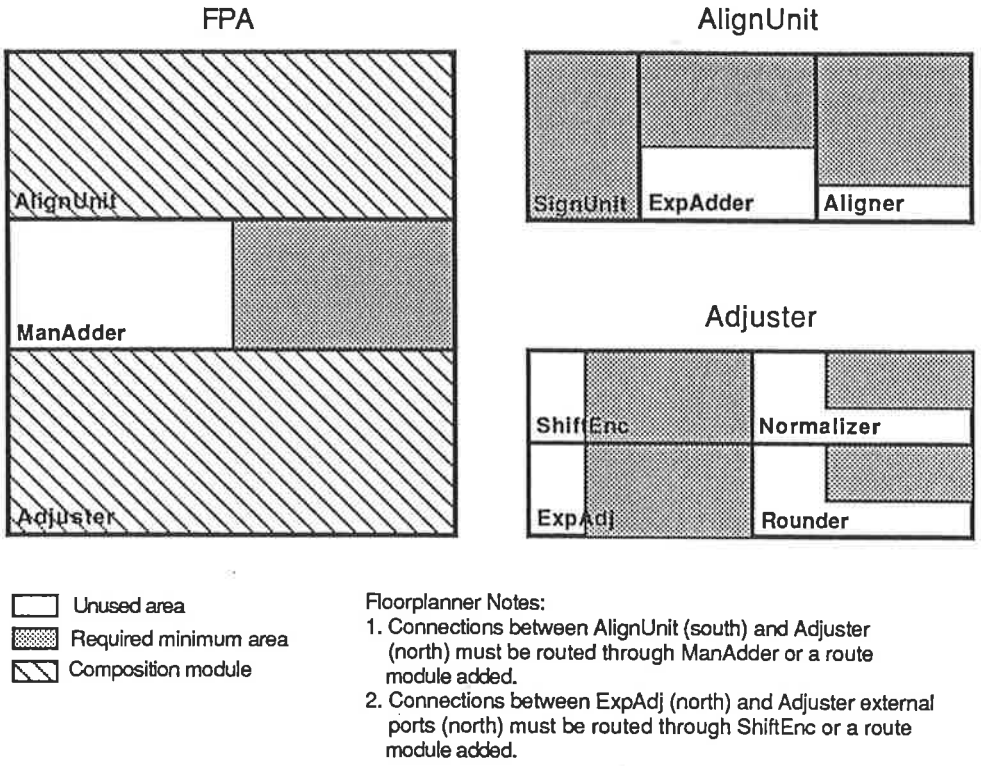

Figure 7.5: A floorplan for the FPA (initial partitioning).

path. Each is likely to have a high cohesion as all the elements will be connected by a bus. The coupling between the two is likely to be low as the two paths are largely functionally independent.

5. If the exponent bus were routed thorough the Shift Encoder then it and the Exponent Adjuster would be able to stack vertically, thus saving area.

The new top level partitioning may be classified as being more along functional lines that data flow: each module implements a particular function: processing of the mantissas or processing of the exponents.

This change in partitioning technique emphasises the importance of modelling in VLSI partitioning. It is not simply sufficient to adopt a partitioning strategy based on the nature of the algorithm.

The new partitioning is imposed by making the following changes:

1. Adding the exponent bus as a route through to the Sign Unit and Shift Encoder.

2. Allocating the Exponent Adder, the Sign Unit, the Shift Encoder and the Exponent Adjuster to a new module called the Exponent Data Path.

3. Allocating the Aligner, the Mantissa Adder, the Normalizer and the Rounder to a new module called the Mantissa Data Path.

The new hierarchical partitioning thus created is shown in Figure 7.6.

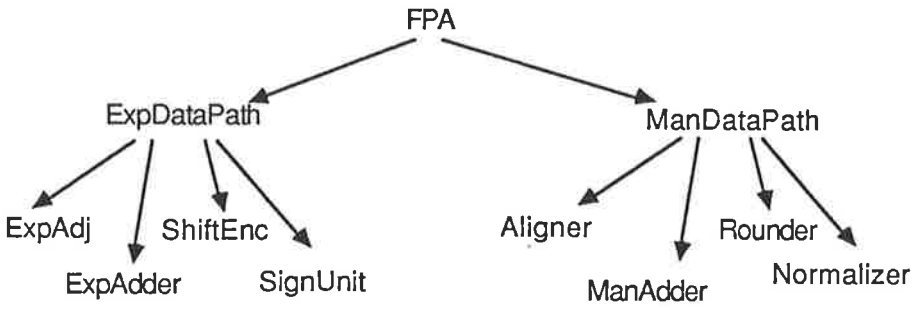

Figure 7.6: The design hierarchy for the revised partitioning.

The profiling results illustrated in Figure 7.7 suggest that the revised partitioning is quite well balanced in terms of coupling and cohesion. The coupling between the two new modules is only 27648 compared to 74752 and 40960 in the initial partitioning. The four most tightly coupled modules, those that process the mantissa, are now all contained in one highly cohesive module.

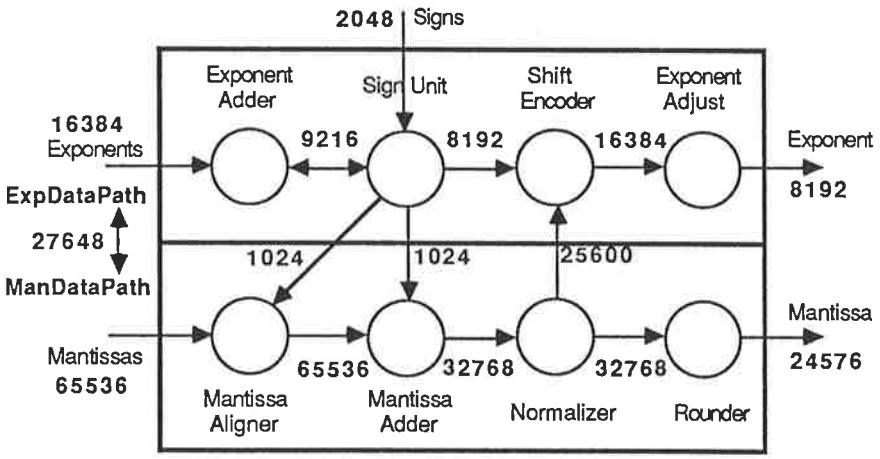

Figure 7.7: Profiling data derived from the FPA model (revised partitioning).

The revised floorplan for the FPA is show in Figure 7.8. There are no further planarity faults and all connections are feasible. Only 13% of the area of the revised partitioning

floorplan is unused space as opposed to 35% for the initial partitioning. The total area occupied by the revised FPA floorplan is 72% of that occupied by the initial floorplan.

