Algorithm To Layout (ATL) Systems For VLSI Design

M.A. Lynch

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU

April 1986

ABSTRACT

The complexities involved in custom VLSI design together with the failure of CAD techniques to keep pace with advances in the fabrication technology have resulted in a design bottleneck. Powerful tools are required to exploit the processing potential offered by the densities now available. Describing a system in a high level algorithmic notation makes writing, understanding, modification, and verification of a design description easier. It also removes some of the emphasis on the physical issues of VLSI design, and focus attention on formulating a correct and well structured design. This thesis examines how current trends in CAD techniques might influence the evolution of advanced Algorithm To Layout (ATL) systems. The envisaged features of an example system are specified. Particular attention is given to the implementation of one its features COPTS (Compilation Of Occam Programs To Schematics).

COPTS is capable of generating schematic diagrams from which an actual layout can be derived. It takes a description written in a subset of Occam and generates a high level schematic diagram depicting its realisation as a VLSI system. This diagram provides the designer with feedback on the relative placement and interconnection of the operators used in the source code. It also gives a visual representation of the parallelism defined in the Occam description. Such diagrams are a valuable aid in documenting the implementation of a design.

Occam has also been selected as the input to the design system that COPTS is a feature of. The choice of Occam was made on the assumption that the most appropriate algorithmic notation for such a design system will be a suitable high level programming language. This is in contrast to current automated VLSI design systems, which typically use a hardware description language for input. These special purpose languages currently concentrate on handling structural/behavioural information and have limited ability to express algorithms. Using a language such as Occam allows a designer to write a behavioural description which can be compiled and executed as a simulator, or prototype, of the system. The programmability introduced into the design process enables designers to concentrate on a design's underlying algorithm. The choice of this algorithm is the most crucial decision since it determines the performance and area of the silicon implementation.

The thesis is divided into four sections, each of several chapters. The first section considers VLSI design complexity, compares the expert systems and silicon compilation approaches to tackling it, and examines its parallels with software complexity. The second section reviews the advantages of using a conventional programming language for VLSI system descriptions. A number of alternative high level programming languages are considered for application in VLSI design. The third section defines the overall ATL system COPTS is envisaged to be part of, and considers the schematic representation of Occam programs. The final section presents a summary of the overall project and suggestions for future work on realising the full ATL system.

CONTENTS

CHAPTER 1

INTRODUCTION

The 1980's have seen the realisation of silicon chips containing in excess of 1,000,000 transistors. The term very large scale integration (VLSI) is used for the technology required to produce them. Densities are further expected to increase by at least another factor of 10 before the limits of the technology are reached. A wide variety of complete systems with enormous computing power are now being designed and laid out on a single chip e.g the IMS T424 transputer chip [4] developed by INMOS. The dawn of the VLSI era has also seen an interest in a whole range of special purpose chips, typically designed to function as peripheral devices attached to a conventional host computer. The motivations behind the design of such chips are two fold. Firstly, there is the emergence of a design philosophy [47] and design tools aimed at unlocking the processing potential of VLSI. Secondly, there is the growing belief [40,65] that a significant portion of the next generation of high performance computers will be based on architectures capable of exploiting VLSI modules. In particular, it is desirable to have compact and inexpensive hosts into which interchangeable high performance modules can be plugged to fit various application requirements.

Examples of special-purpose chips include: the programmable systolic chip [38] developed at Carnegie-Mellon University by H.T. Kung and used to implement various functions including two-dimensional convolutions for signal processing applications; special purpose chips for computer graphics developed under the supervision of Henry Fuchs [17] at the University of North Carolina; and SCAPE: an image processing chip by Lea [41] and his co-workers at Brunel University.

The new quick-turnaround chip fabrication facilities will sustain the interest in special purpose chips. Application areas are expected to broaden and more algorithms previously implemented in software will be mapped into silicon [39]. The availability of this micro-chip technology, however, has resulted in a "VLSI design crisis" because of the complexity involved in the design process. This crisis centres around the effort (in terms of time and man power costs) required to translate a behavioural description of what the new chip is supposed to do into the chip layout implementing this behaviour.

The translation process is achieved by passing through several levels of abstraction, illustrated in figure 1.1. Each level attempts to reduce complexity by hiding unnecessary detail. At each level there are a number of design options that may be selected to solve a particular problem. For instance, at the algorithm level, the freedom to choose between a sequential or parallel algorithm is available. At the architectural level, a designer can select either a bit serial or a bit parallel implementation. Above the structural level, the definition of the system is usually implementation independent. During the translation process a designer typically uses several different design notations,

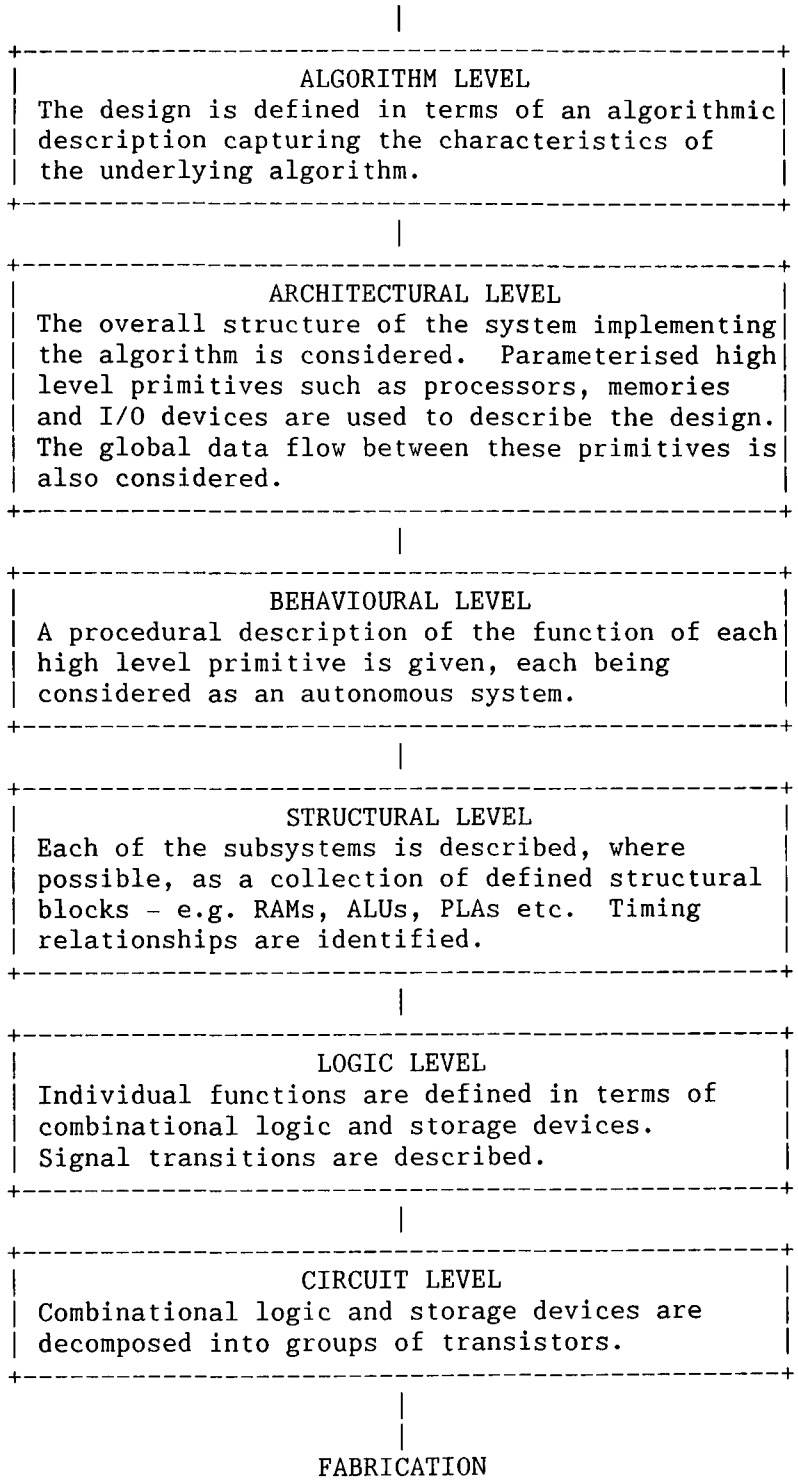


FIGURE 1.1: THE DESIGN PROCESS.

each aimed at representing a particular level of abstraction. The

designer has to verify the equivalence between the successive descriptions.

There has been a substantial growth in the number of CAD synthesis systems developed to meet the challenge of this crisis. However, the majority of these "tools" are oriented towards the transformation of design descriptions at the lower levels of the design process. Their effect has been limited, since the capabilities they offer are badly matched to the sheer complexity now available.

A lack of VLSI design expertise also contributes to the design bottleneck, since present CAD tools are "passive". That is, no assistance is given in design decision making. Designers are expected to have gained some VLSI expertise, enabling them to make such decisions. Without some method for reducing the inherent complexity of designs, the ultimate benefits to be offered by the technology will remain inaccessible.

To date, silicon area has been regarded as a limiting factor in the implementation of a VLSI design. However, if the expected densities of 10-million transistors are achieved, effectively 10 times today's area will be available. This means that for many designs silicon area will no longer be a limiting factor. It is argued that this will remove the current emphasis on layout optimisation thereby reducing the design time and design cost. The possibility of the removal of this emphasis, or at least a reduction of it, has led to an interest in integrated design systems aimed at decreasing design times by automation. One of the most important features of these systems is that a single circuit description is used throughout the design process, ensuring that the same circuit is

laid out on the chip as was originally conceived. These systems can be categorised as either knowledged based, or compilation based. The former use AI approaches to the solution of some of the many problems connected with the design of VLSI chips. In the case of the latter the trend is towards a general purpose design system providing an integrated environment similar to that currently found in software systems.

Both techniques have promise of providing a significant improvement on the traditional graphical based CAD tools. Currently, they are aimed at supporting the designer from the structural/behavioural level. Advanced powerful Algorithm To Layout (ATL) systems are required to fully exploit the processing potential available. Such systems must be capable of transforming an algorithmic description of a design into an actual layout implementation. This thesis considers the selection of an appropriate algorithmic notation for a future ATL system. The approach adopted here was to select a conventional programming language. This removes the need to define and support a special-purpose hardware design language capable of supporting algorithmic descriptions.

The material presented in the thesis gives a background to the environment in which an ATL system would serve. In particular, motivations for the future development of such systems are identified. Current trends in CAD techniques are examined to see how these might influence the evolution of ATL systems. Also, the reasoning behind the choice of the programming language Occam as the source code to an example system are given. This system is called ATLAST (Algorithm To Layout ASsisTant) and is major features are is specified. Finally, the first practical steps toward ATLAST's realisation are presented.

To begin with, Chapter 2 summarises the major features of a VLSI design methodology that has emerged to tackle the problem of design complexity. This chapter also compares the CAD techniques of silicon compilation and VLSI expert systems, both of which incorporate design knowledge. In Chapter 3 parallels are drawn between software engineering and VLSI engineering. These parallels relate the motivations behind the evolution of today's software environments to the future development of design languages and their automated translation. A brief review of the history of these languages is given and the characteristics of two example languages, MODEL and STRICT, are summarised.

Chapter 4 considers the emerging trend of incorporating programming mechanisms into design languages. The advantages of using a conventional programming language in VLSI systems design are considered. A means of identifying which of the many available languages is the most suitable for a hardware design role is presented. This requires assessing the ability of a number of languages (PASCAL, Occam, SmallTalk, Lispkit Lisp and Prolog) to capture the characteristics of an example design. These languages between them represent most of the important categories of programming language. A pattern matching chip, an example of a special purpose VLSI chip, is selected to test their ability. In Chapter 5 each of the selected languages is used to describe this design and a choice made.

Chapter 6 considers the role of Occam in VLSI, in particular, the advantageous of using the language for specifying the behaviour of VLSI systems. The features of a future ATL system (ATLAST) based on Occam are specified. One of the most important issue in its future implementation

will be the mapping of the execution sequence of Occam programs into an appropriate hardware model. The self timed and clocked (synchronous) models are considered. The work to date has concentrated on the the translation of Occam programs into a suitable intermediate graphical (schematic) representation. A compiler capable of translating a limited set of Occam programs into schematic diagrams has been developed. Its approach for extracting topological information from source programs together with brief details of its implementation are discussed in Chapters 6 and 7. The results obtained so far are presented in Chapter 8 and their limitations and implications are considered.

Chapter 9 summarises the work in this thesis and examines related work. Some suggestions for future work on VLSI design systems based on Occam are also given.

CHAPTER 2

TACKLING VLSI DESIGN COMPLEXITY

Before advances in technology made very large scale integration possible, it was usual to design and fabricate an integrated circuit of up to a few thousand gates of 'random' logic. This level of integration is commonly referred to as large scale integration (LSI). There are a variety of design styles available to support the LSI designer in the task of implementing designs. These range from full custom design to semi-custom design using gate arrays and standard cells. Since VLSI is more than just an extension of LSI, the LSI design styles alone are not sufficient for implementing VLSI designs. In VLSI design different issues such as complexity, internal structuring, and communication become relevant. Also, in the design of custom special-purpose VLSI chips flexibility becomes a key issue.

VLSI technology allows complete systems to be laid out on the unstructured domain of a single silicon chip. The designer can take advantage of this freedom to utilise the processing potential offered by the medium. However, design complexity arises from this excess freedom and the wide variety of representations offered by the silicon domain.

Complexity is a serious limitation on the implementation of random logic in VLSI designs. A suitable design methodology and set of design tools are essential if the designer is not to be overwhelmed by the problem. One approach is to restrict the designer to certain architectural styles and control the alternatives available in implementing a design. This chapter considers the use of a structured design methodology as such an approach for tackling design complexity. Also, the role of of computer aided design tools in tackling design complexity is reviewed. These tools are discussed in terms of traditional and future approaches.

2.1 THE EMERGENCE OF A DESIGN METHODOLOGY.

The complexities now involved in designing and debugging very large scale integrated systems can, it is generally agreed, only be managed by adopting some type of regular, structured design methodology. One approach, which has gained considerable recognition, was first formulated by Mead and Conway [47]. It represents a marked departure form earlier LSI approaches since the emphasis is moved from logic-level design to layout of more complex elements from symbolic descriptions or even high level behavioural descriptions.

2.1.1 Principles And Origins

The Mead and Conway methodology for VLSI system design was strongly influenced by the structured programming approach stimulated by Wirth [77], Dijkstra [13], Jackson [32], and others. The main features of the structured programming approach are:

1. Stepwise Refinement

   The creative activity of programming is considered as a stepwise refinement process. In each step, a given task of a program is decomposed into a number of simpler sub-tasks.

2. Modularity

   The degree of modularity obtained by refinement determines the maintainability (with respect to change in objectives and/or additions to its purpose) and portability of a program.

3. Notation

   A notation natural to the problem should be used as long as possible during the process of stepwise refinement. Decisions which concern details of representation should be deferred as long as possible.

4. Design Decisions

   Each refinement implies a number of design decisions based upon a set of design criteria (such as efficiency, clarity and regularity of structure), and the existence of alternative solutions. Decisions should be decomposed as much as possible.

This method has many advantages including: simplification by partitioning the program into small manageable segments (procedures, modules, or subroutines), writing and testing each segment independently, and producing more understandable programs. These advantages are achieved by placing restrictions on the programmer. However, the wide spread use of structured programming languages has shown that a structured methodology has given benefits far outweighing those that were

removed by the restrictions imposed on the programmer.

The four features outlined above were used by Mead and Conway as the beginnings of a "top down, bottom up", structured design methodology for VLSI systems. Their approach makes use of:

Hierarchical Decomposition

This corresponds to stepwise refinement. The technique of "divide and conquer" has long been used to design complex systems in a variety of fields. In applying it to VLSI design, the designer first partitions the overall system into a number of functional blocks. These are then recursively decomposed into sub-functional blocks until manageable segments of design are reached. The geometric shapes, relative sizes, and interconnection topologies of all these primitive segments are planned. Then, the system is constructed from the bottom, by assembling the segments, with a minimum of space and time wasted by interconnect wiring. A system designed in this fashion is seen as a hierarchy of building blocks, from the very lowest level device and circuit constructs up to the high level functional blocks.

Regularity

The design problem is simplified if regularity is introduced into the system. Regularity reduces the total number of different modules that have to be designed. Using this technique results in designs which exhibit one or more of the following properties [37].

(a) The system is implemented by only a few different types of simple cell.

(b) The system's data and control paths are simple and regular, so that the cells may be connected by a network with local and regular interconnections. Long distance or irregular communication is thus minimised.

(c) The system uses extensive pipelining and multiprocessing. In this way a large number of cells are active at one time so that the overall computational rate of the simple cells is high.

Abstraction

In this top down, bottom up approach the chip is viewed at a number of different levels. Abstraction, by a set of simplifying assumptions, is used to hide the lower level details of a component. Through abstraction the designer is able to focus attention on the high level design tasks, without concern for irrelevant detail. Abstract descriptions have simpler specifications than complete solutions and are generated more quickly. Boundaries (interfaces) between levels should be well defined, and kept as simple possible to avoid merging design issues. The use of a bounding box with input and output ports is a common example of abstraction. Such boxes are referred to as "cells". In the simplest case they are an abstraction from the geometry implementing a primitive function.

Notation

There are a number of possible levels of representation for circuits, ranging from symbolic layout geometry to textual descriptions. The notation used is very important since an inappropriate choice can make the design task more difficult. It should capture the features and

structuring properties of the implementation medium. Productivity is improved by using high level notations, as these are concise and easier to understand.

A design methodology in itself is not enough to overcome design complexity. Mechanical aids incorporating design methods are required to help the designer. Computer Aided Design (CAD) tools [52] are used to provide considerable assistance in the design process. By making use of the power and accuracy of computers in mapping design ideas to silicon, these tools enhance the creativity of the designer.

2.1.2 Computer Aided Design Approaches

The traditional CAD approach in VLSI design is characterised by the designer making all design decisions. The approach gives the designer an efficient environment in which to implement such decisions, by providing graphic editors, design verification and simulation tools, and databases. These tools encourage the designer to practice a design methodology, such as the one outlined above.

Designs are decomposed until seen in terms of primitive building blocks (cells). Symbolic layout tools are used to make the design of these cells more efficient and easier. Symbols, representing gates and interconnections, are sketched out on a display device and manipulated by means of a graphic editor. These symbols are then automatically replaced by the appropriate geometries for implementing the represented function. Primitive cells are assembled to form larger cells in the bottom up construction. This involves placement and routing of interconnections. Automatic layout programs are available. Usually there is a tradeoff

between area and design time in this stage of the design. This tradeoff determines the amount of human design effort required and the extent to which automatic design methods can be used. Design verification tools are used to check for human design errors. For example, a design rule checker is a commonly used tool. It ensures that the dimensions of the shapes and their relationship with other shapes in the layout comply with predefined values. Simulation tools then check that the derived layout implements the intended logic function of the circuit.

The advantage of this approach is that it can result in designs which are of high quality, since humans are very good in optimising designs. The major disadvantage is that the human designer is slow and error-prone. Also, the design tools are totally dependent upon the expertise of the designer. They are limited by the fact that they are based on graphical input. The approach places the emphasis on layout and a 'bottom up' design style. Higher levels of abstraction are needed to tackle the complexity of designs which can now be implemented. Textual descriptions, in formal hardware description languages, are more powerful than symbolic representations since they are capable of supporting a structured top down approach. These limitations have led to an interest in approaches supporting textual design descriptions and which incorporate "design knowledge". Two approaches are being proposed: VLSI expert systems and silicon compilation.

A VLSI expert system (see section 2.2) is centred around a knowledge base, holding design expertise in the form of design rules. Such a system "assists" the designer in the design process by providing expert advice. It tackles local design issues and implementation details,

freeing the designer to concentrate on global issues. Expert systems are intended to be less error-prone than traditional CAD techniques and more efficient than the human designer. In order to achieve these improvements a comprehensive knowledge base accumulating the experience of many experts is required. Knowledge can be incrementally added, and new design styles and new architectural concepts can be catered for by including additional design rules in the knowledge base [18]. High density designs can be handled, but with a corresponding increase in design time.

Complete design automation is implied by silicon compilation (see section 2.3). The philosophy behind this approach is that design knowledge is algorithmic and translators can be written to synthesise a solution, or part of one, automatically from a high level description. Such an approach is completely opposite to the traditional and expert system approach,since the human designer is replaced rather than assisted. A complete design automation system significantly shortens design times. However, the cost of reduced design times is measured in terms of the excessive use of silicon area.

An interesting approach to the synthesis of data paths and control sequences from an algorithmic description is under development at AT&T Bell Laboratories [36]. Researchers have combined expert systems with algorithmic approaches to automate as much as possible of this synthesis process. Their design strategy is to break the integrated circuit design process into stages, and implement each stage as programs. Some of these programs are expert systems, while others are compilers. The former are used for high-level hardware synthesis such as quality floor planning,

whereas the latter are used for the lower stages of chip design e.g. layout and placement.

2.2 EXPERT SYSTEMS

Expert systems have been developed by researchers in artificial intelligence to help solve non-numeric problems. These system attempt to construct a model of the human ability of pattern recognition. The essential feature of such a model is that problems can be solved by recognising patterns and linking previously obtained solutions. Sufficient "processing power" is now available in today's computers to apply this approach in VLSI design [1,25].

A VLSI expert system is capable of providing solutions to a range of design problems. A top down, incremental refinement design process is supported by using multiple abstraction levels for system description. Such an approach provides design leverage by enabling a designer to deal with critical issues early and across the breadth of a design. Usually these systems consist of two components: a design knowledge base, and inference mechanisms to manipulate it. The knowledge base is a set of rules and facts summarising design expertise. Symbolic expressions are used to embody expertise and represent the rules and facts. Inference mechanisms are programs which direct the system in its search for a particular solution. Traditionally symbolic languages such as Lisp have been used to implement these systems, but logic programming languages are expected to replace these languages [26].

Palladio [7,60] is an example of an integrated VLSI expert system. It uses multiple description levels similar to those employed by Mead and Conway. Incremental refinement of design specifications is supported, with periodic validation of the specification by simulation. Knowledge base design aids perform some of the refinement necessary to move from an abstract description level, termed a perspective, to a more physical circuit specification. The multiple perspective framework provided by Palladio simplifies the implementation of such expert systems. The design aids provide the designer with feedback on the consequences of circuit refinement decisions, quickly enabling him to avoid unnecessary layout errors.

A Palladio perspective is either structural or behavioural. This explicit decoupling of behavioural concerns from structural concerns allows a user to adopt a modular design style. Each perspective has a set of terms and a set of composition rules. Terms define the allowed types of subsidiaries which can be used to describe a component with respect to a particular perspective. Composition rules limit the way in which terms can be interconnected and combined. The concerns of each perspective are characterised by specific classes of bugs that can be avoided when the composition rules are followed. A design is first specified at a high level architectural perspective, and then incrementally refined with the help of a set of integrated knowledge based tools to a symbolic layout perspective.

An expert systems approach should capture more of the creative design process than the traditional or silicon compilation approaches and so should produce better designs [64]. The multi-level representations

typically supported allows design tradeoffs between more detailed modelling and model simplification. In addition, an expert systems approach enables formal verification of transformations from one representation to another. Feedback from the lower levels can be used to control high level synthesis. It is easy to add rules since the knowledge base is separate from the inference mechanism. Performance improves as knowledge is added.

Specifying knowledge as symbolic rules and facts make such systems easy to understand. With a knowledge based system it is possible to automatically generate an explanation of how it derived its solution. For example, the program may indicate the chain of "if-then" rules which was used to make a decision in the design of some circuit element. Also, the approach supports easy interaction with a designer. These features simplify the task of determining what is incorrect or incomplete about the system's knowledge base.

A few expert systems have been developed for design tasks, for example DAA [36], VEXED [48], REDESIGN [61], and Fujistu's computer aided logic design system [67]. However, a generic framework for constructing such systems is yet to emerge. Current VLSI expert systems are slow, requiring considerable amounts of CPU time. For efficient designs the knowledge base required must be large and comprehensive. Moreover, the rules of thumb that a human expert designer uses are often very difficult to quantify and to express in a form that an expert system can use. Additionally, VLSI design is an emerging art with, as yet, no formal design methodology. This makes it difficult to express explicitly all the reasoning and background knowledge used by good designers.

Experience with expert systems is limited, it is still a very new field with more research needed. One of the most promising applications of expert systems is in front-end design. This is an area where very few design aids have previously existed other than limited simulation techniques. For example, the work of Kowalski of CMU on a VLSI Design Automation Assistant uses an expert systems approach to manipulate one level of design description into a lower level [36]. Expert systems and logic programming languages such as Prolog can also be used to solve some basic VLSI design problems in the areas of design for testability, functional simulation, fault diagnosis, and automatic test generation [30].

2.3 SILICON COMPILATION

The term silicon compilation was first introduced by Johannsen [34] to describe the concept of assembling parameterised pieces of layout. A silicon compiler can be defined as a design tool that automatically translates a high level functional or behavioural description of a chip into a layout implementation [75]. This translation is usually seen as a two step process. Firstly, a brief high level description of a design is translated into an expanded intermediate description, which is still implementation independent. Then a chip layout is generated automatically from this description. An "ideal" silicon compiler would work on a general class of designs, cater for many design techniques, and not restrict the architecture that designs are implemented in.

The technique of silicon compilation creates design leverage [35] by: keeping the design activity at a high level; allowing quick architectural exploration of design alternatives; synthesising intermediate views of the design from one common abstraction; integrating the support tools (e.g. simulators, layout generators etc.); and allowing an incremental or successive design approach. Of great importance is the source language. This is the language in which the designer describes the behaviour to be performed by the integrated circuit. A source language for silicon compilation must [3] provide:

1. a means for directly specifying behaviours that are supported in the target silicon (e.g. parallelism);

2. overall integrity so that the language remains a language of behaviour, rather than merely a language of layout.

Many of the software systems classed as silicon compilers are more accurately described as structure compilers. Essentially, these compilers remove the logic description stage and circuit-design stages for some finite set of functions, or operators. Their source codes are hardware description languages which are usually restricted to a high level structural description of a number of the implemented functions. Often, the syntax and semantics of these languages resemble those of assembly languages. Consequently, they have limited usefulness, since it is difficult to write a valid description of any significant length or complexity. Moreover, the structural information users are required to provide can be at an extremely detailed level.

Some of the silicon compilers in use today do produce a layout description from a behavioural description of a design. However, in order too achieve this, the design problem has been simplified. Instead of handling a wide variety of design styles, designs are implemented in a "target" architecture. Examples of this approach are: Dumbo [78] which implements a standard cell array architecture, MODEL [23] which uses a gate array architecture, FIRST [6] which derives a bit serial architecture, and the Data Path Generator [57] for constructing a standard data path architecture. This simplification makes mapping straight forward, enabling results to be produced quickly. Their source codes cannot be classed as general purpose circuit design languages as the semantics of the languages reflect specific architectures.

MacPitts [59], for example, supports routines which automatically synthesise a Data Path from an algorithmic description of a design. It is aimed at designs which can utilise parallelism in such an architecture. MacPitts allows a designer to specify an algorithm as though completely general and sufficient parallelism existed in the data path of some general purpose computer. Then, the MacPitts compiler derives the minimum hardware micro-programmed machine which executes that parallel algorithm. The compiler consists of routines at two levels. The higher level routines extract a technology independent intermediate level description in terms of data path specifications, control equations, and state assignments. The lower level routines translate the intermediate description into mask data. The resulting structure is topologically similar to any micro-programmable machine's architecture.

2.4 CONCLUSIONS

There are two distinct and opposing views on the future impact of silicon compilation [74]. On the one hand, chip layout is regarded as analogous to deriving machine code from a high level program. By enabling a chip design to go from a high level description to a layout, via a compiler, the IC development effort should be reduced by a factor of perhaps 20. The opposing view holds that chip design has nothing to do with the problem of writing software. Software is a one dimensional problem, whereas chip design is a two dimensional one. The routing and placement skills of a human designer can never be matched by automatic techniques - total automation is just inappropriate. Instead, CAD tools should support the human designer who directs the design. An expert systems approach captures more of the creative design process and so produces better designs.

Certainly, in the commercial field there is a competitive need for optimum performance in terms of speed and area used. On commercial scales of production the overheads incurred through manual optimisations can be recouped for viable chips. Current silicon compilers can produce, at best, design two or three times as large as manual designs. They are limited in application by the simplification of adopting target architectures. Experience in implementing such compilers is limited. However, as more experience is gained, researchers in the field are optimistic that technical problems will be overcome. Compilers are expected to improve until they are competitive with traditional techniques, and ultimately surpass them.

Here, silicon compilers are not seen as a tool which can replace the need for skilled VLSI design engineers. Rather, they are seen as tools which will bring the following benefits.

1. Reduce Costs

   An efficient compiler with a wide range of application will greatly reduce design costs and times. This will make the design of "one-off" special purpose VLSI chips more feasible. Reducing costs and times will also stimulate designs in smaller establishments with no large system production facilities.

2. Serve as a training tool

   VLSI designers are skilled engineers. Their skills are obtained through training and hands on experience. Any tool which could shorten the learning period is advantageous. A silicon compiler would enable a designer to quickly discover design approaches and quickly become familiar with architectural concepts.

3. Expand Application Areas

   High level software compilers have made the underlying hardware much more usable. As a result, application areas have vastly expanded and diversified. High level silicon compilers will also make the micro-chip technology much more accessible to people in other fields (i.e. non VLSI design specialists). The range and function of special purpose chips will therefore expand rapidly.

4. First Time Right Implementations

   Providing the compiler has been verified, complete automation implies 'correctness by construction' and so the silicon produced is guaranteed to be correct.

The impact of silicon compilers has so far been limited [15]. For the technique to gain a greater acceptance the simplification of restricting a compiler to a specific architecture needs to be removed. Also, higher levels of abstraction for design descriptions need to be supported. An automated tool based on a high level design language and capable of handling a wide range of applications is required. One approach to realising such a tool would be to develop a single general purpose compiler capable of solving all a designer's problems. Alternately, several special purpose compilers could be integrated into a single design system. Such a system would take an algorithmic design description and transform it into an intermediate structural/behavioural description. This description would serve as a common form of abstraction for the compilers. Each compiler would be dedicated to a particular architecture, with the complete suite of compilers representing the various important architectures. A system of this type is defined here as an Algorithm To Layout System. In effect such systems are "high level silicon compilers". The next chapter considers how techniques used in software engineering and the evolution of design languages might influence the development of such systems.

CHAPTER 3

AUTOMATED VLSI DESIGN

The current VLSI design complexity crisis is very similar to the crisis faced by software engineers at the end of the 1950's. Their response was to move from assembly languages to "high level" programming languages, such as FORTRAN and COBOL. An essential feature of this move was the development of the technique of software compilation. The degree of complexity which could then be tackled was greatly increased. However, the unstructured nature of these early programming languages meant that the complexity problem was not removed, only pushed further back. Consequently, very high level programming languages and corresponding compilers evolved. Such languages attempt to reduce the complexity problem further by supporting design methodologies, structuring techniques and documentation styles.

This chapter compares VLSI design with software design, and identifies software techniques which can be transferred to the VLSI design domain. Next, the development of hardware description languages is reviewed. Finally, the emerging trend towards high level design description languages is considered in connection with the development of

the technique of silicon compilation.

3.1 A COMPARISON BETWEEN SOFTWARE AND VLSI ENGINEERING

Software and VLSI designers share a common goal - layout. The software designer lays out a one-dimensional array of memory, whereas the VLSI designer lays out a two-dimensional area of silicon. For both, various constraints must be satisfied in order to obtain a working product. Also, in each case, there is a design path from a system specification to its actual implementation. Over the past twenty five years a software discipline has evolved to support the translation process involved in the design path. Recently, a VLSI design discipline has also evolved, but compared to that for software it is still in its infancy. Techniques similar to those developed by software engineers are being re-discovered by VLSI engineers, which suggests that a number of parallels can be drawn in their route from specification to implementation [28,54]. If the evolution of the VLSI discipline is regarded as analogous to that for software, then many of the lessons learnt in the software domain can be transferred to the VLSI domain. Some of the parallels between the two will be explored in this section.

3.1.1 Specification Levels

Today's programmers have powerful compilers which generate executable machine code from specifications written in very high level programming languages. These languages have evolved from the first machine languages via assembly languages and then procedural languages. Each new generation brought with it a higher level of abstraction of the

system under design. In the early days attention was focused on physical issues at the the assembly language and machine instruction levels because memory layout was a serious limitation. Coding was a time consuming and error prone activity. Technological advances lessened the physical memory constraints and, freed from this bottleneck, the ambition of the systems designed increased. However, the low level of abstraction offered by assembly languages restricted the complexity of designs which could be tackled in a realistic time period. This restriction was overcome by the development of automated tools - software compilers. Compilers translate an abstract specification into low level assembly or machine code.

It appears that a similar hierarchical evolution is occurring in the VLSI domain, but at a faster pace. Chapter one identified several levels of design specification in use. VLSI designers have tended to concentrate their effort at the lower levels of abstraction. Again this is due to a physical constraint - chip area. Recent advances in the scaling technology are beginning to lesson this constraint, enabling designers to implement much more complex systems in the same chip area. As their software counterparts discovered, low levels of abstraction severely limit the complexity which can be tackled. Consequently the higher levels of specification are taking on more significance. Just as the higher level programming languages have evolved as the basic building blocks of software engineering, so perhaps will higher level design description languages become the building blocks in VLSI engineering. For this to happen, the translation technology needed to support such abstractions must mature.

3.1.2 Complexity

During the 1970's software complexity increased to the point where the system specification phase became critical. With the ever-increasing size of software projects being undertaken, design techniques became the predominant area of software research. Structured top down techniques enabling designers to think in terms of "function" rather than "code" replaced bottom-up software techniques. These techniques impose constraints upon the designers to help solve the problem of complexity.

The problem of complexity at the system level has already been encountered in the VLSI domain. Semi-custom design techniques developed by LSI designers are not applicable for systems with the complexity of VLSI. This is because the building blocks (e.g. cells) used in these techniques are too small to serve as a starting point. Top-down techniques as used in software have already being adopted. The use of a top down approach for VLSI design was considered in the previous chapter.

