VLSI DESIGN METHODOLOGY

by

Javeed I. Mhar

A thesis submitted to the Faculty of Science,

University of Edinburgh,

for the degree of Doctor of Philosophy

Department of Electrical Engineering

1990 (resubmitted 1991)

VLSI DESIGN METHODOLOGY

by

Javeed I. Mhar

Abstract

The development of FIRST was a significant step in the field of silicon compilation. With FIRST, bit–serial signal processing systems could be rapidly implemented in silicon by high–level designers without requiring layout expertise.

This thesis explores extensions to the compiler, but the methodology and techniques are not specific to FIRST and could be used in the more general VLSI arena. One major theme is the use of process independent layout, allowing the rapid update of a cell library to current state of the art process rules. After surveying other layout strategies, one particular layout style, gate matrix, was evaluated through the manual layout of a bit–serial, two's complement, multiplier utilising novel architectural features. The operation and architectural features of the multiplier are described, as these features were to be incorporated as options in newly generated cell libraries.

SECOND, a full span silicon compiler, taking the high–level input description of FIRST but synthesizing layout to a process independent form (gate matrix) was developed using ideas gained from the manual assembly procedure. SECOND maintains and extends the hierarchy of FIRST using different assembly strategies for differing levels of hierarchy in the synthesis procedure. The hierarchy is described and the placement, routing and assembly procedures of the new elements of the hierarchy are covered. The automation tools used to generate the gate matrix layout of the lowest hierarchy level of SECOND are covered in a separate chapter.

Using the same concepts of hierarchy, a tool ENGEN which transforms FIRST intermediate code to a gate level network description in HILO is also described as an alternative to SECOND in the search for process independence. The thesis ends with a suggestion of a bit–serial/ bit–parallel frame for encouraging the acceptability of bit–serial systems.

Declaration of Originality

The material contained within this thesis was researched and developed as part of a group in the Department of Electrical Engineering at the University of Edinburgh, between October 1985 (1 year MSc converted to PhD) and May 1989. This is in accordance with regulation 3.4.7 of the rules governing a university thesis. I have contributed ideas, discussed and developed methodology throughout the work. I have made significant original contributions in the work behind SECOND (chapter 6) and the HILO generation work (chapter 8). I have tried to indicate other members contributions through the references.

Work described in Appendix 1, was done whilst at Thorn EMI, my cooperating industrial partner under the SERC CASE award scheme. This work is original and I have received permission from Thorn EMI for its inclusion within the thesis.

The current chapter 7 was added after comments by the external examiner. It contains extra relevant material obtained after consultation with my project supervisors.

Acknowledgements

I would first like to thank my supervisor Professor Peter Denyer for his constant support and encouragement, who continued to provide further help and encouragement, after I had left University surroundings, without which this thesis would not have been completed. Thanks are due to all members of the group who contributed to the work on SECOND, the composition of the group had altered during the work, but each has made valuable contributions to the endeavours. This includes Dr Kevin Coplan, Mike Keightley, Mike McGregor, Prof. Kunihiro Asada, and Dr Stewart Smith (who provided much energy, ideas and co-ordination to the SECOND project). I would also like to thank my second supervisor Dr David Renshaw, for encouragement and suggestions, and Dr Lawrence Turner (University of Calgary) who, during his stay at Edinburgh, provided ideas and code which contributed to the work of Chapter 8.

Signed

[redacted]

Javeed Iqbal Mhar

Contents

Chapter 1

Introduction: Bit–Serial Methodology and Silicon Compilation

1.1. Introduction

Through the rapid advancement of fabrication technology in the field of microelectronics, designers can now produce VLSI integrated circuits of complexities which were impractical earlier. These devices have the capability to perform complex functions in real time. The designer is confronted with the task of the integration of hundreds of thousands of devices onto a single silicon component, greatly increasing the complexity of the design process and also the possibility of mistakes. To make this task more manageable, structured design styles and software tools have been developed.

A software tool which would be of great value to the designer of integrated circuits is the *Silicon Compiler*. The term *Silicon Compiler* was used in original work by Johannsen[1], [2] to describe the process of translating a language description of a silicon device to mask information. An ideal silicon compiler would be able to take a high level behavioural description of a system and produce from it the low level mask information for an integrated circuit without any further intervention of the designer, but as the field of silicon compilation is still developing such a compiler is yet to be produced.

It is currently possible to transform high level descriptions (which contain structural descriptions) to low level mask detail if we are prepared to restrict ourselves to a limited domain. Two domains where progress has been made have been with datapaths and with digital signal processing systems. Datapaths have a regular, though flexible architecture which lends itself to automation and their specification can be done through the use of register transfer languages. With digital signal processing systems, their specification can take the form of a flow graph using small elements such as adders, multipliers and delays, and these can be used to construct the system.

In this thesis, we shall concentrate on developments of the *FIRST*[3] silicon compiler. *FIRST* was one of the earliest silicon compilers to be produced and it is targeted at the bit–serial, signal processing domain. FIRST was developed at Edinburgh University and takes as input a structural description of the system for which integrated circuits are required. The structural description describes the interconnection of PRIMITIVEs contained in the library of FIRST. The PRIMITIVE functions have been designed to correspond to those used in digital signal processing algorithms, allowing PRIMITIVE interconnection to correspond directly with the nodes of a signal flow graph. The output from FIRST is the mask information needed for the integrated circuits, for use with a 5μ NMOS process.

This chapter gives an introduction to the topics of silicon compilation and bit–serial architecture which forms the background for the work described in this thesis.

1.2. Background

The layout of digital integrated circuits can be a tedious and time consuming chore, especially considering that a designer may most productively expend effort on the algorithmic aspects of his/her design. Design Automation has thus become essential in VLSI design environments. CAD tools help manage the complexities of the design process leading to fewer design errors and to a faster turnaround. One tool the *Silicon Compiler* has attracted much attention in the recent past because of its potential to simplify and speed up considerably the most tedious aspect of silicon design, the layout of VLSI designs.

The term *Silicon Compiler* has been the subject of much controversy and interest, with the suggestion that the term *Silicon Assembler* be used in its place, as some feel this is a fairer analogy to the compilation of software. Both terms have their merits and certainly there is no dispute that software compilers are in a much more advanced stage than their hardware counterparts, the compilation of a general behavioural description to silicon mask information has yet to be achieved as many complex problems are involved.

The term *Silicon Compiler* will be used in the remainder of this thesis with the understanding that this is not a tool which is capable of transforming a program written in a high level language such as Pascal, Ada or Fortran into a piece of silicon artwork. What we do claim is that we are able to transform a rigidly defined subset of a language into a piece of working silicon, though it is left up to the designer to ensure that functions performed by the silicon and by his program correspond.

FIRST[3], [4] was one of the earliest silicon compilers to be developed. The aim of FIRST is to get from a formal system specification through to a good working system. The specification is written in the FIRST language, and the working systems are the interconnected chips which are produced after processing at a silicon foundry. In this way a 'naive' user of silicon circuit design methodology could use the tool to produce signal processing systems which he/she has familiarity with, without the inconvenience of having to learn VLSI circuit and layout design techniques.

FIRST is restricted to bit–serial systems, and it is this focus of architecture which makes practical the translation to silicon artwork.

Other early *Silicon compilers*, for example MacPitts[5] restrain themselves to other types of specific architecture, for example datapaths. Problems of algorithmic complexity are avoided as specific simplified cases of the very complicated general floorplanning and routing problems can be used. The complexity of the these problems in the general case has made the wider spanning silicon compilers impractical.

A further simplification allowing the compiler to be a practical project has been the use of the bit–serial signaling convention. Signals are communicated along single wires instead of multiwire busses in bit–parallel systems, reducing dramatically the number of wires which need to be routed in systems employing this technique. The impetus to use bit–serial systems was supplied by earlier workers constructing architectures for digital filters and more general signal processing systems.

The foundation work on using bit–serial architectures for signal processing systems was presented by Jackson *et al.*[6] in 1968. In their work they identify features which make the serial scheme attractive for use in signal processing :–

– Filters can be constructed from a small set of simple digital circuits.

– The configuration of these circuits is highly modular and well suited to LSI.

– The flexibility to realise a wide range of filter forms is provided by the circuits.

– The filter may easily be multiplexed, either to process multiple data inputs or realise different functions with the same hardware, providing efficient hardware utilisation.

The above being true for systems which can be clocked at rates much higher than the sampling rate, which is the case in audio signal processing. Using bit–serial keeps the size of the digital circuits small, or when implemented as elements on a VLSI chip allows greater numbers to be used.

Powell and Irwin[7] achieved very high performance architectures by advocating the use of *Functional parallelism*, that is the parallel use of an array of high density, low cost, lower performance devices to achieve a high performance function. They multiplex bit–serial components in large arrays to achieve high throughput rates, even in excess of the bit rate of the bit–serial components.

Lyon[8] proposed a design of a fully serial pipeline multiplier capable of operation on two's complement numbers. This element added another powerful and useful element to the range of bit–serial components. Lyon later proposed an architectural methodology[9] based on the bit–serial technique. This paper established a set of principles on clocking, number format, signaling, etc. which allow bit–serial systems to be constructed. It mentions the type of components required and how multiplexing can be used to achieve a range of performances.

Lyon's methodology was extended by Denyer, Renshaw and Bergmann[10] using it as the basis for FIRST. The useful bit–serial components were included into the compiler as PRIMITIVEs, including extra elements useful for processing. Lyon's principles were incorporated as a rigorous set of conventions [11] which enabled any modules generated with the tool to successfully communicate to each other. The result of strict adherence to these signaling conventions guaranteed correctly functioning circuits, *correct by construction*, aided by a certain amount of overdesign in the underlying PRIMITIVEs.

A designer using the tool need not be aware of these conventions, as they are incorporated in the design of the PRIMITIVEs so allowing him/her to concentrate on the design of complete systems. Any mistakes made by the designer can be spotted easily and corrected quickly because the specification is at a higher and more easily understood level.

1.3. Digital Signal Processing

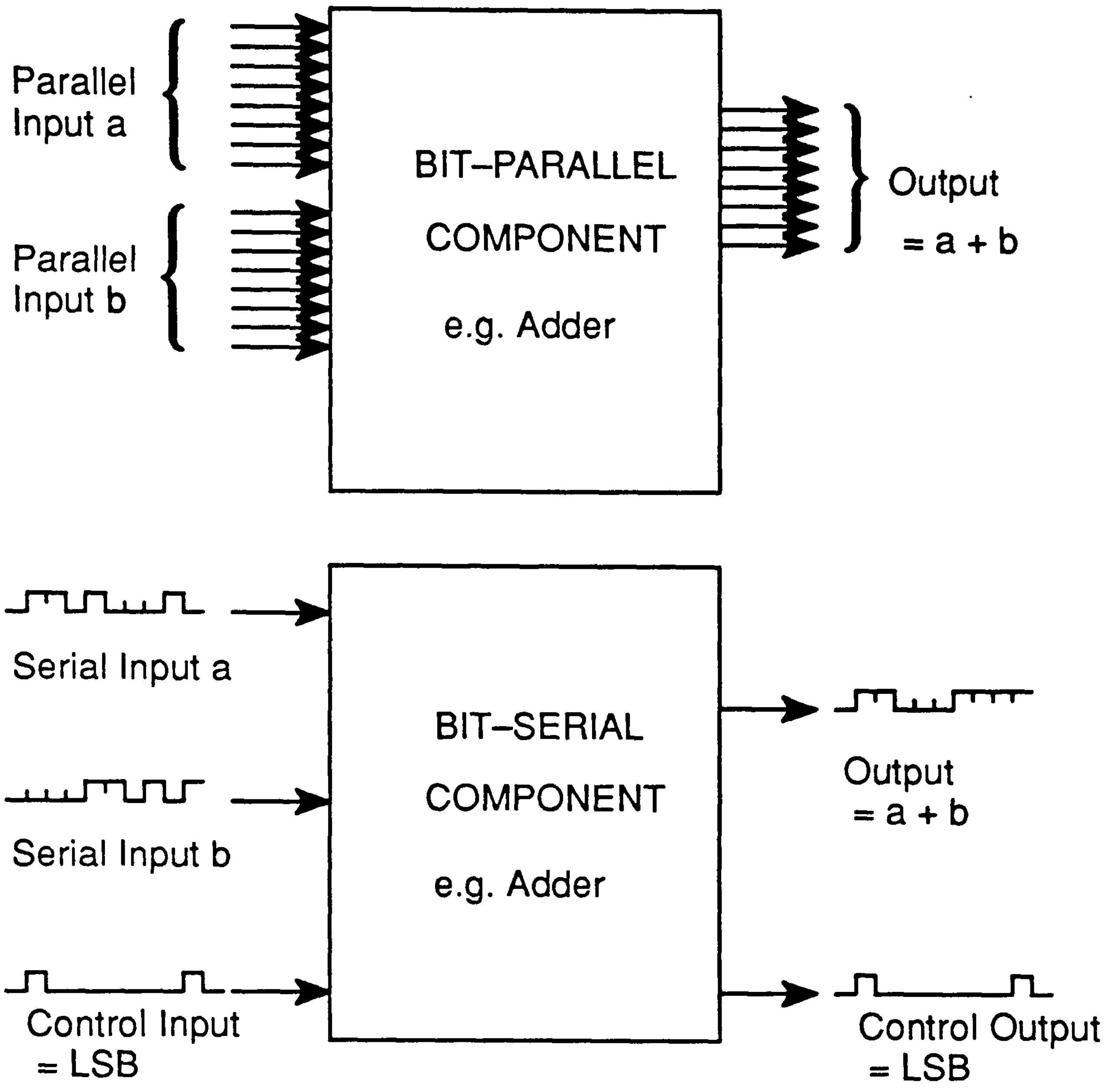
Digital signal processing algorithms implement signal processing functions using digital as against analogue circuitry. Digital signal processing has become deservedly popular and has applications in nearly all branches of electronics from instrumentation to telecommunications. Digital circuits do not suffer from the problems of power supply sensitivity, drift and aging as do their analogue counterparts. Signal processing systems can have a greater range of functionality when composed of digital building blocks, as there are digital elements, to which there are no analogue component equivalents.