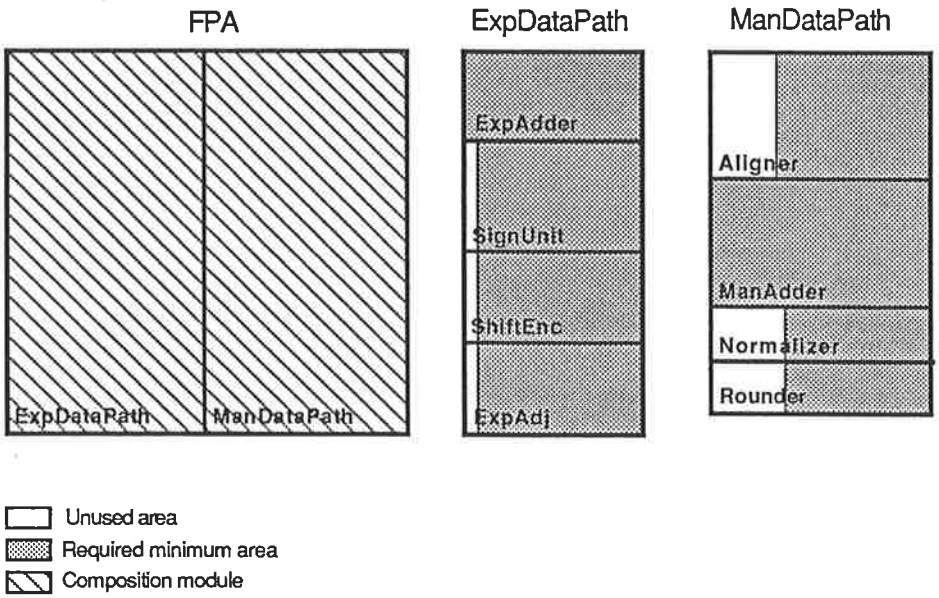

Figure 7.8: A floorplan for the FPA (revised partitioning).

7.6 Refining the Exponent Adder

In both the initial and revised specification, the Exponent Adder is implemented by the following functional module:

```
DEFINITION ExpAdder;
   STATE
    -- Ports:
    Switch, Cout : PortType;
    ExpA, ExpB, Exp : ARRAY [0..7] OF PortType;
    Shift : ARRAY [0..5] OF PortType;
    -- Variables:
    expav, expbv, exprv, shiftamountv : INTEGER;
    coutv : BOOLEAN;
    PORTS
      DeclareBus(ExpA, INP); -- Exponent of A.
      DeclareBus(ExpB, INP); -- Exponent of B.
      DeclareBus(Exp, OUTP); -- Exponent positive difference.
```

```
    DeclareBus(Shift, OUTP); -- Encoded shift amount.
    DeclarePort(Switch, BIT, OUTP); -- Mantissa shift selection.
    DeclarePort(Cout, BIT, OUTP); -- Carry out.
  INIT
    -- None.
  FUNCTION
   expav := READBUS(ExpA);
   expbv := READBUS(ExpB);
   IF (expav-expbv) > 0 THEN
     coutv := TRUE;
   ELSE
     coutv := FALSE;
   END;
   IF coutv THEN
     shiftamountv := expav - expbv;
   ELSE
     shiftamountv := expbv - expav;
   END;
   IF shiftamountv > 23 THEN
      shiftamountv := 23;
   END;
   IF coutv THEN
     exprv := expav;
   ELSE
     exprv := expbv;
   END;
   WRITEBUS(Shift, shiftamountv);
   WRITEBUS(Exp, exprv);
   WRITE(Cout, FROMBOOL(coutv));
   WRITE(Switch, FROMBOOL(coutv));
   ENDFUNCTION;
END ExpAdder;
```

This specification demonstrates the use of the abstract Modula-2 subtraction operator for the simplified specification of a function. Decomposition of the Exponent Adder consists of dividing the function between three submodules:

1. *Inverter* inverts the signals on the A and B exponent bus. By tying the inputs of the following adders high and using the inverted version of one of the two exponents the adders may be used to perform subtraction.
2. The *AdderA* takes the difference `ExpA - ExpB`.
3. The *AdderB* takes the difference `ExpB - ExpA`. The carry output of the adder will be high if ExpA is the larger of the two.

4. The *Multiplexor* uses the carry out of AdderB to select the positive difference and transfer it to Shift. Similarly the larger of the two input exponents is switched to Exp.

The refined specification of the Exponent Adder now appears as the composition module:

```
DEFINITION ExpAdder;
   STATE
    -- Ports:
    Switch, Cout : PortType;
    ExpA, ExpB, Exp : ARRAY [0..7] OF PortType;
    Shift : ARRAY [0..5] OF PortType;
    -- Variables:
    expav, expbv, exprv, shiftamountv : INTEGER;
    coutv : BOOLEAN;
    PORTS
      DeclareBus(ExpA, INP); -- Exponent of A.
      DeclareBus(ExpB, INP); -- Exponent of B.
      DeclareBus(Exp, OUTP); -- Exponent positive difference.
      DeclareBus(Shift, OUTP); -- Encoded shift amount.
      DeclarePort(Switch, BIT, OUTP); -- Mantissa shift selection.
      DeclarePort(Cout, BIT, OUTP); -- Carry out.
    TOPOLOGY
      MakeInstance(InvertBus, InvertBus);
      MakeInstance(AdderA, AdderA);
      MakeInstance(AdderB, AdderB);
      MakeInstance(Multiplexor, Multiplexor);
      -- Input connections:
      Connect(ExpA & InvertBus/Ain);
      Connect(ExpB & InvertBus/Bin);
      -- Output connections:
      Connect(Exp & Multiplexor/ExpOut);
      Connect(Shift & Multiplexor/ShiftOut);
      Connect(Switch & AdderB/Cout);
      Connect(Cout & AdderB/Cout);
      -- Submodule interconnect:
      Connect(InvertBus/Aout & AdderA/Ain);
      Connect(InvertBus/Bout & AdderA/Bin);
      Connect(InvertBus/InvAout & AdderA/InvAin);
      Connect(InvertBus/InvBout & AdderA/InvBin);
      Connect(AdderA/Aout & AdderB/Ain);
      Connect(AdderA/Bout & AdderB/Bin);
      Connect(AdderA/InvAout & AdderB/InvAin);
      Connect(AdderA/InvBout & AdderB/InvBin);
```

```
      Connect(AdderA/Sum & Multiplexor/Ain);
      Connect(AdderB/Sum & Multiplexor/Bin);
      Connect(AdderB/Cout & Multiplexor/Select);
      Connect(FixedHi/Out & AdderA/Cin & AdderB/Cin);
  ENDTOPOLOGY
END ExpAdder;
```

Functional modelling of this specification reveals that the refined module behaves identically to the earlier more abstract version. Modelling the form of the specification produces the simple floorplan shown in Figure 7.9. Neither result suggest that changes need be made to the partitioning.