3.1.3 Compilers

The complexity and scope of designs that can now be tackled by software engineers have been accomplished by advances in compilation techniques. Such techniques have matured to the point where parsers can be automatically produced from formal descriptions of the language to be compiled. Software compilers map high level descriptions down to the level of the actual implementation (assembly or machine code level). In this mapping process many errors are caught before execution through the use of techniques such as type checking. In addition, optimisation techniques, which are an intrinsic part of compilers, result in object

code whose performance is comparable to manually generated assembler code for most applications. Compilers produce this code far more quickly and reliably than any programmer. However, skilled programmers, willing to devote a large amount of time, can still outperform compilers by a factor of 2 or 3 in small program segments. This effort is only justified if there are hard constraints such as real-time requirements or the need to fit a program into a limited amount of main memory.

The compilation technique is also being applied in the VLSI domain, but current silicon compilers are nowhere near as advanced as their software counterparts. Software compilers map from the programming level to the machine level, whereas silicon compilers map from the design description level to the geometry level. The machine level is characterised by a one-dimensional array of memory and sequential activity in the time dimension. The geometry level is characterised by two spatial dimensions (even three dimensions) and a set of concurrent operations in the time dimension. The mapping for hardware is therefore far more complicated and involves significantly different problems. Current silicon compilers use simplifying assumptions (e.g fixed geometry) which ease their task, but limit their application.

3.1.4 Operating Systems

The productivity of software designers and the magnitude and complexity of systems implemented has increased dramatically over the last twenty five years. In part, this has been due to the advances in programming languages and compilers. But the increase can also be attributed to the development of operating systems (O/S). O/S enable

scarce resources such as the memory and the central processing unit to be shared among a number of users in a manner transparent to each user. Each user is provided with a virtual machine. They relieve users of redundant coding of commonly used functions. The global throughput is improved although individual performance is occansionally sacrificed. They provide an integrated software environment in which a user can interact with a wide variety of tools e.g. compilers, editors, debuggers, databases, filing systems etc. The entire software development process is supported by O/S.

Integrated VLSI design environments are starting to receive attention and there are aspects of operating systems which may be applicable in the VLSI domain. Smith and Dallen [58] suggest that a "Silicon Operating System" (SOS) might:

1. Improve designer productivity by allowing large numbers of designers to share (or work on) the same chip without concerns about the effects on other designers, or physical limitations. As in software O/S, the need to re-implement commonly used functions would be eliminated.

2. Provide a medium in which design components could be integrated to form a VLSI chip.

3. Handle some of the implementation details, such as timing (synchronisation) constraints, chip input and output, and the assignment of components to physical areas of a chip.

In such a SOS, designers of individual components would work with virtual chip area (cf. the virtual machine in software O/S), and would not be concerned with the components physical location, its position relative to other components it interfaces with, or their relative speed. A "chip area manager" would be responsible for assigning (placing) components to physical areas on the chip and the communication (routing) between components. Constraints on the maximum amount of virtual chip area allocated to a design component and guidelines for its aspect ratio would probably have to be made for the approach to be realistic.

3.1.5 Differences

Several software engineering techniques have been identified as being readily applicable to VLSI design. However, to conclude this section some of the problems specific to the VLSI field must be outlined. These problems arise out of the two dimensional nature of integrated circuits and the need for physical interconnections between components.

In software implementation programmers need not (usually) concern themselves with the overhead of jump (GOTO) instructions, since they cost only the memory required for the jump instruction itself. The distance between the location of the instruction and the target address in no way effects either execution time or the amount of memory used. In contrast a silicon jump between communicating components requires area for the connecting "wire", and time for signals to propagate. For two components with a high interaction, physical proximity is crucial since the overheads for long, high bandwidth interconnections are severe. Furthermore, a connecting wire must be routed to avoid obstacles and

prevent unwanted short circuits. A software jump has no obstacles.

The wire carrying a signal may be represented on any one of several layers. At some point, a signal has to change layers in transit and this requires a complex of three or more shapes to make the transition. In software all locations in memory are on the same layer. Wires must also compete for chip area, which places topological restrictions on implementations. There are no equivalent restrictions in software.

Re-arranging software procedures (components) is easily done with the use of an editor. Changing the placement of a component in a layout is much more difficult. Very few design tools re-route the attached wires automatically when a block is moved to a different position in the chip floorplan. Graphical editors for two dimensional layout are still in the research stage. Also, changing the size and shape of one component may require a change in neighbouring components. Changing the size of a software procedure has no effect on other procedures unless memory size is very limited.

The software design process is characterised by short iteration times. Once a program is completed and its syntax is correct it can be compiled and executed. Logical errors can be quickly discovered by running the program. Debuggers are available to help designers track down less apparent errors. The performance of the program can be evaluated by time measurements and the effects of modifications quickly discovered. Software is maintainable - changes to a program's specification can easily be accommodated. The very long iteration times in the VLSI design process give designers limited scope for modifications. The effect of a minor design change can take weeks to be

seen in a finished chip. Lengthy simulations are required to detect errors and give performance estimates. Why a particular chip does not meet the required specification may have to be guessed at or deduced from indirect evidence. As yet, there is no equivalent of a debugger for VLSI chips. Once committed to silicon there is little designers can do with regards to changes in a chip's specification. In short, the pressure on the VLSI designer is to get it right first time.

By supporting high level structured specifications and providing an integrated design environment incorporating tools utilising the techniques of compilation, ATL systems will increase the chances of achieving 'right first time' implementations. In addition ATL systems will support fast design turnaround time. The algorithmic notations used by these systems will replace the current design languages now being used. In the next section the evolution of these languages will be reviewed in order to gain an insight into the expected features of the algorithmic design notations. These will then be considered in the next chapter.

3.2 HIGH LEVEL DESIGN LANGUAGES

The emergence of a design methodology was reflected in the development of layout languages. Originally, these languages were very low level (e.g. CIF [47]). However, they allowed the user to design in a more structured and regular manner than had previously been available. These features enhanced the creativity of the designer. The basic idea behind layout languages was the arrangement of geometric shapes into patterns which represented the integrated circuit being designed. The

low level of abstraction offered by these languages made them tedious to use and as a result design descriptions were error prone. These languages were improved by embedding procedure calls to generate layout information in a high level programming language. This technique made features such as variables, assignment, iteration and parameterised procedures available to the designer. An example of this approach is PLAP [73], which was developed at the University Of Newcastle. This design tool is based on the programming language PASCAL. The drawback of this technique is that it is limited to layout concerns. Also, as the technology advanced, to maximise the increasing densities still required a considerable design effort.

Gradually, design languages which were independent of the ultimate layout began to appear. Such languages enable the designer to describe a design in terms of its behaviour and how this description can be implemented structurally. These structural/behavioural languages are similar in philosophy to regular high level programming languages. In a language like PASCAL, which has a straight forward compiling mechanism, the programmer has control over the memory organisation during execution, but not the physical memory elements actually used. Similarly, the designer using a structural/behavioural language has control over the organisation of the chip, but not the individual gate elements.

High level languages that allow both behavioural and structural specifications can be divided into two classes: procedural and non-procedural. The former includes languages such as MODEL [22] and VHDL [56] which provide a hardware description in the manner of a plotting program that specifies the hardware components and their

interconnections (i.e. a net list description). In the later, hardware is described as a function composed of sub-functions. The composition of the functions is well-structured and does not allow the unsystematic specification of interconnections. Examples of non-procedural languages are: CONLAN [50], ZEUS [19,42] and STRICT [8]. The hardware design language ELLA [49] supports both explicit net list descriptions and implicit interconnection of components through functional definitions. As an example of procedural structural/behavioural languages MODEL will be examined. STRICT will be examined as an example of non-procedural languages.

3.2.1 MODEL

The MODEL language [22] was developed to support only structured designs. Potential ambiguities in the interpretation of design descriptions are removed by only considering such designs. It is used as the source language to a silicon compiler, written by Lattice Logic. Both the language and compiler are in commercial use. Design descriptions in the language are implemented as semi-custom integrated circuits using CMOS ULAs. This is in contrast to other design languages which attempt to support custom designs. It also means the compiler has a target implementation and is therefore not technology independent.

Descriptions in MODEL are hierarchical. They are translated via the compiler into an intermediate design file, which is used by a physical design subsystem to generate masks, test pattern generators, functional and timing simulators, and placement and routing tools. Descriptions are easy to read and compact.

The MODEL notation is strongly influenced by the structured programming style. The fundamental structuring feature is the concept of a part. A part is a module with one or more input signals and one or more output signals. Parts can also have numeric parameters. The definition of a part specifies its internal structure in terms of instances of simpler parts and their interconnections. An instance of a part is the use of that part within the definition of a more complex part. Descriptions express the design of the circuit at a number of levels, starting with the entire circuit at the top, and descending to the primitives at the bottom. These levels are represented by parts, each reasonably small and with a well defined interface. This style encourages systematic debugging as each part can be tested in isolation. The style also aids verification as higher-level parts make calls on existing, tested parts, thus making it easier to verify a part formally.

The language supports two data types: signals and integers. Signals are the basic objects manipulated by the language and may appear in one dimensional vectors. The major control structures of high level programming languages are also supported. These allow parts to be more than just simple interconnections of other parts. A comprehensive, parameterised design library adds strength to the language.

MODEL is rather inflexible in that all wiring and connections must be explicitly stated in the description of the design. That is, there is no scope for optimisations by hand wiring. Also, the size of the design is limited, since large or complex designs may not fit onto particular ULAs because of limitations in channel size. However, the fact that it has gained commercial acceptance proves that it is a definite improvement

over previous methods for designing this type of architecture.

3.2.2 STRICT

STRICT [8] was developed as part of a joint project between the Computing Laboratory and the Department of Electrical and Electronic Engineering at the University of Newcastle. It is intended to provide a formal, declarative notation for designing structured integrated circuits in a consistent manner. It was designed for interactive use with a syntax directed editor. This editor produces an intermediate representation, which can be transformed into a format suitable for input to various subsystems, such as layout, simulation, fabrication etc.

The language supports the parallel, sequential and recursive descriptions of systems from modular components. There are two types of component: buses and blocks. A bus transmits information of a specific type from one part of the system to another. A block manipulates information.

Blocks are constructed from two parts: a specification and an implementation. A specification defines the interface to the outside world and describes the block's intended function. An implementation describes how the block's specification can be structurally implemented. Each block is designed separately and through typed design parameters can be tailored to a particular interface specification or hardware implementation.

A specification is composed of an interface specification and a block declaration. An interface specification defines the inputs and outputs to a particular block. These are typed, and can be optionally positioned on particular sides of the block by using edge identifiers. A block declaration specifies the function of the block. It includes any restrictions on the design parameters, declarations for the interface types, and convenient function definitions.

The implementation of a block is defined in terms of instances of other blocks, the connection between these blocks, and how they should be structurally arranged. Connections are made by calling each instanced block with its input parameters. STRICT uses the technique of strong typing to produce consistency in the connections.

The language supports a number of standard types, operations, and functions. User defined types are also supported. This enables the designer to move to a level of abstraction where complex data structures can be considered simply.

The declarative and recursive nature of the language simplify the structuring features required in VLSI design. The level of abstraction employed hides details concerning pads, power and ground lines etc. Unlike MODEL inputs are implicitly connected to outputs. The use of a functional style results in designs which are concise and can be formally reasoned about. However, the syntax and recursive nature of the language can make designs difficult to understand. Some architectural concepts currently employed in VLSI design, such as pipelining, are not catered for in the version of STRICT examined. Encouraging a designer to provide a specification before an implementation is, in principle, beneficial.

However, in practice it is often quite difficult for users to distinguish a boundary between the two.

3.3 THE FUTURE

Design languages have developed from simple "plotting" notations to sophisticated structural/behavioural notations incorporating high level programming techniques. Research into the latter is actively being carried out in both academic and industrial environments. Consequently, there has been a marked increase in the number of high level design languages in use. As illustrated by STRICT and MODEL there are also a variety of styles to choose from. This rapid growth has even led to an attempt by the American Department Of Defence to introduce a standard language, VHDL [55]. The decomposition of descriptions in these languages can be regarded as low level silicon compilation. It is a significant improvement on structural compilation.

Although structural information in design descriptions is still of great importance today, purely algorithmic notations are being considered for the next generation of high level design languages [71]. They are intended to serve as general-purpose circuit design languages. The extreme of such a notation would be a high level programming language, in which the algorithm to be performed by the circuit is written. Compilation of such a description is regarded as "true" silicon compilation [68]. With this in mind several programming languages will now be considered for their potential as the source code to an ATL system.

CHAPTER 4

POSSIBLE SOURCE LANGUAGES FOR A FUTURE ATL SYSTEM

The trend in VLSI design languages is toward 'programming like' languages which can support both the behavioural specification and the structural implementation of VLSI circuit designs. Also, the emphasis is shifting from structural information to behavioural information as higher levels of abstraction are used to tackle complexity in system specifications. Assuming both trends continue they will ultimately converge at an algorithmic, high level programming language. For this language to be supported by an ATL system, it must be amenable to a straightforward translation into a physical VLSI implementation. Probably the most significant factor in determining the effort involved in mapping a programming language onto silicon (and hence its suitability as the source code to an ATL system) is its underlying computational model. If this model accurately represents VLSI designs and closely reflects the structural geometry and properties of their silicon implementations then there will be a direct (i.e. straightforward) translation process. Key implementation features are: parallelism, communication, and localised processing.

Chapter 4 first considers the advantages of using programming languages for VLSI design descriptions. In order to identify which of the many available languages is the most appropriate, a set of design criteria are presented and an example VLSI implementation is also described. In the next chapter several example programming languages will be used to describe the design. Their performance in a design description role will be evaluated against the actual VLSI design implementation and the selected design criteria presented in this chapter.

4.1 PROGRAMMING LANGUAGES FOR VLSI DESIGN

Programming language structures such as the conditional, loop and procedure have been recognised by the designers of Hardware Description Languages (HDLs) as being as powerful for describing hardware as they are for describing programs. In HDLs loops are used to generate repeated structures; conditionals are used to build structures depending on the environment; and procedures are used to describe blocks of related logic. The advantages of incorporating some programmable capability into HDLs have also been recognised. Typically, this feature is seen as the ability of a hardware language to handle arithmetic expressions. Such expressions can be used anywhere the user might specify a value. In particular, expressions are used to compute the actual values for formal parameters of procedures, enabling the designer to develop powerful general purpose procedures. Programmibility relieves the user of the task of manual computation and introduces automation into the design process. More importantly it enables the designer to think in terms of algorithms. Algorithmic specifications are more direct and natural

descriptions and enhance the ability to produce correct and working ICs.

As HDLs are extended to increase their expressive powers they will themselves become complete programming languages. Can the effort of developing specialised programmable HDLs and building reliable and efficient translators for them be avoided? It is argued here and elsewhere [51] that the answer is yes, since current programming languages are sufficiently general to be good hardware description languages.Features exhibited by current high level programming languages which make them attractive as design notations are listed below.

1. They support powerful, precise algorithmic descriptions which can be tested to validate their correctness. (This testing usually involves example runs with different input data).

2. They provide functions and structures which capture implicitly the bulk of a specification thereby reducing the length of specifications.

3. They provide facilities for the definition of abstract data types and for checking that these types are respected. Type checking traps many errors at compile time thus quickly eliminating careless mistakes.

4. They support a modular style of program which encourages correctness and makes them easy to understand and maintain. Separate compilation allows the construction of extensible program libraries.

5. They are flexible i.e they are suitable for a wide range of applications.

6. They can provide concurrency, explicitly or implicitly.

A high level general purpose programming language would make the layout task more like programming. This would enable a designer to concentrate on the problems of the "high level" design algorithm. Concentrating on this is important since, according to H.T Kung [37], the most crucial design decision is the choice of the underlying algorithm. Thus the algorithm design should receive the largest part of the design effort. He also argues that low-level optimisations at the circuit or layout design level are probably not worthwhile, as these will lead only to minor improvements in the overall performance while increasing design time.

A design automation system (i.e. a silicon compiler) based on a conventional programming language will be simpler to implement and more extensible than one based on a specialised hardware description language. In such a system the designer describes a VLSI system in the chosen language and compiles it with a standard compiler. The compiler would need to be extended in order to produce a representation of the resulting circuit as well as executable machine code. When a program description is run, not only would it be capable of processing it would also produce a detailed description of the elements of the circuit and their interconnection. This description could then be used by a layout subsystem to generate a complete VLSI implementation.

Procedural Programming

Computational Model = Control Flow. Concepts: global memory of cells, assignment as the basic action, and implicitly sequential control structures for the execution of statements. Two important classes of procedural programming notations are:

- Conventional e.g. PASCAL

Developed for programming the traditional von Neumann stored program computer. Hence, the semantics reflect the von Neumann Model: global memory, fixed size memory cells, and sequential execution.

- Concurrent e.g. Occam

Extend the control flow model with parallel control structures based on processes plus communication and synchronisation mechanisms.

Object-Oriented Programming e.g. SMALLTALK

Computational Model = Actor. Computation is based upon active objects, sometimes called actors, that communicate by passing messages.

Functional Programming e.g. Lispkit Lisp

Operates by the application of functions to values. Characterised by no sequentiality, no assignment statements, and no side effects.

Logic Programming e.g. PROLOG

Attempts to solve goals, which fail or succeed, when answering a question.

FIGURE 4.1: Programming Categories.

Today, there is a great variety of high level programming languages in use. Obviously, it is not practicable to consider each one for VLSI design. Many of these languages, however, are based on the same computational model. Since the computational model of the language chosen will reflect the structural geometry of VLSI designs, it would be

more applicable to examine languages typical of some of the various models. A classification of programming languages, based on computational mechanisms, is presented in [21]. Several example languages, which between them, represent some of the important categories in this classification will be examined. Figure 4.1 illustrates the selected languages together with a summary of the important features of the category to which each language belongs. The characteristics of each language are given in the next chapter.

4.1.1 Requirements Of A Hardware Design Language

The suitability of a programming language for describing VLSI designs is governed by its ability to satisfy certain design requirements. The criteria used for selection here are based on those presented in [11]. Accordingly, a circuit design language should exhibit the following properties.

1. The language must be able to handle concurrency.

2. The language should capture

   i) Structural Data e.g. what objects are required, how these objects are to be connected.

   ii) Behavioural Data e.g. specifying the overall function/purpose of the design

3. The language should be easy to use. That is, the language should be concise and descriptions should be easy to read and write.

4. The language should encourage structured, regular, and hierarchical (modular) design techniques.

5. The language should be supported by expandable libraries of frequently used layout designs.

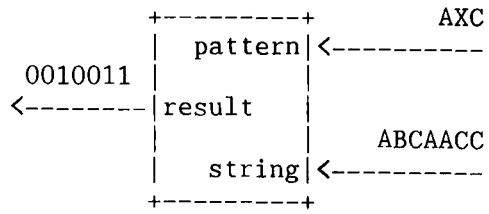
6. The language should provide an effective means of communicating designs between co-operating designers.

4.2 AN EXAMPLE VLSI DESIGN: A PATTERN MATCHER CHIP

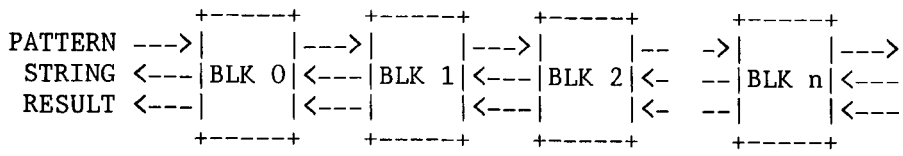
String pattern matching is a good example of a software application which is well suited for implementation as a special purpose VLSI chip connected to a general purpose computer. In this section a specific pattern matching chip is described in terms of its underlying algorithm and its structural implementation. This chip was selected as an example design because it illustrates the importance of the algorithm on the performance and area of the silicon implementation. It also illustrates one of the most important benefits offered by VLSI systems, namely concurrent processing.

4.2.1 Design Description

The pattern matcher chip described by Foster and Kung [16] is a special purpose VLSI chip that performs pattern matching of strings with wild card characters. The input and output characteristics of the chip are shown in figure 4.2(a). There are two input streams: string and pattern. The former is an endless string of characters, whereas the latter contains a fixed length vector of characters containing a wild



(a) Chip Inputs And Outputs



(b) Block Structure

FIGURE 4.2: The Pattern Matcher

card character. The chip generates as output a stream of bits, each of which corresponds to one of the characters in the text string. The data streams move at a steady rate between the host computer and the chip, with a constant time between data items.

Denote the input text stream as S0 S1 S2... , the finite pattern stream as P0 P1 P2...Pn, and the output result stream as R0 R1 R2...,. Characters in the two input streams may be tested for equality, with the wild card character, 'X' say, deemed to match any character in the text string stream. The output bit Ri is to be set to 1 if the sub-string S(i-n) S(i+1-n)...S(i) matches the pattern, and 0 otherwise. As an example consider the following two input streams:

pattern: AXC (where X is the wild card character)

string: ABCAACCQQ...

The following result stream should be generated:

result: 001001100...

The pattern AXC matches the sub-strings S0 S1 S2, S3 S4 S5, and S4 S5 S6 (ie ABC, AAC, and ACC respectively). Result bits R2, R5, and R6 are thus set to 1 and all other result bits are 0.

Formally that is

Ri <== (S(i-n) = P0) AND (S(i+1-n) = P1) AND...AND (Si = Pn)

4.2.2 Algorithm Design

The input strings arrive alternately over the link with the host, one character at a time. The interval during which one character arrives from either stream is termed a beat. During each pair of consecutive beats the chip inputs two characters (one pattern and one string) and outputs one result bit.

The chip is divided into a linear array of modules called character blocks. Each block can compare a pattern and a string character and accumulate a temporary result. Characters are successively 'clocked' through the blocks; on each beat a character moves to a new block. Pattern and string characters flow through the array in opposite directions, with each string character moving past all characters of the pattern. The pattern and string characters are separated by one cell so that each pair of characters meet rather than just pass. Consequently each cell is active on alternate beats.

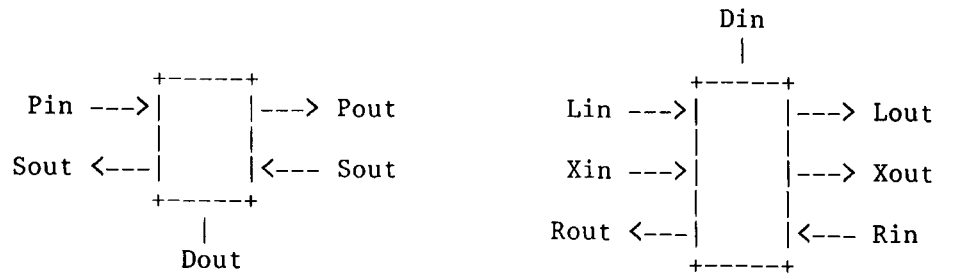
A block diagram for a pattern matcher is shown in figure 4.2(b). Initially, each block of the chip is empty. For simplicity, assume that on the first beat, the first character to be taken off the host link is a string character S0. This character is input to BLK n. On the next beat the pattern character P0 is taken off the link and input to BLK 0. At the same time S0 is moved one cell to the left. On the third beat the string character S1 is placed in BLK n, and P0 is moved one block to the right while S0 is moved one block to the left. By the time the last pattern character Pk leaves a block, the sub-string S(i)S(i+1)...S(i+k) will have met the whole pattern. If partial match results are held in a block and updated whenever a new pair of characters enter the block then, when the last character of the pattern goes through, the result of comparing the two will have been accumulated. A block then outputs this result, which moves along with the string, so that each match result leaves the array with the last character of its sub-string. The pattern is recirculated so that the first character follows two beats after the last one. This enables a block to output a completed result and initialise a new partial result on the same beat.

Each character block performs two functions:

1. it compares characters of the pattern and string,

2. it accumulates and outputs match results.

Each function can be implemented by a separate unit within a block, a comparator cell and an accumulator cell respectively. These are shown in 4.3(a) together with their underlying algorithm. A comparator cell has



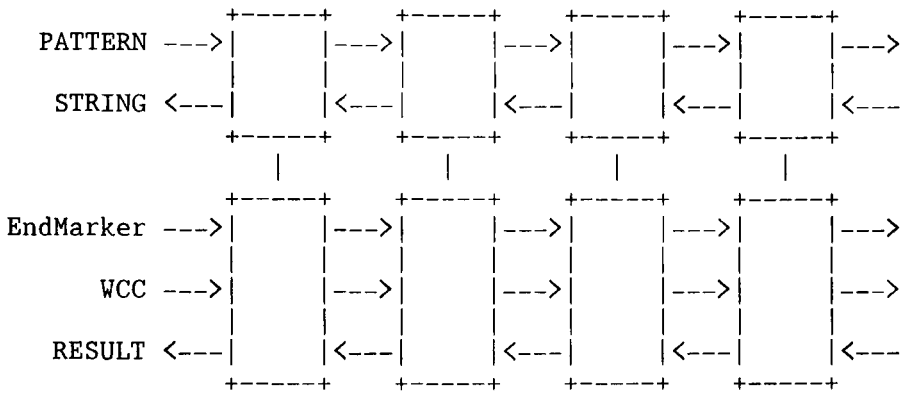

```
Pout := Pin
Sout := Sin
Dout := Pin = Sin
```

COMPARATOR CELL

```
Lout := Lin
Xout := Xin
t := t AND (Xin OR Din)
IF Lin THEN
   begin
        Rout := t;
        t := TRUE
   end
ELSE
   Rout := Rin
```

ACCUMULATOR CELL

(a) Primitive Cells And Their Algorithms



(b) Cell Topology And Interconnection

FIGURE 4.3: The Pattern Matcher Implementation

pattern characters flowing from left to right, string characters flowing from right to left, and the comparison result for a pair of characters flowing from the bottom. Two bits are associated with a pattern

character: the don't care bit and the end marker bit. The former is used to denote the occurrence of the wild card character in the pattern. The latter is used to denote the last character in the pattern sequence. An accumulator cell receives the result from a comparator (Din), the end of pattern indicator (Lin), and the don't care bit (Xin). It maintains a temporary result 't', and at the end of the pattern uses t to replace the result that flows from right to left. The topological arrangement and interconnection of cells are shown in figure 4.3(b).

In the next chapter this design will be described in several programming languages as a means of evaluating their suitability for VLSI design. The most appropriate language will be the one which is most capable of accurately describing the algorithm and representing the implementation presented here.

CHAPTER 5

SELECTING A PROGRAMMING LANGUAGE

In order to select a suitable language for a future ATL system, the computational mechanisms of the example languages will be compared with the structural mechanisms employed in the pattern matcher. To facilitate this comparison, simple programs describing the design will be examined for each language. In these programs the algorithm has been programmed in a relatively simplistic way, attempting only to match the example language to the structures of the algorithm. The segments of code included here, illustrate the different styles and represent prototype solutions. Every attempt has been made to ensure that these solutions represent valid code in the various languages. However, no guarantee of their correctness can be given.

5.1 PASCAL

Pascal is a reasonably extensive language [33] offering data structuring facilities. The primitive data types are: Boolean, integer, char, and real. Structured types are defined by describing the types of their components and by specifying a structuring method. Four such

methods are available: array structure, record structure, set structure, and file structure.

Variables declared in explicit declarations are called static. Dynamic generation of variables in an executable statement is possible. It yields a pointer which serves to refer to the variable.

The assignment statement is the language's fundamental statement. The procedure statement causes execution of the designated procedure. Assignment and procedure statements are construed to form structured statements. These statements specify sequential, selective or repeated execution of their subsidiary statements. Sequential execution of statements is specified by the compound statement, conditional or selective repeated execution by the repeat statement, the while statement, and the for statement.

Statements can be named and referenced through the given identifier. A named statement is then called a procedure, and its declaration a procedure declaration. A procedure may contain a local set of variable declarations, type definitions, and further procedure declarations. These can only be referenced within the procedure itself. A procedure has a fixed number of parameters. There are four kinds of parameters: value, variable, procedure, and function.

Functions are declared in a similar manner to procedures. The only difference lies in the fact that a function derives a result which is confined to a scalar or pointer type. This type must be specified in the function declaration. Functions may therefore be used as constituents of expressions.

5.1.1 Pattern Matcher Implementation

The major features of a pascal program for the pattern matcher algorithm are summarised in figure 5.1, a more detailed form of the program appears in Appendix B.1. A modular structure is achieved through the use of a function and three procedures. These are called from the main program segment. The structure and components of figure 4.2 are represented by a one-dimensional array called "Modules". This array is made of "Ncells" elements (where Ncells is a constant defining the number of pattern modules). Each element is a record of type "PatternCell" and represents a pattern module. They contain two fields: CompCell and AccCell. CompCell is a record designed to represent the state of a comparator cell. It is used to store the string and pattern characters currently associated with such a cell. Similarly, AccCell is a record defined to represent the state of an accumulator cell. It holds the don't care bit, the end bit, the accumulated result, and the output result currently residing in an instance of this cell.

Function "Compare" implements the algorithm for a comparator cell. It has a single value parameter: "Cell" a record of type Comparator. The result of the function is of type BIT, which is the integer sub-range 0..1. If both the characters held by its parameter are identical then Compare is set to "1" otherwise it is set to "0".

Procedure "Move" implements the left to right movement of string characters, don't care bits, and end marker bit between adjacent pattern cells. It also implements the right to left movement of string characters and result bits between adjacent cells. The procedure has two variable parameters of type PatternCells, which correspond to the two

```
FUNCTION Compare(Cell : Comparator) : BIT;
BEGIN...END;

PROCEDURE Move (VAR left, right : PatternCells);
BEGIN...END;

PROCEDURE Accumulate(DataIn : BIT; VAR Cell : Accumulator):
BEGIN...END;

PROCEDURE InOut;
BEGIN...END;

(* Main Body Of Program *)
BEGIN
     Beat := 0;
     WHILE NOT finished DO
     BEGIN
          InOut;
          FOR i := 1 TO (Ncells DIV 2) DO
          BEGIN
               WITH Modules[(i*2)-Beat] DO
                 Accumulate(Compare(Compcell), AccCell);
               Move(Modules[(i*2)-Beat], Modules[((i*2)-1)-Beat]);
          END;
         Beat := 1 - Beat;
     END;
END.
```

Figure 5.1: Pascal Program

adjacent cells. Procedure "InOut" handles the input of pattern and string characters to the array and the output of the result bits.

Procedure "Accumulate" implements the algorithm for accumulator cells. It has a single value parameter "DataIn" which imports the result from a comparison cell. There is one variable parameter, an accumulator record.

In the main program segment there is a conditional while loop, which implements the overall behaviour of pattern modules. Execution of this loop is terminated when the variable "finished" is set to true. The variable "Beat" toggles between 0 and 1 on successive passes through the

body of the loop. Nested within the while loop is a FOR loop in which Beat is used to determine the individual modules active during a pass through the main loop. When Beat is set to 0 odd numbered cells are active. Even numbered cells are active when Beat is set to 1. The FOR loop contains the calling sequence to the Accumulate and Move procedures. In the call to Accumulate, the actual parameter corresponding to the DataIn parameter is a call to the function Compare. This function is first evaluated and its result then passed to the parameter.

The array and record structuring methods enable the abstract data structure 'Modules' to be defined. This structure is intended to represent the structure of figure 4.2. It is a list composed of n elements, each element describes the internal state of a pattern matching module. The elements are sub-divided into two components, one for describing the internal state of an accumulator cell and the other for describing the state of a comparator cell. Although each element accurately captures the internal states, the input and output ports of the cells are not represented.

The function Compare and the procedure Accumulate describes the algorithm which is common to all modules. The main body of the program applies the cell algorithms and attempts to model the connectivity of cells through the update of elements in the array. Both data movement and pattern block algorithm are applied by sequentially accessing each element in "modules". In other words, the main program defines a single computational process which is sequentially applied to update the internal state descriptions held in the array.

The sequential nature of the language is its major limitation in a design description role, since it results in descriptions specifically aimed at a Von Neumann serial processing architecture. The parallelism and localised processing characterising the pattern matcher implementation cannot be represented. An imperative language, such as Pascal, was included in the survey because the underlying sequential control flow model has played a central role in the evolution of programming. No survey of programming languages would be complete with out reference to this model. It is not particularly well suited for capturing the concurrent aspects of VLSI design. The model would be more appropriate for representing finite state machine descriptions.

5.2 OCCAM

Occam [31] is based on dynamically created processes which may be executed concurrently and may communicate using Channels. The fundamental working element in Occam is a process - a single statement, group of statements or group of processes.

Programs are constructed from three primitive processes: assignment, input, and output. To control the order of execution of such processes Occam provides three control mechanisms: sequential (SEQ), parallel (PAR), and alternate (ALT) as well as the traditional WHILE and IF statements.

SEQ and PAR precede a list of processes, defining sequential and parallel execution, respectively. ALT causes exactly one of a list of processes to be executed and will wait until at least one of the "guarding" conditions is true.

5.2.1 Pattern Matcher Implementation

An outline of an Occam implementation for the pattern matcher algorithm is shown in figure 5.2; a complete listing appears in Appendix B.2. The program is divided into five sections. The first section declares vectors (or arrays) of channels, which are used in the main body of the program. So, for example, "pattern[Ncells]" declares five channels named pattern and numbered 0 to 5.

In each of the following three sections PROC is used to declare a name for the text of the process which follows. The text associated with a named process is substituted for all occurrences of that name in the subsequent process. The named processors are: Comparator, Accumulator, and GetChar. Each has a number of formal channel parameters. When any one of the processes is called these formal parameters are replaced by the actual parameters.

Process Comparator consists of two sequential processes. One initialises the local variables "p" and "s". The other is a repetitive process which handles input/output. WHILE TRUE denotes an unbounded loop which executes endlessly. The body of this loop is composed of three sequential processes. The first and second are output and input processes respectively, while the third compares p and s, and outputs the truth value on the channel "dout".