The theory behind synthesising digital circuits is well established. The most commonly needed signal processing component is a filter. This is a linear (usually time–invariant i.e. response only dependent on previous sample values) system with memory (i.e. the output is a linear function of some history of the input). Most commonly, the input and output are single scalar signals, though this is not required; complex– and vector– valued signals are easily accommodated in this architecture. In a digital discrete–time architecture, a filter is a computation that operates on a sequence of numbers as input, and produces another sequence of numbers as output. The usual purpose of such computations is to pass some frequencies of signals, while attenuating others. For detailed information on digital filters, Rabiner and Gold[12] is a good reference.

Bit–Serial Methodology

Bit–serial methodology has some advantages over bit–parallel, especially in size and complexity of the basic blocks. It has shown itself suitable for use in signal processing applications. This section gives an overview of bit serial methodology comparing it with bit parallel and mentioning ways the technique can be extended.

A bit–serial system distinguishes itself from its bit–parallel counterpart by communicating and processing data in a serial–fashion, a single bit from a data–word is processed at a time, rather than processing all the bits of the whole data word as in a parallel system. Communication can take place through a single wire, Figure 1.1, rather than through a parallel bus of width equal to the system wordsize.



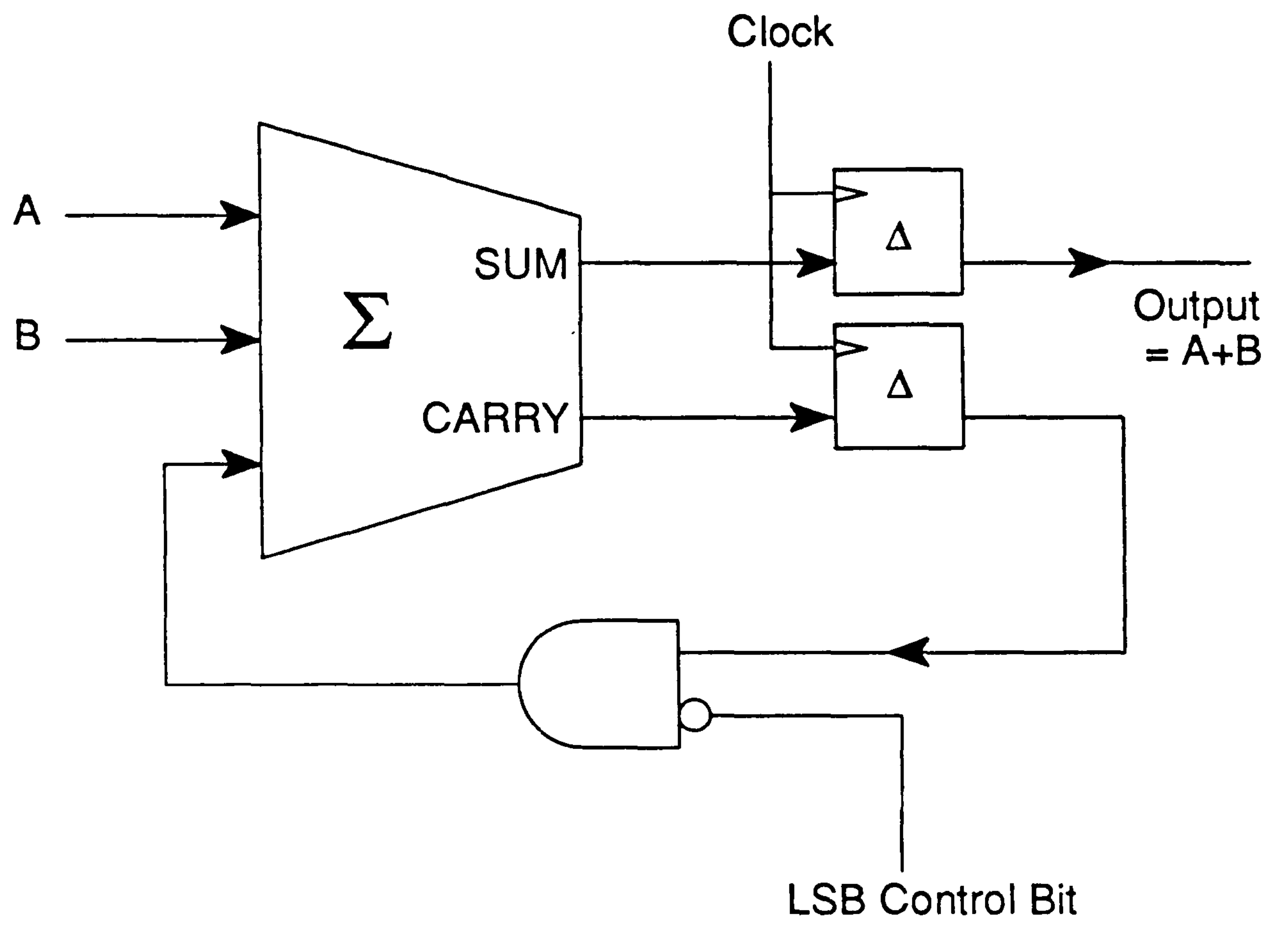
The Bit–Serial Component

Figure 1.1

The routing of signals is a complex task and the routing of large numbers of wires can result in very inefficient routing or to a completely unroutable chip. Therefore the use of a single wire to communicate data which results in less signal routing is an important and advantageous feature in the layout generation process.

Complex systems can be produced with serial components as they are small in size allowing a large number of them to be integrated onto a single chip.

A bit–serial adder is shown in Figure 1.2, it is one of the simplest serial components, which has a size which is independent of the wordlength. More complicated components such as the two's complement multiplier have a regular structure whose size is linear in the wordlength. The structure of this component and similar components has a form amenable to bit–slice implementation. A description of the operation of such a multiplier will be given in chapter 3.



A Bit–Serial Full Adder

Figure 1.2

The order of the bits affects the complexity and size of the components which operates on the bit–serial data stream. In FIRST, the signal convention used is to start with the least significant bit. This is very well suited to adder and multiplier elements but not very well adapted to more advanced signal processing elements such as two input dividers and square–rooters. As very many signal processing systems use only multipliers, adders, subtractors and delay elements, the range of systems which can be implemented with FIRST covers most of the useful signal processing functions.

In summary the main aspects of the bit–serial approach are :–

– Small functional elements

– Regularity of structure; bit–slices are common

– Low communications overhead leading to less routing

Bit–Serial Conventions

In FIRST[11] a set of conventions is used for bit–serial signaling which allow working systems to be constructed from elements with consistent protocols and interfaces. These conventions are important in hiding the low level problems of signal timing, communication and compatibility. The set of conventions built up from those proposed by Lyon[9] used in FIRST ensure that arbitrary networks constructed from PRIMITIVES perform in a sensible and valid way. By fixing these conventions for each element in the FIRST library, we hide from the user these details and so the user can be assured of successful operation. These conventions are :–

– Communication is bit–serial.

– Bit–serial data bits are in order with the least significant bit first.

– Logical 1 is a high signal & logical 0 is a low signal.

– A valid clocking strategy is used throughout the system, in FIRST this is the two phase non–overlapping clocking strategy as proposed by Mead and Conway[13].

– Numerical format is fixed point, two's complement.

– Data wordlength is fixed and constant in each system.

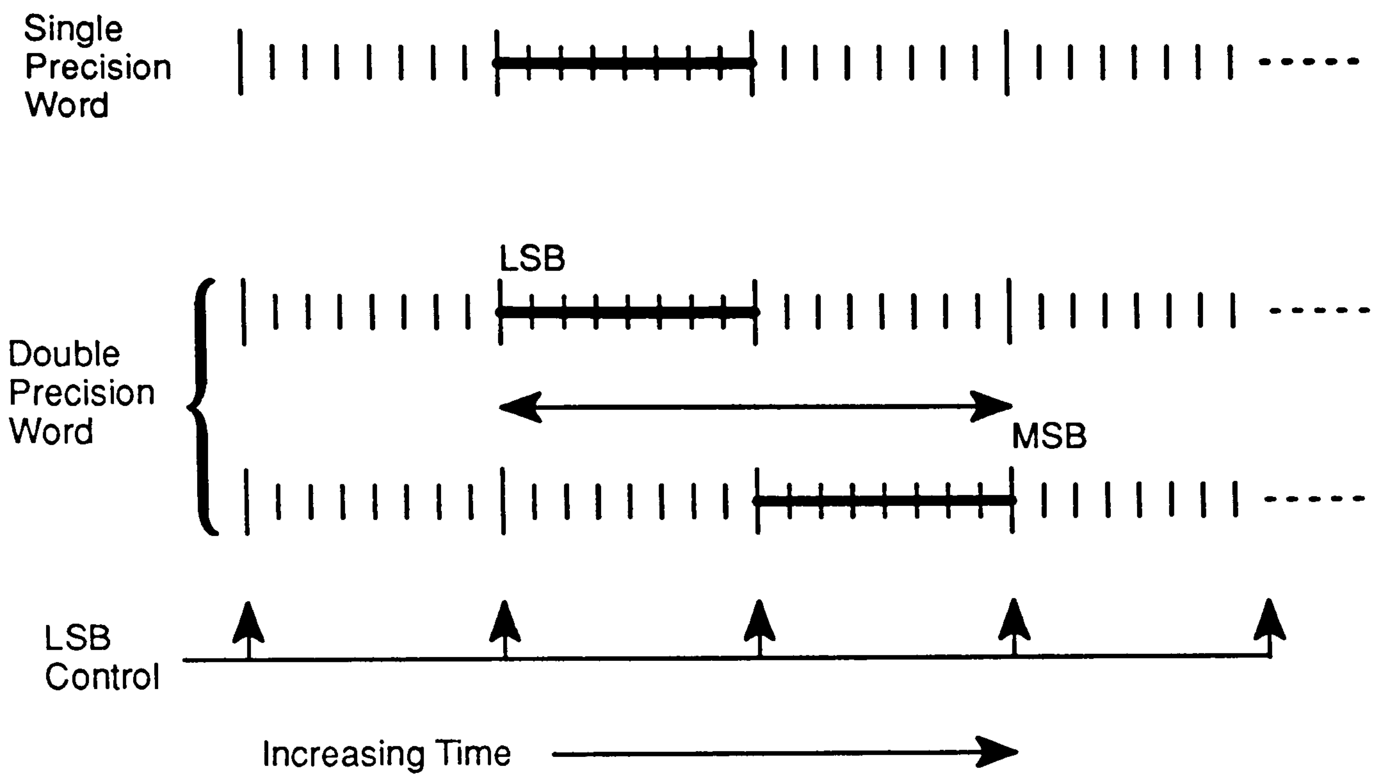
– Multiple precision signals are produced as in Figure 1.3

– Operators have fixed latency.

– Inputs to Operators are time aligned.

– Control cycles have a format as shown in Figure 1.4

– Each system has one CONTROLGENERATOR to generate the control cycles.



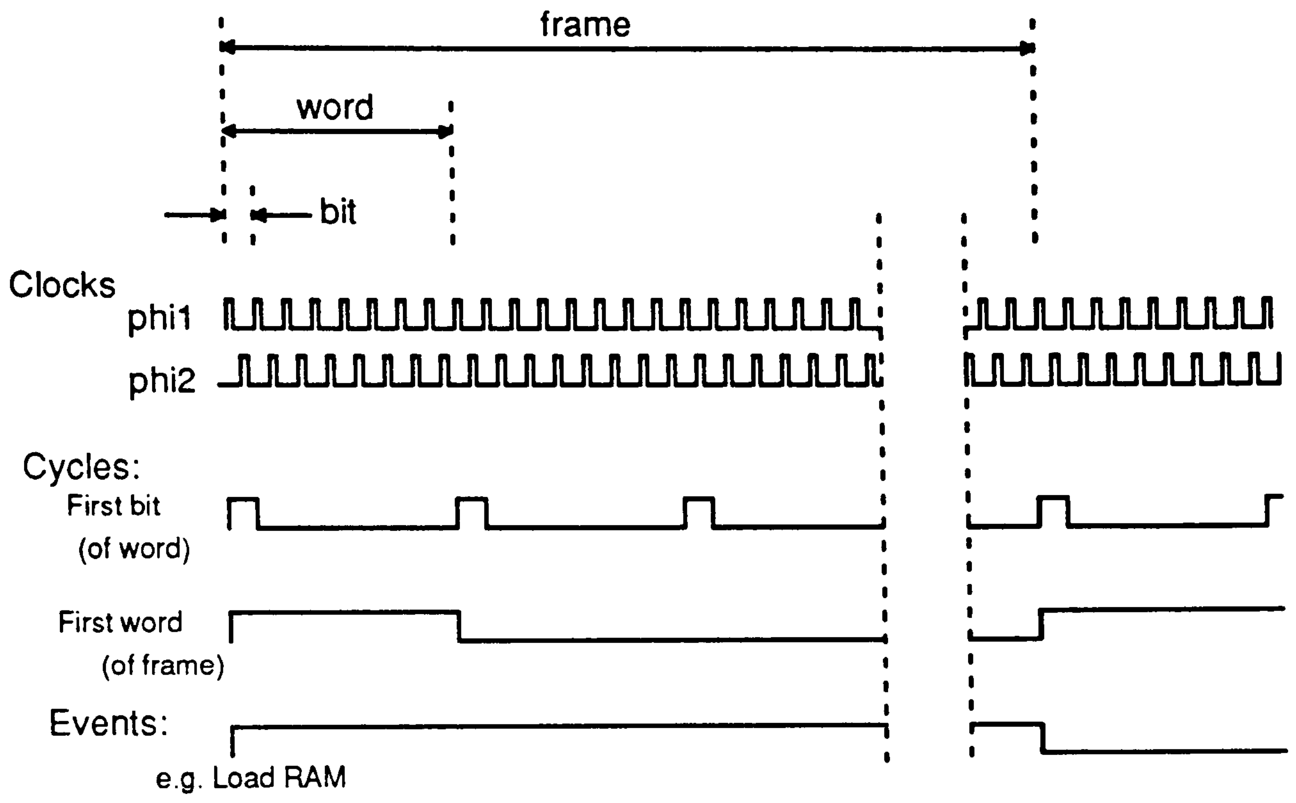
Formatting of multiple–precision words

Figure 1.3

Comparison with Bit–Parallel Systems

As bit–serial components operate on incoming data a single bit at a time rather than on the whole data word, they tend to be smaller than their bit–parallel counterparts. As bit parallel parts need to replicate logic in order to operate on each bit of the data word, the ratio of the area taken by the bit–parallel part to that taken by the bit–serial part is roughly equal to the wordlength. On the other hand the data rate of the bit–serial system will be slower by a factor equal to the wordlength.