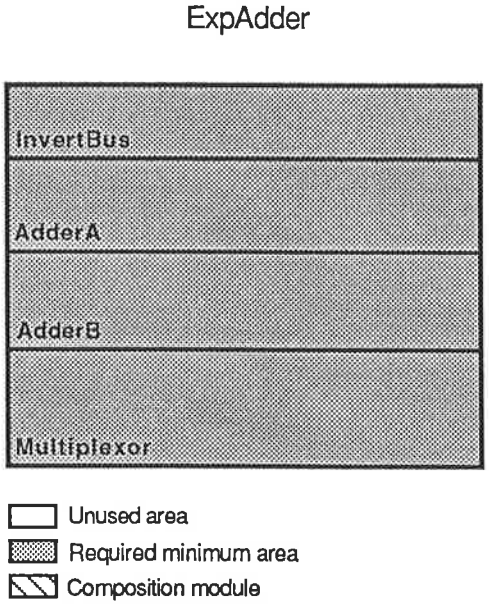

Figure 7.9: Floorplan of the Exponent Adder.

7.7 Refining the Adder

The adder used in the Exponent Adder may be further decomposed to a bitwise composition. It is at this level of the hierarchy that regularity in function and communication become apparent. Note that there is a structural difference between AdderA and AdderB that may be expressed by using two instances of a single definition macro to create different definitions. The definition macro `MakeAdderDef` has three parameters: the name of the definition, and the two signals to be connected to

the respective adder cell inputs: [1]

```
#define MakeAdderDef(NAME,ACON,BCON)
DEFINITION Name;
STATE
  Ain, Bin, InvAin, InvBin, Sum,
  Aout, Bout, InvAout, InvBout : ARRAY [0..7] OF PortType;
  Cin, Cout : PortType;
PORTS
  DeclareBus(Ain, INP); -- A input.
  DeclareBus(Bin, INP); -- B input.
  DeclareBus(InvAin, INP); -- Inverted A input.
  DeclareBus(InvBin, INP); -- Inverted B input.
  DeclareBus(Aout, OUTP); -- A output.
  DeclareBus(Bout, OUTP); -- B output.
  DeclareBus(InvAout, OUTP); -- Inverted A output.
  DeclareBus(InvBout, OUTP); -- Inverted B output.
  DeclareBus(Sum, OUTP); -- Sum of A and B.
  DeclarePort(Cin, INP); -- Carry in.
  DeclarePort(Cout, OUTP); -- Carry out.
TOPOLOGY
  -- Make an array of adder cells:
  MakeInstance(AdderBit, AdderBit<0..7>);
  -- Connect up the through connections:
  Connect(Ain[<0..7>] & Adderbit<0..7>/Ain);
  Connect(Bin[<0..7>] & Adderbit<0..7>/Bin);
  Connect(InvAin[<0..7>] & Adderbit<0..7>/InvAin);
  Connect(InvBin[<0..7>] & Adderbit<0..7>/InvBin);
  Connect(Aout[<0..7>] & Adderbit<0..7>/Aout);
  Connect(Bout[<0..7>] & Adderbit<0..7>/Bout);
  Connect(InvAout[<0..7>] & Adderbit<0..7>/InvAout);
  Connect(InvBout[<0..7>] & Adderbit<0..7>/InvBout);
  -- Connect the sum:
  Connect(Sum[<0..7>] & Adderbit<0..7>/Sum);
  -- Connect the carry's
  Connect(AdderBit<1..7>/Cin & AdderBit<0..6>/Cout);
  Connect(AdderBit0/Cin & Cin);
  Connect(AdderBit7/Cout & Cout);
  -- Strap the appropriate connections:
  Connect(AdderBit<0..7>/A & AdderBit/ACON);
  Connect(AdderBit<0..7>/B & AdderBit/BCON);
ENDTOPOLOGY
END NAME;
```

[1] Line continuation character "\" omitted for clarity.

The definition of adder A and B may now be generated by appropriate calls to the macro:

```
MakeAdderDef(AdderA, Ain, InvBin)
MakeAdderDef(AdderB, InvAin, Bin)
```

The floorplan generated for such a composition is shown in Figure 7.10. The constraints one each cell comprise an environment (shown in Figure 7.11) that may be used to guide a manual or automatic leaf cell design tool.

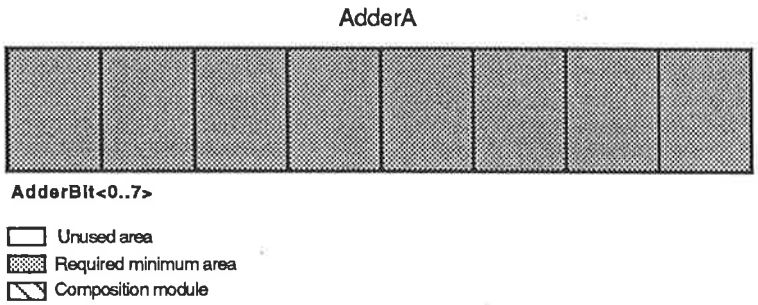

Figure 7.10: Floorplan of `AdderA`.

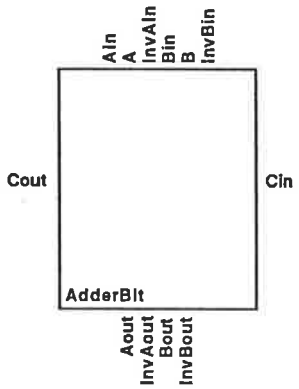

Figure 7.11: Leaf cell environment specification.

Up to this point none of the description has included timing information. Instead "instantaneous" transfers have been used as abstraction from physical delays. At this point however (the leaf module level) it is appropriate to add in delay information that may be used to verify that system wide signal timings are correct. The following is a functional definition with physical delays for a single bit of the adder given in the above description.

```
DEFINITION AdderBit;
-- Adder bit with input and inverted input wire-through.
STATE
  Ain, Bin, InvAin, InvBin, Sum,
  Aout, Bout, InvAout, InvBout,
  A, B, Cin, Cout : PortType;
  av, bv, cinv, outv, coutv : BOOLEAN;
PORTS
  DeclarePort(Ain, BIT, INP); -- A input for wirethrough.
  DeclarePort(Bin, BIT, INP); -- B input for wirethrough.
  DeclarePort(InvAin, BIT, INP); -- Inv A input for wirethrough.
  DeclarePort(InvBin, BIT, INP); -- Inv B input for wirethrough.
  DeclarePort(Aout, BIT, OUTP); -- A output for wirethrough.
  DeclarePort(Bout, BIT, OUTP); -- B output for wirethrough.
  DeclarePort(InvAout, BIT, OUTP); -- Inv A output for wirethrough.
  DeclarePort(InvBout, BIT, OUTP); -- Inv B output for wirethrough.
  DeclarePort(A, BIT, INP); -- A input.
  DeclarePort(B, BIT, INP); -- B input.
  DeclarePort(Sum, BIT, OUTP); -- Sum of A and B.
  DeclarePort(Cin, BIT, INP); -- Carry in.
  DeclarePort(Cout, BIT, OUTP); -- Carry out.
INIT
  -- None.
FUNCTION
  -- Wirethroughs with no delay:
  WRITE(Aout, READ(Ain));
  WRITE(Bout, READ(Bin));
  WRITE(InvAout, READ(InvAin));
  WRITE(InvBout, READ(InvBin));
  -- Actual adder:
  av := TOBOOL(READ(A));
  bv := TOBOOL(READ(B));
  cinv := TOBOOL(READ(Cin));
  outv := (av AND bv AND cinv) OR
          (av AND NOT bv AND NOT cinv) OR
          (NOT av AND NOT bv AND cinv) OR
          (NOT av AND NOT cinv AND bv);
  coutv := (av AND bv) OR
           (av AND cinv) OR
           (bv AND cinv);
  DELAYEDWRITE(Sum, FROMBOOL(outv), 15);
  DELAYEDWRITE(Cout, FROMBOOL(coutv), 15);
ENDFUNCTION;
END AdderBit;
```

When this cell is placed in a composition such as AdderA, the delays in the carry chain for instance are propagated such that the AdderA carry delay is the sum of all the bit cell delays. Modelling the system under these conditions allows the designer to catch synchronization errors, note critical paths, and to estimate and adjust system performance.