Process Accumulator implements the algorithm for the accumulator cells. Like comparator it also consists of two sequential processes: one initialises the local variables, the other is a repetitive. The repetitive process is comprised of three sequential processes.

```
PROC Comparator(CHAN PatrnIn, StrngIn, PatrnOut, StrngOut, DataOut)=
  :

PROC Accumulator(CHAN WildBitIn, EndBitIn, ResIn, DataIn, WildBitOut,
                     EndBitOut, ResOut)=
  :

PROC GetChar(CHAN BusIn,PatternIn,StringIn)=
  :

PAR
  GetChar(Sys.Bus,pattern[0],string[Ncells-1])
  PAR i =[ 1 FOR Ncells-1 ]
    PAR
      Comparator(pattern[i-1],string[Ncells-i],pattern[i],
                       string[(Ncells+1)-i],data[i-1])
      Accumulator(wild[i-1],end[i-1],result[Ncells-i],data[i-1],
                          wild[i],end[i],result[(Ncells+1)-i])
```

FIGURE 5.2: Occam Implementation Of The Pattern Matcher.

The first two are output and input, while the last is a conditional process. If the value of EndMarker (a variable representing the end of pattern) is true an accumulator uses the value of the current comparison result (CurntRes) as the final result and then resets CurntRes to TRUE. Otherwise, it holds a temporary result (TempRes), which is set by the logical expression :

```
TempRes := TempRes /\ (WildCard \/ CurntRes)
```

where "/\", "\/" stand for AND and OR respectively. So, if the current temporary result is TRUE, and WildCard or CurntRes is TRUE, then the new temporary result will be set to TRUE.

Process GetChar models the alternate arrival of pattern and string characters on the system bus. That is, the input to the pattern matcher module. It consists of two sequential processes, the first of which

simply initialises the local variable "Beat". The second is a repetitive sequential process with the two subsidiaries. Its first subsidiary concurrently re-assigns the value of Beat and inputs a value for "Ch" from "BusIn". The second subsidiary is a conditional process: if the value of Beat is 1 then Ch is placed on Channel "PatternIn", otherwise it is placed on channel "StringIn".

The final section corresponds to the main process (or program body). It sets up a 2x3 array of concurrent processes, which models the structure of figure 4.2. It also sets up a process to handle input to this array.

The program provides an accurate representation of the pattern matcher capturing its structure, its concurrent behaviour, and the inter communication between the cells. In the main process a parallel replicator is used to define a pipeline of accumulator cells and a pipeline of comparator cells. An element in one pipeline is connected to an element in exactly the same position in the other pipeline. The channels which link the elements directly model the behaviour of a physical wire. Each element in the pipelines is a distinct process capable of local computation and having its own private storage. The CHAN construct ensures that the inter process communication is synchronised. The WHILE TRUE construct in the two named processes ensure the repeated execution of the processes.

5.3 SMALLTALK

Xerox Palo Research Centre's Smalltalk-80 [20] is built on the model of communicating objects, originally used in SIMULA. It has sequential

control structures. All references in the language are to objects, which consist of some private memory and a set of operations. The private memory is a set of instance variables. A set of methods describe how to implement an object's operations. Objects may be atomic, or may consist of several named fields.

A message is a request for an object to carry out one of its operations. The receiver of the message determines the method to implement the requested operation. The set of messages to which an object can respond is called its interface.

A class describes the implementation of a set of objects that all represent the same kind of system component. Classes are the natural unit of modularity in the language. The individual objects described by a class are called its instances. A class is defined by giving it a name, and naming the fields of its instances. Following this is an optionally categorised list of the methods for processing the messages to which the class responds. Each method consists of a message pattern followed by SMALLTALK code within brackets for computing the appropriate responses. The former consists of a selector and names for the arguments. The code consists of some temporary variable names and expressions to process the received message. Expressions are separated by periods "." and the last one may be preceded by a vertical arrow indicating the value to be returned. These expressions contain conventional expressions that serve a similar role to procedure call.

Objects are created when expressions are evaluated, and they can be passed around by uniform reference, so that no provision for their storage is necessary in the procedures that manipulate them.

The transmission of messages is the only process that is carried on outside of objects. A message-sending expression defines the receiver (cf. the procedure), the selector (cf. the entry point), and the arguments of the message.

Control structures, other than the sequential execution of expressions in a method and the sending of messages that invoke other methods, are based on objects called blocks. Blocks contain a sequence of expressions. Execution of blocks may themselves be controlled by the conditional selectors "ifTrue" or "ifFalse", and by conditional iteration "whileTrue" or "whileFalse".

5.3.1 Pattern Matcher Implementation

Brief details of a SMALLTALK implementation of the pattern matching algorithm are shown in figure 5.3, an expanded form of this appears in Appendix B.3. The implementation centres around three classes "Comparator", "Accumulator" and "PatternMatcher". Two other classes, "PatternBlocks" and "BitStreams" (see Appendix B.3) support initialisation of objects.

```
Class   Comparator

Methods
   Compare: stringchar and: patternchar


Class   Accumulator

Super Class   Object

Instance Variables   accRes

Methods
   initialise

   update: dataIn with: resIn and: bitsIn

   initaccRes


Class   PatternMatcher

Methods
go: nblocks fromhost: charstream tohost: bitstream
   |pattern, string, result, bits, cells, toggle|
   "initialise variables"
   toggle <- 0.
   [charstream isEmpty]
    whileFalse:[i|
      "input string or pattern character"
      i <- toggle + 1.
      [i <= nblocks]
       whileTrue:
         [results at: i <-
            cells at:i (update:(Compare:(string at:i) and:(pattern at:i))
                        with: result at:(i+1)
                        and: bits at:i).
          pattern at:(i+1) put:(pattern at:i).
          string at:(i-1) put: (string at:i).
          bits at:(i+1) put:(bits at:i).
          i <- 1].
      toggle <- toggle - 1]
```

Figure 5.3: SMALLTALK implementation of the pattern matcher

Instances of class comparator represent the results of character comparisons. Comparator responds to a single message pattern with an instance representing the result of comparing a string character with a

pattern character. This message pattern consists of the selector "compare:and:" and two arguments, named stringchar and patternchar. When a message of this type is sent, the expression in Comparator's only method is evaluated. This expression consists of a message to Boolean with the selector "ifTrue:ifFalse:" and two blocks as arguments. The value returned from ifTrue:ifFalse: is the value of the block that was executed. This value is then returned to the sender of the message.

An instance of class Accumulator is used to represent the accumulated result in an accumulator cell. This class has one instance variable "accRes". The implementation description includes one class method and two instance methods. The class method creates and initialises new instances. It has selector "initialise". When a message with this is received, the method creates a new instance by evaluating the expression "super new"; it uses the method for new found in the methods of the superclass "object". It then sends the new instance the message initaccRes. The search for the response begins in the class of the instance i.e. in Accumulator. An instance method is found there which assigns the value of 1 to the instance variable accRes.

The third method of Accumulator implements the operation of an accumulator cell. It is invoked by a keyword message with selector "update:with:and:" and containing the values for the arguments dataIn, resIn, and bitsIn. The argument dataIn represents a comparison result, resIn represents the result flowing through an accumulator cell, and bitsIn is a two element array representing the don't care bit and the end bit respectively. The method uses a temporary variable to hold the value to be returned. It consists of two expressions, the first of which

contains the "ifTrue:ifFalse" selector and two blocks. The ifTrue block uses an important "pseudo-variable" available in every method named self which refers to the message receiver itself. When the expression "self initaccRes" is executed, initaccRes is sent to the same object (self) that the received "update:...". This results in the instance variable being reset to 1.

An instance of class PatternMatcher is used to represent a pattern matching chip. It has a single method which is invoked by a message pattern containing arguments which define the number of pattern modules required (nblocks), the input stream from the host (charstream), and the output stream to the host (bitstream). Once invoked, the methods temporary variables are first initialised. The variables pattern, string, and result are initialised as arrays with nblock elements. Also initialised as arrays are bits and cells. However, the elements of bits are initialised to be arrays of two elements, while those of cells are initialised to be instances of the class accumulator. Having initialised all variables the expression "[...] whileTrue:[...]" is executed. This expression is a control message which repeatedly evaluates the expressions in the second block as long as the condition in the first block holds. The second block contains two expressions; the first increments the temporary variable toggle and the second is another conditional repetition. During successive executions of the second expression toggle alternates from 1 to 2.

The repeated block of the nested conditional consists of five expressions, the first of which places a new value in the i th element of the array results. This value is obtained by sending the instance of

accumulator stored in the i th element of cells the message with selector "update:with:and:". Before this message is sent the expressions defining the values for this message's arguments are evaluated. So, for example, the actual value for dataIn is obtained by sending Comparator the message "Compare:(string at:1) and:(pattern at :i). The parenthesised expressions must also be evaluated before this message can be sent. These expressions obtain the values at the i th position of arrays string and pattern, i.e. the actual values for the arguments stringchar and patternchar respectively. The next three expressions move an element in pattern, string and bits, while the last expression increments i by +2. During the repeated execution of the block the value of toggle is used to determine which elements in the arrays are updated. For example when toggle is 1 the instances of Accumulator at odd numbered positions in cells are used to obtain the new values for results. This is equivalent to odd numbered cells being active. Similarly, even numbered cells are active when toggle is 2.

The class construct provides a good template for describing both the computational aspects of an algorithm and the data it acts on. This feature was illustrated by the class definitions for accumulator and comparator cells. Their associated messages implement explicitly defined entry points (cf. input ports) and attempt to capture the input behaviour of the corresponding cells. The arrays pattern, string, bits and results are used to model the data movement through the pattern matcher. An element in these arrays can be regarded as a connecting wire and its value the current value on that wire. So, for example, the wire on which the i th accumulator cell receives the result from the (i+1) th cell is represented by element (i+1) of results. The current value held in this

element represents the most recent value placed on the wire by the (i+1) th accumulator cell.

The array cells was used to hold a number of identical objects, each representing an accumulator cell. These instances can be thought of as individual processes capable of local computation. The expression used to invoke a particular instance in this array accurately reflects the communication between a comparator cell and an accumulator cell. An instance variable is used to describe the internal state of each cell. This can only be affected by the appropriate message to this type of object. It gives a good representation of the local storage associated with each accumulator cell.

The main drawback of this implementation is the sequential nature of the language. Although local computational elements could be described, parallel operation of these could not be represented. One way round this would be to expand the semantics of the language such that concurrent operation of objects could be accommodated. However, the aim of this chapter is to assess the language's current ability to describe a VLSI implementation, not to consider how the language could be modified so as to be suitable for such a role.

5.4 LISPKIT LISP

The Lispkit Lisp language [27] was developed by Peter Henderson. It is a purely functional derivative of Lisp 1.5. It contains no iterative constructs, instead recursive control structures are used. A data structure, once defined, cannot be altered. This implies that all operations on a data structure have a copying semantics feature.

There are three types of objects: atoms, lists and functions. Atoms are indivisible data objects, either symbolic or numeric, or parenthesis. Lists (or symbolic expressions) are built up from atoms. Two forms of primitive function are available: those which manipulate symbolic expressions and those which handle basic predicates and arithmetic operations.

The language provides a number of functional forms e.g. the conditional form "if x then y else z". User functions are defined by specifying a name and providing the formal arguments, each referenced by an identifier string. The language is weakly typed, and the programmer is responsible for passing arguments of the appropriate type and structure to a function.

Functions can accept each of the three types of objects as arguments, and return each as the result of evaluation. Therefore, a function may also be passed as an argument and/or returned as the result of evaluating a function. In this way, special purpose functions may be created from general purpose functions. These are termed higher-order functions.

5.4.1 Pattern Matcher Implementation

The major features of a Lispkit Lisp implementation of the pattern matcher are given in figure 5.4 (a more complete listing of the program is included in Appendix B4). The function clock simulates the overall behaviour of the pattern matcher chip. It has four input parameters: beat, p, s, and m. The parameter beat is an integer which is either 0 or 1 and determines which character blocks

```
clock(beat, p, s, m) ->
      if eq(beat,0) then
         {cons(car(l), clock(1, p, cdr(s), cdr(l))
          where l = move1(car(s), matchodd(m))}
      else
         cons(nil, clock(0, cdr(p), s,
                   move2(car(p), car(m), matcheven(cdr(m)))))

move1(s_char, m)  ->
      cons(car(car(m)),
           to_even(s_char, car(m), car(cdr(m)), cdr(cdr(m))))

move2(p, head, tail) ->
      cons(left(p, head, car(tail)),
           to_odd(car(tail), car(cdr(tail)), cdr(cdr(tail))))

matcheven(m) ->
          if eq(cdr(m), nil) then
             pmatch(car(m))
          else
             cons(pmatch(car(m)), cons(car(cdr(m)),
                                       matcheven(cdr(cdr(m)))))

matchodd(m) ->
       if eq(cdr(cdr(m)),nil) then
          cons(pmatch(car(m)), cdr(cdr(m))
       else
          cons(pmatch(car(m)), cons(car(cdr(m)), matchodd(cdr(cdr(m)))))

pmatch(inpts) ->
       cons(a1, cons(a2, cons(c1, cons(cons(a3,a4), c3))))
       where a1 = e(1,a),  a2 = e(2,a),  a3 = e(3,a),  a4 = e(4,a),
             c1 = e(1,c),  c2 = e(2,c),  c3 = e(3,c),
             p = e(3,inpts),  s = e(4,inpts),   t = e(2,inpts),
             r = e(1,inpts),
             c = comp(car(p),s),
             a = acc(t, r, car(cdr(p)), car(cdr(cdr(p))), c3)

comp(p,s) ->
     if eq(p,s) then
        cons(p, cons(s,1))
     else
        cons(p, cons(s,0))

acc(a,r,x,l,d) ->
    if eq(l,1) then
       cons(u, cons(1, cons(x,l)))
    else
       cons(r, cons(u, cons(x,l)))
    where u = and(a,or(x,d))
```

Figure 5.4: Lispkit Lisp Implementation

in the pipeline are active. If the value of beat is 0 then even numbered blocks are active, otherwise the odd numbered ones are. The parameter p is a list in which each element is a 'pattern list'. A pattern list has three elements: pch, x, l, where pch is a pattern character, x is the don't care bit, and l is the end bit. The string stream is represented by the character list s. Finally, the input parameter m is a list of n elements (where n is the number of blocks in the implementation), each of which is a pattern block input list. An input list has the format (r,t,p,s), where r is a result bit, t is the accumulated result associated with a block, p is a pattern list, and s is a string character. Once the function is called with the parameters initialised to the start up conditions, it recursively calls itself building up a list of result bits.

If beat is 0 clock simulates the behaviour of the pipeline when odd numbered blocks transform inputs to outputs, which then form the inputs to the even numbered blocks. Also, during this phase, a result bit is output from the left most block, and a string character is input to the right most block. The activity of odd numbered elements is implemented by the function matchodd. The result of this function together with the head of the string list are used by move1, which implements the input/output behaviour and the movement of data from odd numbered blocks to even numbered ones. If beat is 1 then clock simulates the behaviour of the pipeline when even numbered blocks transform their inputs into outputs. No result bit is output during this phase, however, a pattern character and its associated bits are input into the left block of the pipeline. The activity of even numbered elements is implemented by the function matcheven, while move2 implements the input and transfer of data

from even numbered blocks to their right and left neighbours.

The function move1 has two parameters: s_ch and m. These represent a string character and the current outputs of the blocks respectively. The function returns a list, the head of which is the result bit from the left most block, and the tail of which is the list returned by the function to_even. This tail list is similar to the input parameter m. Odd numbered elements in it are identical to the corresponding elements in m, while even numbered elements are updated versions of the corresponding elements in m. The function to_even implements the transfer of data from odd numbered elements to even numbered elements and the input of a string character. It has four arguments s,l,c,r where s is the string character, l is an input list for an even numbered block's left neighbour, c is the input list for this even block, and r is the list containing the input lists for all the blocks to the right of the even block. If the tail of r is empty then function right is called, otherwise c is updated and the remainder of the even numbered elements in r are updated. Although right is not defined here, it simply updates the input list for the right most block using the next string character and the appropriate outputs of its left neighbour.

Function move2 has three input parameters: p, head, and tail. The parameter p is a pattern list, head is the input list associated with the left most pattern block, and tail is a list containing the input lists for the remaining blocks. The function returns a list whose head is the input list returned by the function left and whose tail is the list returned by to_odd. Although not included, left is a simple function that defines the input list for the left most block using the pattern

list p and the appropriate elements of its right neighbour. The function to_odd is similar to to_even, but implements the transfer of data from even numbered blocks to odd numbered blocks.

Both to_odd and to_even use the function left_right. This function demonstrates how a block's inputs are obtained form the appropriate outputs of its left and right neighbour. The function is not defined here, but it has three parameters: left, centre, right. Each is an input list, left for a block's left neighbour, right for its right neighbour, and centre for that block. It returns a parameter list based on the elements in these lists which represents the updated inputs to the centre block.

The function matchodd is used to represent the pipeline when odd numbered pattern blocks are active. It has a single parameter, m, which is a list of pattern block input lists. It returns a similar list in which all the even numbered elements are identical to the corresponding elements in m. The odd numbered elements, however, are updated versions of the corresponding elements in m. Function, matcheven is similar, only it represents the pipeline when even numbered pattern blocks are active. Both use the function pmatch to update the input list associated with an active block.

The function pmatch implements the behaviour of a pattern block. This function has a single parameter, Inpts, which is an input list for a block. It returns an updated version of this list, which represents the outputs of a pattern block. That is, the function captures a block's transformation of inputs into outputs. Within the function body, local definitions are used to make it more readable. A primitive function

e(n,l) is used to return the n th element in the list l. The output list is built up from two subsidiaries. One of these lists is returned by the function comp and the other by the function acc. The former function implements a comparator cell, while the latter implements an accumulator cell.

Lispkit Lisp allows a concise hierarchical description of the pattern matcher to be written. The function pmatch captures the input/output behaviour of a pattern matcher block. It uses the functions acc and comp to illustrate how a block is sub-divided into two primitive elements (or cells). The parameters to functions and the values returned by them describe the connectivity between elements. In the language, no restrictions are placed upon the order of evaluation of sub-expressions. Parallelism can therefore be implied without the need for it to be explicitly defined by the programmer. This is particularly true in the functions matcheven, matchodd, to_even, and to_odd. These functions can be interpreted as replicating an operation a number of times with the occurrences operating concurrently.

The lack of an assignment statement in the language and the absence of a shared memory posed problems in representing the state of a block. It was not possible to hide the internal state of a block. Instead the current value associated with a block was fed in as an input parameter. Its updated value was then returned as an element of the output list generated by pmatch. This suggests a feedback loop for the accumulated result. Although there is no problem with this in terms of circuit design, it does not truly reflect the structure of figure 4.2.

The major drawback of the language is that a function can only return a single value. This meant that for multiple outputs a single list had to be constructed. Combining values into such lists is awkward, it distracts from the main purpose of a function and makes it less readable. Another drawback of the language is the lack of an explicit representation of control flow. This makes it difficult to describe purely sequential circuits.

5.5 PROLOG

PROLOG [10] is a programming language well suited to solving problems that involve objects and the relationships between objects. A program in the language consists of facts about a certain subject, expressed as a collection clauses of which express information that can be used to solve problems or to answer questions. A predicate defines a relationship, and is either an assertion or an implication.

PROLOG attempts to sequentially solve the composite goals of a predicate. For a given goal, it attempts to find a clause whose head can be made to match the goal. If the clause is an implication then it, in turn, attempts to solve its subgoals. The possible results of a goal will be failure or success, plus possible values associated with variables. To achieve success for a goal, all the subgoals must succeed. If one of the subgoals cannot be solved, PROLOG backtracks and tries to find another clause whose head matches the goal. If no untried clauses remain, then the failure is returned for the goal.

The scope of a PROLOG variable is limited to the statement in which it appears. It is a non-procedural language. A statement in a PROLOG program corresponds to an entire subroutine of a conventional programming language. Thus programs are extremely modular. The order in which statements occur is irrelevant.

5.5.1 The Pattern Matcher Implementation

Details of a Prolog implementation for the pattern matcher are given in figure 5.5 and in Appendix B5. The overall behaviour of the pattern matcher is implemented by the predicate "pipeline", such that pipeline(Plist,Slist,Rlist,Mods) emulates the repeated operation of the chip. Plist is the pattern list; it is an endless list in which each element is a sub-list containing three atoms: [P_ch,Xbit,Lbit]. P_ch is a pattern character and Xbit and Lbit denote the don't care bit and the end of pattern bit respectively (either 0 or 1). Rlist is a list of atoms which are either 1 or 0 and represents the results output stream. The head of this list defines the last result to be output from the pattern matcher. Mods is a list of n objects, where n is the number of pattern blocks in the implementation. Each object is a list describing the parameters to a block: [R,T,P,S]. R is an atom representing the result, T is the accumulated result, P is a list of type [P_ch,Xbit,Lbit], and S is a string character.

The predicate pipeline generates an endless list of result bits. When it is first invoked Rlist should be initialised to an empty list i.e. [], whereas the other three arguments should be initialised to define the start up conditions. In the body of pipeline there are five

```
pipeline([P_ch|Pattern], [S_ch|Strng], Res, Blocks):-
         matchodd(Blocks, Blocks_1a),
         transfereven(S_ch, Rout, Blocks_1a, Blocks_1b),
         matcheven(Blocks_1b , Blocks_2a),
         transferodd(P_ch, Blocks_2a, Blocks_2b)
         pipeline(Pattern, Strng, [Rout|Res], Blocks_2b).

matchodd([Ablock, Lblock], [Nblock, Lblock]):-
         process(Ablock, Nblock).

matchodd([Lblock|[Rblock|Tailblks]], [Nblock|[Rblock|Rest]]):-
         process(Lblock, Nblock),
         matchodd(Tailblks, Rest).

matcheven([Blk_j,Blk_k], [Blk_j,Nblk]):-
          process(Blk_k, Nblk).

matcheven([Lblk|[Rblk|Tailblks]], [Lblk|[Nblk|Rest]]):-
          process(Rblk, Nblk),
          match(even, Tailblks, Rest).

transfereven(S_in, [Blk_j,Blk_k], [Blk_j,Ublk]]):-
             right(S_in, Blk_j, Blk_k, Ublk).

transfereven(Sin, [Blk_i|[Blk_j|[Blk_k|Rest]]], [Blk_i|[Ublk|Tailblks]]):-
             exchange(Blk_i, Blk_j, Blk_k, Ublk),
             transfer(Sin, [Blk_k|Rest], Tailblks).

transfereven(Schar, Res, [Blk1|Tail], Nblks):-
             out(Blk1, Res),
             transfer(Schar, [Blk1|Tail], Nblks).

transfer([Blk_i,Blk_j,Blk_k], [Blk_i,Ublk,Blk_k]):-
         exchange(Blk_i, Blk_j, Blk_k, Ublk).

transfer([Blk_i|[Blk_j|[Blk_k|Tail]]], [Blk_i|[Ublk|Rest]]):-
         exchange(Blk_i, Blk_j, Blk_k, Ublk),
         transfer([Blk_k|Tail], Rest).

transferodd(Pin, [Blk_1|[Blk_2|Tail]], [Ublk|Rest]):-
            left(Pin, Blk_1, Blk_2, Ublk),
            transfer([Blk_2|Tail], Rest).

exchange([_,_,P1,_], [_,T2,_,_], [R3,_,_,S3], [R3,T2,P1,S3]).

process([Rin, Temp, [Pin,Xin,Lin], Sin],
                [Rout, Temp2, [Pout,Xout,Lout], Sout]):-
        comp(Pin, Sin, Cout, Pout, Sout),
        acc(Rin, Temp, Xin, Lin, Cout, Rout, Temp2, Xout, Lout).
```

Figure 5.5: Prolog Pattern Matcher Implementation

clauses, the first four of which divide the cyclic behaviour of the pattern matcher into four stages. These can be specified as:

1. Odd numbered blocks in the pipeline are active.

2. Odd numbered blocks output data to their left and right neighbours. Also, the left most block outputs a result bit and the right most block receives the character at the head of the string input stream.

3. Even numbered blocks in the pipeline are active.

4. Even numbered blocks pass data to their left and right neighbours. Also, the left most module receives the head of the pattern stream.

The final clause is recursive and ensures the repeated operation of the above four stages.

Five variables are used in the body of the predicate: Blocks_1a, Blocks_1b, Blocks_2a, Blocks_2b, and Rout. The first four are instantiated in the various clauses to lists of parameter lists with exactly the same format as Mods. Rout is initiated to an integer which is either 0 or 1. The matchodd clause implements stage 1. It instantiates Blocks_1a, such that it defines each block's parameters after the first stage. Blocks_1a together with the head of the string list are passed to transfereven which implements stage 2. In doing so, Blocks_1b and Rout are instantiated. The former so that it describes the state of each block's parameters after stage 2, and the latter to the result out of the left most module. Blocks_1b is used by matcheven to

instantiate Blocks_2a. This clause implements stage 3 and when it succeeds Blocks_2a describes all the blocks' parameters after this stage. The transfereven clause then uses Blocks_2a and the head of the pattern list to instantiate Blocks_2b. This clause is responsible for stage four. When it succeeds Blocks_2b is set up such that it describes the parameters after the final stage in the cycle. Blocks_2b together with the tails of the pattern and string lists and the list produced by placing Rout at the head of Res are fed back to pipeline.

The predicate matchodd(State_a,State_b) implements stage 1 above. It succeeds when State_b is a list of parameter lists. The even numbered elements in State_b are identical to the corresponding elements in State_a. The odd numbered elements, however, are the corresponding elements in State_a after the comparator and accumulator algorithms have been applied. Predicate matcheven(State_a,State_b) is similar. Both predicates involve the goal process(Inlist,Outlist), which correspond to the activity of an individual block in the pipeline. This goal succeeds when Outlist is a parameter list generated from the parameter list Inlist such that the first two elements are updated version of the corresponding atoms in Inlist. The remaining elements are copies of the corresponding elements in Inlist.

The argument Inlist defines the inputs to a particular character block and has the format [Rin,Temp,[Pin,Xin,Lin],Sin]. The variable Rin denotes the result bit in, Temp the current accumulated result associated with a block, Pin the pattern character in, Xin the wild card bit in, Lin the end bit in, and Sin the string character in. Outlist has exactly the same format but defines the outputs a block generates from these inputs.

So the predicate process characterises a block's transformation of inputs into outputs. It consists of two clauses, comp and acc, which correspond to the function of a comparator cell and an accumulator cell respectively. The variable Cout is used to represent the connectivity between the two.

The predicates transfereven(Schar,Resout,Blocks_1,Blocks_2) and transferodd(Pchar,Blocks_1,Blocks_2) implement stage 2 and stage 4 respectively. Blocks_1 describes the pattern blocks' parameters before the stages; while Blocks_2 describes them after. The variable Resout is instantiated to the output of the pattern matcher after stage 2. Pchar and Schar are the required inputs for these stages. Both predicates are intended to represent the movement of data between the pattern blocks and the input/output behaviour of the chip. They build up Blocks_2 such that it represents the situation where the outputs of the most recently active blocks have become the inputs to those blocks which are active on the next beat. Each predicate is recursive and involves the clause exchange.

The goal exchange(Left,Centre,Right,Inpts) is used to instantiate the variables in the single parameter list Inpts. This is achieved through matching of instantiated variables in the other three arguments, which are also single parameter lists. It uses the anonymous variable for the elements which do not play a part in this matching. The predicate is used to update a block's inputs. It demonstrates how the new values are obtained from the appropriate outputs of a block's left and right neighbours.

Prolog enables a very modular description of the pattern matcher to be written. It accurately describes the structure and behaviour of a single character block. The predicate process is used to represent such a block. Its input and output ports are represented by named variables. Recursion was used to represent the replication of pattern blocks and to simulate their concurrent behaviour. The transformation of inputs to outputs is modelled by unification through named variables. A character block is divided into two component modules (or cells). These are are represented by the comp and acc clauses which make up the body of the process predicate. The internal connection between these two elements is represented by the variable Cout. This variable does not appear in the head of the process clause, and therefore stresses that it is a local, internal connection.

The fact exchange also uses unification through named variables to represent the connections between a block and its left and right neighbours.

The lack of global variables means that the accumulated result associated with a block has to be represented by an input parameter. Similarly, its updated value is represented as an output parameter of the block. Above it was said that the named variables in the head of the 'process' clause could be used to represent the inputs and outputs of a pattern block. The fact that the accumulated result of a block and its updated value are represented by named variables would therefore imply that a pattern block has six inputs and six outputs. This is not the situation previously described for a block.

Although the 'process' predicate was intended to represent structure and connectivity not all the predicates of the program represent structural components. For example, the predicates implementing the movement of data and those determining the active blocks in the pipeline represent behaviour and not structure. There is no means of distinguishing which predicates represent structure and which represent behaviour.

The relational approach on which Prolog is based together with the associated matching of structures allows very flexible predicates to be defined. For example, consider a predicate with n arguments. Given (n-1) instantiated arguments, the predicate will instantiate the remaining argument irrespective of which argument it is. This feature is sometimes called "reversible programming", and abstracts away from the notion of formal inputs and outputs to a procedure. Although this abstraction is advantageous in software design, it makes describing a VLSI implementation slightly confusing. This is because interpretation of which arguments are inputs and which are outputs is less than obvious. The order of arguments does not necessarily imply anything. Comments would be required to help a reader distinguish between the two.

5.6 CONCLUSIONS

Five example programming languages representing five different computational models have been used to describe a single example of a VLSI implementation. It cannot be said that the selected example is representative of all the techniques employed in VLSI design, nor can it be said that the full strength of each language has been illustrated.

Instead the pattern matcher chip can be regarded as an example which captures some of the important features of VLSI. The descriptions presented in this chapter can be viewed as introducing the 'flavour' of the example languages. With this in mind the languages can be assessed for their ability to handle some of the important features in VLSI implementations. This will give an insight into which languages are well suited for a VLSI design description role.

All the languages were able to support a description of the pattern matcher. Pascal illustrated the advantages of abstraction through powerful data structuring techniques. The language can be used to represent structure, but showed that the sequential control flow model gives rise to a design which utilise a Von Neumann serial processing architecture. The sequential control flow model was, therefore, not well suited to this particular design. SmallTalk demonstrated that the class construct in the Actor model is useful for defining the data and processing characteristics of computational elements. However, the sequential nature of the language meant that the operation of such elements in parallel could not be represented.

Lispkit Lisp showed that the functional model is well suited for representing wiring, connectivity, and parallelism. It is capable of accurately describing combinatorial circuits. However, the lack of the assignment statement makes state difficult to model. Also, the lack of explicit control structures makes it difficult to describe sequential circuits. The major difficulty found with the language was representing multiple outputs as lists. For the logic model, Prolog illustrated how unification with named variables can be used to represent connections

between circuit elements. Parallelism and replication can be implied by recursion. The reversible nature of predicates suggests that the relational approach in this model is too abstract for many basic devices.

Based on the parallel control flow model, Occam enabled the system to be described as a collection of concurrent processes, which communicated through named channels. This description closely reflected the actual structure and behaviour of the VLSI implementation of the pattern matcher.

For this particular design Occam provided the best description. The language is well suited for describing concurrent elements which communicate with each other. The language is not 'ideal', the following are considered to be shortcomings of the language:

1. It lacks abstract data structures and types.

2. It cannot describe processes which do not have state. That is, all processes require state.

3. It currently does not handle recursion.

However, based on its performance for describing the pattern matcher, it will be selected as suitable language on which to base a future ATL system.

CHAPTER 6

THE ROLE OF OCCAM IN VLSI DESIGN

VLSI design involves the consideration of the two dimensional silicon area and the operation of the system in the time dimension. Both are a source of complexity in the design of large systems. In the previous chapter Occam was identified as a suitable programming language for describing some of the important features of VLSI systems. An Occam description of a system can be used to tackle the dual nature of design complexity by providing a consistent, abstract representation for both the spatial and time dimensions. The organisation of the program (i.e. its structure) can be used to define a topology for the system, while its control sequence (i.e. its operation) can be used to specify the timing behaviour of the system.

This chapter considers the advantages of using Occam for behavioural specifications of VLSI systems. The features of an ATL system based on the language are specified. A key issues to be considered for the future implementation of this system is: the mapping of the algorithmic execution sequence in an Occam description into the physical timing behaviour of a corresponding VLSI system. The reasons for adopting an

asynchronous, self timed approach over the synchronous, clocked approach are given. Also considered in this chapter is the generation of schematic diagrams from Occam programs. Such diagrams help clarify the nature of a VLSI design and aid the comprehensibility of Occam descriptions. Their generation from Occam programs demonstrates the ability of the language to represent both topological information and timing behaviour.

6.1 USING OCCAM AS A BEHAVIOURAL SPECIFICATION LANGUAGE FOR VLSI SYSTEMS

Occam is a language particularly well suited to describing a system consisting of many interacting components (processes) which operate concurrently and communicate through channels [45]. It is primarily intended for describing and programming transputer systems. As shown in the previous chapter it can also be used to provide a behavioural specification language for VLSI designs. Occam actually is a hardware design language, in the sense that it enables systems whose basic components are transputers to be described [46]. Its simple model of processes and communication corresponds exactly to the behaviour of real electronic systems [62]. That is, there is a direct mapping of the Occam representation onto a physical architecture of functional blocks connected via wires.

As a behavioural specification language for VLSI systems Occam is capable of supporting a variety of design styles. Design descriptions can range from being purely sequential to fully concurrent. One way in which to use the language is to decompose a problem hierarchically, using only the parallel constructor and channels until the individual processes

are as simple as possible. This approach can potentially result in descriptions which maximise concurrency. Another approach for exploiting concurrency is to use the sequential constructor to combine parallel constructs. This represents the action of a number of processors which all synchronise after each step they perform. In general, the parallel constructor can be used to allow communication and computation to proceed together.

The advantage of using an Occam program as a specification is that the program can be executed as a simulator, or prototype, of the system [61]. The language has a very efficient implementation, which enables fast execution of a system description. Compilation of a source program into a machine executable implementation enables the algorithm characterising a system to be investigated. The designer can check the efficiency of the Occam description and validate its correctness. It can be 'fine tuned' to derive an optimum representation of the algorithm. Also, alternative solutions can be evaluated quickly. Once the designer has demonstrated that the program provides a satisfactory representation of a system's behaviour, the program can be used as the specification for a special purpose chip (or application specific integrated circuit). Such an approach would involve transforming the Occam program into an intermediate, detailed (in terms of structural components) design description.

Occam has a number of other features which make it an attractive language for hardware design. These include:

1. It contains the necessary concepts of concurrency and communication to enable it to be used for the design of digital hardware at any level of detail.