The usual method of implementing a bit–parallel system is through the use of a datapath. This method has the advantage of being very flexible, but throughput is limited as computational resources are limited. With an ALU (arithmetic logic unit) all arithmetic operations must take place within this element. The processing algorithm must queue data and wait for the resource to become available.



Format of control cycles in bit–serial machines

Figure 1.4

The bit–serial approach is to use a flow graph model of the computation and elements are provided for each of the computational nodes in the flow graph. This gives the bit–serial system a greater computational throughput as all nodes in the flow graph are exercised, leading to high speed efficient systems. This also limits bit–serial architectures to data independent computations.

Multiplexing for Hardware Efficiency

With an iterative algorithm, we can multiplex the bit–serial processor to reduce the amount of hardware needed, when the sampling rate is less than the throughput rate. This leads to a more efficient usage of the available processing power.

The multiplexing of the serial processor can be done to various degrees, from an array type architecture with no multiplexing to a von Neumann type architecture with the maximum possible amount of multiplexing, so only requiring a single processor. The degree of multiplexing used needs consideration of both the gain in available silicon area leading to smaller systems and the reduction of bandwidth as processing speeds are reduced through multiplexing.

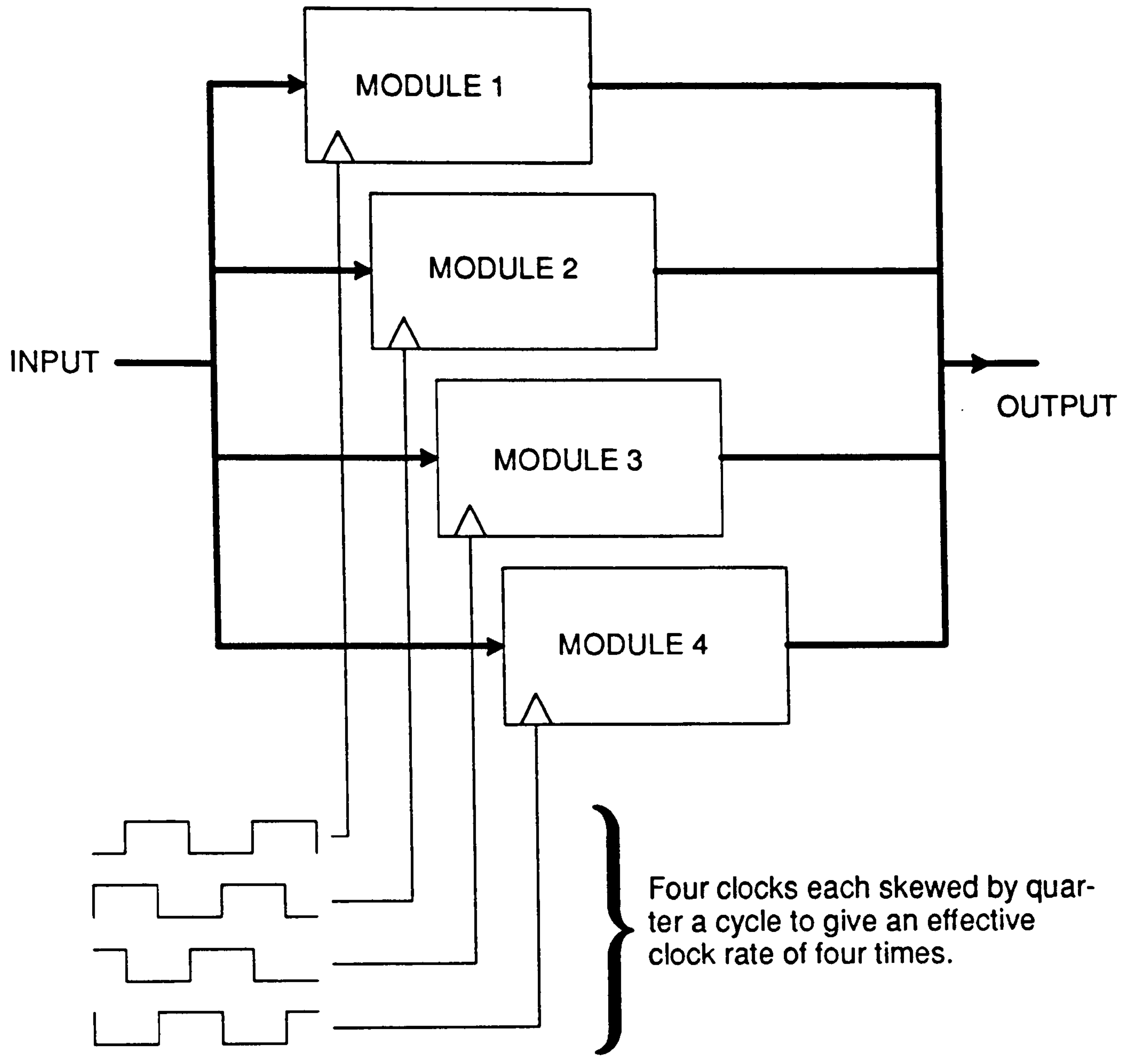
Functional Parallelism

The opposite of multiplexing is to parallel up processors so that greater speeds can be obtained. Here by parallelling up a system and sampling at times which are skewed, we can map regular algorithms into hardware which has a sampling rate greater than word rate of each of the processors in the system. Figure 1.5 shows how the scheme is implemented for algorithms which such a structure.

The scheme has been used by Powell and Irwin[7] to implement the Fast Fourier Transform algorithm using small low cost bit–serial processors. Their hardware attained very high performances with a sampling rate which was faster than the bit rate of each processor.

Signal Processing in real time

There is a demand for systems capable of processing signals in real–time, this is already possible in the audio frequency range, but in the video, systems for advanced processing are not available. One way of achieving the high processing rates essential for video signal processing is through functional parallelism of small interconnected systems. This is an application where bit–serial systems show an advantage. Their small size combined with their high throughput makes them ideally suited for parallelism and consequently high speed processing.


MODULE 1
MODULE 2
MODULE 3
MODULE 4
INPUT
OUTPUT
Four clocks each skewed by quarter a cycle to give an effective clock rate of four times.

Functional Parallelism in bit serial systems

Figure 1.5

As the architectural methodology of bit–serial is geared to fixed function, high speed custom processors, there is an application in processing signals in real time, especially in the audio range. Higher speeds are achievable giving a greater range of application than general commodity signal processing components such as the TMS320.

Interconnection of bit–serial processors is straightforward, leading to high–speed, easy to design systems. The only complicating factors are control signals which require some attention on the part of the designer.

Hardware Acceleration

Another application of bit–serial processors is in hardware acceleration. These small asynchronous processors, specific to a particular function could be attached to existing bit–parallel systems and thus retain versatility and familiarity. They can be attached through a bit–serial to bit–parallel interface which can be incorporated into the silicon–compilation schema through the use of a design–frame, described in a later chapter.

A design frame is a consistent piece of hardware which incorporates standard timing and signal conventions, by including the serial–parallel interfaces into this boundary between systems, serial units and parallel units can cooperate without signal contention. This makes the whole exercise of designing a hardware accelerator straightforward and with less likelihood of a mistake.

1.4. FIRST

This section describes some more details of FIRST, which are built upon later in the thesis in trying to extend the compiler. FIRST lends itself to automation as the scheme is based on a small library of elements built up to make systems hierarchically.

Hierarchical Design

Hierarchical design is the method by which larger system components can be built from smaller system components. It is a very powerful design strategy which simplifies the design of complex systems.

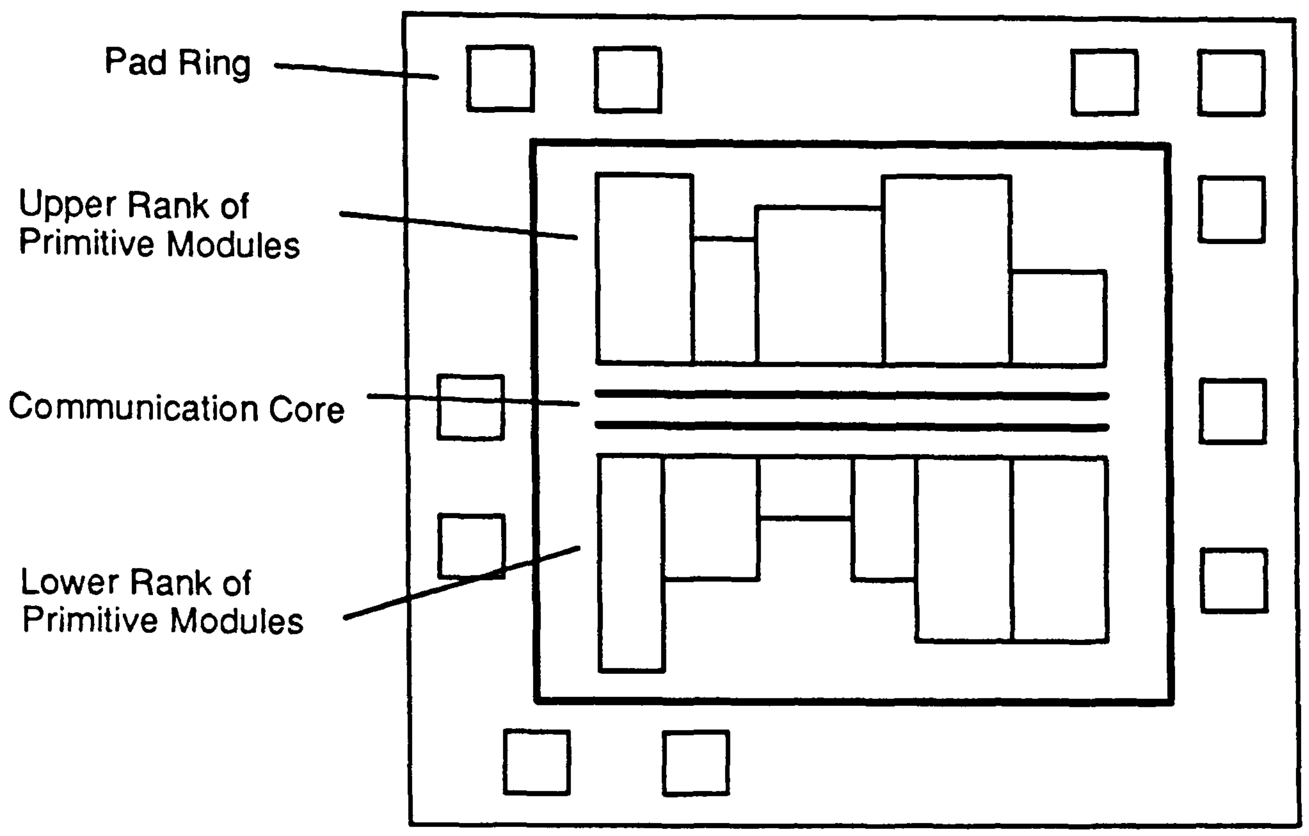
The bit–serial methodology lends itself to a hierarchical design methodology as all operators have similar data and control communication conventions. This means operators built from lower level elements will adhere to the same signaling conventions as those lower level operators.

In FIRST, three typed hierarchical levels exist which mark physical boundarys of the level. These are the SYSTEM level, this is a complete system being developed, it is used for simulation purposes, the CHIP level, this is where partitioning of the system into single chips takes place, it is at this level that floorplanning and routing are needed, and the PRIMITIVE level, this consists of the basic operators of the system, specific layout methods for each primitive are used.

Floorplan and Routing

A constrained floorplan is used within FIRST for the layout of primitives in chips. The primitives are pieces of full custom layout, or else are composed of full custom *leaf cells* through individual software assembly procedures. The leaf cells are dense pieces of layout and connect by abutment to form the PRIMITIVE.

The assembled PRIMITIVEs form rectangular sections of layout, with communication signals on one side of the rectangle, preferably a shorter side, as this is placed next to the communication channel, which determines the length of this resource. An example of the floorplan schema is shown in Figure 1.6, which has been called 'Manhattan Skyline' because of the similarity it bears to a cluster of skyscrapers and their reflections in the water. It comprises a central communications channel, which has two rows of bit–serial primitive modules on either side.