7.8 Summary

An algorithm that is to be implemented as a VLSI system may be initially partitioned using appropriate techniques from software structural design. This reduces the complexity of the design problem by dividing it into a number of subproblems with well defined interactions.

The various physical costs and constraints involved in implementing an algorithm in VLSI impact the partitioning of the system. In particular the high cost of communication between modules, and the requirement for planarity in a structured floorplan, may require that the initial partitioning be modified. This is illustrated in the example outlined in this chapter by the need to restructure the partitioning such that it allows communication based around bus structures. The resulting revised partitioning led to reduced area and communication complexity in the final design.

The example has illustrated the utility of supplying the designer with design tools that aid in the evaluation of the partitioning by modelling the physical function and geometrical form of the design.

Chapter 8

Conclusion

The complexity of VLSI systems may be attributed to the large number of interacting components they contain, the inherently unstructured nature of the medium, and the costs and constraints of interconnect.

This research has shown that managing the complexity of VLSI systems design may be aided by the adoption of a design philosophy and associated computer aided modelling tools that assist in *structural design*. By identifying the similarities and differences between the structural design of software and hardware it has been possible to examine the transfer of software structural design techniques to the VLSI domain.

As a first step towards supporting structural planning, the design process has been divided into a designer intensive, exploratory *planning* phase and an automated *construction* phase. Rather than perform structural planning in a horizontally partitioned set of abstract descriptions, the task is carried out within a single structural representation of the design hierarchy.

A language has been described that encourages the application of software structural design techniques to VLSI. In particular the language provides mechanisms for the abstract representation of intermodule communication and regular structure. It encourages the top-down refinement of the design by the recursive decomposition of functional modules into structural entities, and the refinement of abstract communication mechanisms to physically delayed signal transfers.

Validation and analysis of the design specification requires that a facility be available for the *modelling* of the composite function of the specification. A program has been described that allows the designer to model the system in a similar way to a

software designer "debugging" a program. Test vectors are used to drive the model of a particular system module to validate (with a degree of confidence) the correct function of the specification.

The quality of the partitioning inherent in the structural specification is evaluated with the aid of a set of statistics recorded by the program during simulation. These are mainly concerned with capturing the data-flow properties of the design as communications have a high priority in the VLSI medium. Additional statistics on module activity aid in the estimation of parallelism. The summary of these statistics and their presentation to the designer provides a mechanism for the "profiling" of the design, assisting in the location of trouble spots, and giving an estimate of the *coupling* and *cohesion* of the various modules.

A mechanism has been described that allows for the modelling of the function of the specification based on a representation of each module as a process. Communication between modules is simulated by the passing of messages between processes, and a scheduling mechanism is provided that ensures that process activations occur in an appropriate sequence. The overall effect of this approach to simulation is that fewer module activations are required, increasing the efficiency of the modelling process.

The two-dimensional constrained nature of the silicon surface requires that the designer be able to model the physical *form* of the partitioning in addition to its function. This may be done through the generation of *structured floorplans* for each composition module as it is specified. Such floorplans may also of course be used to guide the module assembly process.

The automatic top-down generation of custom floorplans is complicated by a number of issues: primarily the details of the modules in each design are typically uncertain, as they themselves are yet to be designed. In addition the search space for any such space-planning problem is very large, resulting in exponentially increasing computational complexity.

The approach taken to the floorplanning problem in this research has relied on the use "knowledge-based" techniques to deal with these problems. Configuration factors have been proposed as a mechanism for representing and reasoning with uncertain knowledge of module implementations. Domain specific knowledge has been used to prune the solution space and thus minimize the number of design alternatives that must be evaluated during a design. Several distinct classes of knowledge required for

the floorplanning task have been identified, and a knowledge representation designed or selected for each.

Finally, a case study has been presented that has illustrated the successful application of the structural methodology to a VLSI system design.

Due to the lack of comparable research in the area of structural VLSI design, details of the implementations and their performance have not been emphasised. Rather the intent has been to present a *philosophy* for the structural design of large VLSI systems, and to indicate how it may be supported.

References

Ackland, B., Dickinson, A., Ensor, R., Gabbe, J., Kollaritsch, P., London, T., Poirier, C., Subrahmanyam, P., and Watanabe, H., 1984. *CONCORD - A System of Cooperating VLSI Design Experts.* Technical Memorandum 11354-841215-06TM, Bell Laboratories, Holmdel, New Jersey, December.

Ackland, B., Dickinson, A., Ensor, R., Gabbe, J., Kollaritsch, P., London, T., Poirier, C., Subrahmanyam, P., and Watanabe, H., 1985. CADRE - A System of Cooperating VLSI Design Experts. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 99–104, New York, October.

Ahuja, S., Carriero, N., and Gelernter, D., 1986. Linda and Friends. *IEEE Computer*, 19(8):26–34, August.

Anceau, F. and Reis, R. A., 1982. Complex Integrated Circuit Design Strategy. *IEEE Journal of Solid-State Circuits*, SC-17:459–464, June.

Aylor, J. H., Waxman, R., and Scarrat, C., 1986. VHDL – Feature Description and Analysis. *IEEE Design and Test*, 3(2):17–27, April.

Azema, P., Diaz, M., and Doucet, J., 1975. Multilevel Description Using Petri Nets. In *Computer Hardware Description Languages*, New York.

Baker, A. L., 1979. The Use of Software Science in Evaluating Modularity Concepts. *IEEE Transactions on Software Engineering*, SE-5(2):110–119, March.

Barbacci, M. R., 1981. Instruction Set Processor Specifications (ISPS): The Notations and its Applications. *IEEE Transactions on Computers*, C-30(1):24–40, January.

Barrow, H. G., 1984. Proving the Correctness of Digital Hardware Designs. *VLSI Design*, July.

Baybars, I. and Eastman, C. M., 1980. Enumerating Architectural Arrangements by Generating Their Underlying Graphs. *Environment and Planning*, 7:289–310.

Baybars, I., 1982. The Generation of Floorplans with Circulation Spaces. *Environment and Planning*, 9:445–456.

Bergland, G. D., 1981. A Guided Tour of Program Design Methodologies. *IEEE Computer*, 14(10):13–37, October.