2. The language employs explicit control of concurrency and communication.

3. It is a system description language which brings methodology to system design. Occam supports the use of parameterised processes which enables the hierarchical decomposition of problems. That is, the language supports a structured design style.

4. It has a formal basis which opens up the potential of formal reasoning and transformation as design techniques.

5. It is easy to understand since it uses the minimum of concepts (three primitive processes). It is concise. The syntax is regular.

6.2 ATLAST: AN EXAMPLE ATL SYSTEM

Occam's ability to describe concurrent systems in a manner suggesting an architectural implementation makes it an attractive notation on which to base an ATL system. In principle, an Occam program describing the behaviour of a system could be mapped into an equivalent hardware implementation. In order to obtain such an implementation the program's control sequence must be mapped into an appropriate timing model. Also, topological information must be extracted. This requires

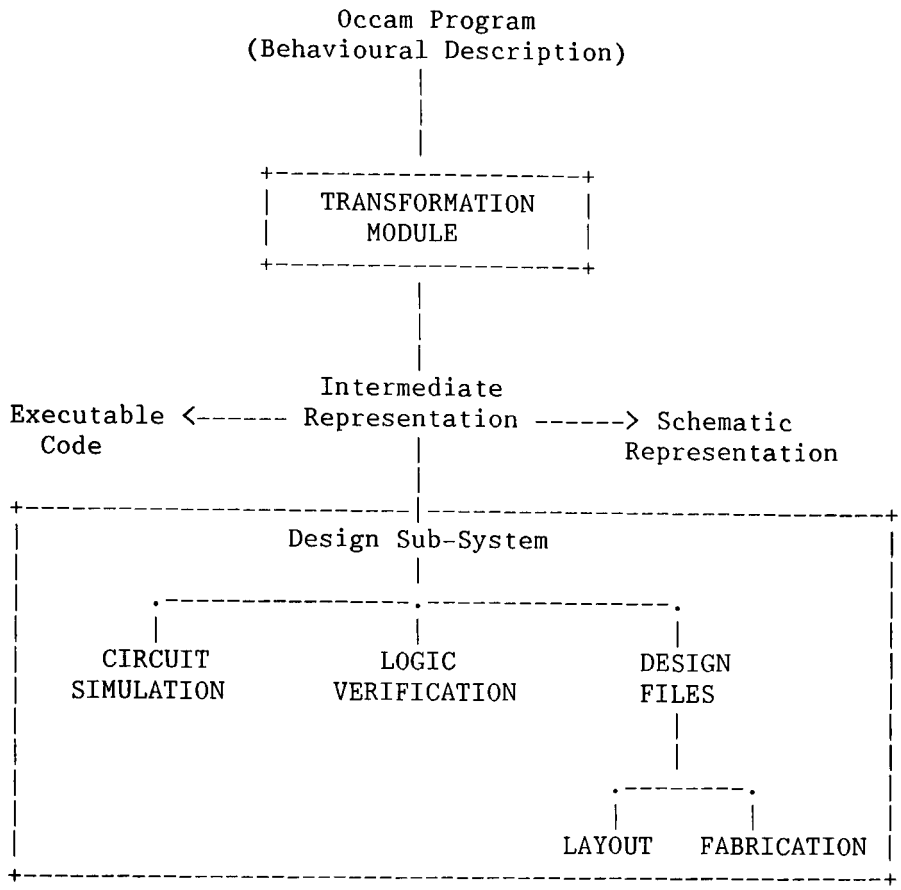


FIGURE 6.1: DESIGN PHILOSOPHY BEHIND ATLAST

an efficient set of transformations which, when applied to the description, map Occam components into appropriate VLSI circuit elements in accordance with an appropriate timing model. The proposed philosophy on which an example ATL system could be founded is illustrated in figure 6.1. This figure represents the specification for a future system called ATLAST (Algorithm To Layout ASsisTant). It is envisaged that this type of design system will provide a 'software type' design environment to support the translation of algorithmic descriptions into special purpose VLSI chips.

The central feature of ATLAST will be a powerful dual purpose transformation module supporting both software and hardware design. This module will check the syntax and semantics of source programs and generate executable machine code. That is, a standard Occam software compiler will be an intrinsic part of it. Also, the module will derive detailed information for specifying VLSI implementations. This information will provide an interface to a design sub-system capable of producing an actual implementation. It is intended that the approach taken for implementing the sections of the transformation module responsible for forming the VLSI interface will differ form that currently used for silicon compilers. Rather than deriving detailed physical information for an adopted target architecture, an Intermediate Representation (IR) will be generated. This IR will be a structurally optimised Occam description. In effect, Occam's concept of concurrency and communication will be adopted as a target model. The design sub-system will be responsible for deriving an appropriate hardware implementation from this description.

6.2.1 Timing Considerations

Complexity in the time dimension stems from the need to determine the discrete intervals in time at which signal events may occur. A signal passes through a sequence of time intervals when it is correct, then incorrect, then correct again. To interpret the signal voltage, the times at which the signal represents valid logic must be determined. In a complex VLSI system there are many different signals which need to be considered. The physical concept of a time metric is at too low a level for describing the times when these signals are valid and the

relationships between them. A structured design methodology, such as the one described in Chapter 2, is more applicable. So instead of using a low level means for describing time relationships, a much more powerful, abstract level can be used - the concept of sequence.

An Occam description of an algorithm contains an explicit definition of execution sequence. The PAR construct and named channels are used to describe concurrent components which at certain points in time communicate. Channels enable the components to synchronise their execution, exchange the relevant data, and then continue executing independently until the next communication. Sequential ordering of events is achieved through the SEQ construct. Usually, all three execution constructs are used together to describe a system composed of communicating sequential processes. The inherent sequence in an Occam description can be regarded as the abstract specification for the timing behaviour intended for a corresponding VLSI implementation. This specification must be mapped into physical timing in such away that performance objectives are achieved. This involves connecting abstract sequence to an appropriate timing discipline. These disciplines can be divided into two categories [47]: the synchronous approach and the self-timed approach. In the former sequence and time are connected by means of a global clock signal. In the latter sequence and time are connected in the interior of parts called elements.

The synchronous approach is the most widely used. A clock with two or more non overlapping phases is normally used to remove constraints on the minimum delay in a clocked system. The clock signals serve two purposes. Firstly, they act as a sequence reference, the transitions

serve the logical purpose of defining successive instants at which system state changes may occur. Secondly, they provide a physical time metric. The period or interval between signal transitions is used to handle the elements and wiring delays in the paths from the output to input of clocked elements. This dual role of the clock binds the system sequencing and timing so closely that "timing" is the source of numerous difficulties in the design, maintenance, modification, and reliability of synchronous systems.

The problems of managing complex designs in which all system parts must operate together has led to an increased interest in the alternative self-timed approach. Self-timed systems are interconnections of parts, which are called elements. Time and sequence are related inside elements, so that events such as signal transitions at the terminals of an element may occur only in certain orders. Initiation of a given computational step depends on completion signals produced by its sequential predecessor. The approach involves the design of elements and the interconnection of these elements in a system. In the design of elements logic, physics, and timing are brought together. This is made easier since the designer works within a domain small enough to make the design manageable. System design involves tackling the problem of synchronising communication of data from one element to another. One of the major difficulties in the self-timed approach is ensuring that this communication is achieved reliably without an enormous area overhead in additional logic.

Future scaling down of feature size and scaling up chip area will not only increase the complexity of VLSI chips, but will also change relationships in parameters. These parameters describe the physical (i.e. electrical) characteristics of switching devices, circuits, and wires. The increased wire delay associated with the increased resistivity of scaled down wires will have a dramatic impact [44]. It will result in propagation delays within a chip causing significant wiring delays among functional blocks. Unless a technology based solution is found, clock skew will become such a problem that synchronous behaviour through the use of a system wide clock may no longer be achievable. The lack of alternatives to the aluminium/doped polysilicon interconnection systems is convincing some designers [55,77] that the solution lies with self-timed approach. Also, this approach is more in keeping with a rigorous discipline of modularity [5]. For these reasons the self-timed approach will be adopted for system implementations by ATLAST. The execution sequence of an Occam source program will be used to divide a system into modular parts that are self-timed elements. These elements will be implemented as as synchronous systems with an internal clock.

6.3 SCHEMATIC REPRESENTATION OF OCCAM DESCRIPTIONS

A complete implementation of the ATLAST system is beyond the scope of this thesis. Instead, the aim of the work to date has been to demonstrate the automatic generation of schematic diagrams corresponding to the IR. Such diagrams are desirable for a number of reasons. Firstly, they help clarify the nature of the intermediate Occam description. Although textual descriptions are powerful and concise

their overall meaning can be difficult to comprehend. A schematic diagram corresponding to a textual description provides a valuable aid to documenting and understanding that description. Secondly, schematic diagrams reflect the two dimensional nature of VLSI design. They are traditionally used to provide abstract topological information form which an actual layout can be derived. That is, schematic diagrams are used to illustrate the relative placement and interconnection of high level hardware primitives (functional blocks). Thirdly, schematic diagrams can be used to provide a visual representation of the data flow and control flow in a system.

Since the IR will itself be an Occam description the approach adopted concentrated on mapping key features of the language into suitable schematic representations. The overall objective was to demonstrate how both topological and timing information can easily be obtained from an Occam description. As mentioned above, it is envisaged that systems implemented by ATLAST will be based on the self-timed approach. So it was decided that the schematic representations should reflect this approach.

6.3.1 The Occam Subset

Rather than considering the complete Occam language, schematic generation for programs written in a subset has been demonstrated. Only a subset of Occam was selected so as to simplify the problem of generating schematics to diagrams which occupy a single sheet. This simplification only allows very simple descriptions to be considered. The syntax for the selected subset of Occam is defined in Appendix C.

This subset is powerful enough to enable the fundamental and most important feature of Occam to be described, namely communicating concurrent processes. It contains three types of primitives: operators, channels and variables. The set of operators is comprised of the primitive process operators and the arithmetic operators. There are three primitive process operators: input "?", output "!", and assignment ":=". The arithmetic operators are: "+", "-", "*", and "/". Channels and variables are the fundamental objects manipulated by the subset and both are denoted by identifier strings.

The operator, channel, and variable primitives represent the fundamental components from which a wide variety of programs may ultimately be composed. They are the building blocks from which processes are formed. They are combined, according to the subset's syntax rules, to form primitive (or simple) processes. Simple processes can be combined, by means of a constructor, to form a complex process. A construct governs the order of execution of its component processes. In the subset of Occam handled by COPTS there are two constructs: sequential (SEQ) and parallel (PAR). A complex sequential process is constructed by SEQ and parallel process is constructed by PAR.

In the approach adopted for generating schematic representation of Occam programs the three types of subset primitives are mapped directly into schematic equivalents. Operators are mapped into elements called schematic cells (see section 6.4). Variables are mapped into elements called registers, which have input and output data paths (see section 6.4.1). Channels are mapped into communication paths (see section 6.4.2). These schematic elements serve two purposes. Firstly, each

gives a high level structural representation of a hardware element which will implement the corresponding software primitive. Secondly, each depicts the function of its primitive. Just as operators, channels and variables are the building blocks of software descriptions so cells, registers, data paths and communication paths are the building blocks from which schematic diagrams are constructed. In a schematic diagram, each element corresponds to a software primitive identified in a corresponding Occam program.

6.3.2 COPTS: A Schematic Compiler

A schematic compiler called COPTS (Compilation of Occam Programs To Schematics) has been written to demonstrate the generation of high level schematics. From an Occam program written in the subset, COPTS can generate a schematic diagram depicting its realisation in silicon. The diagram provides a visual representation of parallelism present in the described system. It also illustrate the flow of the control and data through the system and the communication paths between its components. Ideally such diagrams could be used as structural specifications for VLSI implementations. The arrangement of schematic elements in a diagram could be used to specify the actual placement of their associated hardware structure. Substituting detailed physical representations for the elements, in accordance with each one's position, would generate an implementation. For this approach to be practical future advances in the fabrication technology will have to remove the silicon area constraint, since a schematic diagram is very wasteful of space. Also the problem of power consumption must be tackled. A more realistic approach to obtaining implementations is suggested in the final chapter.

In translating an Occam design description into a schematic implementation, every primitive occurring in the program is identified. Its corresponding schematic element is generated and placed accordingly. The set of elements generated for an arithmetic expression visually represent the function of that expression. Their arrangement and interconnection reflects the data flow and execution sequence in the expression. The set of elements derived for simple and complex processes illustrates the various functional steps involved and the order in which they occur.

Source code for an Occam compiler was unavailable, so a parser for the subset had to be written. This builds up an internal data tree to represent the block structure and execution sequence of source programs. The conceptual structure of this tree is then used to define the placement and interconnection of the schematic elements. The assembly of these elements into a network portraying a source program is based on a simple mapping strategy. This strategy can be summarised as:

1. parallel behaviour is used to define vertical placement;

2. sequential behaviour is used to define horizontal placement;

3. data flow is, in general, left to right;

4. only vertical and horizontal routing is allowed, crossovers are permissible;

5. compaction is not a primary concern.

The remainder of this chapter defines the schematic elements and summarises the function associated with each. A more detailed consideration of the strategy used for spatially arranging these elements and the implementation of the graphics compiler is given in the next chapter.

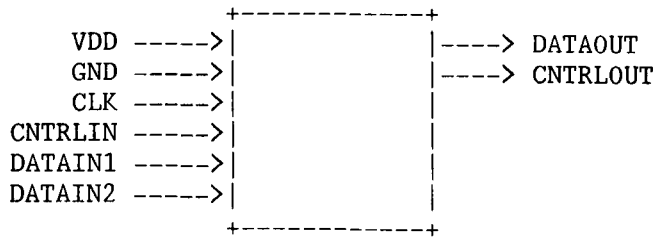
6.4 SCHEMATIC OPERATOR CELLS

All the primitive operators in the subset are mapped into schematic elements called operator cells. The behaviour of a system, as specified in an Occam source program, is represented by an interconnected set of these cells. COPTS identifies the required members of this set. It then determines their topological arrangement and interconnection in a manner reflecting a self-timed implementation of the program's execution sequence. Each operator cell depicts a self-timed element (or macro-cell), which is implemented as a synchronous system with an internal clock. From inside an element this clock appears independent of the clocks in other elements of the implementation. However, the operation of each element's clock is governed by signals taken from a global clock. So, at the system level, element clocks are partly dependent. In an actual implementation of the wire carrying the global signals, a low signal rate would be necessary to avoid clock skew. This would not be a problem, since an element would generate faster, internal clocking signals from the these signals. The internal clock rate of an element would not be dependent on that of the other elements.

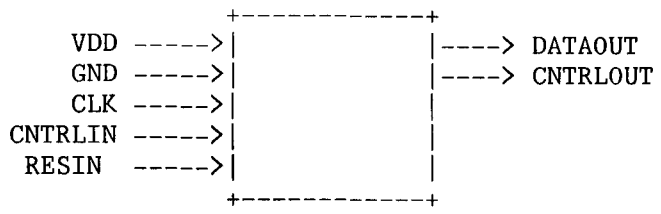
In an implementation, initiation of a macro-cell would depend on a completion signal produced by its sequential predecessor. In effect, such a signal is a control token and implies a physical link between the two - a control line. The control token informs a cell that its inputs are available and instructs it to commence processing them. On arrival the token is consumed by a cell. When that cell has completed its function and formed its output it generates a new token. This token is then sent to the next element in the execution sequence. Once enabled, a cell cannot accept another token until it has generated the completion signal for the current task.

Since the function of a system is implemented as the collective behaviour of the component macro-cells, every cell must, at some stage, receive a token. For correct functioning, the interconnection of cells, by a control line, must be such that each receives a control token in a defined sequence. That is, a constraint on the topological arrangement of cells is needed to ensure correct operation of a system. This topological constraint is reflected in the schematic diagrams generated by COPTS. Macro-cells which would operate in parallel in an implementation are represented by schematic operator cells placed in a vertical array. Macro-cells which would operate one after another (i.e. sequentially) are depicted by schematic cells placed in horizontal arrays.

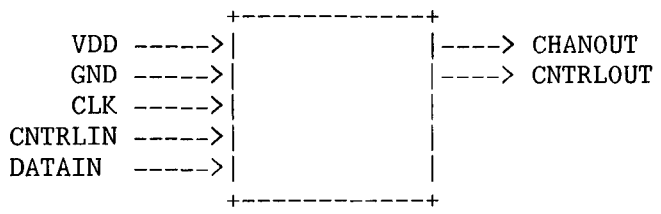
The representation for each of the macro-cells is illustrated in figure 6.2(a)-(d). It is envisaged that a textual description of an actual hardware element will be associated with each of the schematic cells. This description could be a composition routine in some high



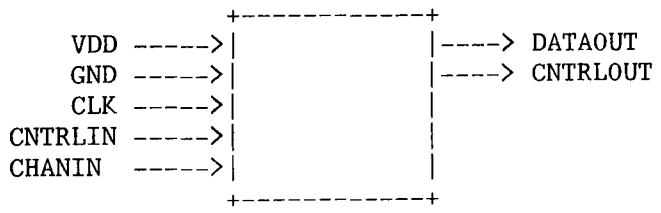
(a) An Arithmetic Operator Cell.



(b) An Assignment Operator Cell.



(c) An Output Operator Cell.



(d) An Input Operator Cell.

FIGURE 6.2: THE REPRESENTATION FOR PRIMITIVE CELLS.

level notation, for example a PLAP description. From it, the complete hardware implementation (wires, boxes, polygons, etc.) of the

corresponding primitive could then be generated automatically. Such descriptions could also incorporate information required by a simulator. At present each schematic depicts a primitive's realisation in silicon. They can be regarded as being at a very high level of structural abstraction with much of the implementation detail hidden, cf. abstraction by means of a bounding box. The boxes hide details of the physical layout of the components implementing the function of an operator. Only the required inputs and the generated outputs are shown. The combination of operator cells in a schematic diagram forms an overall structural specification for the intended VLSI system.

There are four inputs common to every operator cell. These are labelled VDD, GND, CLK, and CNTRLIN, and denote the power, ground, clock and control inputs to each macro-cell. The first three inputs are derived from system inputs, whereas the fourth is derived from an operator cell's sequential predecessor. A schematic cell's power input denotes the metal line which would supply the operating voltage to the corresponding macro-cell; while the ground input denotes the metal line for sinking a voltage. The clock input represents the line from which a macro-cell would take signals to generate its internal clocking signals. The control input depicts the metal line on which a macro-cell's initiation signal (or control token) would arrive. As well as the four inputs common to all schematic cells, there is also one output common to them all, labelled CNTRLOUT. This denotes the wire along which a macro-cell would signal that it had completed its execution.

Arithmetic operator cells represent hardware blocks which take two numeric inputs and compute a result according to the function of the operator they are implementing. Their behaviour can be summarised as:

Result <-- Operand1 Operator Operand2

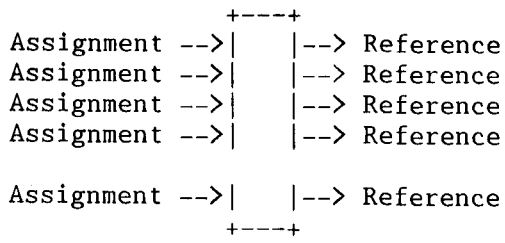
In the schematic representation, figure 6.2(a), DATAIN1 and DATAIN2 correspond to Operand1 and Operand2 respectively. Each line denotes the data path for its binary operand. The actual implementation of such a path is determined by the type of data transfer required e.g. bit serial or bit parallel. For bit serial transfer a data line is implemented as a single metal wire; while for fast parallel transfer a data line is decomposed into 'n' (where n is the number of bits per word in the implementation) metal wires. After computing the result its value is placed on DATAOUT - the output data path.

An assignment operator cell, figure 6.2(b), depicts the transfer of the value of an associated expression to a register (see Variables and Data Lines). Such a cell is needed because the data path carrying the result of the expression may not necessarily be compatible with the input data path of the register. It represents either a sequential in parallel out (sipo) block, or a parallel in serial out (piso) block. In the case of the former, RESIN denotes a single metal wire, DATAOUT 'n' metal wires and the cell represents an element for converting from bit-serial transfer of a word to fast parallel transfer. For a piso block RESIN denotes 'n' metal wires, DATAOUT a single metal wire, and the cell represents an element for converting from parallel transfer to bit-serial transfer. Should the two data paths be compatible then in an implementation neither block would be required.

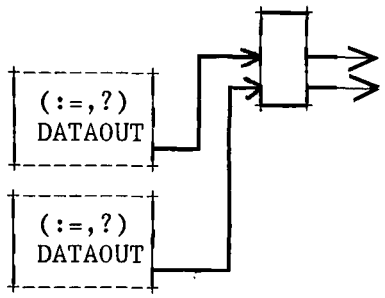
6.4.1 Variables And Data Lines

A variable identifier is used to hold a value within the scope of a complex sequential process. Such a process may use several variables. Variables must be initialised, and may well be re-initialised with different values in the process. All of a process' variables which are initialised in the same subsidiary are mapped into a schematic element called a register. This register is placed immediately to the right of the set of schematic operator cells corresponding to that software subsidiary. In an implementation, a register would be comprised of one or more storage locations, the actual number being defined by the variables it represents. Each location would be capable of holding an n-bit word and would be associated with a particular variable. In the schematic representation of a register, see figure 6.3(a), there is an input and an output line for each word. These are referred to as the assignment line and the reference line respectively. New values to be stored in the word for a particular variable arrives on the former, while a copy of the contents of that word are sent on the latter. So, the assignment line represents a write line and the reference line a read line. Data flow on both lines is left to right.

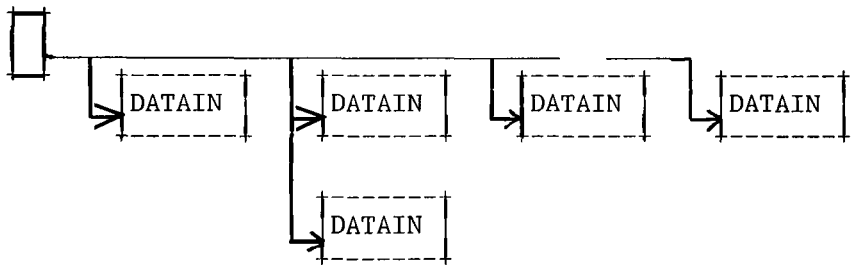
A variable is initialised by either an assignment or an input primitive process. This is represented schematically as a data line linking the DATAOUT output of either an input or an assignment operator cell to an assignment line of a register. Data flow on this line is always left to right. To help illustrate the mapping strategy, an outline of the schematic representation for the initialisation of two variables is given in figure 6.3(b).



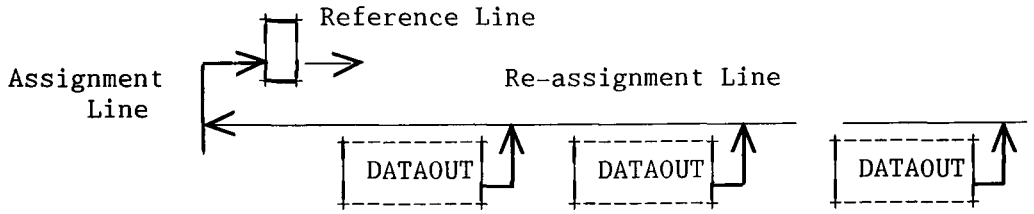
(a) Schematic Representation Of A Register Cell



(b) Initiation Of A Register



(c) The Reference Line



(d) Register Update

FIGURE 6.3: The mapping of variables

The two boxes labelled with ":=,?" represent the assignment or input

operator of the primitive process responsible for placing initial values in the variables. Vertical placement of the two cells means that in the source program these two processes were defined to operate in parallel (see section 7.2). For clarity only the DATAOUT outputs of the cells are illustrated. In keeping with a left to right representation of data flow, the register for the two variables is placed to the right of the cells. The register has two sets of assignment-reference lines, each one is always associated with the same variable. The connecting data lines represents the communication path between the two cells and the register. It also implies a communication protocol for handling the transfer of data from the operator cell to the register cell. For bit-serial communication the line is implemented as a single wire, whereas for concurrent communication the line is expanded into n wires.

Having been initialised, a variable is typically referenced as an operand to a number of operators. Each of these operators could be in separate expressions. This is represented schematically by the reference line for that variable being routed to the appropriate DATAIN input of every operator cell in which the value is required. These cells will always be to the right of the register. As an example consider figure 6.3(c). This gives an outline of the schematics for an initialised variable which is as an operand of five arithmetic operators. The boxes labelled with DATAIN represent arithmetic operator cells. For clarity only the appropriate DATAIN input of the cells are shown. The position of the boxes is intended to merely illustrate a possible arrangement of cells and is not intended to define any particular placement strategy. The value being carried on the reference line is sent down each fork of the line. Each branch maintains the left to right convention for data

flow.

If a variable is assigned a number of different values in a process, then this is mapped into a subsidiary data path connected connected to the assignment line corresponding to the variable. This path is called the re-assignment line, and it denotes a bus for fetching updated values for the word associated with the variable. Data flow is reversed on this line (i.e. right to left as opposed to left to right). From it there are branches to the DATAOUT outputs of all operator cells (either assignment or input) responsible for updating the register corresponding to the variable. Again these cells will always to be to the right of the register. An outline of the schematics representing three different updates of a variable is given in figure 6.3(d). The cells labelled DATAOUT denote the operator cells updating the variable. Obviously updates occur sequentially hence the left to right horizontal arrangement of cell (see section 7.2)

6.4.2 Communication Primitives

Channels are used to communicate between concurrent processes. Each channel provides a one way connection between two concurrent processes. A channel identifier is associated with an input component and an output component of a parallel construct. The two components are said to be connected by the channel. Only one input and one output process can be connected by a channel. Output and input operators are represented by the cells shown in figure 6.2(c) and figure 6.2(d) respectively, while a channel identifier is mapped into a line called a communication path. A communication path links the CHANIN input of an input operator cell to

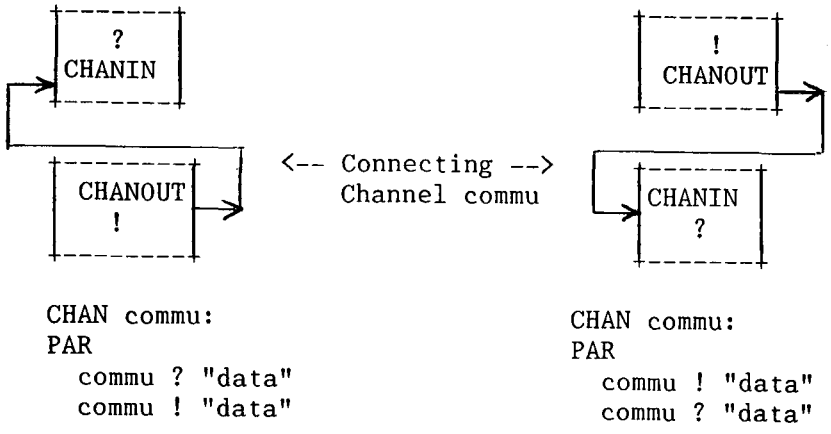


FIGURE 6.4: Schematic Representation Of A Channel Identifier.

the CHANOUT of an output operator. Data flow in such a path is always from the output cell to the input cell.

An outline of the schematics for an input cell linked to an output cell by a communication path is given in figure 6.4. The boxes labelled "?" denote input cells, the boxes labelled "!" denote an output cell, and the communication path is labelled "commu". For simplicity only the communication path is shown. Since, the cells are associated with primitive processes which ultimately belong to the same parallel process they are placed vertically (parallel behaviour defines vertical placement, see section 7.2). There are two possible arrangements for the operator cells, both are outlined in figure 6.4 together with the segments of code from which they were mapped. The term "data" is used to denote the value of a word and not a variable. If the primitive input process is the first to be defined in the parallel process then the input cell is the upper most. If, however, the output process is the first to be defined then the output operator cell is upper most. The connecting

communication line, commu, is routed between the two cells. This line defines the routing of the communication path between the two cells. It also depicts the implementation of the necessary synchronisation protocol (e.g. handshaking) between the two.

If the input and output processes occur on consecutive lines of the source program, as in figure 6.4, then COPTS places the schematic representation for the corresponding input cell vertically adjacent to that for the output cell. However, if the two processes are separated by intermediate processes, then the cells are separated vertically by other cells. The operator cell associated with the first process to reference a channel identifier is termed the channel's source cell. The operator cell for the corresponding process is termed the channels destination cell. Channels declared for the main process may represent external inputs or outputs. In which case, the corresponding channel lines are only connected to source cells. If this source cell represents an input process then the line is routed to the west edge of the diagram. On the other hand, if the source cell represents an output process then the line is routed to the east edge of the diagram.

An output operator cell denotes a hardware element for transferring (cf. a driver) the value of an expression on to a communication channel. An input operator cell depicts a hardware element for transferring a value from a communication path (cf. a stabilising input buffer) to a register. Together these hardware elements would provide communication between concurrent components in a system. The communication path between the two schematic cells represents the physical connection (link) along which communication between the two hardware elements would occur.

This communication is synchronised. An input element will not complete execution until the corresponding output element has executed a write. Equally an output element will not complete execution until its corresponding input element has executed a read from the link. When both are ready to communicate, the value to be output is copied, via the physical link, to the input cell. After the communication has taken place both cells generate a control token signalling the termination of their execution. The concurrent component each one belongs to then continues to execute independently.

6.5 SUMMARY

This chapter has considered how Occam can be used to help clarify the nature of a VLSI design. Since Occam was designed to describe systems composed of a number of communicating processes operating in parallel, it can be used to provide behavioural specifications of VLSI systems. Its features make it relatively straightforward to generate corresponding schematic diagrams. Such diagrams help document an Occam VLSI description. A schematic compiler (COPTS) which can generate schematic diagrams for a limited sub-set of Occam has been developed. The primitive elements used to build up these diagrams have been introduced. The next chapter considers in more detail the approach adopted by COPTS for building up diagrams.

CHAPTER 7

GENERATING SCHEMATICS

The task of generating a schematic diagram for an Occam description is divided into mapping the main processes into abstract cells. Each of these is individually defined according to whether they correspond to a complex or primitive process. The resulting abstract cells are then interconnected to form a network representing the program. This chapter defines the various types of abstract cells and considers how they are built up from the primitive schematic elements. Brief details of the current implementation of COPTS are also given.

7.1 ABSTRACT CELLS

In deriving a schematic diagram, the compiler uses the top down nature (i.e. the hierarchy) in the source program to structure its representation. This involves viewing a program at a number of levels of abstraction, ranging from the entire program down to the actual primitives it is constructed from. A particular component of the Occam subset is associated with each level. The primitives described in section 6.3.1 characterise the lowest levels. Above these, occurrences

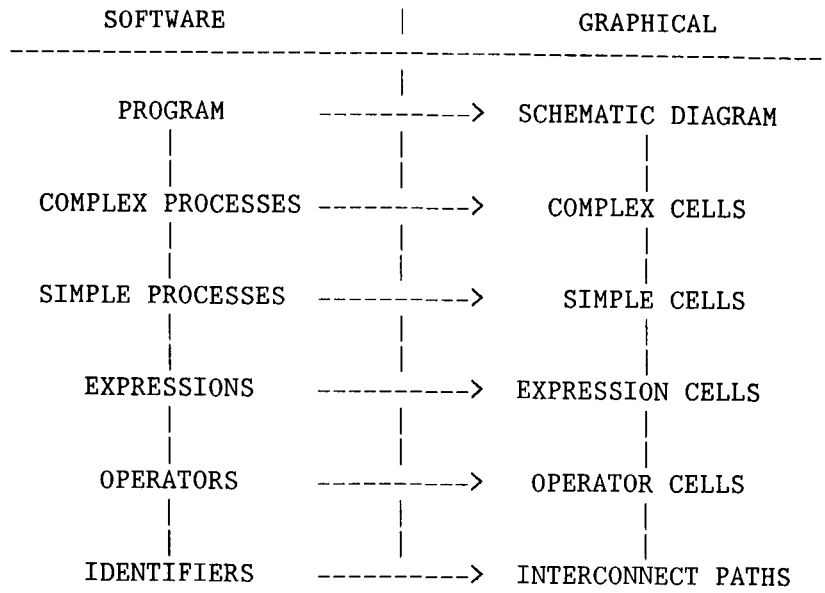


FIGURE 7.1: Schematic Mapping

of a level's component are used to partition the schematic implementation into collections, or blocks, of interconnected schematic operator cells. These blocks are regarded as abstract cells. The software components, their hierarchical arrangement and their mapping into schematic counterparts are shown in figure 7.1. In this figure, abstract cells are classified according to whether they represent a complex process, or a simple process, or an expression. A simple process is comprised only of primitive operators, whereas a complex process is comprised of simple process and/or other complex processes.

The arrangement and interconnection of schematic operator cells in an abstract cell depicts the function of a corresponding software component. It defines the placement and interconnection of hardware elements which would collectively implement that function. Defining the layout of operator cells for a complex abstract cell entails recursively

partitioning the cell into component abstract cells. This recursive decomposition is continued until abstract cells for simple processes can be considered. These are then treated separately and according to whether they will represent an assignment, output, or input process. Both assignment and output abstract cells may involve defining expression cells (see section 7.5). Abstract input cells are composed entirely of operator cells and lines defining their interconnection and communication paths. The defined cells are interconnected to build up the representation of the parent complex cell. This top down, bottom up approach produces schematic diagrams having a hierarchical structure corresponding exactly to that in an Occam source description.

Abstract cells are treated as bounding boxes with pins located around the periphery to identify the interconnect points for inputs and outputs. They enable the diagram to be partitioned into distinct areas which can be considered individually. This allows the diagram to be built up gradually. To derive abstract cells, the compiler first routes the appropriate lines to the input points. Then the placement and interconnection of the schematic elements is considered. Finally, the routing of the outputs is tackled. The definition of an abstract cell represents the 'glue' which binds together the schematic elements corresponding to each of the associated component's primitives. The compiler uses the bounding box to delimit the area occupied by the elements. It is a conceptual feature and does not appear in the overall schematic diagram generated from an Occam description. Only operator cells, their interconnection paths, power, ground, clock and control lines are drawn.