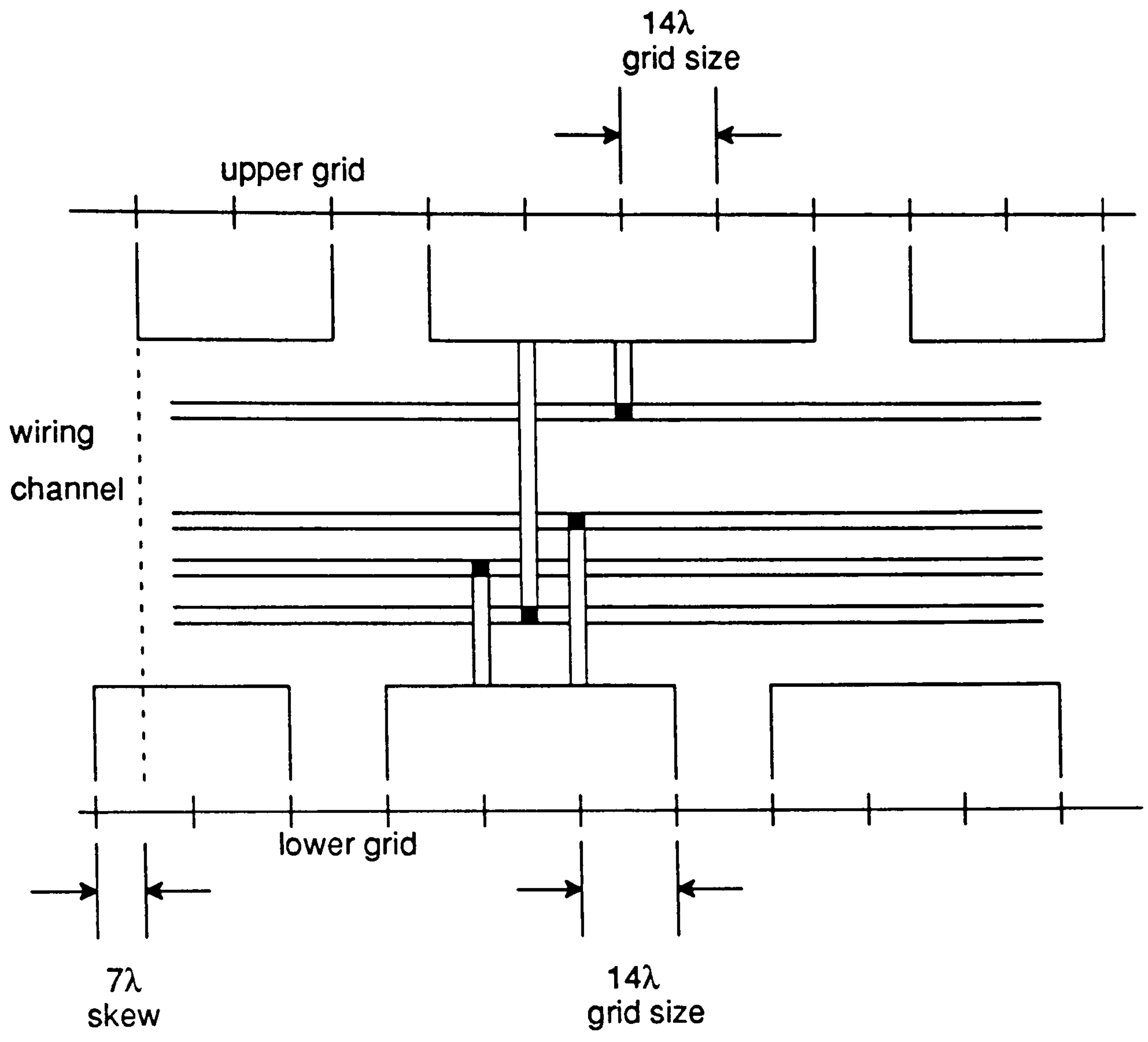

Pad Ring
Upper Rank of Primitive Modules
Communication Core
Lower Rank of Primitive Modules

Typical Floorplan in FIRST

Figure 1.6

As the communication between PRIMITIVEs is bit serial, only single wires are required for data communication between the PRIMITIVEs. This leads to a small area communication channel. The two clock phases, power and ground are distributed along either side of the communication channel and around the perimeter of the chip. The buffers to the PRIMITIVEs are allowed to spread beneath these lines in the communication channel.

For channel routing, the allowable positions of input and output ports are quantised to be at even multiples of the wire pitch of the vertical wiring layer. In FIRST a standard wire width of 4 lambda is assumed, where lambda is a relative unit of linear dimension, and a wire separation of 3 lambda, giving a total pitch of 7 lambda. Thus input and output ports can be placed at positions separated by twice this value, 14 lambda. The routing scheme is illustrated in Figure 1.7, the 7 lambda skew, between the two sides allows an arbitrary connection of signals to the wiring channel.



Routing in FIRST

Figure 1.7

1.5. Limitations of Silicon Compilers

The main problems with a silicon compiler are its limitations. These tend to be of the form :–

1. Limited Application – restricted only to finite state machines, datapaths, etc.

2. Limited Process – restricted only to a single process.

3. Limited Technology – restricted to a single technology, e.g. NMOS.

4. Too General – the compiler is too general and gives very little advantage over hand layout combined with a structured layout methodology.

5. Limited efficiency and performance of generated circuits.

These problems have tended to restrict the use of silicon compilers, where silicon area is an expensive resource, meaning that time spent optimising layout can pay off, especially for integrated circuits produced in mass quantities. A designer may well choose to use a full custom approach to his design, where the compiler is too inefficient or does not exist, for the layout of the work. Many tools such as routers, compacters and floorplanners of full custom modules are useful for this type of work and can be incorporated in manual layout design packages.

On the other hand, many integrated circuits are now application specific, meant to replace boards of discrete components, and may only be produced in hundreds. Here the time for design may well be the overwhelming cost consideration. Silicon compilers offer a rapid turn over allowing impressive design times. The area usage of layout produced by automatic compilation is approaching that of full custom; but, even when minimum silicon area is of prime importance, automatically compiled layout may be favoured, because of the smaller likelihood of a design error.

Limitations of Cell Libraries

FIRST possesses all the features of a usable design tool, but it has a severe limitation which prevents its wider adoption, that is the total reliance on an underlying cell library. The process library used in the FIRST compiler was a 5 micron NMOS process, which has by now become outdated. Although it is possible to complete a new cell library for new processes, this is a major undertaking involving too high a use of resources in a University environment. The amount of work involved in maintaining such a library is easily seen by examining such a library, for instance that by Newkirk and Mathews[14] which is for a particular NMOS process. Some of the practical aspects for a CMOS cell library are covered in [15].

Each time layout for an updated set of process rules is required, the whole cell library artwork must be redesigned. Each library element is a piece of full–custom layout, on which much time can be spent on optimising to achieve efficient results.

Statement of Problem this Thesis Addresses

A major limitation of FIRST is the process bound cell library, preventing its use with new processes without the layout of a new cell library. This is the problem which this thesis tackles. It looks into the automatic generation of the cell library using symbolic layout techniques.

Process portability would be achieved since the cell library could be rapidly updated to reflect advances in newer higher density process rules when and as provided by silicon fabricators. Furthermore, flexibility could be added to the compiler, since new primitives could be added to the compiler on demand without the tedium of becoming involved with full–custom layout.

The problem of process portability of the cell library of the FIRST silicon compiler has been identified. New PRIMITIVEs require the layout of new full custom leaf cells deterring users from including their own specific components. This thesis looks at one way in which process portability and flexibility can be added to a compiler such as FIRST through the automatic generation of process portable layout.

Circuit Techniques

Circuit techniques which enhance the throughput of bit–serial systems were developed during the layout automation project described in this thesis. We sought to include these techniques in any further library to FIRST to enhance performance of the systems produced.

In most cases these techniques do not affect our layout strategy as they can be incorporated into the boolean description of the constituent logic blocks. However, layout automation proceeded with the inclusion of these techniques in mind and have affected the layout software, so they are described in a later chapter.

The architecture of the multiplier which includes these techniques and was used as a layout evaluation vehicle is also described.

Chapter 2

Methodologies for the Layout of VLSI systems

2.1. Introduction

The first chapter covers the background and applicability of the silicon compiler FIRST. The major problem of this compiler through limitation to a single set of process rules was identified. A means of achieving greater process portability is needed and this chapter surveys some layout methodologies which could form a suitable base for the automatic generation of the cell library in FIRST.

A structured layout methodology is a means by which a system description can be translated to silicon mask information in a rapid manner following a regular design procedure. Characteristics of a good methodology are regularity of structure, portability between different processes or technologies and a consistent algorithm lending itself to automation.

The description of a digital microelectronic system can take many forms, as a truth table or boolean logical expression (low level), as a flow diagram or flow chart or as a high level software description (structural or behavioural). Translating this to layout is an arduous task which takes valuable time and effort to perform. In the competitive world of electronics, there is great pressure in producing working circuits from a designer's description of a system in as short a time as possible and to use the latest processing technologies to gain the benefit of higher performances and greater integration densities. A layout methodology offers savings in time and effort in the layout task and also offers a degree of process independence. Particularly useful is the ability to automate the process, alleviating completely the need for manual layout.

2.2. Programmable Logic Arrays

If the digital circuit to be designed can be specified as a small finite state machine, then programmable logic arrays (PLA's) provide a rapid means for a designer to implement such a system and obtain a working circuit. Much effort has gone into the problems of logic minimisation, [16], [17], [18] and to PLA folding techniques to achieve smaller structures. The layout of a PLA is an easily automatable process and is well understood. The PLA minimisation problem is NP hard, meaning that no algorithm exists which produces an optimal solution in a time proportional to a polynomial function of the size of the PLA. PLA synthesisers thus use heuristic algorithms which give solutions in a practical time using heuristic algorithms, which will settle with a good solution rather than producing the optimal solution.

There are many structures for PLA's to improve performance and increase logic density. With CMOS technologies, where the direct static approach is inefficient on counts of area and performance, various dynamic implementations exist[19].

The PLA structure is highly regular whatever implementation is adopted. PLA's can be built up from a small number of easily changed elements which can be built up to form a complete PLA. Thus a PLA implementation has advantages in designing a maintainable and easily automatable system :–

– Changing to a different process is easily accomplished by changing a base library of the basic elemental blocks.

– PLA structures can be generated automatically from a logical boolean description.

Despite these advantages, PLA's suffer from a major drawback which prevents their use for a system of any complexity. A PLA will tend to be large and uncompetitive with other forms of layout. A circuit is implemented as two levels of gates which although sufficient to formulate all boolean expressions may not be the most, or even an, efficient method of implementation.

The PLA is still an important component when incorporated into larger silicon systems such as the control sections of a microprocessor, but it would be naive to implement all digital systems as PLA's. In particular in the case of bit–serial components, a PLA implementation is inefficient. A single PLA would be too large and an implementation in terms of multiple PLA is still not as compact as other layout methods and introduces the problem of partioning.

2.3. The Semi–Custom Approach

The semi–custom approach[20] allows the designer of a system a greatly simplified route to the layout of an integrated circuit. Gates or cells are pre–formed conceptually, if not physically, and the job of the user is then to specify the interconnection between elements. Software or the vendor of the gate array library will then route these interconnections between gate array elements.

Commercial vendors of silicon processing capability offer traditionally two methods of implementing a semi–custom silicon system, the gate–array and the standard cell library. The vendors usually offer the latest in processing technology, as the uncustomised silicon slices of a gate–array are likely to be one of the first mass produced articles using a new process.

Gate Arrays

Here the basic uncommitted components are preprocessed onto the silicon slice before customisation takes place, where only a few masks (typically 2–4) are used. The designer has access to the latest technology as the vendor can produce the uncustomised silicon slices with pre–processed gates in large numbers and can easily update these low level components to take advantage of the latest or specialised processing capabilities.

The designer needs only to specify the interconnection of these components by way of a netlist. Software which performs most or all of the routing of the user supplied netlist on the uncustomised array exists and is used in a commercial environment.

A development of the gate array concept is a layout style termed '*sea–of–gates*'. With this style high transistor densities are achieved by packing transistor gates as close together as the technology permits, but where not all the transistors can be used, allowing impressive numbers of gates to be fabricated on a chip; over 100K gates is typical. The philosophy of the approach is to fabricate many more transistors than are needed by the designer as not all the transistors are available, since some will need to be sacrificed to allowing routing over them. Developments on this concept, employing a hierarchy in the chip array and design has resulted in the *gate forest*[21], [22] concept, which has extra unique features. These approaches require state of art

CAD approaches, as the layout of these massive arrays cannot be undertaken manually without a great deal of trouble.

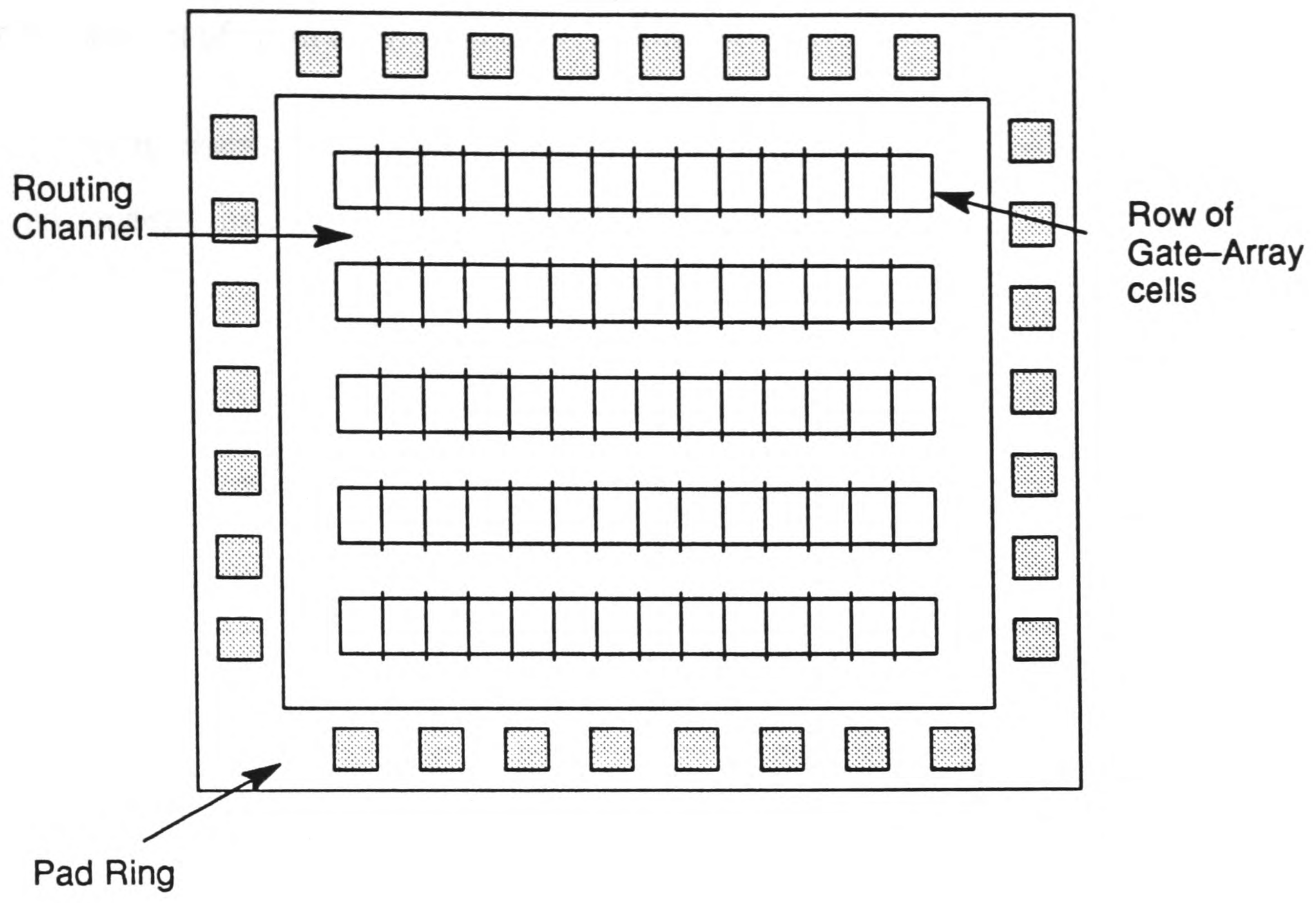
Another variation, core–custom, where the uncustomised gate–array chip incorporates a well known VLSI component such as a commercial microprocessor, leaving a significant portion of chip area for the designer to include their own logic, increase the attractiveness of the approach. A designer can take his board level solution and port it over to the gate–array chip using components and techniques he/she is already familiar with.