Birmingham, W. P., Joobbani, R., Kim, J. H., Siewiorek, D. P., and York, G., 1985. CLASS: A Chip Layout Assistant. In *Proceedings of the IEEE Int. Conf. on Computer Aided Design*, pages 216–218, November.

Birtwistle, G., Joyce, J., Liblong, B., Melham, T., and Schediwy, R., 1986. Specification and VLSI Design. In *Proceedings of the Edinburgh Workshop on Formal Methods in VLSI Design*, North Holland.

Bobrow, D. and Stefik, M., 1983. *The LOOPS Manual.* Technical Report, Xerox PARC.

Böhm, C. and Jacopini, G., 1966. Flow Diagrams, Turing Machines andf Languages with Only Two Formulation Rules. *Communications of the ACM*, 9(5):366–371, May.

Breuer, M. A., 1976. A Class of Min-Cut Placement Algorithms. In *Proceedings of the 13th Design Automation Conference*, pages 284–290, June.

Brooks, F. P., 1975. *The Mythical Man-Month.* Addison-Wesley, Reading, Mass.

Brown, H., Tong, C., and Foyster, G., 1983. Palladio: An Exploratory Environment for Circuit Design. *IEEE Computer*, 16(12):41–56, December.

Brownston, L., Farrell, R., Kant, E., and Martin, N., 1985. *Programming Expert Systems in OPS5.* Addison-Wesley, Reading, Mass.

Buchanan, B. G. and Shortcliffe, E. H., editors. *Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison Wesley, Reading, Mass.

Buchanan, B. G. and Shortliffe, E. H., 1984. Uncertainty and Evidential Support. In *Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project*, pages 209–232, Addison Wesley, Reading, Mass.

Buchanan, I., 1980. *Modelling and Verification of Structured Integrated Circuit Design.* PhD thesis, California Instite of Technology, May.

Cameron, J. R., 1983. *JSP & JSD: The Jackson Approach to Software Development.* IEEE Computer Society Press.

Chen, M. C. and Mead, C. A., 1983. A Hierarchical Simulator Based on Formal Semantics. In *Proceedings of the 3rd Caltech Conference on VLSI*, Caltech, Pasadena, Ca.

Dantzig, G. B., 1963. *Linear Programming and Extensions.* Princeton University Press, Princeton, New Jersey.

Davis, R. and Buchanan, B., 1977. Production Rules as a Representation for a Knowledge Based Consultation System. *Artificial Intelligence*, 14(2):15–45, September.

Davis, R., Buchanan, B. G., and Shortliffe, E. H., 1977. Production Rules as a Representation for a Knowledge Based Consultation Program. *Artificial Intelligence*, 8:15–45.

Dennis, J. B., 1980. Data-Flow Supercomputers. *IEEE Computer*, 13(11):48–56, November.

Denyer, P. B., Renshaw, D., and Bergmann, N., 1982. A Silicon Compiler for VLSI Signal Processors. In *ESSCIRC 1982*, Brussels, Belgium.

Dickinson, A. and Eshraghian, K., 1984. Abstracted Description and Simulation of VLSI Systems. In *Pacific/Asian Regional Conference on VLSI*, Melbourne, Australia, May.

Dickinson, A., 1982. *A Video Waveform Analyser/Synthesiser.* Honours Thesis, The University of Adelaide, May.

Dickinson, A., 1984. *Prudence — A Framework for Constructing Object Based Expert Systems.* Technical Memorandum 11354–841215–07TM, AT&T Bell Laboratories, December.

Dickinson, A., 1984. *TICTOC: A VLSI System Language and Simulator.* Technical Report, The University of Adelaide, Dept of Elec. Eng., Adelaide, Australia, March.

Dickinson, A., 1985. *An Expert System Approach to Custom VLSI Floorplanning.* Technical Memorandum 11354-850225-05TM, Bell Laboratories, Holmdel, New Jersey, February.

Dickinson, A. G., 1986. An Application of Domain Knowledge to VLSI CAD. In *Microelectronics '86*, pages 241–248, Adelaide, Australia, May.

Dickinson, A. G., 1986. Floyd: A Knowledge Based Floor Plan Designer. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, New York, October.

Dickinson, A., 1987. *The Pink Manual.* Technical Report, The University of Adelaide, Dept Elec. Eng., June.

Dickinson, A. G., Cole, P. H., Wu, S. W., and Watson, C. R., 1987. A Tool Set for Correct-By-Construction VLSI Design. In *Electronic Design Automation Conference*, London, July.

Dijkstra, E. W., 1968. GOTO Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, March.

Dijkstra, E. W., 1970. Structured Programming. In *Software Engineering Techniques*, pages 84–88, NATO Scientific Affairs Division, Brussels 39, Belgium, April.

Dijkstra, E. W., 1972. Notes on Structured Programming. In *Structured Programming*, chapter I, pages 1–82, Academic Press Inc, London.

Dijkstra, E. W., 1976. *A Discipline of Programming.* Prentice-Hall, Edgewood Cliffs, NJ.

Duda, R., Hart, P., and Nilsson, N., 1976. Subjective Bayesian Methods for Rule-based Inference Systems. In *Proceedings AFIPS 1976 NCC*, pages 1075–1082.

Earl, C. F., 1980. Rectangular Shapes. *Environment and Planning*, 7:311–343.

Erman, L., Hayes-Roth, F., Lesser, V., and Reddy, D., 1980. The HEARSAY-II Speach Understanding System: Integrating Knowledge to REsolve Uncertainty. *Computing Surveys*, 12(2):213–253, February.

Eshraghian, K. E., Bryant, R. C., Dickinson, A. G., Fensom, D. S., Franzon, P. D., Pope, M. T., Rockliff, J. E., and Zyner, G. B., 1984. A New CMOS Architecture for Signal Processing. In *Pacific/Asian Regional Conference on VLSI*, Melbourne, Australia, May.

Eshraghian, K. E., Bryant, R. C., Dickinson, A. G., Fensom, D. S., Franzon, P. D., Pope, M. T., Rockliff, J. E., and Zyner, G. B., 1985. The Transform and Filter Brick - A New Architecture for Signal Processing. In *Proceedings of VLSI 85*, Tokyo, August.

Evangelist, W. M., 1983. Software Complexity Metric Sensitivity to Program Structuring Rules. *The Journal of Systems and Software*, 3(3):231–243, September.

Feigenbaum, E. A., 1977. The Art of AI: Theses and Case Studies of Knowledge Engineering. In *IJCAI 77*, pages 1014–1029.

Feigenbaum, E. A., 1977. The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering. In *IJCAI 5*, pages 1014–1029.

Feldbrugge, F. H. J., 1980. VLSI and Petri Nets. In *Design Methodologies for VLSI Circuits*, NATO Advanced Study Institute, Louvain-la-Nuese, Belgium, July.

Ferry, D. K., 1985. Interconnection Lengths and VLSI. *IEEE Circuits and Devices*, 1(4):39–42, July.