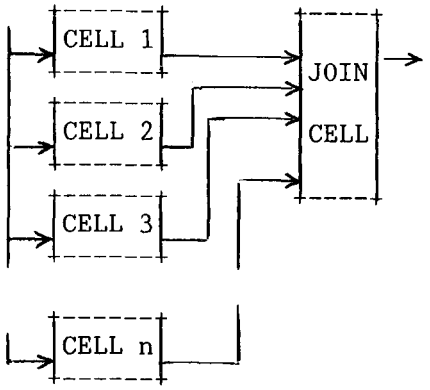
7.2 EXECUTION SEQUENCE AND LAYOUT

The execution sequence specified in source programs is used directly to define the spatial arrangement of cells. The interconnection of these cells by the control line is also determined by the execution sequence. Occam programs typically contain both sequential and parallel behaviour. The SEQ construct is used to explicitly define a process whose subsidiaries execute one after another. Concurrent operation of a process' subsidiaries is defined by the PAR construct. Both sequential and parallel execution are possible in the evaluation of an arithmetic expression (see section 7.3). This section first considers how sequential behaviour is used to define placement and routing and then how parallel behaviour is similarly used. To illustrate the approach adopted, consider an Occam component composed of 'n' subsidiaries (where $n > 1$). Assume that these subsidiaries can either operate in sequence (ie. execute and terminate one after the other), or concurrently (ie. all commence executing at the same time). Assume also that each subsidiary has a 'cell' representation, whether this is abstract or primitive is not important as what follows is applicable to both.

7.2.1 Sequential Behaviour

Sequential execution of the 'n' subsidiaries is mapped into a horizontal array of cells. An outline of the schematic representation resulting from this mode of operation is given in figure 7.2(a). To aid understanding only the control line and cell boundaries are illustrated. The control line is routed such that it first connects the control in point of CELL 1. It then connects the control out point of CELL 2 to the

(a) sequential execution of cells.



(b) concurrent execution of cells.

FIGURE 7.2: Execution Sequence And Interconnection Of Cells.

control in point of CELL 3,..., and the control out point of CELL (n-1) to the control in point of CELL n. That is, there is a single thread of control linking the cells from left to right. A control token arrives from the left and enables CELL 1. This cell executes, and when it has completed its task, places a new token on its control out line, and terminates. The token is carried to CELL 2, which is enabled etc. This sequence is repeated along the array of cells, each one in turn being enabled, executing, and then terminating.

7.2.2 Parallel Behaviour

In the schematic representation for n components operating concurrently, figure 7.2(b), the corresponding cells are arranged in a

vertical array. The control line is routed down the left side and ultimately connecting with control input of CELL n. Along the route to this connecting point are (n-1) "forks". At each fork a segment from the control line connects with the control input of the adjacent cell. So the intervening (n-1) cells are also connected directly to the control line. At each fork the initiate signal propagates along both segments. To the left of the vertical array of cells is a single cell with n inputs. This is a JOIN cell and the top input is connected to the control output of CELL 1, the second from top input is connected to the control output of CELL 2,..., and the lowest input to the control output of CELL n. The join cell has a single output. On receiving a completion signal from each of the n cells, the join cell simply places a completion signal on its output.

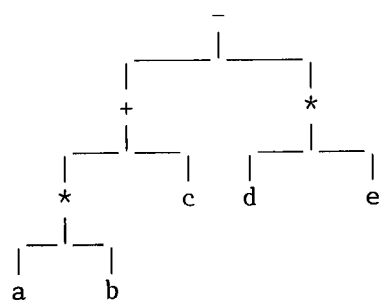
Concurrent execution is represented by the forks on the control line. Return to sequential execution is indicated by the join cell. When the control token arrives at the upper left corner it is carried simultaneously to each of the n cells, which are then enabled. The execution time of a cell is independent of that for the other cells. That is, each cell is time independent of all other cells in the array. Consequently, completion signals are placed on the CNTRLOUT outputs at different times. The join cell receives these, and when all n have arrived places a terminating signal on its output. This is then carried to the next cell in the execution sequence. From this cell, the vertical array of cells appears as a single cell with a single control line in and a single control line out.

7.3 EXPRESSION TREES AND THE LAYOUT OF EXPRESSION CELLS

Arithmetic expressions can be described by binary trees. The tree for a variable identifier in such a description is simply that variable. If e1 and e2 are elements with trees T1 and T2 then the trees for e1 + e2 and e1 e2 (ie. <element> <assoc.op> <element>) are:

The trees for e1 - e2 and e1 / e2 (ie. <element> <operator> <element>) are respectively:

As an example consider the Occam expression ((a*b)+c)-(d*e). The binary tree describing this expression is

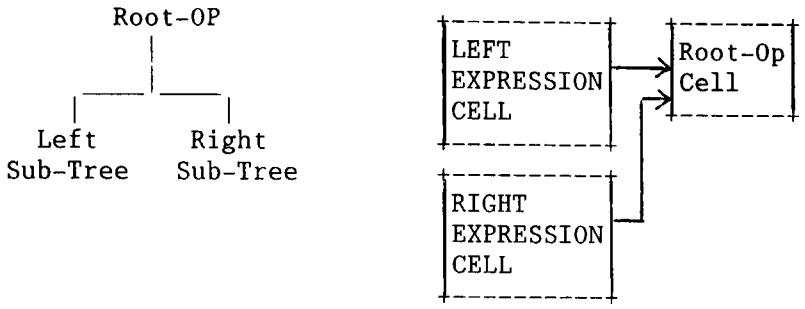


The operator of an expression forms the root of a tree, while each element is a variable identifier described by an end node or a sub-expression described by a sub-tree.

7.3.1 Defining Layout

A tree description is used to define the mapping of an expression into an abstract expression cell. Such cells are composed of primitive operator cells and their interconnection paths. The placement of these operator cells is defined by the implied execution sequence in the corresponding tree description. A tree consists of a left and right sub-tree, see figure 7.3(a), each of which is mapped into a sub-expression cell. When the sub-expressions described by the left and right sub-trees have been evaluated, their results form the operands of the root operator. This operator can then compute the overall value of the expression. The value obtained for the left sub-tree is independent of that obtained for the right tree. Both depend only on the values of the variables occurring in their sub-expression. Therefore, they can be evaluated concurrently. Since parallel behaviour is represented by vertical placement, the abstract cells defined for the right and left sub-trees are placed as shown in figure 7.3(b). The operator cell for 'root-Op' is placed to the right of the two expression cells.

The parallel behaviour means that the control line from the sequential predecessor has to be split into two segments. One segment is routed to the left expression cell and the other to the right expression cell. A join cell immediately after the two expression cells is required. Its output is connected to the CNTRLIN input of the root operator cell. Neither the join cell or the control line is shown in figure 7.3(b). This mapping into sub-expression cells and associated routing of the control line is identical for all left and right sub-trees. In figure 7.3(b) the root operator cell is connected to each



(a) Expression Tree (b) Mapping

FIGURE 7.3: Topological Arrangement Of Expression Cells.

expression cell via a data line. These lines carry the value of the associated sub-expression. This mapping strategy ensures that the binary nature of expressions is reflected in the schematic representation.

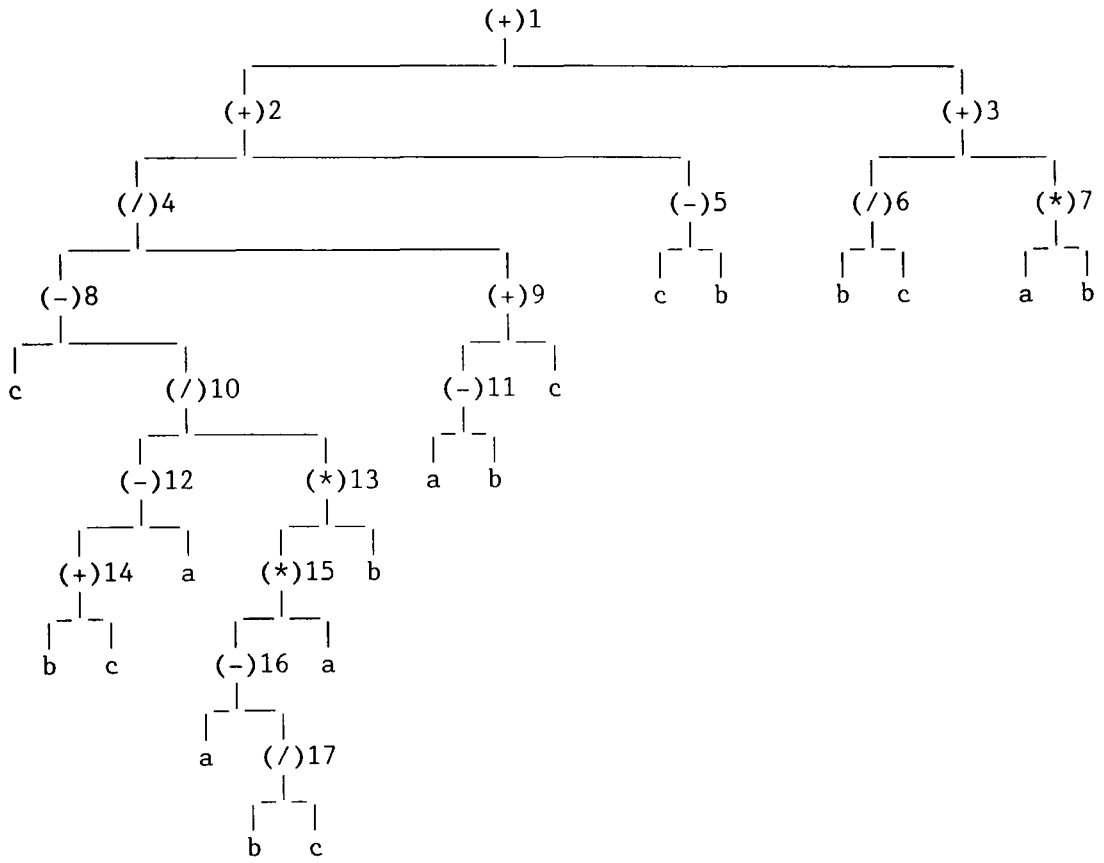
Data lines depicting the variables referenced in an expression are routed to the appropriate operator cells. Power, ground and clock lines are routed to each of the cells in the expression cell.

7.3.2 An Example

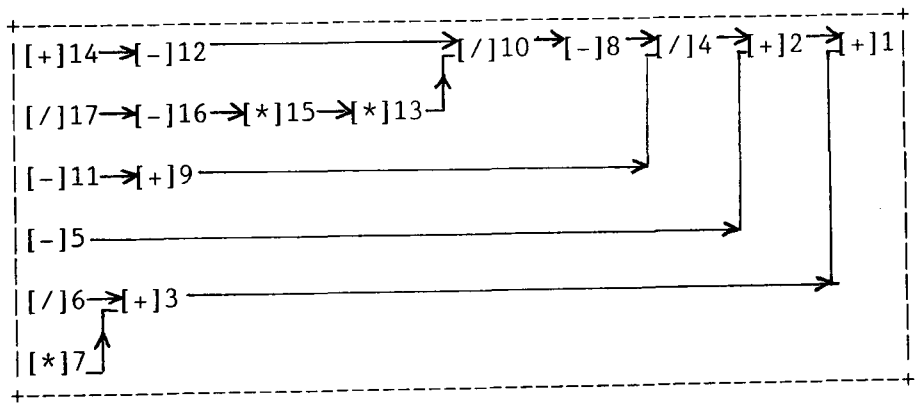
To illustrate the mapping strategy outlined above consider the expression

$$((c-(((b+c)-a)/(a-(b/c))*a*b)))/((a-b)+c))+(c-b)+(b/c)+(a*b)$$

The expression tree for this is shown in figure 7.4(a). An outline of its corresponding schematic representation is given in figure 7.4(b). To aid clarity only the relative placement of cell elements and the routing of each cell's RESOUT data path are outlined. Each cell is labelled



(a): Expression Tree



(b): Cell Arrangement

Figure 7.4: Example Expression Tree And Its Schematic Layout

according to the node in the tree it represents.

The root of the tree is labelled '(+)1'. Both its siblings are themselves trees, so before the corresponding cell can be placed, the expression cell for each tree must be placed. The left tree, with root '(+)2' is considered first. Its left path is descended until a terminal (a node with a variable for each sibling) is reached. In this descent, if a node with a left path directly leading to a variable is encountered (e.g. '(-)8'), then the right sub-tree is descended. Again, the left path of this is followed unless it leads to a variable. Eventually the terminal labelled '(+)14' is reached. The cell corresponding to this is placed in the top left hand corner of the conceptual bounding box. In figure 7.4(b) this cell is represented by '[+]14'. Having placed the cell, its parent is considered, i.e. node '(-)12'. The right branch of this leads directly to a variable and so the corresponding cell is placed immediately to the right of the previous cell. The next node up, '(/)10' is then considered. Its right sibling is a tree and so the expression cell for it must be placed before the node's cell can be. This involves descending the sibling tree, as described above, until the terminal node '(/)17' is reached. Its corresponding cell is placed directly below that for node '(+)14'. Cells can then be placed for the nodes'(-)16', '(*)15', and '(*)13'. These cells are placed to the right of cell '[/]17', as shown in figure 7.4(b), forming a horizontal array.

Both expression cells for the node '(/)10' have now been placed. The left expression cell consists of two operator cells arranged in a horizontal array, while the right consists of the horizontal array of four cells immediately below. The cell for the node is placed to the

right of these, as shown in figure 7.4(b). Note a join cell, indicating the return to sequential behaviour, would be placed just to the left of cell '[/]10', but this is not shown. Node '(-)8' can now be placed since its right expression cell has been laid out. This cell is placed to the right '[/]10'. Before the cell for node '(/)4' can be placed the expression cell for its right sub-tree must be positioned. The same process is repeated, generating an expression cell consisting of '[-]11', '[+]9', and '[-]5'. A cell for node '(/)4' is then placed to the right of '[/]10'. This top down, bottom up traversal of the tree is continued until a cell for each node has been placed.

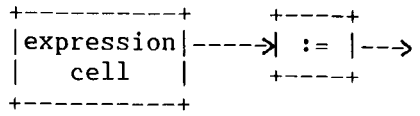
The interpretation of the structural representation of behaviour implied by figure 7.4 will now be considered. The execution sequence is identical to that defined by the tree in figure 7.4(a) and is reflected in the data flow. In the following discussion, integers refer to the actual hardware implementations of the correspondingly labelled schematic cell outlines in figure 7.4(b). In an implementation, 14, 17, 11, 5, 6, and 7 would receive a control token simultaneously, and hence operate concurrently. After 14 generates a control token 12 would commence operating. The same applies to 17 - 16 and 11 - 9. After 16 has completed, 15, and 13 operate one after the other. Before 10 could be activated both 13 and 12 would have had to of terminated. When 9 and 10 have generated their control tokens, then 8 would be initiated, followed by 4. The operation of 3 is dependent on the completion of 6 and 7; 2 is dependent on 4 and 5; and finally 1 is dependent on 2 and 3.

7.4 SIMPLE CELLS

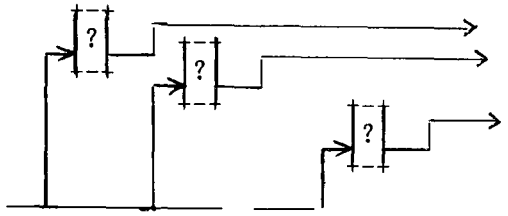
The simple processes are mapped into abstract simple cells, which consist of one or more operator cells and lines defining their interconnection and communication. The arrangement of these cells within the bounds of a simple cell is determined by the implicit sequence of operations in the corresponding process. An outline of the schematic representation for each of the three simple processes are given in figure 7.5.

An assignment process is mapped into two cells: an abstract expression cell and an assignment operator cell. The abstract representation of an expression is used, the actual layout of constituent operator cells and their interconnection was described in the previous section. External data lines are routed to the expression cell. These lines fetch the value of the variables referenced in the corresponding expression. A data line connecting the two cells is used to transfer the value of the expression to the operator cell.

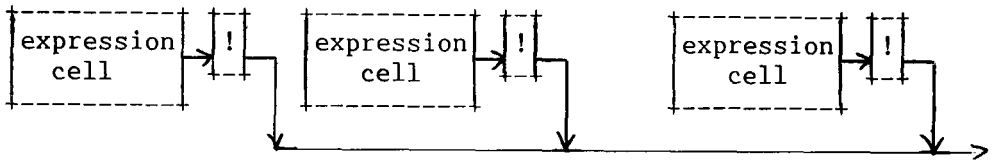
An input cell is comprised of n input operator cells, where n denotes the number of constituent read operations in the corresponding input process and is greater than or equal to one. That is, the mapping produces an input operator cell for each read operation in the process. These cells are placed sequentially to form a horizontal linear array. A power, ground, and clock line is routed to each cell, while the control line links the component cells together for sequential execution. The data output line from each cell is routed along the top of the array, from left to right, forming a bundle of lines. The channel line corresponding to the process is routed from left to right, connecting



(a) An Abstract Assignment Cell



(b) An Abstract Input Cell



(c) An abstract Output Cell

Figure 7.5: Arrangement Of Components In Abstract Simple Cells

with each subsidiary cell's CHANIN input, and terminating in the right most.

An output cell consists of n expression cells and n output operator cells, where n represents the number of write operations occurring in the corresponding process and is grater than or equal to one. These cells are arranged into expression/operator pairs, with each pair corresponding to one write operation. The n pairs of cells are placed sequentially forming a horizontal linear array, as illustrated in figure .7.5(c). Power, ground and clock lines are routed into each cell. The control

line connects the control output of each cell to the control input of its right neighbour, so the cells execute sequentially. The output channel line is routed from the channel output port of the left most cell to the east edge of the array, passing underneath the intervening pairs of cells. Each intervening operator cell is also connected to this channel line.

7.5 COMPLEX CELLS

Complex processes are comprised of a number of subsidiaries which are either simple or other complex processes. The abstract cell for each subsidiary is defined by the compiler and placed as described in section 7.2. So, a sequential process is mapped into a horizontal array of cells (see figure 7.2(a)), while a parallel process is mapped into a vertical array of cells (see figure 7.2(b)). The cells in these arrays are either simple abstract cells or other complex cells. In figure 7.2(a) the left most cell depicts the cell for the sequential process' first subsidiary, while the right most cell depicts that for the last (n th.) subsidiary process. In figure 7.2(b), the top most cell is the abstract cell derived from the first subsidiary process defined, the second from top is that for the second process,..., and the bottom cell is that representing the final subsidiary.

7.6 IMPLEMENTATION DETAILS OF THE GRAPHICS COMPILER

The current version of COPTS is a Pascal program (approximately 6500 lines) which translates a source program into an equivalent graphical representation. A more detailed description of its implementation is

presented in Appendix E. Source programs for it are written in the language defined by the syntax given in Appendix C. These programs are first analysed and then their corresponding schematic representation are generated. An integral part of the compiler is its internal data structure. This is used to save the information obtained during both these steps. The data structure is represented by the record type '**CmpntRec**' (see Appendix F). Pointers to records of this type are used by different parts of the program to access and manipulate the data structure.

The 'CmpntRec' record type was designed to implement a data base which reflects the block structure and hierarchical nature of the source language. This internal data base has a tree structure. It is used to hold both the internal representation of the source program and the information defining the corresponding graphical representation.

The compiler's SYMBOL TABLE is implemented by records of the type '**SymTabRec**' (see Appendix F). References to this type of record are distributed throughout the data structure. The record associated with such a reference holds information on an identifier and represents a symbol table entry.

The phases of the compiler, in order of execution, are as follows.

7.6.1 Parse Phase

Consists of two parts: a SCANNER and an ANALYSER. The scanner reads in the characters of the source program and constructs the symbols of the program. These symbols are classified into integers, identifier

strings, keywords, and delimiters ('+', '-', '*', '/', ':='...). The symbols are passed to the analyser in an internal form. Each symbol is represented by a TOKEN. A token has an integer value associated with it, denoting the symbol it represents.

The analyser performs a syntax and semantic check of the program. This involves building up the internal form of the program - the PARSE TREE - in the compiler's data structure. Information on the declaration of identifiers is gathered and stored. Expressions are analysed and stored in their Reverse Polish form. As each source language construct is recognised it is checked for semantic correctness. For example, during the parse of a declaration the identifiers are checked to see if they have been declared twice.

7.6.2 Graphical Specification Phase

This phase completes the internal representation of the source program. This involves a preorder traversal of the compiler's internal data tree. During this traversal information on the occurrences of identifiers is gathered and stored. Also, any Polish expressions in the data structure are transformed into corresponding tree representations. These "expression trees" specify the order of execution for the operators of the expression. The conceptual structure of such a tree is later mapped directly into a structural representation. Information on an expression tree is held in a record of the type '**XprsnRec**'.

7.6.3 Graphical Definition Phase

This phase translates the internal source program into the

corresponding internal definition of the graphical representation. This involves conceptually building up a structural implementation of this representation on an imaginary grid. Information defining the implementation is stored in the compiler's data tree. The recursive algorithm developed to generate this information incorporates a preorder traversal of the internal tree.

During the traversal information in the internal tree is examined. Information on each primitive operator is used to define, in terms of co-ordinates and units of the grid, the occurrence of the corresponding graphical (cell) in the full representation. The resulting cell definition information is stored along side the internal representation of the primitive. Information specifying the occurrences of each identifier is examined. It is used to define a set of vertical and horizontal lines. This set of lines maps the occurrences of a particular identifier in the source program to the corresponding graphical representation. Again the implementation information generated is stored.

The parse phase stores information on the execution sequence of the source program. This is also examined and used to generate plotting information on a set of lines. The lines in this set link all the primitives on the grid and represent the mapping for the control sequence.

7.6.4 Schematic Output Phase

The final phase of the compiler generates the object code defining the graphical representation. Another preorder traversal of the internal

tree is used to access the internal definition of the graphical representation. During the traversal information on each line defined for the graphical representation is written out. The resulting output is then sorted and formatted to produce the object code. A line plotting program later uses the object code to display the schematic representation of the Occam source program.

The first two phases correspond to the analysis step, while the last two phases correspond to the generation of the graphical representation.

There is a fifth phase to the program, the dump phase. The execution of this phase is not essential. When used it writes out the contents of the tree data structure after the parse and specification phase. It was designed as an aid in developing the program.

CHAPTER 8

RESULTS

Six example programs written in the defined subset were used to test the implemented graphics compiler. The first three programs together with their corresponding schematic diagram are included here, while the remaining examples are included in appendix D. These programs are intended to merely illustrate the features of Occam currently handled and do not describe any particular algorithm. The defined subset limits the complexity of programs which can be developed. Within these bounds the example programs range in 'complexity', the simplest consists of twenty-three lines of code, while the two most complex each contain fifty-eight lines. In this chapter the schematic diagram of one example program (Program One) will be discussed in some detail. A brief description of the remaining programs and important features of their schematics is also included. An overall assessment of the schematics is made with reference to their limitations and some suggested improvements.

Before considering the first example some general introductory notes to the diagrams included here are required. All red boxes are operator cells, yellow boxes are join cells, and a purple box represents a

register corresponding to one or more variables (i.e. a register bank). A bounding box (drawn in black) surrounds each diagram. The top four inputs to this box represent the system's power, ground, clock and control input lines respectively. Any remaining lines on the west edge are input communication lines. On the east edge of the bounding box outputs are communication lines and the control line out. The latter can be identified by the fact that it is the only output which can be traced back to a join cell. Blue and green are used to distinguish between horizontal and vertical routing of lines.

8.1 A SIMPLE EXAMPLE OF COMMUNICATING PROCESSES

Program One, shown in figure 8.1, inputs two values, computes some intermediate values and then outputs two results. The schematic diagram generated for this program is shown in Plate 1. A key to this diagram is given in figure 8.2. The boxes in this figure are labelled according to the operator of the cell it is associated with. Each operator is numbered according to its position in the source code. So, for example, "(*)5" represents the fifth multiply operator in Program One. Registers are also identified and labelled. In Plate 1 there are six inputs to the bounding box, the top three are the power, ground, and clock lines respectively. Each operator cell and register element is directly linked to these lines. The fourth from top is the control input line. This line threads its way through the diagram, forking and joining until it eventually becomes an output. The other two input lines are the communication paths corresponding to the channels 'In1' and 'In2'.

```
CHAN In1, In2, Out1, Out2:
VAR a, b:
SEQ
  PAR
    In1? a
    In2? b
  CHAN Comm1, Comm2:
  PAR
    VAR t1, t2:
    SEQ
      PAR
        t1 := a * a
        t2 := b * b
      PAR
        Out1! t1+(a*b)+t2
        Comm1! t1
        Comm2! t2
    VAR asq, bsq:
    SEQ
      PAR
        Comm1? asq
        Comm2? bsq
      Out2! (asq*b) - ((a/b) - (bsq*a))
```

Figure 8.1: Listing Of Program One.

The program defines a sequential process which is comprised of two subsidiaries, each of which is a parallel process. The first simply inputs a value from each of the channels 'In1' and 'In2' and stores them in the variables 'a' and 'b' respectively. This is represented by the two left most cells, labelled "(?)1" and register element R1. The communication line for each channel is routed to the corresponding input cell. R1 is a register with two storage locations (words), one for 'a' and the other for 'b'. Its upper input is the assignment line for 'a' and the lower that for 'b'. Since the two inputs are defined to occur in parallel, the control line forks to link with the CNTRLINs of the input cells. A line form the CNTRLOUT of each cell is routed to the join cell immediately to the left of "(?)1". This represents the return to sequential behaviour.

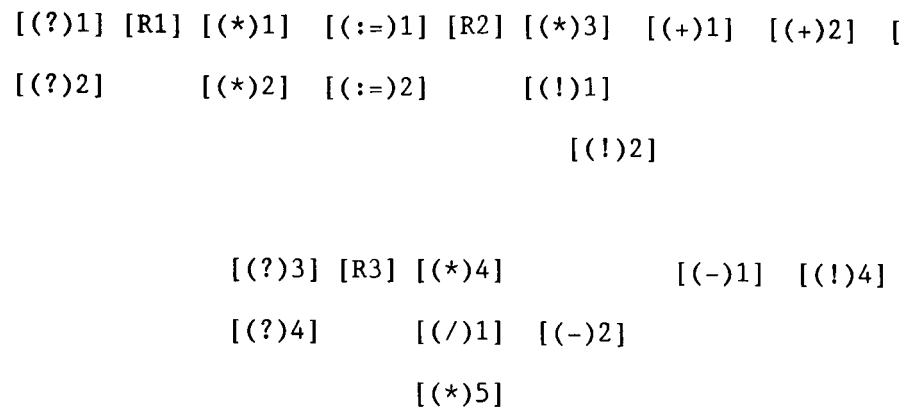


Figure 8.2: Key To Plate 1.

The remainder of the schematic diagram represents the second parallel process to be declared in the program. This is more involved and consists of two sequential processes. These communicate, and hence synchronise behaviour, via the channels 'Comm1' and 'Comm2'. The line linking the CHANOUT of cell "(!)1" to the CHANIN of "(?)3" represents the former. Comm2 is represented by the line connecting the CHANOUT of cell "(!)2" to the CHANIN of cell "(?)4".

The first sequential process squares 'a' and 'b' in parallel, assigning the results to the variables 't1' and 't2'. It then concurrently outputs these to the other sequential process, and outputs the value of an arithmetic expression on 'Out1'. Cells "(*)1" and "(:=)1" represent ':= a * a', while cells "(*)2" and "(:=)2" represent ':= b*b'. The register R2 represents the two variables t1 and t2. Both cells "(*)1" and "(*)2" two DATAIN lines can be traced back to the outputs of R1. The former's to the upper output (i.e. the reference line for a) and the latter's to the lower output (i.e. the reference line for b) Cells "(*)3", "(+)1", "(+)2" and "(!)3" represent the output process

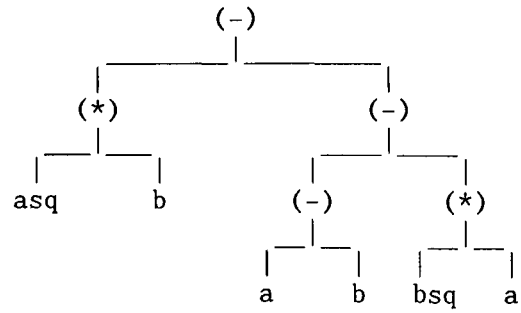


FIGURE 8.3: Expression Tree For '(asq*b)-((a/b)- (bsq*a))'

using Out1. The other two output processes are represented by "(!)1" and "(!)2. Placement and interconnection of cells for all three subsidiaries reflects the sequential execution of the process.

The second sequential process concurrently inputs the squared values of a and b, storing them in asq and bsq. It then computes the value of an arithmetic expression, placing its value on Out2. Cells "(?)3" and "(?)4" represent the input operators. The register R3 represents the variables asq and bsq. The output process evaluates the expression

(asq * b) - ((a/b) - (bsq*a))

Its expression tree is shown in figure 8.3. The arrangement of the cells "(*)4", "(/)1", "(*)5", "(-)2", and "(-)1" illustrates how the conceptual structure of the tree is mapped into arrangement of operator cells. Careful tracing back of an operator cell's data inputs reveals its operands.

In the schematic diagram there are seven join cells. Each one depicts the return to sequential behaviour after concurrent processing. These cells will now be considered in left to right order. The left most cell signals the completion of the program's first parallel process. The second from left signals the completion of the parallel process which inputs values for asq and bsq. Completion of the parallel process which squares a and b is signalled by the third from left. Both the fourth and fifth cells signal the completion of intermediate parallel steps in the evaluation of the expression given above. The second from right cell signals the completion of the parallel process consisting of three outputs. Finally, the right most cell produces the control out line signalling the completion of the program's two subsidiaries.