Standard Cell Approach

Unlike gate arrays, all layers of the mask set are customised and the silicon has to be processed in one go. A vendor of this type of semi–custom process will supply a catalogue of components, which are pieces of full–custom layout, thereby being efficient in terms of area and performance.

Automatic placement and routing software [23], [24] for these cells which does the job wholly or partially are used by vendors to connect user supplied netlists. It is up to the vendor to update their library to new processes as and when they become available.

The floorplan of a gate array takes the form of a row of gates separated by routing channels. Two computing problems arise, that of the most efficient placement of the gates and that of routing the channels. Figure 2.1 shows a typical gate array floorplan. Placement algorithms[25], [26] determine where to place the cells for the good routing results, after which routing algorithms[27], [28], [29], [30], [31], [32] route the wires for signals around the chip. Good placement is important as this can affect dramatically the routing results produced with the router. The routing will be bad, even with an efficient routing algorithm if the placement of the gate array cells is poor.



Gate Array – Typical Floorplan

Figure 2.1

Software such as SOLO supplied by ES2, which is an update of the earlier Chipsmith[33], which is mentioned in a later chapter, produces mask information for all the layers of what they call 'optimised' gate arrays. This doesn't have the advantage of requiring only the customisation of the metal layers in a traditional gate array but produces more compact layouts as routing channels can be of an optimum size. The cost of making masks is unimportant in this case as ES2 uses a direct write on the silicon wafer using an electron beam in their fabrication process. The technique allows quick updatability to new process rule sets[34] allowing the software to be used for processes supplied by other fabrication processes.

2.4. Structured Layout Methodologies

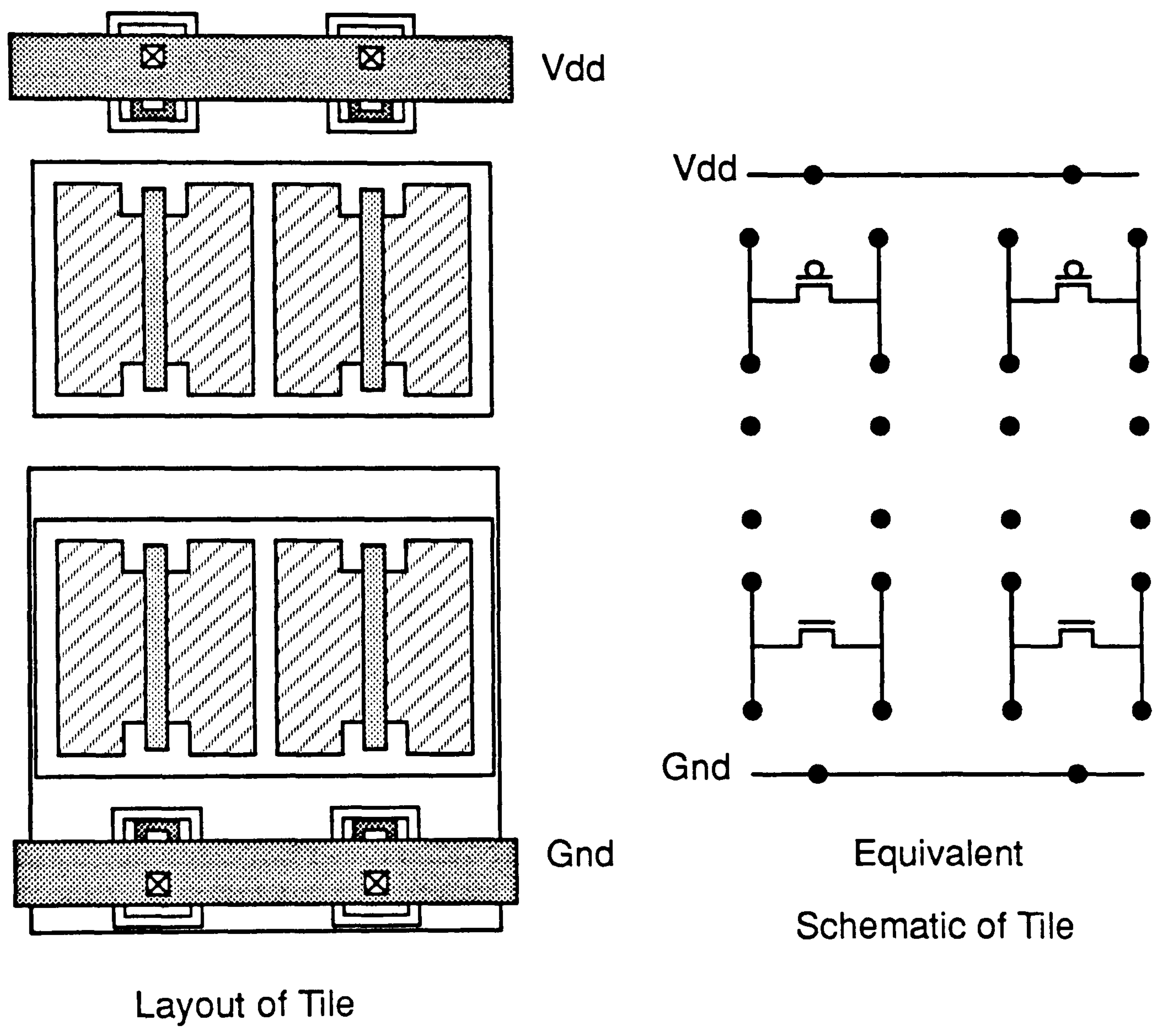
If the full custom layout route is adopted then full use can be made of all mask layers to produce the most compact and efficient layout possible in practice. The problem with the straightforward use of full custom layout is that it is a difficult and time consuming process requiring skill and patience to do well. It is also non–portable. If a new process becomes available using a new set of process rules, the complete layout has to be repeated.

What is needed is some way to abstract the details of the design rules, leaving the topological and connectivity information supplied by the designer, so that it can be used whatever design rules are given.

This is what is done with a structured layout methodology or symbolic layout system. The two terms will be taken as synonymous throughout this thesis.

Tiling Scheme

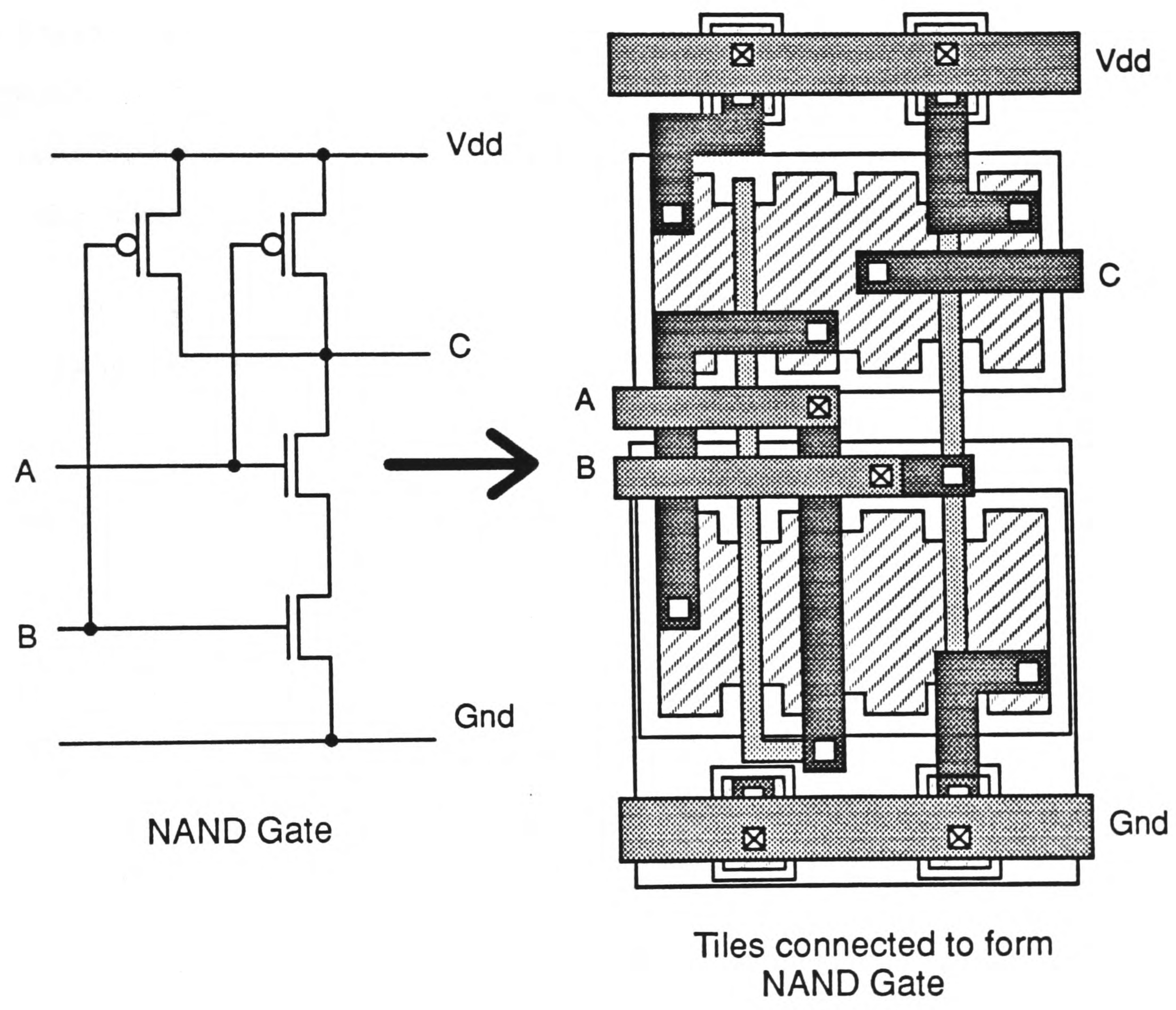
This is a layout method that was developed at Edinburgh University. Fourth year electrical engineering students have already used the system with success in their projects, on low cost computers, using CAD software written at the University.



An Uncustomised Tile

Figure 2.2

To reduce the design time of a circuit, standard tiles are used, Figure 2.2 shows an uncustomised blank tile and its equivalent schematic. There are four transistors in a tile; two p–type, near the supply line (Vdd), and two n–type, near ground. Connections are made using all layers to form a completed circuit. Figure 2.3 shows an example of how these tiles may be connected to form a nand gate.



Tiles connected to form a Nand Gate

Figure 2.3

To change to another process only the tiles need be changed, connections between tiles can be stored in a library and reconnected automatically from the library or, without much pain, manually. It is easier to lay out circuits and change process using this tiling scheme rather than trying to perform the lay out completely in a full custom fashion.

Weinberger Arrays

As long ago as 1967, the benefits of a regular layout style were understood. Weinberger[35] proposed an array of nor–gates to implement sections of digital logic. The array differs from a PLA in that, logic is constructed from multilevel nor logic, whereas only two levels are used in PLA's. Weinberger arrays are thus more general, but more suited for NMOS circuits for which they were originally developed, (NMOS nor–gates can possess large numbers of inputs without a great degradation in performance, unlike CMOS equivalents).

The circuits can quickly be moved to another process because the structures are very regular and hence the redesign of layout can be achieved with ease.

Gate–Matrix

The gate matrix[36] structured design methodology was originally developed at Bell Laboratories[37] and has found the greatest favour for use in the layout automation system described earlier. It can be viewed as a generalisation of the Weinberger array, since it has a similar structure. Unlike the Weinberger array, the types of logical circuits that may be laid out are limited to multilevel nor gate implementations.

The major characteristics applying to gate matrix are:

– Polysilicon runs only in one direction and is constant in pitch.

– Diffusion runners exist between polysilicon columns.

– Metal runs in both directions and is constant in width except for power busses.

– Transistors exist only on the polysilicon columns where diffusion rows intersect it.

These features give the style a regularity which is important from the standpoint of computer automation. Also the layout density is comparable to full custom layout. This may not be so clear when comparing small blocks of layout, but in larger pieces of layout, the advantages of the regularity enable easy interconnection of smaller relatively independent blocks. A major benefit of this style is the ability to update to a new set of process rules quickly and without much effort. Pieces of layout can be stored in the symbolic form, from which actual mask data (incorporating process rule dependencies) can be generated.

The style is well adapted for computer aided design[38] as well as a manual design style [39] on low cost terminals. The style has been used for the layout of a large section of control logic in a 32– bit CPU[40] demonstrating the usefulness and utility of the layout style.

Algorithms for the automated generation of gate matrix layout exist, [41], [42], [43], [44], [45], but as the optimal layout of circuits using the gate matrix style is intractable, the algorithms are heuristic by necessity. It is also possible to extract netlists[46] for simulation with SPICE directly from the gate matrix layout.