Fey, C. F., 1985. Custom LSI/VLSI Chip Design Productivity. *IEEE Journal of Solid State Circuits*, 20(2):555–561, April.

Fidducia, C. M. and Mattheyses, R. M., 1982. A Leinear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, June.

Foderaro, J. K. and Sklower, K. L., 1983. *The FRANZ LISP Manual.* UC Berkley, June.

Forgy, C. L., 1981. *OPS5 Users Manual.* CMU, July.

Forgy, C. L., 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, January.

Foyster, G., 1986. VLSI Functional Verification from Specification to Test. In *Microelectronics 86*, pages 287–293, Adelaide, Australia, May.

Genesereth, M., 1982. Diagnosis Using Hierarchical Design Models. In *AAAI 82*, pages 278–283.

Genesereth, M. R., 1983. An Overview of Meta-level Architecture. In *AAA 83*, pages 119–124.

German, S. M. and Lieberherr, K. J., 1985. Zeus: A Language for Expressing Algorithms in Hardware. *Computer*, 18(2):55–65, February.

Gerner, M. and Johansson, M., 1986. VLSI Testing: DFT Strategies and CAD Tools. In *Summer School on VLSI Tools and Applications*, ETH, Zurich, Switzerland, July.

Ghosh, S., 1986. Software Technologies in ADA for High-Level Hardware Descriptions. *IEEE Circuits and Devices*, 2(2):32–47, March.

Giambiasi, N., 1985. An Adaptive and Evolutive Tool for Describing General Hierarchical Models, Based on Frames and Demons. In *Proceedings of the 22nd Design Automation Conference*, pages 460–466, June.

Gilleard, J., 1978. LAYOUT- A Hierarchical Computer Model For the Production of Architectural Floor Plans. *Environment and Planning*, 5:233–241.

Goldberg, A. J. and Robson, D., 1983. *Smalltalk-80: The Language and its Implementation.* Addison wesley, Reading, Mass.

Gordon, J. and Shortliffe, E. H., 1984. The Dempster-Shafer Theory of Evidence. In *Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project*, pages 272–294, Addison Wesley, Reading, Mass.

Gordon, R., 1979. Measuring Improvements in Program Clarity. *IEEE Transactions on Software Engineering*, SE-5(2):79–90, March.

Grason, J., 1970. A Dual Linear Graph Representation for Space-Filling Allocation Problems of the Floor Plan Type. In *Emerging Methods in Environmental Design and Planning*, pages 170–178, MIT Press.

Halstead, M. H., 1977. *Elements of Software Science.* Elsevier North-Holland, New York.

Hartenstein, R. W., 1986. High Level Simulation and CHDLs. In *Summer School on VLSI Tools and Applications*, ETH, Zurich, Switzerland, July.

Hayes-Roth, F., Waterman, D., and Lenat, D., 1983. *Building Expert Systems.* Addison Wesley, Reading, Mass.

Healey, S. T. and Gajski, D. D., 1985. Decomposition of Logic Networks into Silicon. In *Proceedings of the 22nd Design Automation Conference*, pages 162–168, June.

Hedlund, K. S., 1987. Aesop: A Tool For Automated Transistor Sizing. In *Proceedings of the 24nd Design Automation Conference*, pages 114–120, ACM/IEEE, Miami, Fl., June.

Heller, W. R., Sorkin, G., and Maling, K., 1982. The Planar Package Planner for System Designers. In *Proceedings of the 19th Design Automation Conference*, pages 253–260, June.

Henrion, M., 1978. Automatic Space Planning: A Post-Mortem? In Latcombe, E., editor, *Artificial Intelligence and Pattern Recognition in Computer Aided Design*, North Holland.

Henry, S. and Karfura, D., 1984. On the Relationship Among Three Software Metrics. *Software Practice and Experience*, 14(6):561—573, June.

Hild, M. and Piednoir, J. O., 1985. Efficient Placement Algorithms for VLSI. *VLSI Design*, :46–50, April.

Hillis, D., 1986. *The Connection Machine.* Association for Computing Machinery, New York.

Hoare, C. A. R., 1978. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August.

Intellicorp., 1985. *The Knowledge Engineering Environment.* Knowledge Systems Division, Intellicorp, Menlo Park, Cal.

Jabri, M. and Skellern, D., 1986. A Hybrid Rule-Based/Algorithmic Approach to VLSI Circuit Floorplanning. In *Microelectronics '86*, pages 225–231, Adelaide, Australia, May.

Jackson, M. A., 1975. *Principles of Program Design.* Academic Press, New York.

Jensen, K. and Wirth, N., 1978. *Pascal: User Guide and Report.* Springer-Verlag, New York.

Jepsen, D. W. and Gelatt, C. D., 1983. Macro Placement by Monte Carlo Annealing. In *IEEE Conference on VLSI in Computers*, pages 495–498.

Jhon, C. S., Sobelman, G. E., and Krekelburg, D. E., 1985. Silicon Compilation Based on a Data-Flow Paradigm. *IEEE Circuits and Devices*, 1(3):21–28, May.

Joobani, R. and Siewiorek, D. P., 1985. Weaver: A Knowledge Based Routing Expert. In *Proceedings of the 22nd Design Automation Conference*, June.

Joseph, R., Ensor, R., Dickinson, A., and Blumenthal, R., 1986. Describe – An Explanation Based Facility for an Object based Expert System. In *The Artificial Intelligence and Advanced Computer Technology Conference*, Long Beach, California, April.

Jouppi, N., 1983. Timing Analysis for NMOS VLSI. *Proceedings of the 20th Design Automation Conference*, June.

Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., and Adler, M. A., 1986. Software Complexity Measurement. *Communications of the ACM*, 29(11):1044–1050, November.

Kernighan, B. W. and Lin, S., 1970. A Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49, February.

Keyes, R. W., 1981. Fundamental Limits in Digital Information Processing. *Proceedings of the IEEE*, 69(2):267–279, February.

Kimm, J., McDermott, J., and Siewiorek, D. P., 1984. Exploiting Domain Knowledge in IC Cell Layout. *IEEE Design and Test*, 1:52–65, August.

Kollaritsch, P. W. and Weste, N. H. E., 1984. A Rule Based Symbolic Layout Expert. *VLSI Design*, :62–66, August.

Kollaritsch, P. W. and Weste, N. H. E., 1985. Topologizer: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout. *IEEE Journal of Circuits and Systems*, June.

Korf, R. E., 1977. A Shape Independent Theory of Space Allocation. *Environment and Planning*, 14:37–50.

Kowalski, T. J. and Thomas, D. E., 1983. The VLSI Design Automation Assistant: Prototype System. In *Proceedings of the 20th Design Automation Conference*, pages 479–483, June.

Kowalski, T. J., 1986. *An Artificial Intelligence Approach to VLSI Design.* Kluwer, Boston, Mass.

Kozminski, K. and Kinnen, E., 1984. An Algorithm for Finding a Rectangular Dual of a Planar Graph for Use in Area Planning for Integrated Circuits. In *Proceedings of the 21st Design Automation Conference*, pages 655–656, June.