8.2 OTHER EXAMPLE PROGRAMS HANDLED BY COPTS

The second example is intended to demonstrate the ability of the implemented compiler to handle a program comprised of more than two complex subsidiary processes. A listing of the program is given in figure 8.4. This example also demonstrates the compiler's ability to handle the re-assignment of variables. Program Two defines a sequential process consisting of four parallel sub-processes. Its first subsidiary inputs a value from each of the channels 'input1', 'input2', and 'input3'. The second subsidiary is responsible for assigning values to 't1', 't2' and 't3'. It consists of two assignment processes and two sequential processes which communicate via the channel 'intrn'. The third subsidiary consists of three assignment processes which re-assign values to a, b and c. In the last subsidiary three output processes write the value of an expression to their associated channels.

```
CHAN input1, input2, input3, output1, output2, output3:
VAR a, b, c, t1, t2, t3:
SEQ
  PAR
    input1 ? a
    input2 ? b
    input3 ? c
  CHAN intrn:
  PAR
    t1 := a + (b * c)
    t2 := a - ((b+c)*(b/(a+c)))
    VAR temp1, temp2:
    SEQ
      PAR
        temp1 := a + (b - c)
        temp2 := c * (b + a)
      intrn ! (temp1*temp2) + (temp1/temp2)
    VAR M,N:
    SEQ
      PAR
        intrn ? M
        N := a*b*c
      t3 := (N*(M-a)) + (N-b)
  PAR
    a := a * a
    b := b * b
    c := c * c
  PAR
    output1 ! (t1*t2*t3) + (a-(b+c))
    output2 ! (t1/(t2+t3)) + a + b + c
    output3 ! (t3-(t1*t2)) - ((a*b)+c)
```

Figure 8.4: Listing Of Program Two

In the schematic diagram corresponding to this program four 'blocks' of cells can be distinguished from left to right. Each block represents the abstract cell generated for the four processes defined in the program. The left most block illustrates the parallel input of three values and the register bank in which they are stored. Second from left is the block corresponding to the second process in which the variables t1, t2, and t3 are assigned a value. This block shows the routing of the assignment lines for these variables and the placement of the register

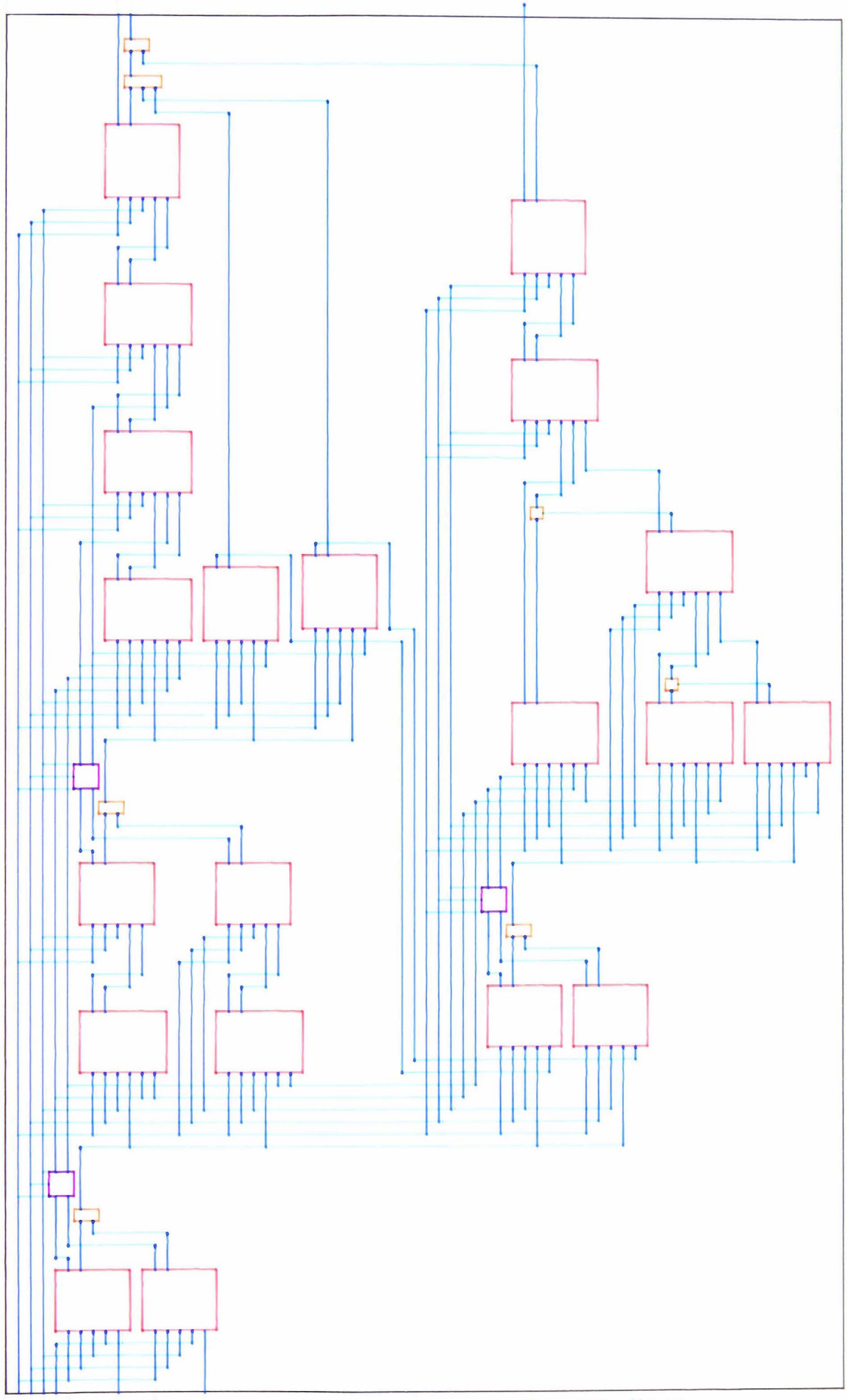
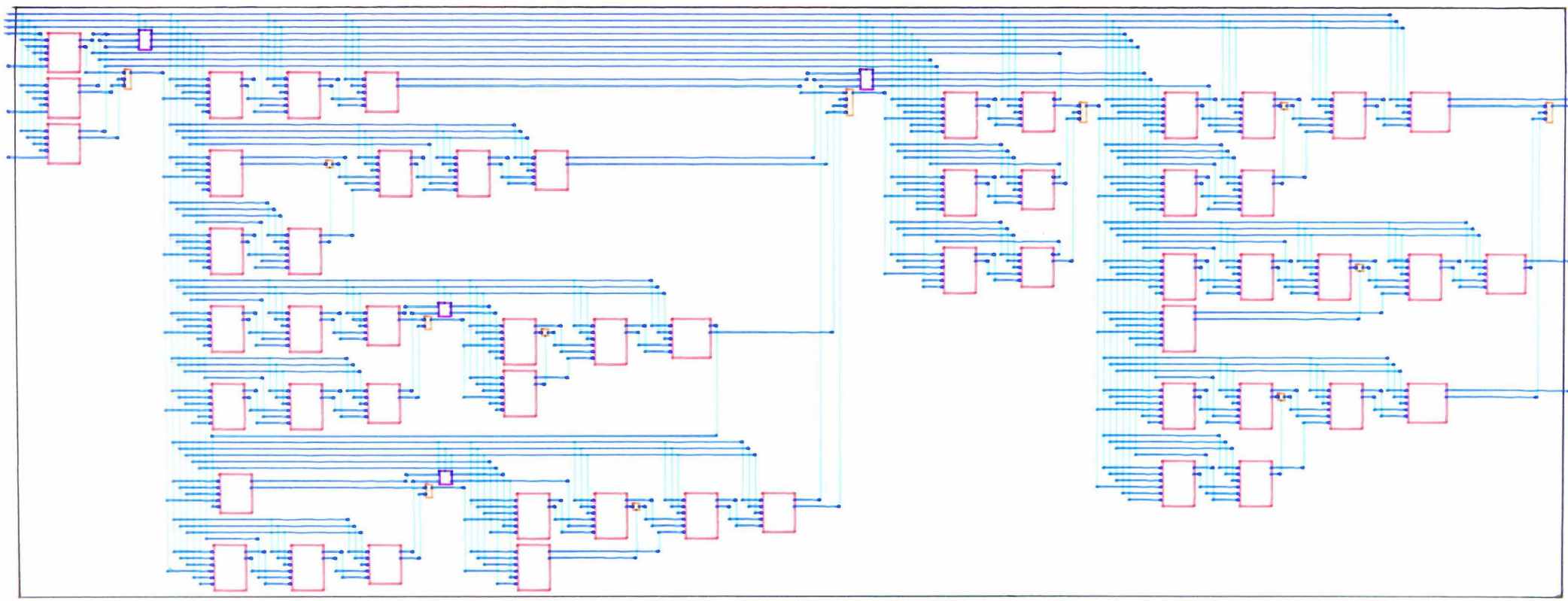


PLATE ONE

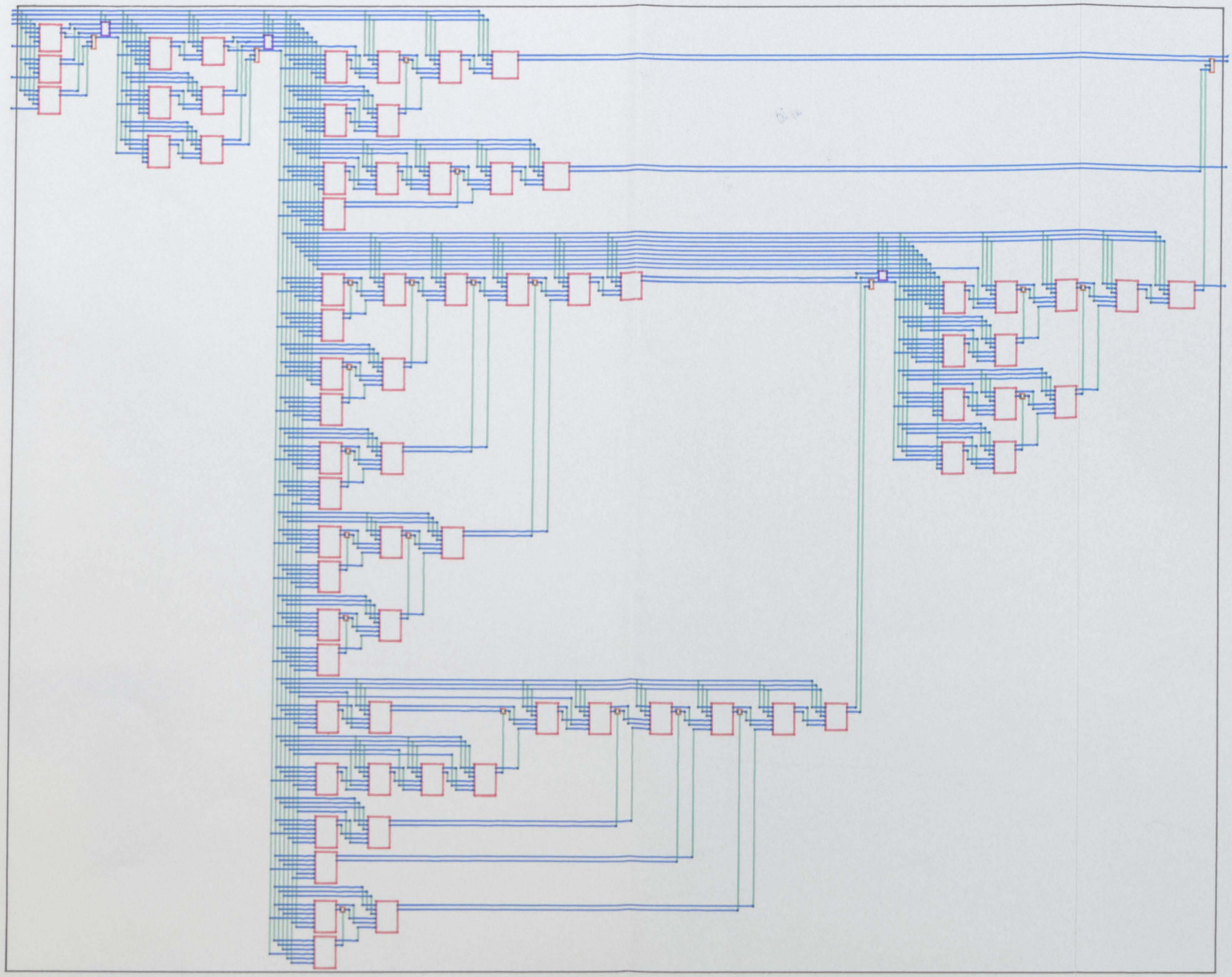
—> x

PLATE TWO



y
x

PLATE THREE



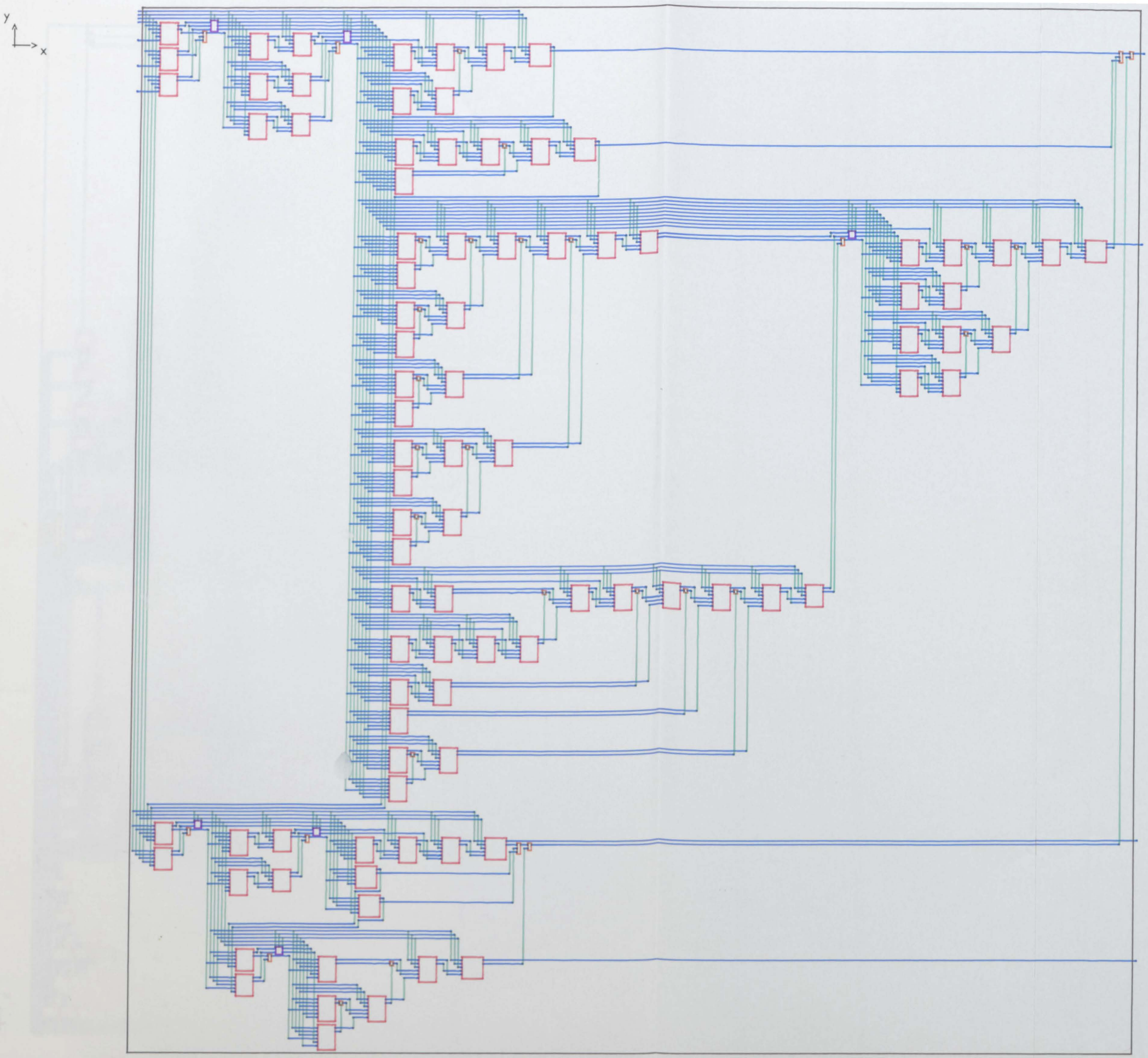


PLATE
FOUR

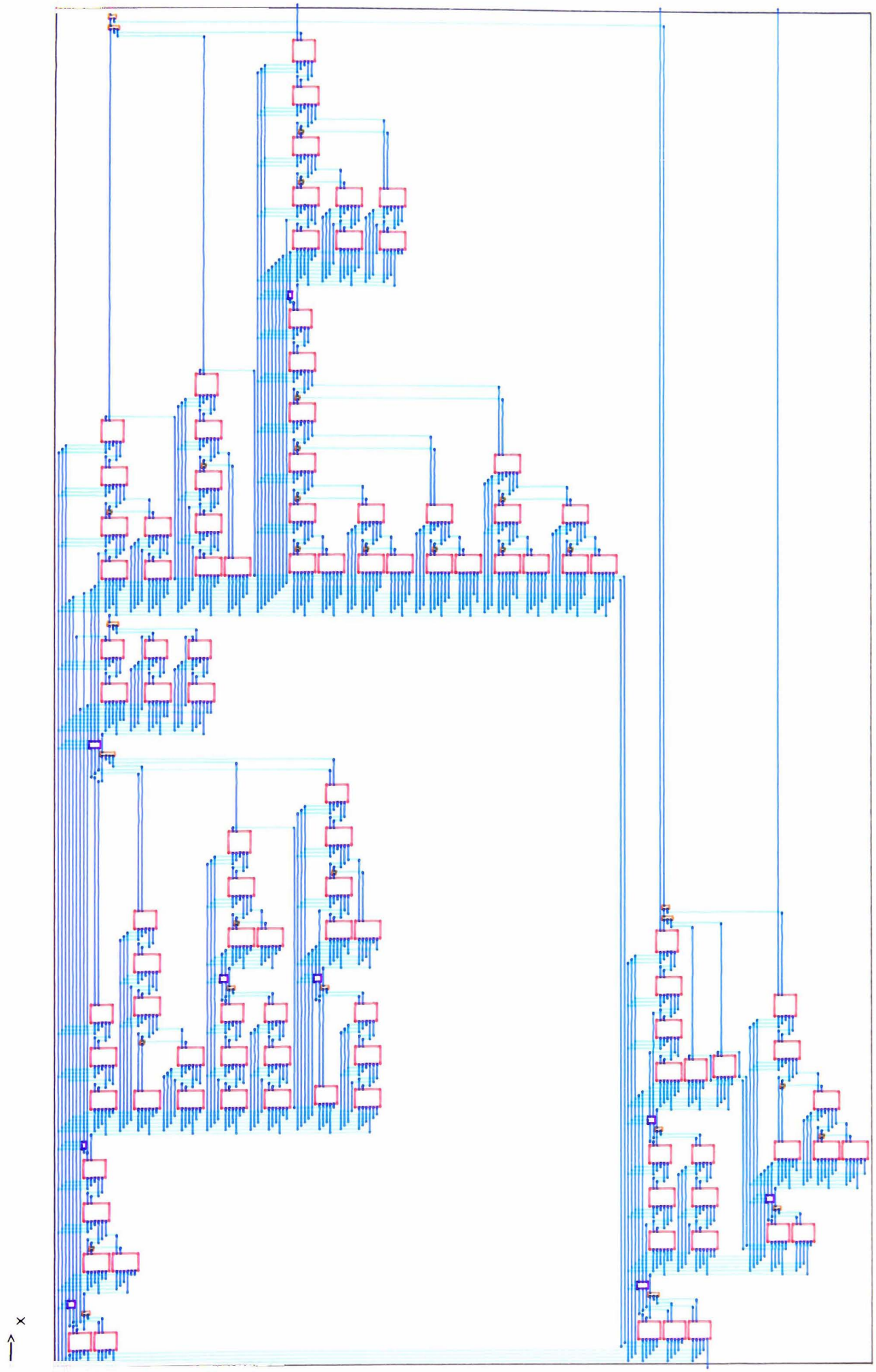


PLATE FIVE

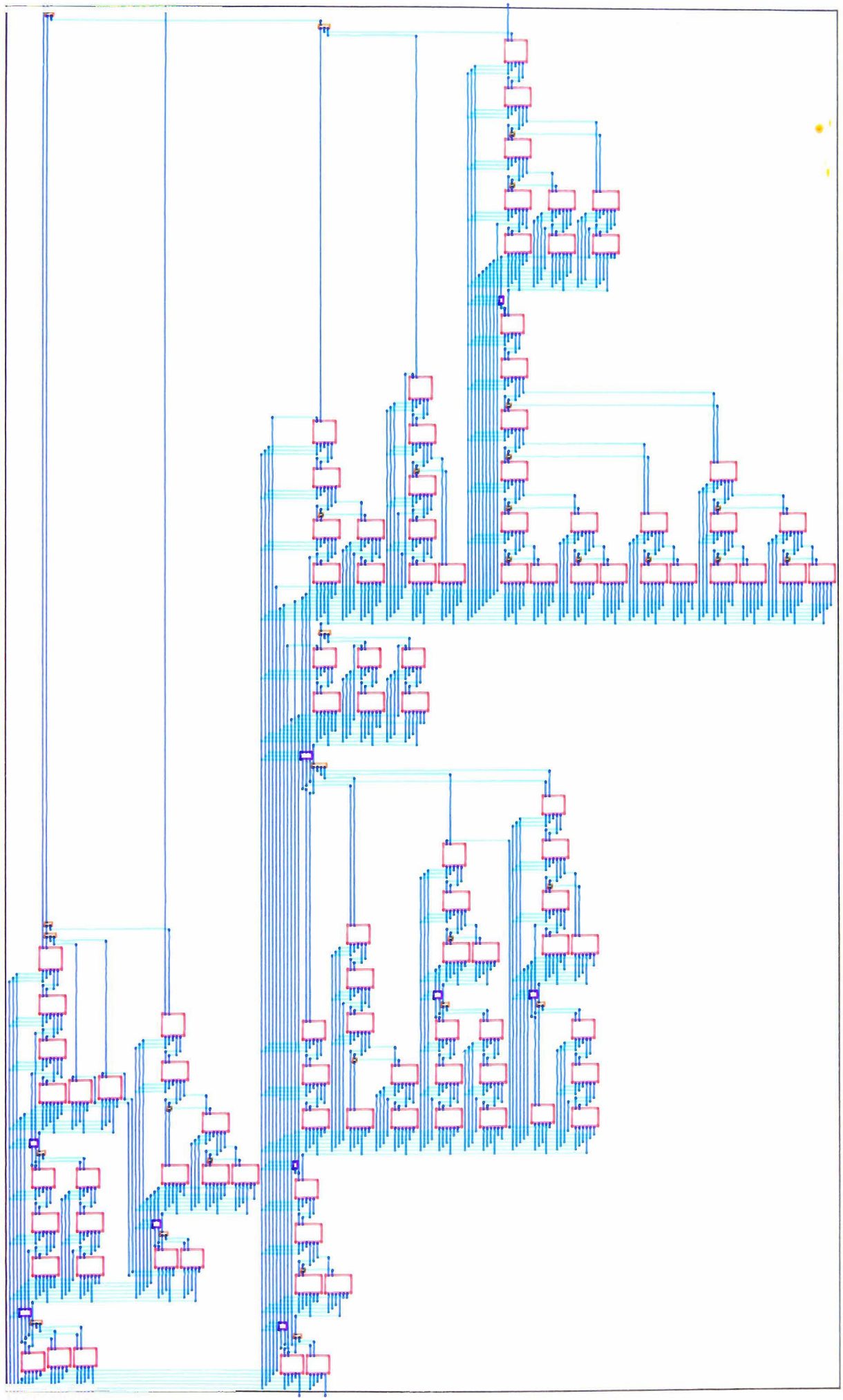


PLATE SIX

bank intended to hold their values. It also shows the routing of the communication path representing channel intrn. This path is routed from the second right most cell in this block to the second from bottom left most cell (i.e. the output and input cells) The second from right block illustrates the re-assignment of the tree variables a, b and c. Note that the DATAOUT line of the assignment cells (the right most red boxes in this block) are routed back to the register bank adjacent to the left most block. The layout for three output processes in the program's fourth subsidiary is shown in the right most block.

Program Three is intended to demonstrate the ability of the compiler to layout detailed arithmetic expressions. A listing of the program is given in figure 8.5. It defines a sequential process consisting of three parallel subsidiaries. There is no communication between these processes. The first subsidiary to be declared simply inputs three values in parallel. In the schematic diagram, the tree left most cells correspond to these inputs. Three simple assignment processes are defined in the second subsidiary. Their schematic representation is immediately to the right of the input cells. It consists of two vertical columns, each containing three cells. Cells in the first depict the multiply operator, while those in the second the assignment operator. The remainder of the schematic diagram represents the final subsidiary. this consists of two output processes and a sequential process. The latter contains two assignments involving the evaluation of complex expressions. One of these is the example described in section 7.3.2. The schematic representation previously outlined for it can now be seen in full in the bottom third of the diagram.

```
CHAN input1,input2,input3,output1,output2,output3:
VAR a, b, c, t1, t2, t3:
SEQ
  PAR
    input1 ? a
    input2 ? b
    input3 ? c
  PAR
    t1 := a * a
    t2 := b * b
    t3 := c * c
  PAR
    output1 ! (t1*t2*t3) + (a-(b+c))
    output2 ! (t1/(t2+t3)) + a + b + c
    VAR x,y:
    SEQ
      PAR
        x:=(a+b)+(b+c)+(t1+t2)+(a+c)+(a-b)+(c-a)+(c-b)+(a/b)+(b/c)+(a*b)
        y:=((c-(((b+c)-a)/((a-(b/c))*a*b)))/((a-b)+c))+(c-b)+(b/c)+(a*b)
      output3 ! (t3-(t1*t2)) + ((a*b)+c) + (x*x*x) + (y*y*y)
```

Figure 8.5: Listing Of Program Three.

The fourth example is a program which links the process defined by Program One to that defined in the previous example. That is, Program Four defines a parallel process consisting of two subsidiaries. The first is equivalent to the process defined by Program One, while the second is equivalent to the process defined by Program Three. These two subsidiaries communicate and synchronise behaviour through the channels 'intn1' and 'intrn2'. The resulting diagram can be divided into an upper and lower block. A very close similarity between the upper, larger block and the diagram for Program Three and between the lower smaller block and the diagram generated for Program One can clearly be seen. The diagram illustrates the routing of the communication paths between the two subsidiaries. These lines are routed from the east edge of the top two right most cells in the upper block to the west edge of the top two left most cells in the lower block.

The last two examples describe the same program, they differ only in the order in which subsidiary processes are declared. They are intended to demonstrate how re-ordering the sequence of process declarations changes the schematic diagrams generated. The program consists of two sequential processes which operate concurrently. They communicate and synchronise through the two channels named 'intrnal1' and 'intrnal2'. Both processes are modified versions of previous programs. One is a slightly modified version of the process defined by Program One, while the other is a slightly extended version of the process defined by Program two. The schematic diagram for each can be divided into an upper and a lower. Each block represents the layout of the abstract cell for the program's two complex subsidiary processes. The largest block corresponds to the process based on Program Two and the smaller to that based on Program One. In the two diagrams the position of the blocks is reversed. This illustrates how the order in which a process is declared determines the position of the abstract cell and its components.

In reversing the order in which the two processes are declared, the positions in the program of the input and output statements defining the communication between the two processes is altered. In Program Five the first process declared outputs data to the second, but in Program Six the first process declared inputs data sent by the second. Note that input/output behaviour between the two is unaffected, the two programs are semantically equivalent. The routing of the data lines for 'intrnal1' and 'intrnal2' is, however, different for the two diagrams. This is because the routing strategy for communication lines is based upon the position in the program of the input and output processes for the associated channel. In the diagram for Program Five these lines are

routed from the top two right most cells in the upper block to the top two left most cells in the bottom block. While in the diagram for Program Six they are routed from the top two left most cells in the upper block to the top two right most cells in the lower block.

8.3 SUMMARY

The selected examples illustrate the current compiler's ability to generate schematic diagrams for a limited subset of Occam. Their corresponding diagrams provide an adequate representation of structure, data flow, and control flow. The necessary information for representing these is extracted from the source programs. The diagrams, however, suffer form a number of shortcomings. In particular, the cells and registers are not labelled. This makes it difficult to examine the correspondence between a node and a software primitive in the source code. It would be desirable to have each cell labelled with the operator it depicts and each register element labelled with the variables it represents. Another problem with the diagrams which effects their clarity is the number of lines displayed. One solution would be to reduce the number by hiding (i.e. not displaying) the power, ground and clock lines since these are so fundamental to MOS IC design that their presence can be assumed. Alternatively, future advances will make it possible to route these lines in other layers, in which case a separate diagram for their routing could be generated.

Although not yet demonstrated, the diagrams as they stand could conceivably be used as the starting point for prototype implementations. Such implementations would be far from ideal and would suffer from a

massive area overhead and poor performance. Nevertheless, they would demonstrate a feasible route from Occam programs to geometric layouts. To obtain implementations would involve substituting the schematic cells for actual blocks of layout. The necessary protocols for synchronised data communication and register update would have to be implemented. If bit parallel communication is to be used then the data paths involved would have to be decomposed into the appropriate number of metal wires. The timing mechanisms would also need to be carefully considered. As a result of the simplicity of the mapping strategy the diagrams currently contain a lot of white space. However, layout compactors are available [76] which would reduce the wasted silicon area in an implementation. A lack of time and power resources (the work to date has been a single effort) prevented this route from being explored.

The most serious limitation of the graphics compiler is the subset of Occam it can handle. This contains the bare minimum so as to just allow complex communicating processes to be declared. It cannot at the moment be practically used describe real applications. The subset and compiler must be extended to incorporate conditionals, iteration and replication. Named processes would be advantageous in the long term but need not necessarily be included in a first extension.

CHAPTER 9

CONCLUSIONS

The densities made available through advances in the fabrication technology enable sophisticated systems to be laid out on a silicon chip. Single chip solutions are now possible for many systems previously implemented with printed circuit boards containing many components. Also, special-purpose (or application specific) chips as opposed to programmed general-purpose microprocessors can now be considered. However, the complexities now involved in custom design together with the failure of CAD techniques to keep pace with the advances have resulted in a design bottleneck. Consequently, the ability of designers to maximise the benefits now available have been limited. Powerful tools aimed at exploiting the processing potential offered by the technology are required. The traditional graphical based CAD tools are no longer appropriate since they place the emphasis on layout and a 'bottom up' design style.

A structured 'top down' approach similar to that used by software engineers has been recognised as appropriate for the implementation of 'complete systems on a chip'. Such an approach allows the designer to

abstract away from detailed physical considerations and concentrate on higher, functional issues. As VLSI designs become more and more ambitious, higher levels of abstraction are needed if the designers are not to be overwhelmed by complexity. Textual descriptions, in formal design languages, are more powerful and precise than graphical representations for these higher levels since they encourage a more structured approach. Therefore, design languages are playing an increasingly significant role in VLSI design. They have evolved from simple plotting notations to the point where they can now handle both the behavioural specification of a design and its structural implementation. Structures such as the conditional, loop, procedure, and arithmetic expression are as important features of these languages as they are of programming languages.

High level descriptions in design languages are translated into an equivalent circuit implementation. Much of this translation is carried out manually and typically involves passing through several intermediate levels of abstraction. Two different CAD approaches are emerging to support the designer in this process: expert design systems and silicon compilers. The former assists the designer down the design hierarchy, while the latter completely automates the translation process. The technique of silicon compilation is favoured for making the design of "one-off" application specific chips feasible. The reasons being: it enables very fast design turnarounds, greatly reduces design costs, makes the silicon medium accessible to non VLSI design specialists, and produces a working chip first time. The overhead for these advantages is an excessive use of silicon area. However, the expected ten-fold increase in densities will lessen the constraint imposed by silicon area

giving silicon compilers a wider acceptance.

Insight into the future development of design languages and their automated translation can be gained by drawing parallels form the software field. The motivations behind the evolution of today's software environments are similar to those forcing the development of advanced CAD techniques. This similarity arises from a common problem - complexity. Many of the techniques developed by software engineers to tackle this problem may be applicable in the VLSI domain. In particular, high level programming languages, compilation and operating systems.

The source code, or input design language, to a silicon compiler requires careful consideration since it determines the ease of mapping design descriptions into silicon implementations. At the levels of abstraction now being considered for design, the emphasis is on the behavioural characteristics of a system rather than its structural implementation. This shift in emphasis reflects the importance of the choice of the underlying algorithm in a design. It is the most crucial design decision and determines the performance and area of the silicon implementation. For this stage of the design to receive the required attention algorithmic notations are expected to replace the current structural/behavioural ones. Two choices are available for an algorithmic notation: a fully programmable design language or a conventional programming language. The latter was selected here since it eases the task of implementing a silicon compiler and excludes the need to design and support a special purpose hardware design language. There are numerous programming languages which could be considered. Each one's performance in a design description role is determined by the ability of

its underlying computational model to represent the features of VLSI systems. Important features are: parallelism, communication, and localised processing. Five computational models were examined: sequential control flow, parallel control flow, actor, functional, and logic. An example language from each category was selected and its ability to describe the VLSI implementation of a pattern matching chip assessed.

Pascal illustrated the use of abstraction through powerful data structures but showed that the sequential control flow model gave rise to designs which utilise a Von Neumann serial processing architecture. SmallTalk demonstrated that the class construct in the Actor model can be used to define data and processing characteristics of computational elements. However, the sequential nature of the language meant that the operation of such elements in parallel could not be represented. Lispkit Lisp showed that the functional model is well suited for representing wiring, connectivity, and parallelism. The major drawback of this model is the lack of the assignment statement making state difficult to model. For the logic model, Prolog illustrated how unification with named variables can be used to represent connections between circuit elements. The relational approach in this model is too abstract for many basic devices. Based on the parallel control flow model, Occam enabled a system to be described as a collection of concurrent processes, which communicate through named channels. This closely reflects the structure of VLSI systems composed of a large number of processing components operating together and linked via wires. The language also handles concurrency in a formal manner. The major disadvantage with the language is the lack of abstract types.

Occam's ability to express some of the important characteristics of VLSI design makes it an attractive language for describing hardware. An Occam description of a system can serve two purposes. Firstly, a designer can execute the program as a simulator, or prototype, of the system, and investigate it and modify it in the usual software way (i.e. through a standard editor and re-compilation). Secondly, when a designer is confident that the program represents a satisfactory description of a system's underlying algorithm, it can be used as the behavioural specification for a VLSI implementation. That is, an Occam program can be used to characterise the input/output behaviour of a VLSI chip. Deriving a layout description for such a chip would involve transforming the program into a detailed circuit description. The envisaged features of an ATL system (ATLAST) for achieving this transformation have been specified. The VLSI implementations obtained from ATLAST will be based on the self timed approach.

A prototype implementation of one of ATLAST's features has been developed - COPTS. This is a tool for generating schematic diagrams corresponding to Occam descriptions. The ability of COPTS to automatically generate a high level schematic representation of programs written in a subset of Occam has been demonstrated. These schematics depict an Occam description's realisation in silicon. They illustrate the flow of data and control information in a program. The diagrams also provide a visual representation of the parallelism present in it. The topological information required to define a schematic representation is extracted from the natural hierarchy found in Occam programs. It is envisaged that future advances in the fabrication technology will be such that silicon area may no longer be considered a scarce resource. In

which case, the schematics generated by COPTS will serve as a high level structural definition of a system from which an actual implementation can be directly derived. Probably a more realistic approach is to regard the schematics as defining a virtual architecture from which an actual implementation could be derived. This would involve removing some of the replication of operators. To achieve this some of the parallel behaviour in a system could be transformed into sequential iterative (multiplexed) behaviour.

9.1 RELATED WORK

The technique of silicon compilation has received considerable attention since the term was introduced by Johannsen [34] in 1979. At least two text books on the subject have been published. Ayres [3] describes a methodology for implementing a silicon compiler (or more accurately a silicon assembler), while Denyer [12] presents a study of a particular structural compiler for signal processing applications. It is argued here that this approach to VLSI design needs extending beyond structural silicon compilation toward automated algorithm to layout systems. The advantages of using a conventional programming language as the input to ATL systems has been considered. Occam was chosen as a suitable language. Others have also considered using programming languages in VLSI design. For example, Clocksin [9] has argued that Prolog is an appropriate language, whereas Robinson and Dion [51] favour Modula-2. More specifically, Trikey [66] has reported the implementation of a system for automatically translating Pascal programs into silicon. A literature search revealed only one other design system for generating VLSI implementations form Occam programs.

9.1.1 Occam To CMOS

Researches at Fujistu have implemented a prototype expert system intended to map an Occam specification into a standard data path architecture [43]. It is intended that this architecture will be implemented by CMOS circuits. The process involves three stages of design: functional, circuit, and CMOS. Each stage is supported by a separate design subsystem composed of several modules. Functional design involves algorithmic investigation of the characteristics of a system. The final Occam description is translated into an optimised functional description of a finite state machine. This translation involves designer interaction to determine word sizes for variables and the implementation of Occam's inter-process communication. The state machine description is transformed into a structural description for the circuit design stage.

Circuit design involves generating information on the data paths and control paths. This information is in the form of logical expressions. These expressions are decomposed into sub-expressions in such away that each sub-expression can be implemented by a single CMOS functional cell. Such cells are produced during the CMOS design stage. The logic expressions are translated into combinational circuits which are then implemented on an array of CMOS transistors. Also, during this stage components such as registers, memories, decoders, adders, and I/O pins are assembled from a library of basic cells. After the basic cells and functional cells have been assembled a facility to optimise the CMOS circuit is available.

ATLAST and the above system share a common objective: to translate algorithmic notations in Occam into VLSI implementations. When fully implemented both will enable a designer to investigate Occam programs in a familiar software environment. ATLAST will completely automate the translation of optimised programs, while the above requires the designer to steer this translation. The Fujistu system uses a target architecture (the standard data path) in its approach, whereas ATLAST uses a target model (that of distributed processing). Adopting a target model rather than a target architecture will give ATLAST greater leverage since its model may be implemented by various architectures. The architecture used for a particular design will depend on the performance requirements.

9.2 FUTURE WORK

Occam has been identified as an appropriate programming language on which to base a future high level automated algorithm to VLSI layout design system. The proposed features for an example system, ATLAST, were outlined in chapter 6 (see figure 6.1). Future scaling down of feature size and scaling up of chip area will result in propagation delays within a chip causing significant wiring delays among functional blocks. Clock skew will become such a problem that synchronous behaviour through the use of a global clock will no longer be achievable. For this reason the self-timed model has been adopted for systems implemented by ATLAST. Also, such an approach is more in keeping with a rigorous discipline of modularity.