The gate matrix methodology is very practical and usable. The style has the following features which we are looking for :–

– A Structured Methodology: allowing the possibility of computer automation and interconnection of independently generated blocks.

– Competitive Layout Densities: the achievable layout density is competitive to that produced by hand.

– Process Updatable: mask information can be generated quickly from a symbolic form which is process rule independent.

– Demonstrated Usefulness: the layout style was developed by AT & T laboratories who have successfully laid out large sections of logic[47] using gate matrix.

Virtual grid Layout

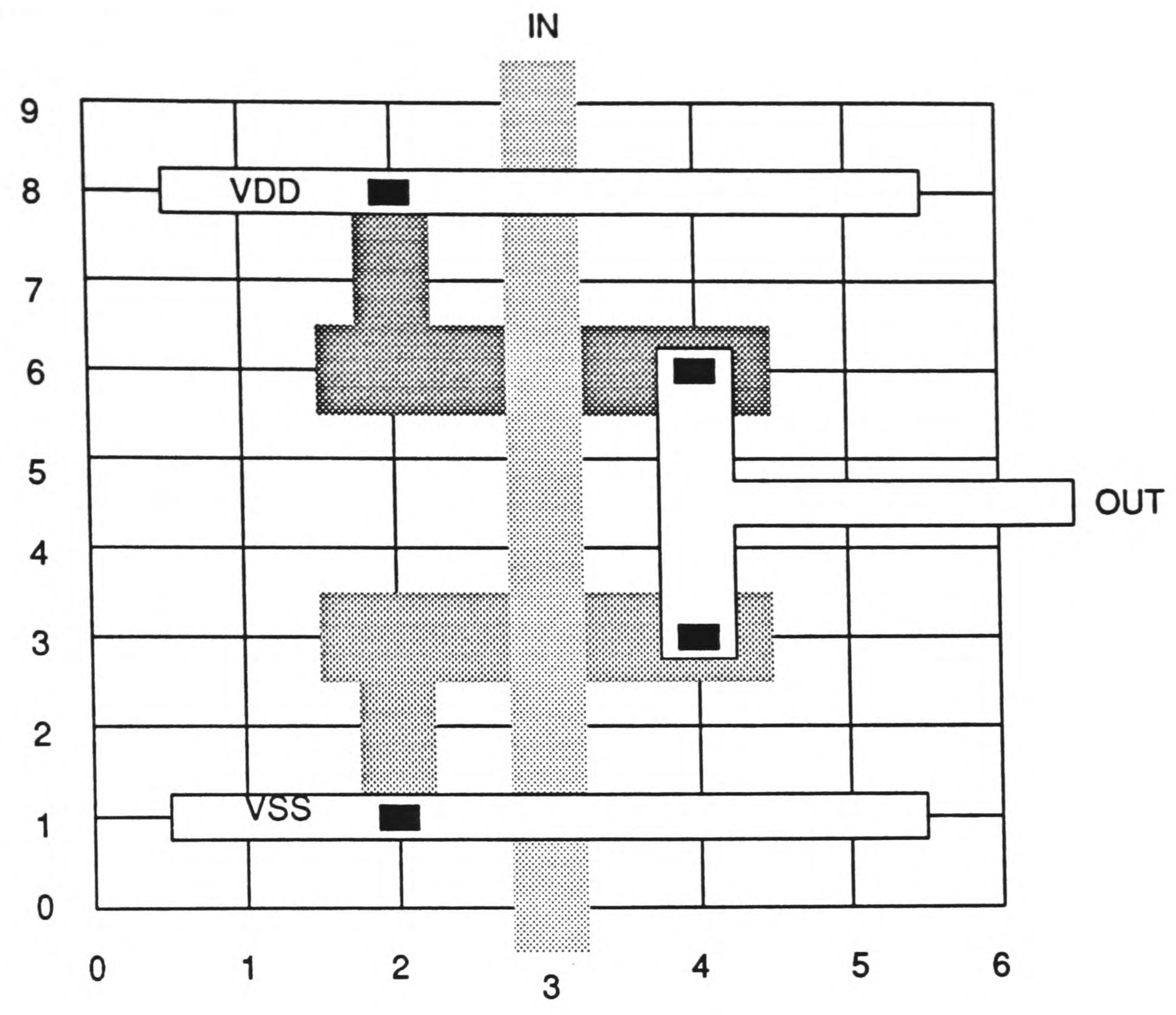
The virtual grid [48] scheme also known as the 'sticks' approach offers another method of achieving design rule independent layout. The method allows the designer to specify the topological placement of transistors and interconnections. These primitive elements are linked both electrically and topologically through the concept of a *virtual grid*. This relative placement network defines layout topology without specifying the physical distance between elements. Compaction and the production of design rule dependent layout is performed by software on the virtual grid data structure.

A design system using the virtual grid concept is the tool MULGA[49] used for design rule free symbolic layout and verification of MOS integrated circuits.

Figure 2.4, shows a virtual grid representation of an inverter. Absolute positions are unimportant, it is the relative positions of components which is important. The symbolic language used within MULGA contains functional and topological descriptions but no explicit size or spacing information. To translate the symbolic design description into physical mask information requires assignment of physical coordinates to the symbolic grid positions. This is done in two stages. In the first the cell is shrunk to its minimum physical size as determined by process design rules, keeping objects aligned to the grid, in a compaction phase, after which layers can be fleshed. The second stage involves expanding the cell to satisfy global interconnection requirements allowing adjacent cells to abut.

The compaction phase is performed using the technique of *virtual grid* compaction. Elements are constrained to their original grid positions, but the grid is allowed to contract or expand in a non–uniform manner. Compaction proceeds by starting with the X–direction, by comparing grid lines and elements on them for any violations of design rule in that direction ignoring oblique interferences. A minimum spacing between each grid line is used, allowing the grid to vary, which avoids violations. The process then takes place in the perpendicular Y–direction this time taking all oblique violations fully into account.

In this manner mask data can be prepared for many sets of different process rules starting from the same description of the circuit. The designer is left only with the task of specifying the topological layout of his/her design, the inclusion of process dependent data can be left to software.



An Inverter on a Virtual Grid

Figure 2.4

Another design system is STICKS[50], developed at the HIRST Research Centre of GEC. The system consists of a symbolic editor and flesh and compaction software, also using a virtual grid. Extraction of the circuit can be done from the symbolic network and a SPICE file generated.

A similar concept based on the virtual grid symbolic scheme is that of generalised CMOS[51] which includes automatic well generation and is valid in four classes of CMOS technology, P–well, N–well, twin–well and silicon on insulator.

2.5. Knowledge Based Techniques

The use of artificial intelligence or knowledge based techniques have become increasingly popular and have been applied to the design and layout of microelectronic systems. They look promising as they can also operate in a process independent way, generating the layout as required. They rely on a set of base rules which specify what can be done to the state of layout at that moment. The activation of a rule can be looked upon as performing a specific optimisation.

Knowledge based systems try to mimic the human designer, and the set of rules are obtained from experts in the field in which the system is meant to operate. It is because of the way they operate that they can be used to solve problems where algorithms for a solution are difficult to produce.

A knowledge based expert system has been described[52] which is able to synthesise the datapaths of CPUs and has successfully done this with the 6502 microprocessor and the IBM SYSTEM/370 with favourable reaction from expert designers.

There have been developments with the automatic generation of VLSI circuit blocks using knowledge–based techniques. TALIB[53] is a program implemented using the general purpose production system language OPS5. It accepts a description of an NMOS circuit in net-list form, which is to be laid out as input and produces the description of mask geometry as output. The technique is not limited to NMOS technologies. Another program is TOPOLOG-IZER[54] which is an experimental expert system for the design of CMOS cells containing fewer than 100 transistors. It uses as input a transistor connection description and a description of the ports required at the boundary of the cell. Its output is a symbolic description using a virtual grid. The tool MULGA[49] can then be used to form the mask information.

2.6. Manual Layout

Manual layout despite being the most difficult to do quickly is also the most flexible. In some cases such as the design of analogue circuits, layout automation is still in its infancy and manual layout tools must be used. With digital circuits the densest layout achievable will remain manual full custom, as the layout can be optimised to make full use of the design rules. In many cases it can be justifiable to use a manual editor at the final stage of the layout process in order to optimise certain cells, although without careful use it is not recommended, as functionality of an otherwise correct layout can be destroyed.

Although manual editors allow the most basic operations to be performed on a piece of layout, they are not basic tools. They can possess sophisticated functions within them. *Caesar*[55] is a basic mask editor, which was used throughout our project for layout purposes, allowing operations on rectangular areas for 'Manhattan' geometry layout. It includes the facility to edit hierarchically and possesses features such as marking points or regions to allow work on large area layouts. It has been updated as *magic*[56] and uses a better data structure internal to the program allowing algorithms which operate on the layout to be much quicker. Design rule checking is done by a separate program, which means that a user must correct these errors after completion of an area of layout. It has the disadvantage that an error may be difficult to correct when surrounded by remaining layout, but does allow the editor to be used to layout any arbitrary shapes, making it easy to use with any set of design rules.

The Racal–Redac ISIS workstation which is dedicated to the function of integrated circuit layout, on the other hand employs an incremental design rule checker which checks whilst layout is in progress. It stops a user form laying out any piece of logic which would include a design rule error.

These tools make the task of full custom layout far more manageable. They may include useful routines within the editor software, such as a compacter or router, but although giving a designer maximum flexibility suffer in the amount of time and effort needed in using them for any large digital design.

2.7. Other Work on Silicon Compilation for Signal Processing

An alternative approach to signal processing hardware has been taken by Pope and Brodersen[57] who use parallel processing elements in their implementations. They assemble systems from macrocells, logical units capable of performing operations at a high level of complexity and not needing many signals for external communication. Typical macrocells which they have identified are:

1. Processors, consisting of data memory and a pipelined arithmetic unit.
2. Control sequencers, consisting of counters, ROM and PLA.
3. Control Modification units, such as address arithmetic units and conditional logic operators.
4. Buffer memories and external input/output interfaces.

They have managed to produced an automated layout system[58], [59] using this methodology. By starting with a symbolic description of the system they wish to construct, then using software consisting of a tiler, floorplanner, and a tool for global placement and routing, which is capable of supporting a wide range of technologies, they produce a system consisting of a number of pipelined and microprogrammed signal processors operating in parallel.

The workers on the CATHEDRAL project have produced a sophisticated piece of software capable of generating complex signal processing systems. CATHEDRAL I employed a bit serial cell library [60] of components for use in the system and consisted of software which constituted a complete environment[61] for the design of VLSI signal processing systems. CATHEDRAL II [62], [63] uses bit parallel components and uses multiple processors to achieve performances demanded from the systems designed using the software. CATHEDRAL III allows the use of systolic arrays to achieve very high performance systems. The specification of the system is given in the language SILAGE, data path synthesis tools produce layout from this specification.

A symbolic layout scheme[64] is also used within the system to design a process tolerant cell library. CAMELEON[65] is the name of the symbolic layout software. Cells are specified in a process rule independent symbolic form and can be edited with a sticks editor (LUDIEC). A compaction program (SPACE) compacts this representation and produces the cells suitable for a specific set of process rules, after consulation with a process rule description file. The tools combine to form a powerful and flexible system.

2.8. Conclusion

Structured design methodologies allow the rapid generation of the layout for cells. Some methods are restrictive in the type of circuit they cater for and are efficient for this type of circuit, for instance PLA's for implementing finite state machines. All the methodologies allow a rapid update when process rules change, making them suitable as a basis for an automatic cell library generation scheme.

The gate matrix approach is a layout methodology which allows rapid update to new process rules and also has considerable flexibility. The layout procedure lends itself to automation through its regular style and is the style preferred for use in our cell library generation scheme. The automatic generation of gate matrix layout and its assembly to form complete components is explored in the remainder of this thesis.

Chapter 3

Circuit Techniques and Architectures

3.1. Introduction

Novel circuit techniques have been developed at the University to increase the throughput of bit–serial systems and reduce the size of their implementations. These techniques we wish to include into future cell libraries of FIRST. The techniques alter the generation of the layout to some extent so they are described in this chapter.

A multiplier using some of these techniques was laid out using a gate matrix style in order to evaluate both the gate matrix layout and the circuit techniques. The layout and results of this exercise are described in a later chapter. The multiplier uses an architecture which doubles its throughput relative to a normal serial–parallel multiplier and this architectural device is also described.

The techniques and architectural devices used to increase throughput and layout efficiency covered are :–

– single phase clocking.

– mixed tree logic.

– twin pipe performance enhancement.

– architecture of the serial/parallel multiplier.

3.2. Clocking Strategies

This section looks at various clocking strategies and explains the merits of the single phase clocking strategy developed and used at the University.

Synchronous and Asynchronous Operation

For any logic circuit, which is not entirely combinatorial, a notion of time is needed. Such logic circuits function as finite state machines which possess at any given moment a definite state, which evolves into another depending on the inputs to the circuit. A form of storage is required to hold the state, in the way of latches or feedback loops.

If this storage is performed by way of feedback, no clock is necessary and the circuit is termed *asynchronous* . Asynchronous circuits need great care and analysis in their design as they are vulnerable to a large number of complex timing hazards which can prevent them from functioning as intended. They have the advantage of working at their inherently fastest rate and also of being able to work without global signals. These considerations are becoming more important with VLSI circuits where high speed operation and local communication are desirable. Asynchronous circuits with these aspects in mind are gaining increasing popularity as *self–timed* circuits[66].