Krambeck, R. H., So, H. C., Law, H. S., Blahut, D. E., and Shichman, H., 1982. *Hierarchical Layout and Design of a Single Chip 32 Bit CPU.* Technical Memorandum 82-52154-2, Bell Laboratories, August.

LaPotin, D. and Director, S., 1985. Mason: A Global Floor-Planning Tool. In *Proceedings of the IEEE Int. Conf. on Computer Aided Design*, pages 143–145, November.

Lattin, W. W., Bayliss, J. A., Budde, D., Rattner, J. R., and Richardson, W. S., 1981. A Methodology ofr VLSI Chip Design. *Lambda*, :34–44, Second Quarter.

Lauther, A. M., 1979. A MIN-CUT Algorithm for General Cell Assemblies Based on a Graph Representation. In *Proceedings of the 16th Design Automation Conference*, June.

Lenat, D., Davis, R., Doyle, J., Genesereth, M., Goldstein, I., and Shrobe, H., 1983. Reasoning About Reasoning. In *Building Expert Systems*, pages 219–240, Addison Wesley, Reading, Mass.

Leske, M. W. and Schmidt, E., 1975. *LEX - Lexical Analyzer Generator.* Technical Report 75-1274-15 39199 39199-11, Bell Laboratories, Murray Hill, NJ, July.

Ligtenberg, A. and O'Neill, J. H., 1985. *Towards an Automatic Layout for Digital Signal Processing Algorithms.* Technical Memorandum 11355-840201-01TM, Bell Laboratories, February.

Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., and Lederberg, J., 1980. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project.* McGraw-Hill, New York.

Linger, R. C., Mills, H. D., and Witt, B. I., 1979. *Structured Programming Theory and Practice.* Addison-Wesley, Reading, Mass.

Lipp, H. M., 1983. Methodical Aspect of Logic Synthesis. *Proceeding of the IEEE*, 71(1):88–97, January.

Lipsett, R., Marschner, E., and Shahdad, M., 1986. VHDL – The Language. *IEEE Design and Test*, 3(2):24–41, April.

Maling, K., Mueller, S. H., and Heller, W. R., 1982. On Finding Most Optimal Rectangular Package Plans. In *Proceedings of the 19th Design Automation Conference*, pages 663–670, June.

McCabe, T. H., 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(6):308–320, December.

McDermott, J., 1980. R1: An Expert in the Computer Systems Domain. In *AAAI 1*, pages 269–271.

McFarland, M. C., 1983. Computer-Aided Partitioning of Behavioral Hardware Descriptions. In *Proceedings of the 20th Design Automation Conference*, pages 472–478, June.

Mead, C. and Conway, L., 1980. *Introduction to VLSI Systems.* Addison Wesley, Reading, Mass.

Mead, C., 1982. Personal Communication, May. Adelaide, Australia.

Miller, G. A., 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review*, 63(2):81–97, March.

Minsky, M., 1975. A Framework for Representing Knowledge. In Winston, P., editor, *The Psychology of Computer Vision*, pages 211–277, McGraw Hill, New York.

Misunas, D., 1973. Petri Nets and Speed Independent Design. *Communications of the ACM*, 16(8):474–481, August.

Mitchell, W. J., Steadman, J. P., and Liggett, R. S., 1976. Synthesis and Optimization of Small Rectangular Floor Plans. *Environment and Planning*, 3:37–70.

Moore, G., 1979. VLSI: Some Fundamental Challenges. *IEEE Spectrum*, :30–37, April.

Morel, R. J. L., Luchtmeyer, R. C. C., and Spaanenburg, L., 1982. STAS: A Mixed-Level Specification Tools for VLSI Chip Assembling and Simulation. In *Tenkon '82*, Hong Kong.

Mudge, J. C., Herrick, W. V., and Walker, H., 1980. A Single-Chip Floating-Point Processor: A Case Study in Structured Design. In *Design Methodologies for VLSI Circuits*, NATO Advanced Study Institute, Louvain-la-Nuese, Belgium.

Mudge, J. C., Peters, C., and Tarolli, G. M., 1980. A VLSI Chip Assembler. In *Design Methodologies for VLSI Circuits*, pages 329–355, NATO Advanced Study Institute, Louvain-la-Nuese, Belgium, July.

Mudge, J. C., Clarke, R. J., Potter, R. J., Smith, G. J., and Watson, C. R., 1984. Speaker-Dependent Word Recognition — A Case Study of a Custom VLSI Chip Development. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, October.

Murphy, A. C., 1984. *TicToc — A First Evaluation.* Honours Thesis, The University of Adelaide, May.

Nash, J. D. and Saunders, L. F., 1986. VHDL Critique. *IEEE Design and Test*, 3(2):54–65, April.

Navarro, J. J., Llaberia, J. M., and Valero, M., 87. Partitioning: An Essential Step in Mapping Algorithms Into Systolic Array Processors. *Computer*, 20(7):77–89, July.

Newell, A., 1969. Heuristic Programming: Ill Structured Problems. In Aronofsky, J. S., editor, *Progress in Operations Research*, chapter 10, pages 361–414, John Wiley and Sons.

Nixon, I. M., 1984. *I. F. An Idiomatic Floorplanner.* Internal CSR-170-84, University of Edinburgh, October.

Noyce, R. E., 1977. Microelectronics. *Scientific American*, 237(3):62–69, September.

O'Neill, J., 1987. Plausible Reasoning. *The Australian Computer Journal*, 19(1), February.

Otten, R. H. J. M., 1982. Automatic Floorplan Design. In *Proceedings of the 19th Design Automation Conference*, pages 261–267, June.

Payne, T. S. and Van Cleemput, W. M., 1982. Automated Partitioning of Hierarchically Specified Digital Systems. In *Proceedings of the 19th Design Automation Conference*, pages 182–192, June.

Petajan, E. D., 1986. An Architecture for High Speed Contour and Region Coding of Threshold Images. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 502–505, October.

Peterson, J. L., 1977. Petri Nets. *Computing Surveys*, 9(3):223–252, September.

Petri, C. A., 1966. *Communication with Automata.* Technical Report RADC-TR-65-377, Rome Air Force Base, New York, January.

Piloty, R. and Borrione, D., 1985. The Conlan Project: Concepts, Implementations and Applications. *IEEE Computer*, 18(2):81–92, February.

Quin, M., 1975. The Placement Problem as Viewed from the Physics of Classical Mechanics. In *Proceedings of the 12th Design Automation Conference*, June.

Resnik, M. L., 1986. Sparta: A System Partitioning Aid. *Computer-Aided Design of Integrated Circuits and Systems*, CAD-5(4):490–498, October.

Ritchie, D. M. and Kernighan, B. W., 1978. *The C Programing Language.* Prentice-Hall.

Ritchie, D. M. and Thompson, K., 1974. The UNIX Time-Sharing System. *Communications of the ACM*, :365–375, July.

Rose, C. W., Ordy, G. M., and Parke, F. I., 1983. N.mPc: A Retrospective. In *Proceedings of the 20th Design Automation Conference*, pages 497–505, June.