When fully implemented, ATLAST will be a general purpose VLSI design system which will broaden the spectrum of algorithms implemented by special purpose VLSI chips. It will help bridge the gap between software design and hardware design by providing an automated means for directly mapping software implementations into hardware implementations. It is intended that the system will be easy to use and understand, and it is hoped that it will make the VLSI technology more accessible to the non specialist. Users of the system will not require detailed hardware knowledge, since the design approach is based on a very high level of abstraction. However, it is assumed that a user will have some experience in programming. The central feature of ATLAST will be a transformation module. This will be used to automatically lower the level of abstraction in a specified design.

The specification of a design, its implementation, and its structural properties will be obtained from a single source - an Occam program. A designer will implement the behaviour of a system as an algorithmic description written in the language. This can then be compiled and executed as a simulator, or prototype, of the system. When satisfied that it is a satisfactory description the program will be passed to ATLAST to generate a VLSI implementation. Its decomposition into a layout description will involve two stages. Firstly, the transformation module in ATLAST will convert the Occam program into an optimised structural/behavioural intermediate representation (IR). This IR will also be an Occam program. The completed version of COPTS will then generate a schematic representation of the IR, providing the designer with feedback on the relative placement and interconnection of the function blocks described in the IR.

In the second stage, a design sub-system will be responsible for transforming the intermediate Occam description into the appropriate VLSI circuits. A number of different architectural implementations can be derived from such a description. For example, a description could be transformed into a finite state machine description and then implemented by a standard data path architecture. Varies degrees of concurrency and redundancy in replicated processes could be removed by introducing multiplexing on channels and sequential iterative behaviour. The resulting description could then be implemented by a bit serial architecture. The concurrency and communication of a description could be mapped into a systolic architecture. Macro-cells corresponding to software primitives (+,-,*,/,?,! etc.) could be substituted for the occurrence of these primitives in the description thereby giving a cell architecture. Alternatively, different sections of the description could be implemented by different architectures giving a multi-architecture system. It is envisaged that the design sub-system will contain a suite of silicon compilers for the various target architectures. The choice of architecture will be user defined. A designer will therefore be able to explore alternative architectures simply and quickly.

The selected compiler will decompose the IR into a geometrical layout of the system. A set of design files defining the complete layout will be generated. Each file will hold a graphic description in the form of a machine-readable representation - a layout language (e.g. CIF). These descriptions will be used to derive other forms for layout plotters and pattern generators.

The IR will also serve as the input specification language to a simulator. Simulation of a design produces information on the behaviour of the circuit network intended to implement the design. It provides the designer with feedback on the performance of the system. This may encourage design alternatives which improve performance to be examined. Simulation is carried out at several levels of abstraction, each level yielding information on different aspects of the design. The information obtained is used to confirm physical functioning and make critical design decisions. Delays, critical paths, and design errors can be detected through simulation. Additionally, estimates on the speed of the circuit and its power consumption are obtained.

Implementing the entire ATLAST design system is no small undertaking, as each of the utility tools outlined above are themselves complex systems. One of the major problems to be tackled will be the definition of the IR, which serves as an interface between the design sub-system and the transformation module. Its form is determined by the information extracted from it by each tool for its particular application. A complete definition is only possible after the input requirements of the units have been considered. As is usually the case, these requirements are not fully unveiled until the problems of implementing each tool are tackled. As much use as possible should be made of existing tools and approaches. Powerful workstations are available [69,71] which incorporate many of the features needed in the design sub-system. One approach to obtaining this sub-system would be to integrate available structural silicon compilers with an appropriate workstation.

Automatic generation of high level schematic diagrams from programs written in a subset of Occam has been presented as the first steps toward the realisation of the system. Much work still needs to be done, in particular, as stated in section 8.3 a route from these diagrams to working prototype implementations is required. One practical approach to developing such a route would be to take advantage of Occam's association with the transputer. Occam is the native language of the transputer. A transputer is a computer on a chip and consists of a processor coupled to an on chip RAM and four links for communication with other transputers and transputer devices. The processor is capable of 10 million instructions/sec. and provides efficient support for the Occam model of concurrency and communication. The schematic diagrams generated by COPTS could be implemented as transputer based systems.

Transputer based systems would enable real time experimentation. Critical sections of code in a design description could be identified. The schematics corresponding to these sections could be used as the structural specifications for one or more dedicated hardware devices, or "code accelerators". The approach of substituting hardware primitives (i.e. macro-cells) for the schematic elements could then be used to obtain these accelerators. An accelerator could be linked to the appropriate transputer via one of its four links. In such a configuration, the accelerator can be regarded as a process executing concurrently with the process on the transputer. Synchronised communication between the two processes could be implemented by the transputer's link. The code harnessing the critical sections would be used to program the supporting transputers. Again the schematics could help in determining the allocation of processes among the constituent

transputers. This approach is not the system on a chip solution intended for ATLAST. However, it is quickly achievable, and could form an interim solution while an efficient algorithm to layout route is developed.

A key feature of Occam which should be exploited in future work on a system such as ATLAST is its semantic properties. The language's formal basis together with its ability to conveniently express concurrent problems, emphasise its suitability for a VLSI design role. Occam is directly based on Hoare's mathematical notation of Communicating Sequential Processes [29]. This has two important and interesting consequences. Firstly, it enables behavioural specifications written in Occam to be formally verified and validated. As the complexity of implemented VLSI systems expands this will become an essential requirement of design. Secondly, techniques for formally validating the equivalence between high level and low level Occam representations can be developed. Such techniques will enable semantic preserving transformations to be explored, in particular, transformations to 'push' high level behavioural specifications toward equivalent detailed low level implementations. Making these an intrinsic part of the route to layout removes the necessity for simulation and verification since they guarantee 'correctness'. Future work should concentrate on developing software to support these transformations. Current research into an automated system for semantic preserving transformation of an Occam system description into either custom silicon or microcode for a processor [2] may be able to make a significant contribution to this work.

To sum up, application specific chips will play a significant role in fifth generation computers. Automated algorithm to layout systems will enable a wide variety of such chips to be implemented. The concurrent programming language Occam is well suited to this approach since it is capable of accurately describing their behaviour in a manner which reflects their implementation. The language is also well suited for providing system level specifications for these components. In addition, Occam can be used to describe their overall integration into the systems envisaged for the next generation of computers.

APPENDIX A

REFERENCES

1. Adshead G., "AI Techniques in VLSI design", Silicon Design, VOL 2 , No. 9, Sept 1985, pp10-11,16.

2. Alvey CAD015 Project, "Development of Advanced System Description Language Transformation and Verification Tools", 1985.

3. Aryes R.F., "VLSI silicon compilation and the art of automatic microchip design", Prentice-Hall, Inc., 1983.

4. Barron I., et al, "Transputer does 5 or more MIPS even when not used in parallel", Electronics, VOL 56 no. 23, Nov 1983 pp109-115.

5. Barton E.E., "A non-numeric design methodology for VLSI", VLSI 81, Ed J.P. Gray, Academic Press, 1981, pp25-34.

6. Bergmann N., "A Case Study Of the F.I.R.S.T. Silicon Compiler", Third Caltech Conference On VLSI, Ed R. Bryant,1983, pp413-430.

7. Brown H., Tong C., & Foyster G., "Palladio: An Exploratory Environment for Circuit Design", IEEE Computer Magazine, VOL 16(12), Dec. 1983, pp41-55.

8. Campbell R.H., Koelmans A.M., & McLauchlan M.R., "STRICT: a design language for strongly typed recursive integrated circuits", IEE PROCEEDINGS 132(2), March/April 1985, pp108-115.

9. Clocksin W.F., "Logic Programming and the Specification of Circuits", University of Cambridge, Computer Laboratory, Technical Report No 72, 1984.

10. Clocksin W.F., and Mellish C.S., "Programming in PROLOG", Springer-Verlag, 1981.

11. Collins B. & Gray A., "The INMOS Hardware Description Language And Interactive Simulator", VLSI 81 Ed. J.P. Gray, London, New York: Academic Press, 1981, pp107-116.

12. Denyer P., & Renshaw D., "VLSI Signal Processing: A Bit-Serial Approach", Addison-Wesley Publishing Co., 1985.

13. Dijkstra E.W., "The Humble Programmer", Communications of the ACM, vol 15(10), 1972, pp859-866.

14. Enderle G., Kansay K., and Pfaff G., "Computer Graphics Programming: GKS - The Graphics Standard", Springer-Verlag, 1984.

15. Evanzuk S., "Silicon Compilers: No Automatic Route To Acceptance", VLSI Systems Design, Vol VI No 10, Nov 1985, pp42-44.

16. Foster M.J., and Kung H.T. "The Design of Special-Purpose VLSI Chips", IEEE Computer Magazine, Jan 1980, pp26-40.

17. Fuchs H. & Abram G., "VLSI Architecture for Computer Graphics" presented at NATO Advanced Study Institute On Microarchitecture of VLSI Computers, Urbino, Italy. July 84.

18. Gajski D.D, Kuhn R.H., "New VLSI Tools", Computer Magazine, VOL 16 No 12, Dec 83, pp11-14.

19. German S.M., & Lieberherr K.J., "ZEUS: A Language for Expressing Algorithms in Hardware", Computer Magazine, Vol 18 No 2, Feb 1985, pp55-65.

20. Gouldberg A., and Robson D., "Smalltalk-80 The Language and its Implementation", Addison-Wesley Publishing Company, 1983.

21. Gouveia Lima I., "Programming Decentralised Computers", Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, 1984.

22. Gray J.P., Buchanan I., & Robertson P.S., "Controlling VLSI Complexity Using A High-Level Language For Design Description", Proceedings of the IEEE Conference on Computer Design: VLSI in Computers, 1983.

23. Gray J.P., Buchanan I., & Robertson P.S., "Designing Gata Arrays Using a Silicon Compiler", ACM IEEE 19 th Design Automation Conference Proceedings, June 1982, pp377-383.

24. Gries D., "Compiler Construction for Digital Computers", John Wiley & Sons, 1971.

25. Hayes-Roth F., "The Knowledge-Based Expert System: A Tutorial", Computer Magazine, Vol 17 No. 9, Sept 1984, pp11-28.

26. Hayes-Roth F., "Knowledge-Based Expert Systems", Computer Magazine, Vol 17 No 10, Oct 1984, pp263-273.

27. Henderson P., "Functional Programming Application and Implementation", Prentice-Hall International, 1980.

28. Hirschhorn S. & Davis A.M., "The Revolution In VLSI Design: Parallels Between Software and VLSI Engineering", IEEE Workshop Report: VLSI and Software Engineering, Oct. 1982, pp75-83.

29. Hoare C.A.R., "Communicating Sequential Processes" , Prentice/Hall International, 1985.

30. Hortstmann P.W., "Expert Systems and Logic Programming for CAD", VLSI Design Magazine, Nov 1983, pp37-46.

31. Inmos, "Occam Programming Manual", Prentice-Hall International, 1984.

32. Jackson M A, "Principles Of Program Design", New York Academic Press, 1975.

33. Jensen K., and Wirth N., "PASCAL User Manual and Report", 2nd. Edition, Springer-Verlag, 1978.

34. Johannsen D., "BRISTLE Blocks: A Silicon Compiler", Proc. of 16 th. Design Automation Conference, 1979, pp310-313.

35. Johnson S.C., "VLSI circuit design reaches the level of architectural description", Electronics Magazine, May 1984, pp121-128.

36. Kowalski T.J., Geiger D.J., Wolf W.H., & Fichtner W., "The VLSI Design Automation Assistant: From Algorithms to Silicon", IEEE Design & Test of Computers, August 1985, pp33-42.

37. Kung H.T., "Let's Design Algorithms for VLSI Systems", Proc. Caltech Conf. on Very Large Scale Integration, California Institute of Technology, Jan 1979.

38. Kung H.T., "Programmable Systolic Chip", presented at NATO Advanced Study Institute On Microarchitecture of VLSI Computers, Urbino, Italy, July 84.

39. Kung H.T., "Putting Inner Lopps Automatically In Silicon", Lecture Notes in Computer Science, 163, VLSI Engineering Beyond Software Engineering, Ed. T.L. Kunnii, Spinger-Verlag, 1984, pp70-104.

40. Kung H.T. & Yu S.Q., "Integrating High-Performance Special Purpose Devices Into A System", VLSI Architecture, Ed. B. Randell & P.C. Trelevan, Prentice/Hall International 1983, pp205-211.

41. Lea R.M., "SCAPE: A VLSI CHIP ARCHITECTURE FOR IMAGE PROCESSING", presented at NATO Advanced Study Institute On Microarchitecture of VLSI Computers, Urbino, Italy, July 84.

42. Lieberherr K.J., "Toward a Standard Hardware Description Language", IEEE Design & Test of Computers, Vol 2 No 1, Feb 85, pp55-62.

43. Mano T., et al., "Occam To CMOS: Experimental Logic Design Support System", ICOT Technical Report: TR-093, December 1984.

44. Marques J., & Cunha A., "Clocking of VLSI circuits", VLSI Architecture, Ed B. Randell & P.C. Treleaven, Prentice Hall International, 1983, pp165-178.

45. May D., "Occam", SIGPLAN Notices, Vol 18 No 4, April 1983, pp69-79.

46. May D., "Occam", IEE Colloquim on "Software tools for hardware design", Digest No 1983/98, 1983.

47. Mead C., and Conway L., "Introduction to VLSI systems", Addison-Wesley, 1980.

48. Mitchell T.M., Steinberg L.I, and Sulman S., "A Knowledge-Based Approach to Design", IEEE Transactions On Pattern Analysis And Machine Intelligence, Vol. PAMI-7, No. 5, Sept 1985.

49. Morison J.D., Peeling N.E., & Thorp T.L., "The Design Rationale of ELLA, a Hardware Design and Description Language", Presented at 7 th International Conference on Computer Hardware Description Languages, Tokyo, Japan, August 1985.

50. Piloty R., Borrione D., "The Conlan Project: Concepts , Implementations, and Applications", Computer Magazine, Vol 18 No 2, Feb 1985, pp81-92.

51. Robinson P., & Dion J., "Programming Languages For Hardware Description", ACM IEEE 20 th., Design Automation, 1983, pp12-16.

52. Russel G., Kinniment D.J., Chester E.G., & McLauchlan M.R., "CAD for VLSI", Van Nostrand Reinhold (UK), 1985.

53. Seitz, C.L., "Self Timed VLSI Systems", Proc. Caltech Conf. On VLSI, Jan 1979, pp345-255.

54. Sequin C.H., "Managing VLSI Complexity: An Outlok", Proceddings of the IEEE, Vol 71 No 1, Jan 1983, pp149-166.

55. Shahdad M., et al, "VHSIC Hardware Description Language", IEEE Computer Magazine, Feb 1985, pp94-102.

56. Shiva S.G., Klon P.F., "The VHSIC Hardware Description Language", VLSI Design, Vol VI No 6, June 1985, pp86-106.

57. Shrobe H.E., "The Data Path Generator", In Digest Of Papers Spring Compcon 82: High Technology In The Information Industry. San Fransico, CA, USA 22-25 Feb. pp340-4 1982.

58. Smith C.U., & Dallen J.A., "Future Directions For VLSI and Software Engineering", In: Lecture Notes In Computer Science (No 163), "VLSI Enginnering Beyond Software Engineering", Ed T.L. Kunni, Springer-Verlag, 1984 pp2-20.

59. Southard J.R., "MacPitts: An Approach to Silicon Compilation", IEEE Computer Magazine VOL 16(12), Dec 1983, pp74-82.

60. Stefik M., et al, "The Partioning Of Concerns In Digital System Design", Conference on Advanced Research in VLSI Proceedings, Ed., P. Penfield, Jan 1982.

61. Steinberg L.I., Mitchell T.M., "A Knowledge Based Approach To VLSI CAD - The REDESIGN System", ACM IEEE 21 st Design Automation Conference Proceedings, 1984, pp412-418.

62. Taylor R., "Signal processing with occam and the transputer", IEE Proceeding, Vol 131, Pt.F, No 6, Oct 1984 pp610-614.

63. Taylor R., Wilson P., "Occam", Electronics, Nov 1982, pp89-95.

64. Thomas D.E., et al, "Automatic Data Path Synthesis", Computer Magazine, VOL 16 No 12, Dec 83, pp59-70.

65. Treleaven P.C. & Gouveia Lima I., "Future Computers: Logic, Data Flow, ..., Control Flow?", IEEE Computer Magazine 17(3), March 84 pp47-59.

66. Trickey H., "Compiling Pascal programs into silicon", Ph.D Thesis, Stanford University, July 1985. (STAN-CS-85-1059).

67. Uehara T., "A Knowledge-Based Logic Design System", IEEE Design & Test of Computers, Vol 2 No 5, Oct 1985, pp27-34.

68. Ullman J.D., "Computational Aspects of VLSI", Computer Science Press, 1984

69. VAX-11 PASCAL Language Reference Manual.

70. VLSI Design Staff, "A Perspective On CAE Workstations", VLSI Design, VOLUME VI, No. 4 April 1985, pp52-77.

71. VLSI Design Staff, "Silicon Compilers: Part 1, Drawing a Blank", VLSI Design Magazine, Sept 1984, pp 54-58.

72. VLSI Systems Design Staff, "Survey OF IC Layout CAD Systems", VOL VI, No. 9, Sept 1985, pp45-54.

73. VLSI Design Tools, Dept. of Electrical Engineering, Newcastle University 1983.

74. Werner J., "The Silicon Compiler: Panacea, Wishful Thinking Or Old Hat?", VLSI Design Magazine 3(5), Sept/Oct 1982.

75. Werner J., "Progress Toward The 'Ideal' Silicon Compiler", VLSI Design Magazine 4(5), Sept 83.

76. Weste N. Eshraghian K., "Principles of CMOS VLSI Design, A Systems Perspective", Addison-Wesley Publishing Company, 1985, pp297-302.

77. Wirth N., "Program Development by Stepwise Refinement", Communications of the ACM 14(4) April 1971, pp 221-227.

78. Wolf W., Newkirk J., Mathews R., & Dutton R., "DUMBO, A Schematic-To-Layout Compiler", Third Caltech Conference on VLSI, Ed R. Bryant, 1984, p279-393.

79. Yakovlev A., "Designing Self Timed Systems", VLSI Systems Design, September 85, Vol VI No 9, pp70-91.

APPENDIX B

THE PATTERN MATCHER IMPLEMENTATIONS

B.1 PASCAL

```
CONST
     Ncells = 4;
TYPE
    BIT = 0..1;
    Comparator = RECORD
                       String, Pattern : CHAR;
                 END;
    Accumulator = RECORD
                        EndBit,WCC,TempRes,Result : BIT;
                  END;
    PatternCells = RECORD
                         AccCell : Accumulator;
                         CompCell : Comparator;
                   END;

VAR
   Modules : ARRAY[1..Ncells] OF PatternCells;
   Beat : BIT;
   i : 1..Ncells;

FUNCTION Compare(Cell : Comparator) : BIT;
BEGIN
     WITH Cell DO
       IF Pattern = String THEN
          Compare := 1
       ELSE
          Compare := 0
END;

PROCEDURE Move (VAR left, right : PatternCells);
BEGIN
     right.CompCell.Pattern := left.CompCell.Pattern;
     left.CompCell.String := right.CompCell.String;
     right.AccCell.WCC := left.AccCell.WCC
     right.AccCell.EndBit := left.AccCell.EndBit;
     left.AccCell.Result := right.AccCell.Result;
```

```
END;

PROCEDURE Accumulate(DataIn : BIT; VAR Cell : Accumulator);
BEGIN
     WITH Cell DO
       BEGIN
           IF NOT ((TempRes=1) AND ((WCC=1) OR (DataIn=1)) THEN
              TempRes := 0;
           IF EndBit = 1 THEN
           BEGIN
                Result := TempRes;
                TempRes := 1;
           END
       END
END;


(* Main Body Of Program *)
BEGIN
     Beat := 0;
     WHILE NOT finished DO
     BEGIN
          InOut;
          FOR i := 1 TO (Ncells DIV 2) DO
          BEGIN
               WITH Module[(i*2)-Beat] DO
                 Accumulate(Compare(Compcell), AccCell);
               Move(Module[(i*2)-Beat], Module[((i*2)-1)-Beat]);
               Beat := 1 - Beat;
          END;
     END;
END.
```

B.2 OCCAM

```
DEF Ncells = 4:
CHAN Sys.Bus:
CHAN pattern[Ncells], string[Ncells], data[Ncells-1]:
CHAN end[Ncells], wild[Ncells], result[Ncells]:

PROC Comparator(CHAN PatrnIn, StrngIn, PatrnOut, StrngOut, DataOut)=
  VAR patrn, strng:
  SEQ
    PAR
      patrn := 0
      strng := 0
    WHILE TRUE
      SEQ
        PAR
          PatrnOut ! patrn
          StrngOut ! strng
        PAR
          PatrnIn ? patrn
          StrngIn ? strng
        DataOut ! patrn = strng :

PROC Accumulator(CHAN WildBitIn, EndBitIn, ResIn, DataIn, WildBitOut,
                      EndBitOut, ResOut)=
  VAR CompRes, WildBit, EndBit, Result, PartialRes:
  SEQ
    PAR
      WildBit := FALSE
      EndBit := FALSE
      Result := FALSE
      PartialRes := TRUE
    WHILE TRUE
      SEQ
        PAR
          WildBitOut ! WildBit
          EndBitOut ! EndBit
          ResOut ! Result
        PAR
          WildBitIn ? WildBit
          EndBitIn ? EndBit
          ResIn ? Result
          DataIn ? CompRes
        PartialRes := PartialRes /\ (WildBit \/ CompRes)
        IF
          EndBit = TRUE
            SEQ
              Result := PartialRes
              PartialRes := TRUE :
```

```
PROC GetChar(CHAN BusIn,PatternIn,StringIn)=
  VAR Ch, Beat:
  SEQ
    Beat := 0
    WHILE TRUE
      SEQ
        PAR
          Beat := 1 - Beat
          BusIn ? Ch
        IF
          Beat = 0
            PatternIn ! Ch
          Beat = 1
            StringIn ! Ch :

PAR
  GetChar(Sys.Bus,pattern[0],string[Ncells-1])
  PAR i =[ 1 FOR Ncells-1 ]
    PAR
      Comparator(pattern[i-1],string[Ncells-i],pattern[i],
                         string[(Ncells+1)-i],data[i-1])
      Accumulator(wild[i-1],end[i-1],result[Ncells-i],data[i-1],
                           wild[i],end[i],result[(Ncells+1)-i])
```

B.3 SMALLTALK

```
Class   Comparator

Methods
   Compare: stringChar and: patternChar
     stringChar = patternChar
       ifTrue:[^1]
       ifFalse:[^0]
```

```
Class   Accumulator

Super Class   Object

Instance Variables   accRes

Methods
   initialise
     ^super new initaccRes

   update: dataIn with: resIn and: bitsIn
     | resOut |
     accRes <- dataIn & (bitsIn at: 1 | accRes).
     BitsIn at: 2 = 1
       ifTrue:[resOut <- accRes.
               self initaccRes]
       ifFalse:[resOut <- resIn]
     ^resOut

   initaccRes
     accRes <- 1
```

```
Class   PatternBlocks

Indexed Instance Variables   blocks

Methods
   Initialise: nblocks | i |
     blocks <- array new: nblocks.
     i <- 1.
     [i <= nblocks]
       whileTrue:[blocks at: 1 put: Accumulator initialise.
                  i <- i + 1]
```

Class BitStreams

Indexed Instance Variables bits

Methods

```
   initialise: N | i |
     bits <- array new: N.
     i <- 1.
     [i <= N]
       whileTrue:[bits at: 1 put: #(0 0).
                  i <- i + 1]
```

Class PatternMatcher

Methods

```
go: nblockks fromhost: charstream tohost: bitstream
   |pattern, string, result, bits, cells, toggle|
   pattern <- array new: nblocks.
   string <- array new: nblocks.
   result <- array new: nblocks.
   bits <- bitStreams initialise: nblocks.
   cells <- patternBlocks initialise: nblocks.
   toggle <- 0.
   [charstream isEmpty]
    whileFalse:[i|
      "input string or pattern character"
      i <- toggle + 1.
      [i <= nblocks]
       whileTrue:
         [results at: i put:
            cells at:i (update:(Compare:(string at:i) and:(pattern at:i))
                       with: result at:(i+1)
                       and: bits at:i).
          pattern at:(i+1) put:(pattern at:i).
          string at:(i-1) put: (string at:i).
          bits at:(i+1) put:(bits at:i).
          i <- 1 + 2].
      toggle <- toggle - 1]
```

B.4 LISPKIT-LISP

```
clock(beat, p, s, m) ->
      if eq(beat,0) then
         {cons(car(l), clock(1, p, cdr(s), cdr(l))
          where l = move1(car(s), matchodd(m))}
      else
         cons(nil, clock(0, cdr(p), s,
                   move2(car(p), car(m), matcheven(cdr(m)))))

move1(s_char, m)  ->
      cons(car(car(m)),
           to_even(s_char, car(m), car(cdr(m)), cdr(cdr(m))))

move2(p, head, tail) ->
      cons(left(p, head, car(tail)),
           to_odd(car(tail), car(cdr(tail)), cdr(cdr(tail))))

to_even(s, l, c, r) ->
       if eq(nil, r) then
          cons(l, right(s, l, c))
       else
          cons(l, cons(left_right(l, c, car(r)),
                       to_even(s, car(r), car(cdr(r)), cdr(cdr(r)))))

to_odd(l, c, r) ->
       if eq(cdr(r), nil) then
          cons(l, cons(left_right(l, c, car(r)))
       else
          cons(l, cons(left_right(l, c, car(r)),
                       to_odd(car(r), car(cdr(r)), cdr(cdr(r)))))

matcheven(m) ->
          if eq(cdr(m), nil) then
             pmatch(car(m))
          else
             cons(pmatch(car(m)), cons(car(cdr(m)),
                                       matcheven(cdr(cdr(m)))))

matchodd(m) ->
       if eq(cdr(cdr(m)),nil) then
          cons(pmatch(car(m)), cdr(cdr(m))
       else
          cons(pmatch(car(m)), cons(car(cdr(m)), matchodd(cdr(cdr(m)))))

pmatch(inpts) ->
       cons(a1, cons(a2, cons(c1, cons(cons(a3,a4), c3))))
       where a1 = e(1,a),  a2 = e(2,a),  a3 = e(3,a),  a4 = e(4,a),
             c1 = e(1,c),  c2 = e(2,c),  c3 = e(3,c),
             p = e(3,inpts),  s = e(4,inpts),   t = e(2,inpts),
             r = e(1,inpts),
             c = comp(car(p),s),
             a = acc(t, r, car(cdr(p)), car(cdr(cdr(p))), c3)
```

```
comp(p,s) ->
     if eq(p,s) then
        cons(p, cons(s,1))
     else
        cons(p, cons(s,0))

acc(a,r,x,l,d) ->
    if eq(l,1) then
       cons(u, cons(1, cons(x,l)))
    else
       cons(r, cons(u, cons(x,l)))
    where u = and(a,or(x,d))
```

B.5 PROLOG

```
pipeline([P_ch|Pattern], [S_ch|Strng], Res, Blocks):-
        matchodd(Blocks, Blocks_1a),
        transfereven(S_ch, Rout, Blocks_1a, Blocks_1b),
        matcheven(Blocks_1b , Blocks_2a),
        transferodd(P_ch, Blocks_2a, Blocks_2b)
        pipeline(Pattern, Strng, [Rout|Res], Blocks_2b).

matchodd([Ablock, Lblock], [Nblock, Lblock]):-
         process(Ablock, Nblock).

matchodd([Lblock|[Rblock|Tailblks]], [Nblock|[Rblock|Rest]]):-
         process(Lblock, Nblock),
         matchodd(Tailblks, Rest).

matcheven([Blk_j,Blk_k], [Blk_j,Nblk]):-
          process(Blk_k, Nblk).

matcheven([Lblk|[Rblk|Tailblks]], [Lblk|[Nblk|Rest]]):-
          process(Rblk, Nblk),
          match(even, Tailblks, Rest).

transfereven(S_in, [Blk_j,Blk_k], [Blk_j,Ublk]]):-
             right(S_in, Blk_j, Blk_k, Ublk).

transfereven(Sin, [Blk_i|[Blk_j|[Blk_k|Rest]]], [Blk_i|[Ublk|Tailblks]]):-
             exchange(Blk_i, Blk_j, Blk_k, Ublk),
             transfer(Sin, [Blk_k|Rest], Tailblks).

transfereven(Schar, Res, [Blk1|Tail], Nblks):-
             out(Blk1, Res),
             transfer(Schar, [Blk1|Tail], Nblks).

transfer([Blk_i,Blk_j,Blk_k], [Blk_i,Ublk,Blk_k]):-
         exchange(Blk_i, Blk_j, Blk_k, Ublk).

transfer([Blk_i|[Blk_j|[Blk_k|Tail]]], [Blk_i|[Ublk|Rest]]):-
         exchange(Blk_i, Blk_j, Blk_k, Ublk),
         transfer([Blk_k|Tail], Rest).

transferodd(Pin, [Blk_1|[Blk_2|Tail]], [Ublk|Rest]):-
            left(Pin, Blk_1, Blk_2, Ublk),
            transfer([Blk_2|Tail], Rest).

exchange([_,_,P1,_], [_,T2,_,_], [R3,_,_,S3], [R3,T2,P1,S3]).

left(P_ch, [_,T1,_,_], [R2,_,_,S2], [R2,T1,P_ch,S2]).

right(S_ch, [_,_,P1,_], [_,T2,_,_], [R,T2,P1,S_ch]):-
     R is 1.
```

```
process([Rin, Temp, [Pin,Xin,Lin], Sin],
                [Rout, Temp2, [Pout,Xout,Lout], Sout]):-
        comp(Pin, Sin, Cout, Pout, Sout),
        acc(Rin, Temp, Xin, Lin, Cout, Rout, Temp2, Xout, Lout).

comp(X, X, C, X, X):-
     C is 1.

comp(X, Y, C, X, Y):-
     C is 0.

acc(R1, T1, X1, L1, Data, R2, T2, X1, L1):-
    andor(X1, T1, Data, C),
    update(L1, C, R1, T1, R2, T2).

update(1, C, _, _, C, T):-
      T is 1.

update(0, C, R1, T1, R1, C).
```

APPENDIX C

THE SYNTAX OF THE OCCAM SUBSET

The syntax of the occam subset is described in Backus-Naur-Form. Actual language symbols and keywords are not surrounded by <>. The ::= symbol is used to define a syntactic category. The handle for the category is given on the left of the symbol, and the valid syntactic forms on the right. Where there are several valid forms of one category, they are separated by the symbol |.

An item between {} indicates that it may be repeated zero or more times.

An item between [] indicates that the item is optional.

```
<program>       ::=  <process>

<process>       ::=  <primitive>
                   | <construct>
                   | <declaration> <qbd>:<qnl>  <process>

<primitive>     ::=  <assignment>
                   | <input>
                   | <output>

<construct>     ::=  SEQ {  <process> }
                   | PAR {  <process> }

<declaration>   ::=  VAR <var> { , <var> }
                   | CHAN <chan> { , <chan> }
                   | DEF <const.def> { , <const.def> }
```

```
<assignment>     ::=   <var> := <expression>

<input>          ::=   <chan> ? <var> { ; <var> }

<output>         ::=   <chan> ! <expression> { <; <expression> }

<var>            ::=   identifier.string

<chan>           ::=   identifier.string

<const.def>      ::=   identifier  = <expression>

<expression>     ::=   <element> {  <assoc.op>  <element> }
                   |   <element> [  <arithmetic.op>  <element>]
                   |   <monadic.op>  <element>

<element>        ::=   number |  <var> | (  <expression> )

<assoc.op>       ::=   + | *

<arithmetic.op>  ::=   + | - | * | /
```

APPENDIX D

MORE EXAMPLE SOURCE PROGRAMS

D.1 PROGRAM FOUR

```
CHAN input1, input2, input3, intrn1, intrn2, output1, output2, output3:
PAR
  VAR a, b, c, t1, t2, t3:
  SEQ
    PAR
      input1 ? a
      input2 ? b
      input3 ? c
    PAR
      t1 := a * a
      t2 := b * b
      t3 := c * c
    PAR
      intrn1 ! (t1*t2*t3) + (a-(b+c))
      intrn2 ! (t1/(t2+t3)) + a + b + c
      VAR x,y:
      SEQ
        PAR
          x:=(a+b)+(b+c)+(t1+t2)+(a+c)+(a-b)+(c-a)+(c-b)+(a/b)+(b/c)+(a*b)
          y:=((c-(((b+c)-a)/((a-(b/c))*a*b)))/((a-b)+c))+(c-b)+(b/c)+(a*b)
        output1 ! (t3-(t1*t2)) + ((a*b)+c) + (x*x*x) + (y*y*y)
  VAR a, b:
  SEQ
    PAR
      intrn1 ? a
      intrn2 ? b
    CHAN Comm1, Comm2:
    PAR
      VAR t1, t2:
      SEQ
        PAR
          t1 := a * a
          t2 := b * b
        PAR
          output2 ! t1+(a*b)+t2
          Comm1! t1
```

```
    Comm2! t2
VAR asq, bsq:
SEQ
  PAR
    Comm1? asq
    Comm2? bsq
  output3 ! (asq*b) - ((a/b) - (bsq*a))
```

D.2 PROGRAM FIVE

```
CHAN input1,input2,input3,output1,output2,output3,intrnal1,intrnal2:
PAR
  VAR a, b, c, t1, t2, t3:
  SEQ
    PAR
      input1 ? a
      input2 ? b
    c := (a*a) + (b*b)
    CHAN intrn:
    PAR
      t1 := a + (b * c)
      t2 := a - ((b+c)*(b/(a+c)))
      VAR temp1, temp2:
      SEQ
        PAR
          temp1 := a + (b - c)
          temp2 := c * (b + a)
        intrn ! (temp1*temp2) + (temp1/temp2)
      VAR M,N:
      SEQ
        PAR
          intrn ? M
          N := a*b*c
        t3 := (N*(M-a)) + (N-b)
    PAR
      a := a * a
      b := b * b
      c := c * c
    PAR
      intrnal1 ! (t1*t2*t3) + (a-(b+c))
      intrnal2 ! (t1/(t2+t3)) + a + b + c
      VAR x:
      SEQ
        x:=(a*b)+(b*c)+(t1-t2)+(a/c)+(a-b)+(c-a)+(c-b)+(a/b)+(b/c)+(a*b)
        output1 ! (t3-(t1*t2)) + ((a*b)+c) + (x*x*x)
  VAR a, b, c:
  SEQ
    PAR
      intrnal1 ? a
      intrnal2 ? b
      input3 ? c
    CHAN Comm1, Comm2:
    PAR
      VAR t1, t2:
      SEQ
        PAR
          t1 := (a * a)/c
          t2 := (b * b)/c
        PAR
          output2 ! t1+(a*b)+t2
          Comm1! t1
          Comm2! t2
      VAR asq, bsq:
      SEQ
```

```
PAR
  Comm1? asq
  Comm2? bsq
output3 ! (asq*b) - ((a/b) - (bsq*a))
```

D.3 PROGRAM SIX

```
CHAN input1,input2,input3,output1,output2,output3,intrnal1,intrnal2:
PAR
  VAR a, b, c:
  SEQ
    PAR
      intrnal1 ? a
      intrnal2 ? b
      input1 ? c
    CHAN Comm1, Comm2:
    PAR
      VAR t1, t2:
      SEQ
        PAR
          t1 := (a * a)/c
          t2 := (b * b)/c
        PAR
          output1 ! t1+(a*b)+t2
          Comm1! t1
          Comm2! t2
      VAR asq, bsq:
      SEQ
        PAR
          Comm1? asq
          Comm2? bsq
        output2 ! (asq*b) - ((a/b) - (bsq*a))
  VAR a, b, c, t1, t2, t3:
  SEQ
    PAR
      input2 ? a
      input3 ? b
    c := (a*a) + (b*b)
    CHAN intrn:
    PAR
      t1 := a + (b * c)
      t2 := a - ((b+c)*(b/(a+c)))
      VAR temp1, temp2:
      SEQ
        PAR
          temp1 := a + (b - c)
          temp2 := c * (b + a)
        intrn ! (temp1*temp2) + (temp1/temp2)
      VAR M,N:
      SEQ
        PAR
          intrn ? M
          N := a*b*c
        t3 := (N*(M-a)) + (N-b)
    PAR
      a := a * a
      b := b * b
      c := c * c
    PAR
      intrnal1 ! (t1*t2*t3) + (a-(b+c))
      intrnal2 ! (t1/(t2+t3)) + a + b + c
```

```
VAR x:
SEQ
  x:=(a*b)+(b*c)+(t1-t2)+(a/c)+(a-b)+(c-a)+(c-b)+(a/b)+(b/c)+(a*b)
  output3 ! (t3-(t1*t2)) + ((a*b)+c) + (x*x*x)
```

APPENDIX E

PROGRAM DOCUMENTATION

This appendix describes the Pascal program developed for COPTS. It is divided into three sections.