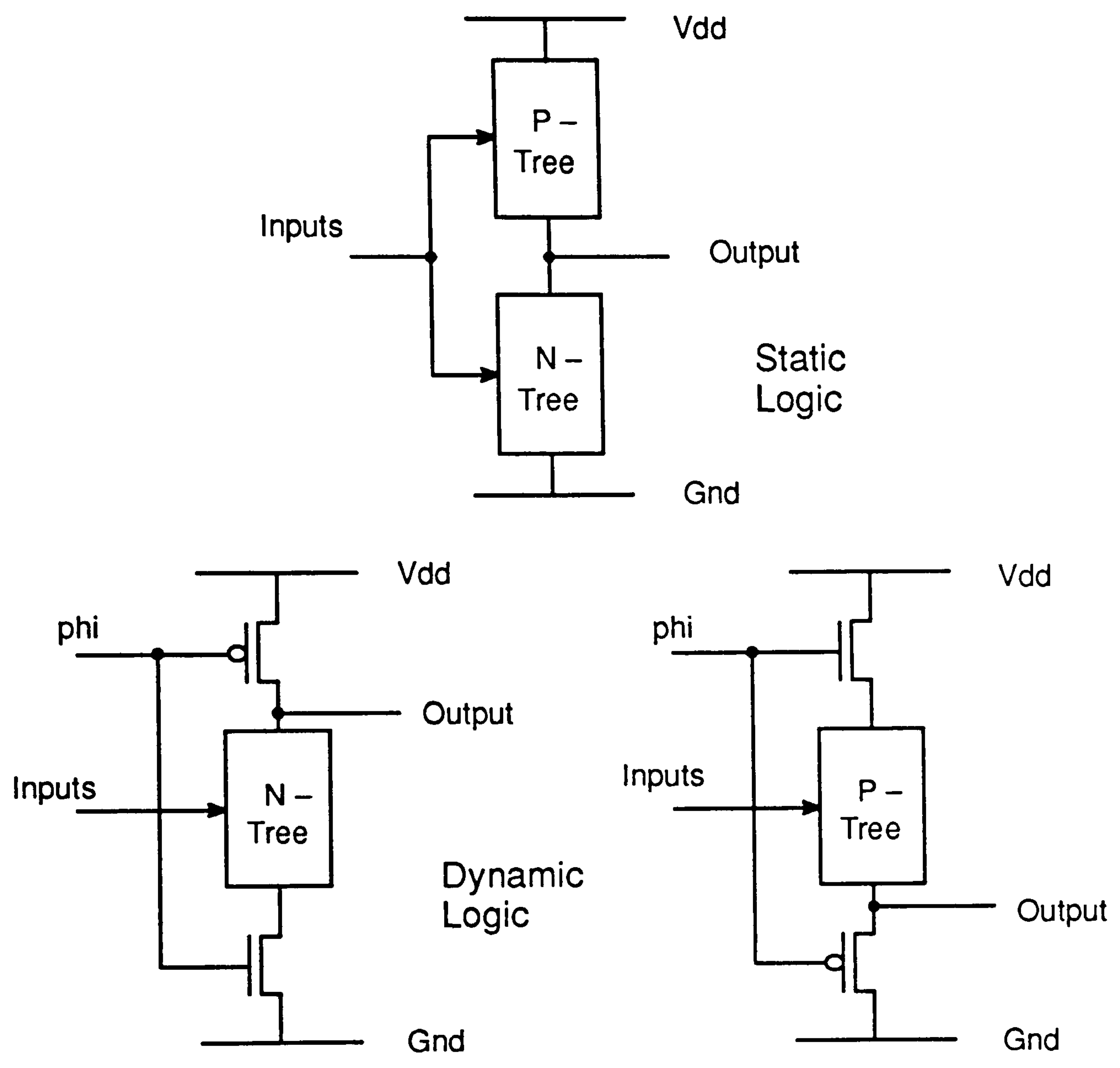
Another form of operation is provided by *synchronous* circuits, here clocked latches are used to store the internal state of the circuit. In such circuits, the timing is coordinated throughout by a global clock. Timing considerations are greatly simplified as logic transitions are synchronised with the clock edge. This type of circuit is presently the most popular and universally used in digital circuit design. The clocked latches can either be implemented dynamically or statically.

The design of both types, synchronous and asynchronous, is well established and can be found in many books[67], [68].

Dynamic Circuits

With static CMOS circuits, there is a redundancy of circuitry. For the n–type pull down tree, there is a dual p–type pull up tree, with the complete logic function being performed by both trees. This leads to increased area and greater capacitive loading as there are twice as many transistor gates connected to each input of the circuit block.

Dynamic circuit schemes eliminate this redundancy by removing one of the circuit trees. A pair of clocked transistors replaces this circuit tree, which on each clock cycle charges the output node. This node is conditionally discharged depending on the inputs to the remaining circuit tree, Figure 3.1 .

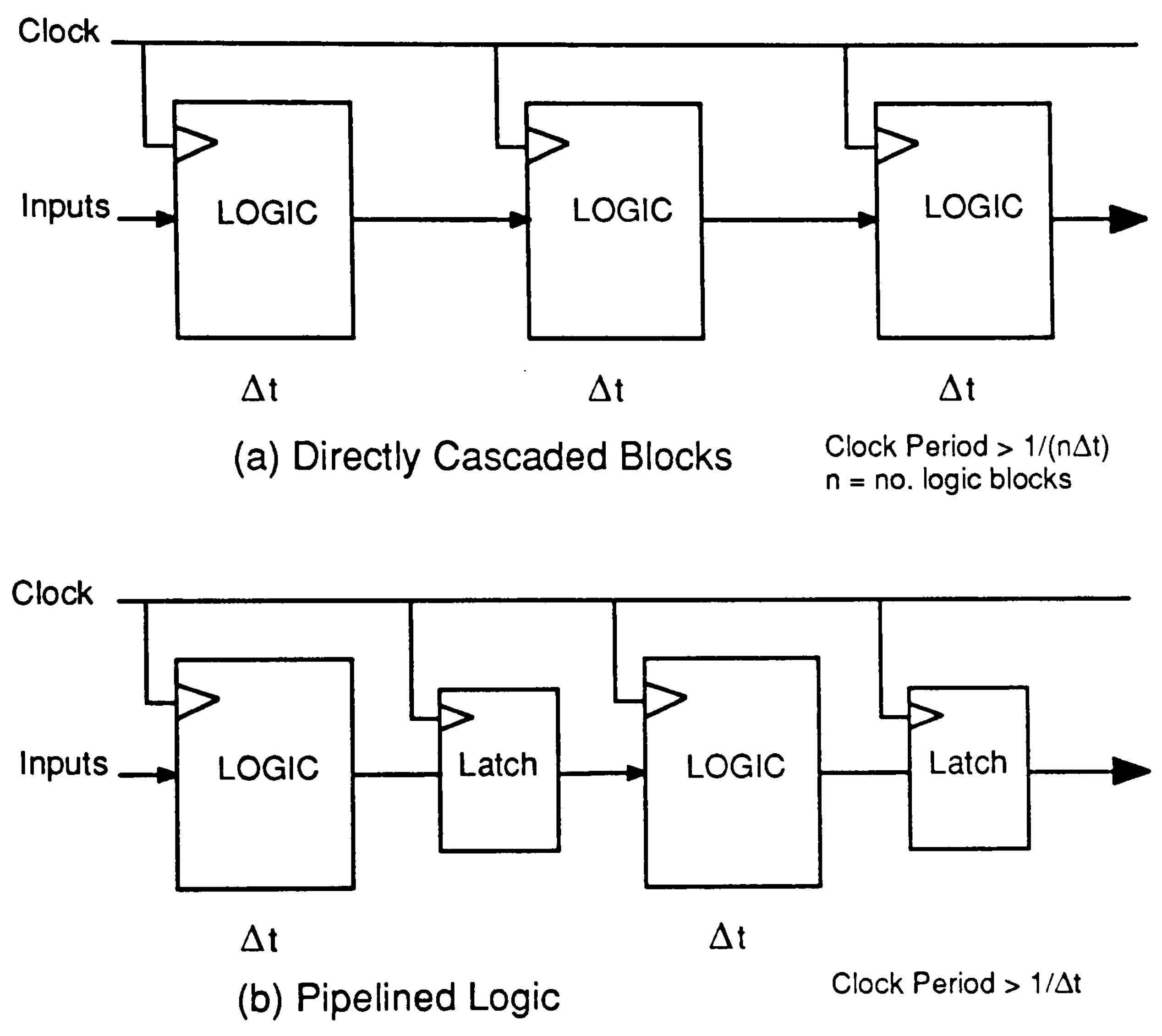


Static and Dynamic Logic

Figure 3.1

Two transistors are necessary to prevent a conduction path between the supply rails when the circuit is charging the logic node. In cascading such blocks care needs to be taken so that valid logic signals are transmitted between stages and this is taken care of by the clocking scheme used.

If many stages are cascaded then throughput of the logic circuitry must be reduced as the clock rate has to be long enough for the logic values to propagate through cascaded blocks, Figure 3.2(a). By inserting latches between logic blocks, we can form a data–pipeline. This allows us to increase the clocking rate, increasing the throughput, as now the longest delay suffered through logic evaluation is only between successive latches, Figure 3.2(b).



Pipelining the Logic

Figure 3.2

Clocking Schemes

We can thus see the following reasons for a clock in digital logic circuits :–

– to coordinate latching in synchronous circuits.

– to provide a pre–charge time in dynamic circuits.

– to provide the clock for latches in pipelined logic.

A number of schemes have been proposed with these functions in mind, Domino Logic[69] and NORA[70] which has been improved so that it now appears similar to the two–phase scheme of NMOS[60], but this chapter will only describe the single–phase scheme used in the multiplier. To understand the scheme and also its advantages it is easiest to describe a simpler scheme first.

Pseudo Two–Phase Clocking

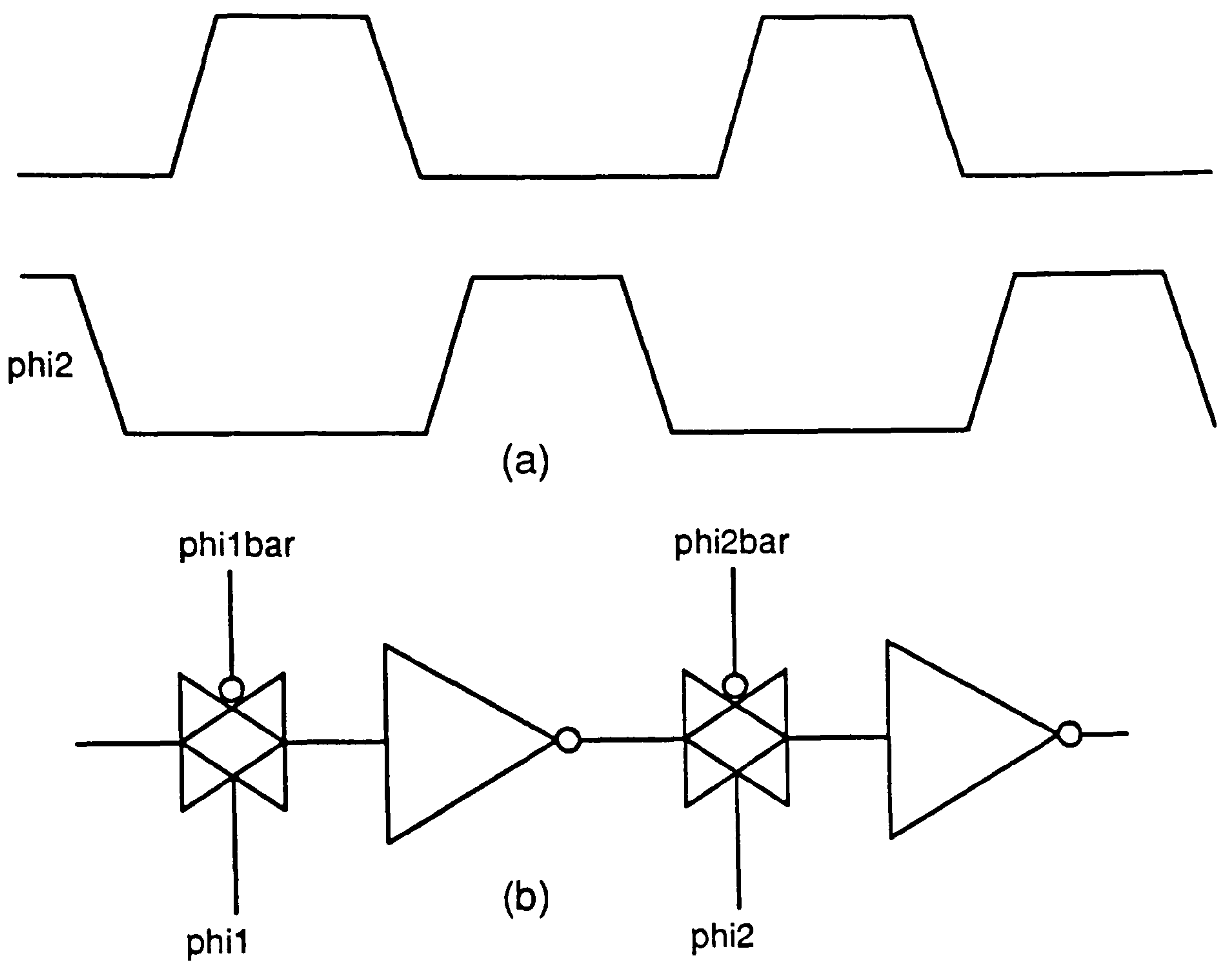
In this scheme two non–overlapping clocks, phi1 and phi2, are used, together with their inverses. This is a CMOS translation of the two–phase non–overlapping NMOS clocking scheme as used in Mead & Conway[13]. This gives four clock phases to route. Clocking waveforms are illustrated in Figure 3.3(a).

A dynamic latch using this scheme is shown in Figure 3.3(b), during the phi1 phase, the first transmission gate conducts allowing the logic value to be stored at the gate of the first inverter, during the phi2 phase, the second transmission gate transfers it, inverted, to the gate of the second inverter. It is held during this phase and reinverted at the output.

Clock skew can prevent this scheme from operating. The clock skew can arise in two ways, by a delay in the clock distribution path causing the clock phases to no longer be non–overlapping or else as a slow rise and slow fall time of the clock edges. Both of these forms of skew can lead to some overlap of clock phases causing both transmission gates to be on simultaneously, allowing the logic state to *race through* , losing the stored state.

To prevent this from happening the period of the clocks must allow for the worst case logic propagation time in combinatorial blocks that are between latches.