Rowson, J. A., 1980. *Understanding Hierarchical Design.* PhD thesis, California Institute of Technology, April.

Sarcerdoti, E., 1974. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5(2):115–135, February.

Schomburg, G., 1984. *A TicToc Description of the Complete TFB Chip.* Honours Thesis, The University of Adelaide, October.

Schweikert, D. G. and Kernighan, B. W., 1972. A Proper Model for the Partitioning of Electrical Circuits. In *Proceedings of the 9th Design Automation Conference*, pages 291–308, June.

Sechen, C. and Sangiovanni-Vincentelli, A., 1985. The Timberwolf Placement and Routing Package. *IEEE Journal of Solid State Circuits*, 20(2):510–522, April.

Segal, R., 1981. *Structure, Placement and Modelling.* Master's thesis, Caltech, February.

Séquin, C. H., 1983. Managing VLSI Complexity: An Outlook. *Proceedings of the IEEE*, 71(1):149–166, January.

Séquin, C. H., 1986. VLSI Design Strategies. In Fichtner, W. and Morf, M. ., editors, *Proceedings of the Summer School on VLSI Tools and Applications*, ETH, Zurich, Switzerland, July.

Shadad, M., Lipsett, R., Marschner, W., Sheehan, K., Cohen, H., Waxman, R., and Ackley, D., 1985. VHSIC Hardware Description Language. *IEEE Computer*, 18(2), February.

Shortcliffe, E. H., 1976. *Computer Based Medical Consultations: MYCIN.* Elsevier Scientific Publications.

Shortliffe, E. H. and Buchanan, B. G., 1984. A Model of Inexact Reasoning in Medicine. In *Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project*, pages 233–262, Addison Wesley, Reading, Mass.

Shrobe, H., 1985. Personal Communication, January. Symbolics, Inc. Cambridge, Mass.

Siewiorek, D., Bell, C., and Newell, A., 1982. *Computer Structures: Principles and Examples.* McGraw-Hill, New York.

Simon, H. A., 1962. The Architecture of Complexity. *Proceedings of the American Philosophical Society*, 106(6):467–483, December.

Southard, J. R., 1983. MacPitts: An Approach to Silicon Compilation. *IEEE Computer*, 16(12):75–82, December.

Stefik, M., 1980. Planning with Constraints. *Artificial Intelligence*, 14(2):111–139, September.

Stefik, M., 1981. Planning and Meta-Planning. *Artificial Intelligence*, 16:141–170.

Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., and Sacerdoti, E., 1981. *The Organization of Expert Systems: A Prescriptive Tutorial.* Technical Report, Xerox Palo Alto Research Centers.

Stefik, M., Bobrow, D., Bell, A., Brown, H., Conway, L., and Tong, C., 1981. *The Partitioning of Concerns in Digital Systems Design.* Technical Report VLSI-81-3, Xerox Palo Alto Research Center, Palo Alto, California, December.

Stefik, M., Aikens, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., and Sacerdoti, E., 1983. Basic Concepts for Building Expert Systems. In *Building Expert Systems*, pages 59–88, Addison Wesley, Reading, Mass.

Sussman, G. and Steele, Jr., G., 1980. CONSTRAINTS—A Language for Expressing Almost Hierarchical Descriptions. *Artificial Intelligence*, 14:115–135.

Sussman, G. J., 1981. Scheme-79 Lisp on a Chip. *IEEE Computer*, 14(7):10–21, July.

Suzuki, N. and Burstall, R., 1982. Sakura: A VLSI Modelling Language. In *Conference on Advanced Research in VLSI*, pages 201–209, MIT, Mass., January.

Szepieniec, A. A. and Otten, R. H. J. M., 1980. A Genealogical Approach to the Layout Problem. In *Proceedings of the 17th Design Automation Conference*, pages 535–542, June.

Taylor, R. T. and Johnson, M. G., 1985. A 1-Mbit CMOS Dynamic RAM with a Divided Bitline Matrix Architecture. *IEEE Journal of Solid State Circuits*, SC-20(5):894–901, October.

Trimberger, S., Rowson, J. A., Lang, C. R., and Gray, J. P., 1981. A Structured Design Methodology and Associated Software Tools. *IEEE Transactions on Circuits and Systems*, CAS-28(7):619–633, July.

Tucker, M. and Scheffer, L., 1982. A Constrained Design Methodology for VLSI. *VLSI Design*, :60–65, May/June.

Van Emden, M. H., 1975. *An Analysis of Complexity.* Volume 35 of *Mathematical Center Tracts*, Mathematisch Centrum, Amsterdam.

Van Ginneken, L. and Otten, R. H. J. M., 1984. Stepwise Layout Refinement. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 30–36, October.

Wardle, C. L., Watson, C. R., Wilson, C. A., Mudge, J. C., and Nelson, B. J., 1984. A Declarative Design Approach for Combining Macrocells by Directed Placement and Routing. In *Proceedings of the 21st Design Automation Conference*, pages 594–601, June.

Watanabe, H. and Ackland, B., 1986. Flute - A Floorplanning Agent for Full Custom VLSI. In *Proceedings of the 23nd Design Automation Conference*, June.

Watanabe, H. and Ackland, B., 1987. Flute: An Expert Floorplanner for VLSI. *IEEE Design and Test*, 4(1):32–41, February.

Watson, C. R., 1985. An Algorithm for the Correct Interconnection of VLSI-level Macrocells. *Journal of Electronic and Electrical Engineering (Aust)*, 5(3), September.

Watson, C. R., 1987. Automated Layout of CMOS Circuits. In *Microelectronics '87*, Melbourne, Australia, April.

Weinreb, D. and Moon, D., 1981. *Lisp Machine Manual.* Symbolics, Inc., Cambridge, Mass., fourth edition edition.

Weinreb, D. and Moon, D., 1981. Objects, Message Passing and Flavors. In *Lisp Machine Manual*, chapter 20, pages 279–313, Symbolics, Inc., Cambridge, Mass.

Werner, J., 1982. The Silicon Compiler: Panacea, Wishful Thinking, or Old Hat? *VLSI Design*, :46–52, September/October.

Weste, N. and Ackland, B., 1981. A Pragmatic Approach to Topological Symbolic IC Design. In *Proceedings 1st International Conference on VLSI*, pages 117–129, Edinburgh, Scotland, August.

Winston, P. H., 1984. *Artificial Intelligence.* Addison-Wesley, Reading, Mass.

Wirth, N., 1971. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, April.

Wirth, N., 1982. *Programming In Modula-2.* Springer Verlag, Berlin, West Germany.

Witt, B. I., 1985. Parallelism, Pipelines, and Partitions: Variations on On Communicating Modules. *IEEE Computer*, :105–112, February.

Yourdon, E. and Constantine, L., 1975. *Structured Design.* Yourdon Inc, New York.

Zadeh, L., 1979. A Theory of Approximate Reasoning. *Machine Intelligence*, 9.

Zyner, G., 1988. *Arithmetic Systems in VLSI.* PhD thesis, The University of Adelaide, In Prep.