The first section provides an introduction to the program and considers the calling sequence for the first level procedures, while the second section summarises the purpose of these procedures. The final section discusses the output produced by the program.

E.1 INTRODUCTION

The compiler program was written in VAX-11 Pascal V2.2-114 [69]. The code for the compiler is divided into five MODULES. Each module contains the set of procedures and functions used to implement one or more parts of the compiler. An ENVIRONMENT file holding the outermost level of definitions of constants, types, variables, and procedures is generated by the main program. The VAX Pascal INHERIT attribute enables each module to use these definitions.

```
[ENVIRONMENT('SETUP')]
PROGRAM DEFINITIONS(INPUT,OUTPUT);
.
.
.

(* main body of program DEFINITIONS *)
BEGIN
     INITIALISE;
     BUILDPARSETREE(MajorPrc);
     IF NOT ErrFlag THEN
        BEGIN
             InitBoxCell(MajorPrc);
             PrcSpec(MajorPrc);
             Dump(MajorPrc);
             PrcImplmntn(MajorPrc);
        END
END.
```

FIGURE E1: The Execution Sequence

The main body of the Pascal program is listed in Figure E1. The variable 'MajorPrc' is a pointer to the root record of the internal data tree. 'ErrFlag' is a global boolean flag used to indicate if there are any syntax errors in the Occam source code.

Firstly, the program calls the procedure 'INITIALISE' to initialise all of its global variables. 'BUILDPARSETREE' then reads in the source program and attempts to parse it. If syntax errors were found in the source code then the global variable 'ErrFlag' is set to false and the program terminates. Otherwise, 'ErrFlag' is set to true and the internal form of the program is completed by 'PrcSpec'. This internal form is then written out by 'Dump'. Calling this procedure is not essential since it merely allows the user to check the internal format. Finally, 'PrcImplmntn' generates the graphical definition of the source code. This procedure also writes out the object code defining the representation.

E.2 BuildParseTree

Initialises 'MajPrc' and store information obtained from parsing the associated process.

E.2.1 Variable Parameters

1. MajorPrc :

   Pointer to the 'CmpntRec' associated with a process.

E.2.2 Description

Scans and analyses the Occam source program and corresponds to the parse phase. The scanning algorithm is adapted from the Scanner presented in Gries [24]. The recursive descent algorithm implemented by the analyser is based on the grammar defined in Appendix C. The procedure also carries out syntax checking and some semantic checking. Limited error messages are produced on error detection. A syntax error causes the analyser to fail, since no error recovery scheme has been implemented.

The scanner acts upon a whole line of the source text. The tokens obtained from a scan of the current text line are held in a buffer. The analyser fetches tokens from this buffer one at a time. When the buffer is empty the next line of the Occam source program is read in. This line is scanned and the tokens, representing the identified symbols, placed in the buffer. The sequence of tokens in the buffer after the scan corresponds to the sequence of symbols in the text line.

The analyser uses the tokens to build up the internal data tree. This requires creating and initialising 'CmpntRec' records. Each record created is associated with a process in the Occam program and identifies whether that process is simple or complex. The level of one of these records in the internal data tree reflects the nesting of its process in the source program. The root record of the tree is associated with the process defined by the program itself. The records are initialised with information on the parse tree - the internal representation of the program.

For a complex process the analyser determines if any declarations are associated with it. If there are, these are parsed and a reference to the information gathered is stored. It also identifies the construct governing the execution of the subsidiary components, counts them, and parses each one according to its type. The resulting information is stored in the 'Cdscrptr' (see Appendix F: CmplxCmpnt) field of the associated 'CmpntRec' record.

For a simple process the analyser initialises a 'SmpleCmpnt' record (see Appendix F). The information gathered during the parse of the simple process is held in this type of record. Additional information is added in the succeeding phases. A reference to the record set up is stored in the process' 'CmpntRec'. The analyser first identifies the process' primitive operator. It uses this to select the appropriate parse routines called. The operator is also used to determine the form of the 'SmpleCmpnt' record to be set up.

If the input operator is identified the analyser obtains and stores a reference to the input channel identifier. Then, references for all the input variables are obtained and stored. If the output operator is identified a reference to the output channel is obtained and stored. Next, each output expression is transformed into its corresponding Reverse Polish form. References to the records representing these Polish forms are stored. Finally, if the assignment operator is identified a reference to the variable identifier assigned a value is obtained and stored. The analyser then transforms the associated arithmetic expression into its corresponding Polish form. A reference to the record representing this form of the expression is stored.

E.3 Dump

Writes out information held and referenced by the record corresponding to ' MajorPrc'.

E.3.1 Variable Parameters

1. MajorPrc :

   Pointer to the 'CmpntRec' associated with the Occam program.

E.3.2 Description

The procedure 'Dump' writes out information on the internal representation of the Occam source program. Firstly, it writes out the parse tree in a form reflecting the block structure of the source code. This form also illustrates the reverse polish representation for each arithmetic expression occurring in the program. Secondly, the contents of the 'SymTabRec' records implementing the symbol table are displayed.

The data displayed for the parse tree and the symbol table represents the information stored in the internal tree after the parse phase.

Finally, the procedure 'Dump' writes out the internal representation of the program after it has been completed in the graphical specification phase. This involves displaying information on the occurrences of identifiers and writing out the tree forms of expressions. This data is presented in a way which illustrates the structure of the parse tree.

E.4 InitBoxCell

Prepares the internal data tree for the graphical specification and graphical definition phases.

E.4.1 Variable Parameters

PrcPtr :

Pointer to the 'CmpntRec' associated with the Occam program.

E.4.2 Description

This recursive procedure implements a preorder traversal of the internal data tree. During this traversal previously stored information is used to initialise elements in the 'CmpntSpec' and 'CmpntImpl' fields of each 'CmpntRec' implementing the internal data tree. If a 'CmpntRec' is associated with a complex process then elements in the 'Cdscrptr' field are also initialised.

The elements initialised are used for holding information on (a) the occurrences of identifiers in the corresponding process and on (b) the graphical (cell) representation of the process. The information used in this initialisation was stored in the data tree during the parse phase.

E.5 INITIALIZE

Sets up the global identifiers used in the program.

E.5.1 Description

This procedure initialises identifiers used as global arrays, identifiers used as global counters, and a global boolean used as a flag.

The global arrays are set up as follows.

1. C

   The class values for the ASCII character set are stored in this array. The position of an element in the array represents the ASCII code for a particular character. The integer value held in the element defines the class of the associated character.

2. DelTable

   The delimiters and their internal code are stored in this array. An element of this array holds the symbol for a particular delimiter and its corresponding internal code.

3. RWTable

   The reserved words and their internal code are stored in this array. An element of this array holds the character string defining a reserved word and its corresponding internal code.

The following global counters are set to zero.

1. Ntoks

   Used to count the number of tokens held in the token buffer.

2. StartCol

   Counts the number of blanks proceeding the characters composing the current line of source text being scanned.

3. TBptr

   Counts the number of tokens removed from the token buffer. This variable represents a pointer to the current position in the buffer.

The global boolean flag 'ErrFlag' is set to false.

E.6 PrcImplmntn

Generates the compiler's object code. This code defines the graphical representation of the Occam source program.

E.6.1 Variable Parameters

1. MajPrc :

   References the 'CmpntRec' record denoting the root of the internal data tree.

E.6.2 Description

The task of this procedure is implemented in two stages. The first stage derives an internal definition of the graphical representation. The second stage uses this to generate the object code. The first stage corresponds to the graphical representation phase, and the second to the output phase. The data symbolising this definition is assembled in the compiler's data tree.

Internal definitions of the graphical (cell) forms of the program's processes are combined, giving one complete definition for the entire source program. This involves examining the data previously placed in the 'CmpntRec' records of the internal tree. A preorder traversal of the tree is used to access these records. During this traversal information in each record is interpreted, in order to define the cell form of the associated process. The resulting definition information is stored in the record.

The procedure separates the defining of the cell representation for a simple process from that for a complex one. However, an intrinsic step in the definition of both, is the placement of the cell on a conceptual grid. Information obtained from this placement defines, in terms of the grid, the co-ordinates of the cell's origin, its input ports, its output ports, and its dimensions. These conceptual co-ordinates are later translated into 'actual' co-ordinates during the plotting of the representation on a display device.

The cell definition of a complex process represents the 'glue' linking the cell representations of its subsidiary components. Deriving it entails defining the cell forms of the subsidiaries. It also requires defining the following.

(a) The graphical representation for the initialisation of each variable declared in the process.

(b) The routing of the power, ground, and clock lines to the subsidiary cells.

(c) The routing of the control line between the subsidiary cells.

(d) The routing of external and internal identifiers.

The algorithm developed to implement both these tasks is iterative, obtaining the definition and associated routing for each sub-process in turn.

Initialisation of a variable is represented graphically by a box linked to a cell. This box represents a register for holding the values assigned to the variable. The cell corresponds to the the assignment operator in the process initialising the variable. Connecting the east side of the cell to the west side of the register is an output line - the assignment line. Obtaining this representation requires defining the routing of this line, and defining the origin co-ordinates of the register. Routing of the output line is determined by the parent process' constructor.

There may be a line branching from the assignment line. This represents re-assignment of the variable in other processes. Segments of this line are connected to outputs of the cells in which re-assignment occurs. The routing of this line to these cells is part of (d). The connecting segments are defined during the definition of the corresponding sub-cells. There is also a line routed from the east side of the register with branches to several sub-cells. This line, termed the reference line, represents references to the variable in other processes. Defining the routing of this and its off-shoots is part of (d) above.

Steps (c) and (d) are both sub-divided into defining the routing into a sub-cell, and the routing out of a sub-cell. The definitions for the input lines are obtained before the sub-cell is defined, and those for the output lines after. Input lines are defined by a routing strategy determined by the execution sequence of the parent process. For identifier lines, the procedure ascertains whether a line is required in the horizontal and/or vertical data path of the sub-cell. The decision on which data path(s) a line is to be routed in, is based on information specifying the occurrences of the identifier in that sub-cell.

Each sub-cell has a control line routed out from its east side. Defining the routing of this output line is determined by the parent process' constructor and the sub-cell's type (simple or complex). For a simple sub-cell the line is routed from its right most constituent operator to the east side of the cell. Similarly, for a complex sequential sub-cell the control line is routed from the east face of its right most sub-component to the east edge of the cell. If, however, the sub-cell represents a complex parallel process the routing strategy is more involved. Firstly, a join cell is defined to be placed in the upper right most corner of the cell. Next, the control output line of each component cell is routed to the west face of this join cell. Finally, the control line for the parent process is routed from the east side of the join cell to the east edge of the cell.

Other possible output lines of a cell are channel lines. These lines connect two cells which represent the input and output processes associated with a channel identifier. Defining these lines involves routing each line from the cell it was created in, the source cell, to the cell it terminates in, the destination cell. The source and

destination cells may be adjacent, or separated by several cells. The implemented routing strategy defines lines for either situation.

Channels declared for the main process may represent external inputs or outputs. In which case, the corresponding channel lines are only connected to source cells. If this source cell represents an input process then the line is routed to the west edge of the main program's cell. On the other hand, if the source cell represents an output process then the line is routed to the east edge of the main cell.

The representation of a simple process is composed of one or more operator cells, several identifier lines, and power, ground, clock, and control lines. The cells represent the operators of the simple process. Their placement and the associated routing of the power, ground, clock, and control lines represents the implicit execution sequence in the process. The internal definition of this representation is derived from data stored in the process' 'CmpntRec' record. Acquiring it means defining each operator in turn, and defining the routing of the required lines. Each primitive process is handled separately. However, defining the representation of expressions is common to both assignment and output processes.

For an input process the procedure defines operator cells for the constituent read operations. These cells are defined sequentially to form a horizontal linear array. Before the definition of an input operator cell, vertical line segments from the parent's power, ground, clock, and control lines are defined. Then, during the cell's definition, a horizontal segment from each vertical line is defined. Each segment connects with the corresponding input port of the cell. The control line is defined such that it links the subsidiary cells together.

The variable output line of each operator cell is defined to connect to a corresponding line in the parent's horizontal data path. The channel line associated with the simple cell is routed from the west boundary of the cell to each subsidiary, terminating in the right most.

The procedure defines an assignment process in two stages. Firstly, the representation of the associated expression is defined (see below). Secondly, a primitive operator cell is defined together with the routing of the lines connected to its input ports. This cell represents storing of the result of the expression in a variable. The data output line of this cell is routed to a register as described above.

The definition of an output process involves two very similar stages which are repeated for each constituent write operation. In the first stage, the representation of the expression associated with a write operation is defined. An output cell and the routing of the lines connected to its input ports are defined in the second stage. This cell represents the writing of the expression's value on the process' channel. The channel is represented by a line defined to be routed from the channel output port of the first cell to the east edge of the parent cell, passing below the other cells. The channel outputs from the cells in between are defined to connect with this line.

Figure 7.6(a)illustrates an arithmetic expression tree. 'XprsnRec' records implement the nodes of such trees. Each node represents the <operator> <operand-1> <operand-2> form of an expression. The operator and a reference for each operand are stored in the node. The reference for an operand is either to a variable/constant identifier, or to a sub-tree describing a sub-expression. 'Ln-Op' denotes a node at level n in the tree. Figure 7.6(b) outlines the graphical representation of the

tree. The north to south arrangement of cells in an array mirrors the left to right order of nodes in the corresponding level. Defining this representation requires mapping the implied structure of the tree into the conceptual bounding box. Information stored in the tree is used to define the mapping. Accessing this information involves a postorder traversal of the tree.

Power, ground and clock lines are routed to each operator cell. Segments from the control line are first routed to the cell representation corresponding to the operand held in each leaf node (i.e. the cells in the left most array). Subsequent routing for the control is determined by the structure of the tree. Each subtree is used to define the routing of the control line from the cell representation of its siblings to the control input port of the cell corresponding to the parent. If a root has only one subsidiary node then the control line is routed directly from the cell representation of the node to that of the root. If two nodes are attached to the root the control output line of each sibling cell is routed to a join cell. A line is then routed from this join cell to the cell corresponding to the root.

The subtrees are also used to define the routing of data lines between the cells. The data output line from the cell representation of each sibling node is routed to the appropriate input port of of the root's operator cell. For a node referencing a variable the corresponding data line is routed to an input port of its operator cell.

After deriving the internal definition for the graphical representation, the procedure uses it to produce the compiler's object code. To produce the object code the procedure writes out, in a text file, the conceptual co-ordinates of all the segments composing the lines

for power, ground, clock, control, each variable, and each channel. The line segments comprising the bounding box of each primitive cell are written out. Associated with each pair of co-ordinates is an integer denoting the colour of the line connecting the two points. This integer is also written out. Obtaining this data involves a second preorder traversal of the data tree. During the traversal the maximum and minimum co-ordinates are calculated. The data in the text file is then used to set up and format another text file. This second text file holds the object code. It is formatted into blocks, each block containing pairs of co-ordinates for line segments plotted in a particular colour.

E.7 PrcSpec

Implements the graphical specification phase.

E.7.1 Variable Parameters

1. MajPrc :

   Pointer to the 'CmpntRec' associated with a process.

E.7.2 Description

The overall task implemented by this procedure is the completion of the internal representation of the source code. This is split into the following sub-tasks.

1. Identifying the subsidiary in which each of a process' declared variables is initialised.

2. Classifying the variable identifiers occurring in a process.

3. Identifying the read and the write subsidiary associated with each of a process' declared channels.

4. Classifying the occurrences of channel identifiers in a process.

5. Translating the reverse polish forms of expressions into their tree representations.

These tasks are carried out during a preorder traversal of the initialised data tree. Data stored during the previous phase is accessed and used to derive the necessary specification information.

A process' variable is initialised by either an assignment operation or an input operation. A reference to the variable occurs in the internal form of the initialising subsidiary process. The procedure recursively searches the internal representations of the process' subsidiaries until the required reference is found. When found, the reference number of the sub-process is stored in the appropriate element of the 'Cdscrptr' field of the parent's 'CmpntRec' record. Also, a reference to the identifier is stored in the 'CmpntSpec' field of the sub-process' 'CmpntRec'. This reference specifies that the identifier is external to the process. It is stored in the list for initialised external variables. Having identified the subsidiary, the remaining ones are searched for any other instances of the identifier. Further occurrences are classified in to two types: re-assignment and reference. Each type is checked for separately. If found a reference to the identifier is stored in the appropriate external list of the 'CmpntSpec' field of the subsidiary.

A process' channel identifiers will be referenced in at most two subsidiaries. The procedure searches for the first occurrence of a channel identifier. When found, the reference number of the subsidiary is stored according to whether it is an input or output process. A reference to the identifier is stored in the 'CmpntSpec' field of the subsidiary's 'CmpntRec'. This reference is placed in one of two lists classifying the occurrence in the process; either the list for external channels read from, or the list for external channels written to. Then the second subsidiary is sought. If found, a reference to it is stored in the appropriate external channel list of the subsidiary. The subsidiary's reference number stored in the parent's 'CmpntRec'.

As each 'CmpntRec' associated with a complex process is accessed, the procedure searches its subsidiaries for occurrences of external identifiers. References to these identifiers were stored earlier in the traversal in the external lists of the 'CmpntSpec' field . Any occurrences of variable identifiers are classified into three types: assignment, re-assignment, and reference. While occurrences of channel identifiers are classified into read and write references. Each type is checked for separately. If found a reference to the identifier is stored in the appropriate external list of the subsidiary.

The reverse polish expression associated with an assignment process and the expressions associated with an output process are translated into a recursive tree representation. This tree representation is used in specifying the graphical implementation of the expression. During this translation the execution sequence of the expression is modified to maximise concurrency. The procedure also obtains the reference numbers for variables occurring in the expressions associated with the sub-trees

of the expression tree. The pointers to the recursive tree representations are stored in the 'SimplCmpnt' record referenced by the 'CmpntRec' record.

E.8 THE OUTPUT

The format of the the "object file" generated by the compiler is illustrated overleaf. Such a file holds the definition of an Occam program's graphical representation. Each item within <> represent an an integer. A line plotting program processes this file in order to implement the definition. This program was also written in VAX-11 Pascal V2.2-114. It uses the VAX EXTERN facility to call a sub-set of the subroutines in the GKS graphics package [14]. These subroutines activate and deactivate the device on which the representation is to be displayed, and display each line segment in the appropriate colour. The program uses the minimum and maximum co-ordinates to scale the definition to the GKS device co-ordinates.

```
<minX> <minY> <maxX> <maxY>
<separator>
<pen number>
<x1> <y1> <x2> <y2>
<x1> <y1> <x2> <y2>
.
.
.
<x1> <y1> <x2> <y2>
<separator>
<pen number>
<x1> <y1> <x2> <y2>
<x1> <y1> <x2> <y2>
.
.
.
<x1> <y1> <x2> <y2>
<separator>
<pen number>
.
.
.
<separator>
<end of file>
```

KEY

<Xmin> :- minimum X co-ordinate in the defined graphical representation
<Ymin> :- minimum Y co-ordinate in the defined graphical representation
<Xmax> :- maximum X co-ordinate in the defined graphical representation
<Ymax> :- maximum Y co-ordinate in the defined graphical representation
<separator> :- delimiter used to split the file into sections
<Pen Number> :- integer denoting the colour a set of line segments are to be displayed in.
<x1> :- X co-ordinate of the start point of a line segment
<y1> :- Y co-ordinate of the start point of a line segment
<x2> :- X co-ordinate of the start point of a line segment
<y2> :- Y co-ordinate of the start point of a line segment

APPENDIX F

IMPORTANT RECORDS AND THEIR FIELDS

This appendix describes several of the important records used to implement the compiler's data tree. A brief summary on the purpose of each identifier and its attributes is included. The identifiers are listed alphabetically.

F.1 CellRec

Defined for holding information on the graphical (cell) definition of a process. The information held in the fields of this type of record relates to the placement of the cell, its dimensions, and the inputs and outputs of the cell. This information is obtained during the definition phase.

- Xorig
The X co-ordinate for the upper left hand corner of the cell.

- Yorig
The Y co-ordinate for the upper left hand corner of the cell.

- Width
The width of the cell in 'grid' units.

- Height
The height of the cell in 'grid' units.

- Control
Pointer to record associated with the routing of the cell's control line.

- Vbundle
A record used for holding references to information on the vertical path for the power, ground, and clock lines.

- Hbundle
A record used for holding references to information on the horizontal path for the power, ground, and clock lines.

- Vdatapath
A record used for holding information on the vertical path for variable lines.

- Hdatapath
A record used for holding information on the horizontal path for variable lines.

- AsgnLines
A record used for holding information on the path for the lines representing variables initialised in the cell.

- Vchanpath
A record used for holding information on the vertical path for channels.

- Hchanpath
A record used for holding information on the horizontal path for channels.

- ThruChans
A record used for holding information on the path routing channels through the cell.

- VarAsgnd
Indicates if a storage structure (one or more registers) is associated with the cell. Such a structure is required if the associated process initialises variables declared in its parent.

- ChanInfo
Record used to hold information on the graphical definition of all channels declared in the associated process.

- JoinReqd
Selector used to indicate if a join cell is associated with the cell representation.

1. JoinReqd = TRUE

    1. - joinhght
       Height of the join box in 'grid' units.

    2. - join
       Record holding the origin co-ordinates of the join cell.

2. JoinReqd = FALSE
   No fields.

F.2 CmplxCmpnt

Defined for linking parse and specification information on a complex process (component). A subset of the fields of this record type are

initialised during the parse phase to hold information on an associated process' subsidiary components, its constructor, and its identifier declarations. The specification phase initialises the remaining fields with information on the occurrences of the identifiers.

- Constrctr
The internal representation for the process constructor.

- Dclration
A pointer to the record holding the information obtained from the parsing of any declaration statements associated with process. If there are no such statements then this pointer is given a nil value.

- Nsubcmpnts
The number of subsidiary processes composing the process.

- SubCmpnts
A list of pointers to the records associated with the subsidiary processes.

- Ichans
The number of channels declared for the process.

- Nlocal
The number of variables declared for the process.

- Xrefs
A list holding information on references to the process' variables.

- Xredefs
A list holding information on re-assignments to the process' variables.

- Xdefs
A list holding subsidiary reference numbers. Each entry in the list identifies the processes in which the process variable associated with that list position is initialised.

F.3 CmpntRec

A record type defined to link all the information obtained on a process (component). This record type always holds some specification and definition information on the associated component. This information is independent of the whether the component is simple or complex. If the associated component is complex then the record holds parse and additional specification information. If, however, the component is simple then a pointer is held. The pointer references a record holding parse, specification and definition information specific to a simple component.

- SubNo
Reference number associated with a process. For the major process this is always zero.

- Pcmpnt
Pointer to the record associated with the surrounding (parent) process.

- CmpntSpec
A record designed for holding information on the associated process' external identifiers. This information specifies which variables are initialised, referenced, and re-assigned in the process. It also specifies which channels are read from, and written to in the process. The information is gathered and stored during the specification phase.

- CmpntImpl
Holds, information gathered during the graphical definition phase, on the graphical (cell) form of the process. (see CellRec)

- ProcType
Selector used to distinguish between simple and complex processes.

1. ProcType = Simple

    1. - SimplPtr
       Pointer to a record containing specification and definition information for a simple process. (see SmpleCmpnt)

2. ProcType = Complex

    1. - Cdscrptr
       Record holding parse and specification information for a complex process. (see CmplxCmpnt)

F.4 PrimUnit

Defined for holding the information on the graphical (cell) definition of a primitive operator. The information held in the fields of this type of record relates to the placement of the cell, its dimensions, and the co-ordinates of the inputs and outputs of the cell. This information is obtained during the definition phase.

- Xorig
The X co-ordinate of the upper left hand corner of the cell representation.

- Yorig
The Y co-ordinate of the upper left hand corner of the cell representation.

- Height
The height, in 'grid' units, of the cell representation for the primitive operator.

- Width
The width, in 'grid' units, of the cell representation for the primitive operator.

- VddIn
The co-ordinates of the input port for the power line.

- GndIn
The co-ordinates of the input port for the ground line.

- ClkIn
The co-ordinates of the input port for the clock line.

- CntrlIn
The co-ordinates of the input port for the control line.

- CntrlOut
The co-ordinates of the output port for the control line.

- UnitType
Selector to identify which primitive operator is implemented by the record.

1. UnitType = AsgnOp

    1. - ResIn
       The co-ordinates of the input port for the line bringing the value to be assigned to a variable.

    2. - DataOut
       The co-ordinates of the output port for the line connecting the operator to the appropriate register.

2. UnitType = InOp

    1. - ChanIn
       The co-ordinates of the input port for the channel being read from.

    2. - VarOut
       The co-ordinates of the output port for the line connecting the operator to the appropriate register.

3. UnitType = OutOp

    1. - XpIn
       The co-ordinates of the input port for the line bringing the value to be output on the channel.

    2. - DataOut
       The co-ordinates of the output port for the line channel.

F.5 SmpleCmpnt

Defined for holding parse, specification, and definition information on a simple process (component). This information is dependent upon the primitive operator associated with the component. During the parse phase the primitive operator of the process is identified and used to determine the form of this type of record. The parse phase also adds a reference to the identifier acted on by the operator together with information on

the part of the statement occurring after the operator.

If the component is an assignment or output one, then, references to associated reverse polish expressions are also stored during the parse phase. In the succeeding phase these referenced expressions are translated into their tree forms and pointers to the records holding these forms are stored.

In the definition phase this type of record is used to hold information on the graphical (cell) form of the primitive operation.

- PrmType
Selector used to identify the type of simple process information is being gathered on.

1. PrmType = Asgnmnt

    1. - VarId
       Pointer to the symbol table record corresponding to the variable being assigned a value in the process.

    2. - RevPolXprsn
       Pointer to the record associated with reverse polish format of the expression.

    3. - AsignUnit
       Record holding the information concerning the graphical definition of the assignment operator. (see PrimUnit)

    4. - AsgnPtr
       Pointer to the record holding the specification and definition information for the expression.

2. PrmType = Inpt

    1. - InChan
       Pointer to the symbol table record corresponding to the channel being read from in the process.

    2. - Nin
       Number of sequential input operations occurring in the process.

    3. - VarList
       List of pointers to the symbol table records of each of the variables reading a value from the channel in the process.

    4. - InList
       List of records each holding graphical definition details for an input operator. The ordering of the list corresponds to the order in which values are read from the channel and stored in the variables.

3. PrmType = Outpt

    1. - OutChan
       Pointer to the symbol table record corresponding to the channel being written to in the process.

    2. - Nxprsns
       The number of values being sequentially written to the channel in the output process.

    3. - RevPolXprsns
       List of pointers to the records holding the reverse polish from of the expressions occurring in the process.

    4. - OutXprsns
       List of records each holding graphical definition details for an output operator. The ordering of the list corresponds to the order in which expressions are evaluated and the results written to the channel.

F.6 SymTabRec

Defined for holding information on identifiers. Records of this type hold information on an associated identifier string and a reference to the process in which the identifier was declared. These records also hold information which is specific to the identifier type.

- IdLen
Length of the identifier string.

- IdStrng
The identifier string.

- NxtRec
Pointer to the next related record.

- ProcPtr
Pointer to the record associated with the process in which the identifer was declared.

- Item
Selector for the variant fields needed for the different types of of identifier.

1. item = Channel

    1. - placed
       Indicates if the channel identifier has already been processed in the graphical definition phase.

    2. - offchip
       Indicates if there are both read and write operations associated with a channel identifier.

3. - InitdFor
   Defines whether the first reference to the identifier was in an input or output statement.

4. - CellNmbr
   Used to reference the current cell representation the channel is being routed from.

5. - UsedIn
   A pointer to the record associated with the process in which the read and write operations occur.

6. - Cell
   A pointer to the record associated with the process in which the identifier is first referenced.

2. item = Constant

   1. - Value
      The integer value associated with a constant identifier.

3. item = Variable No fields.

F.7 XprsnCell

Defined for holding information on the graphical (cell) representation of the tree form for an arithmetic operator, and the left and right expressions associated with it (ie <left expression> <operator> <right expression>). This information is placed in this type of record during the interpretation of associated specification data. The data held in the fields of this type of record relates to the placement of the cell, its dimensions, and the inputs and outputs of the cell.

- Vbundle
Record holding the pointers to routing records for the vertical path of the power, ground, and clock lines. (see BundlRec)

- Hbundle
Record holding the pointers to routing records for the horizontal path of the power, ground, and clock lines.

- Xorig
The X co-ordinate for the upper left hand corner of the cell.

- Yorig
The Y co-ordinate for the upper left hand corner of the cell.

- Width
The width of the cell in 'grid' units.

- Height
The height of the cell in 'grid' units.

- Control
Pointer to routing record associated with the cell's control line.

- Vdatapath
Record used to hold plotting information on the vertical routing path for variables in the cell representation.

- Hdatapath
Record used to hold plotting information on the horizontal routing path for variables in the cell representation.

- PrimCell
Record holding details of the cell definition for the arithmetic operator.

- JoinReqd
Selector indicating if a join cell is associated with the cell.

1. JoinReqd = TRUE

    1. - join
       Record holding the co-ordinates for the upper left hand corner of the join cell.

    2. - joinhght
       Height in 'grid' units of the join cell.

F.8 XprsnOpUnit

A record defined for holding information regarding the graphical (cell) definition of an arithmetic operator. The information held in the fields of this type of record relates to the placement of the cell, its dimensions, and the co-ordinates of the inputs and outputs of the cell. This information is obtained during the definition phase.

- Xorig
The X co-ordinate of the upper left hand corner of the cell representation.

- Yorig
The Y co-ordinate of the upper left hand corner of the cell representation.

- Height
The height, in 'grid' units, of the cell representation for the arithmetic operator.

- Width
The width, in 'grid' units, of the cell representation for the arithmetic operator.

- VddIn
The co-ordinates of the input port for the power line.

- GndIn
The co-ordinates of the input port for the ground line.

- ClkIn
The co-ordinates of the input port for the clock line.

- CntrlIn
The co-ordinates of the input port for the control line.

- CntrlOut
The co-ordinates of the output port for the control line.

- UnitType
Identifies the type of operand being represented by the record.

- DataIn1
The co-ordinates of the input port for the upper data line.

- DataIn2
The co-ordinates of the input port for the lower data line.

- DataOut
The co-ordinates of the output data line for the result of the operation.

F.9 XprsnRec

A record type defined for holding information on the graphical specification and definition of an expression. Records of this type represent the tree forms of expressions. These trees are formed by manipulating the corresponding Polish forms during the specification phase. In the succeeding definition phase the implied structure of the tree is used to place a graphical implementation of the expression on the conceptual grid. The information generated for this placement is stored in this record.

- InUpper
Record referencing the left input to the operator node. The input is either a variable or the result of a sub-expression. For the former this record holds a variable reference number. In the case of the latter the details of the sub-expression are referenced by the record.

- InLower
Record referencing the right input to the operator node (cf InUpper).

- Spec
A record defined for holding the graphical specification information on the tree form of an expression. This information is placed in this type of record during the transformation of the corresponding Polish form. This record identifies the arithmetic operator linking the two sub-expressions trees. It also references each variable occurring in the left subtree and each variable occurring in the right subtree.

- Strctr
Record holding the graphical definition information. (see XprsnCell)