Dynamic logic utilising the pseudo–two phase scheme is illustrated in Figure 3.4(b), with the clock phases in Figure 3.4(a).

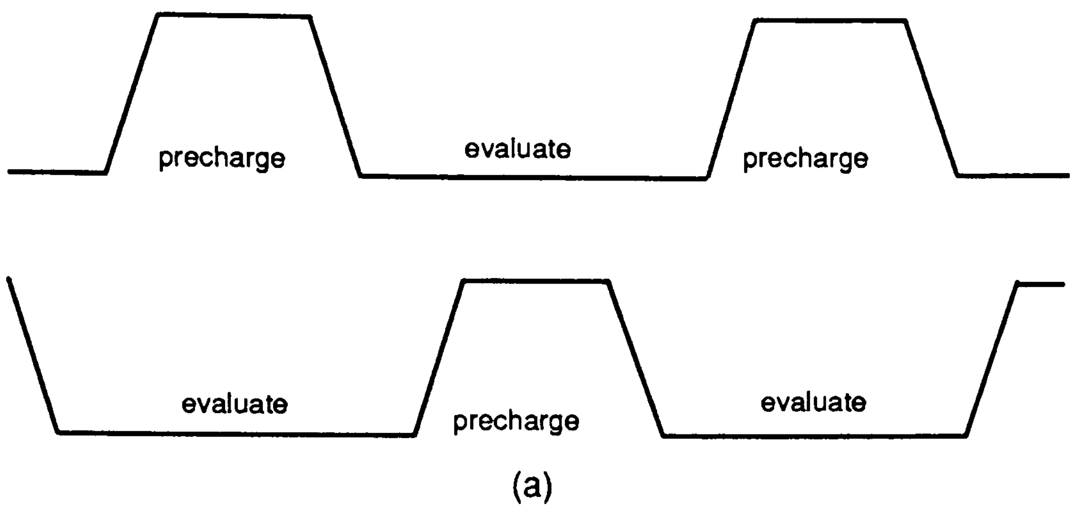


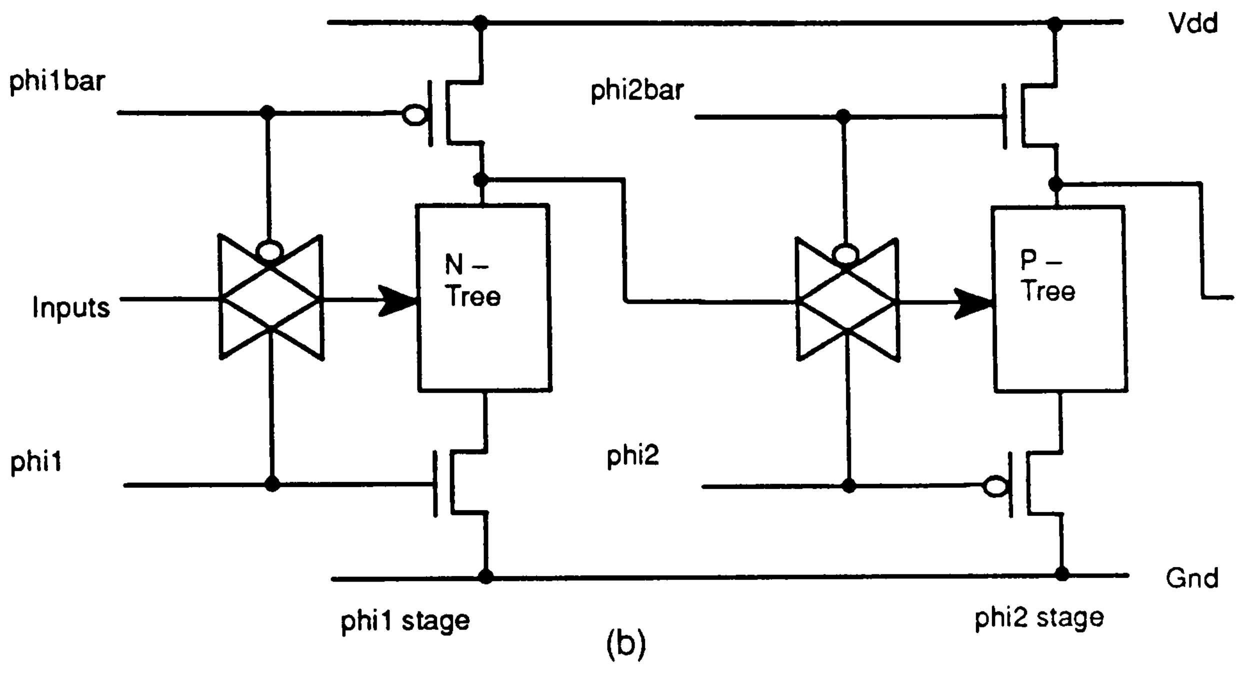
Pseudo Two–Phase Clocking

Figure 3.3

During the phi1 clock phase, the output of the phi1 stage is precharged to be evaluated by the logic whilst the phi1 clock is low. Inputs to the logic block are held steady by the closing of the transmission gate at the input. The operation of the phi2 stage is identical but it operates during the phi2 phase.

The scheme allows us a practical way to implement dynamic synchronous logic circuits at the cost of routing four clock lines. The complementary clocks can be generated locally using inverters. The scheme is vulnerable to clock skew between phases which can cause coupling between logic blocks, leading to a race–through condition.





Dynamic Logic using pseudo two–phase clocking

Figure 3.4

Other schemes have been proposed, for instance two phase and four phase schemes (described in Weste and Eshraghian[19]) to either reduce the number of clock lines or of the susceptibility to clock skew.

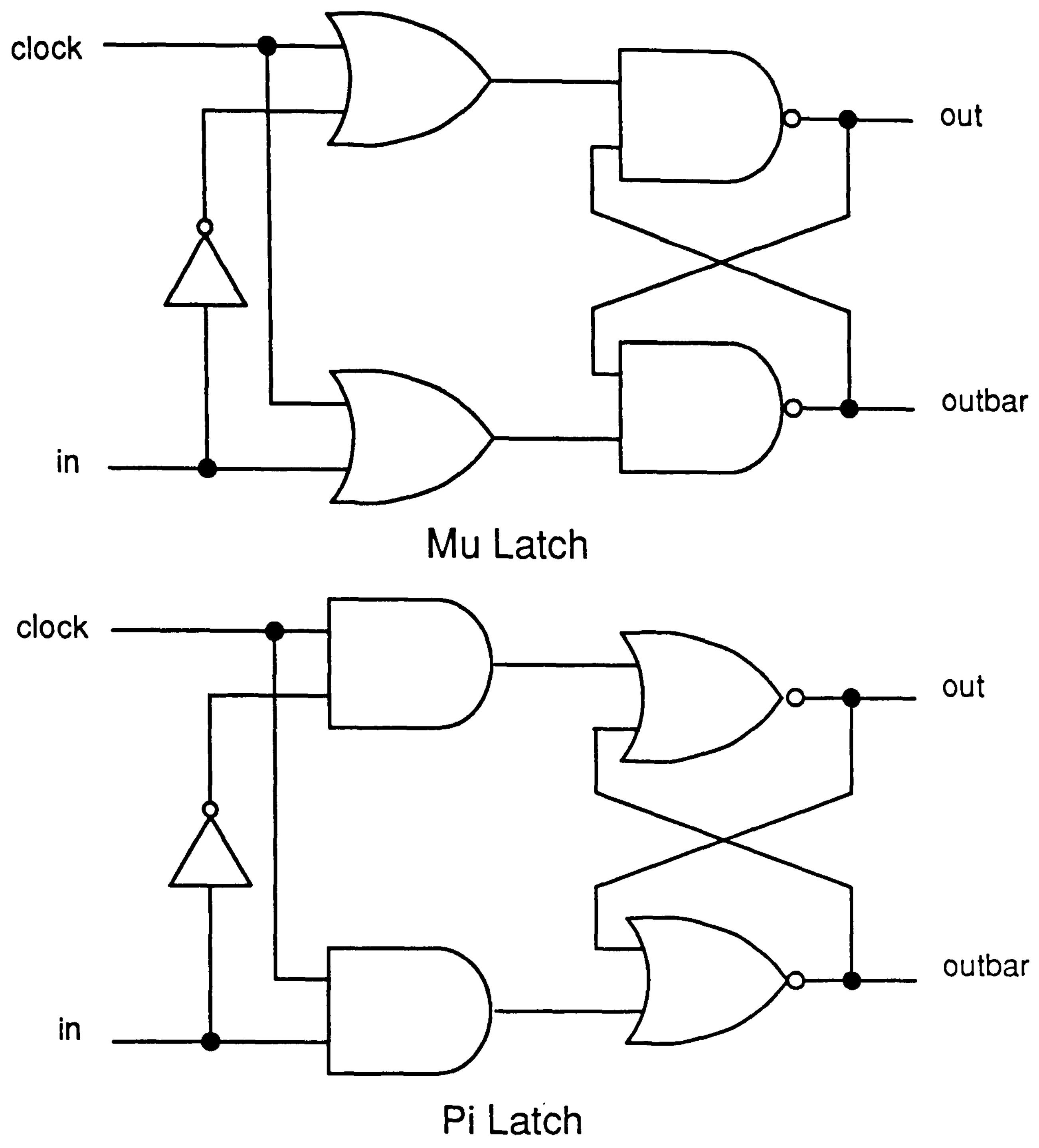
The NORA scheme[70] reduces the susceptibility to clock skew by making circuit operation tolerant to a degree of clock skew.

Single–Phase Clocking

The clocking scheme used in our multiplier uses only a single clock phase[71] which will be described in this section.

Instead of using multiple clock phases, the scheme uses two types of latch which are designated by the greek letters μ (*mu*) and π (*pi*) , which are mnemonics representing the holding phase of the two latch types. Each latch operates on a different half of the clock cycle. The latches operate as the two transparent D–type latches found in TTL circuits. The *mu* latch holds its output whilst the clock is low, signified by the lowered joining bar on the μ character, and samples whilst the clock is high, the *pi* latch working in a similar but complementary fashion, i.e. holding whilst the clock is high, signified by the raised joining bar on the π character.

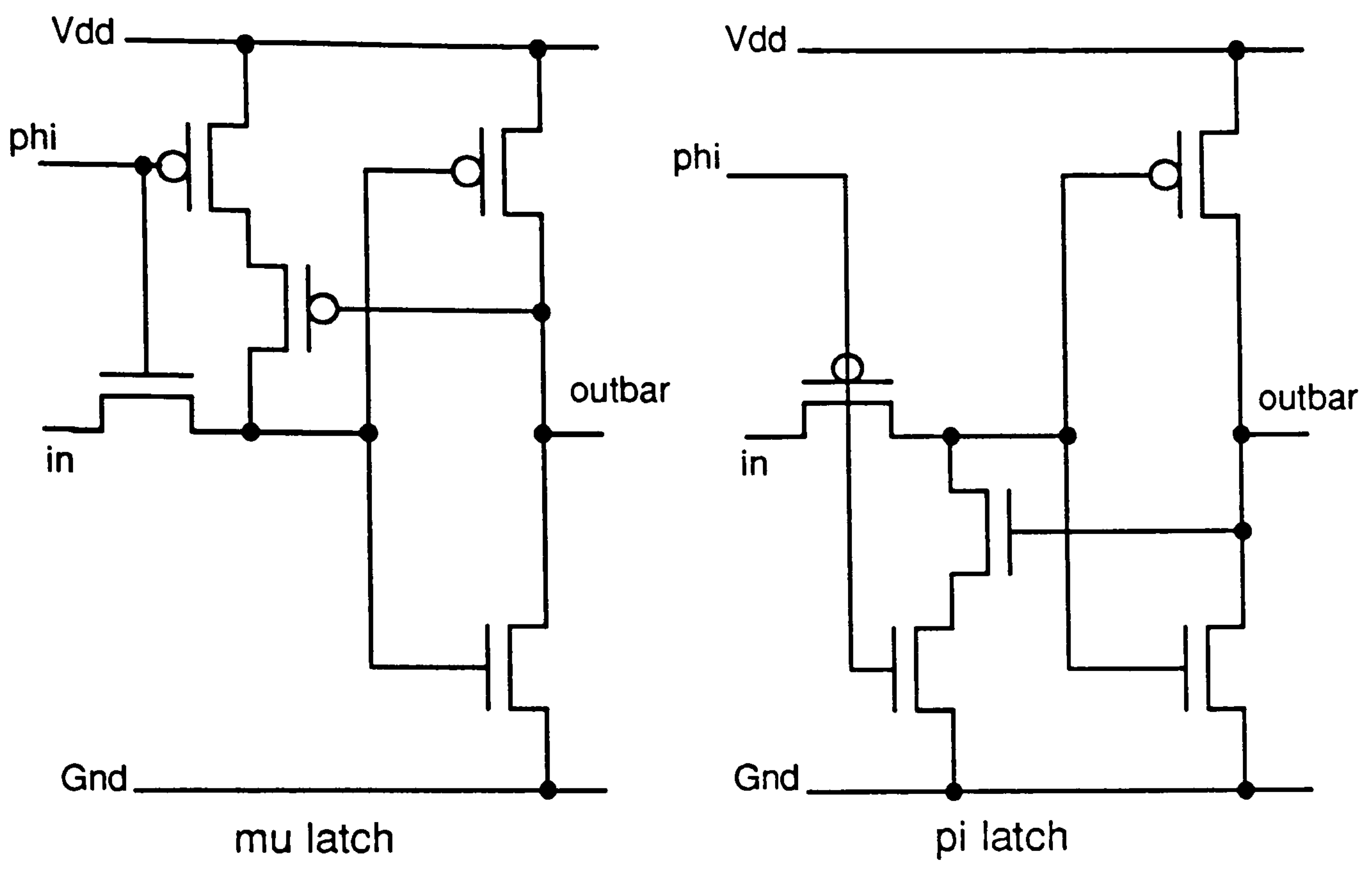
Figure 3.5 show the logic for these two types of latches which can be used to implement static forms for them.



Mu and Pi Latches (Static Forms)

Figure 3.5

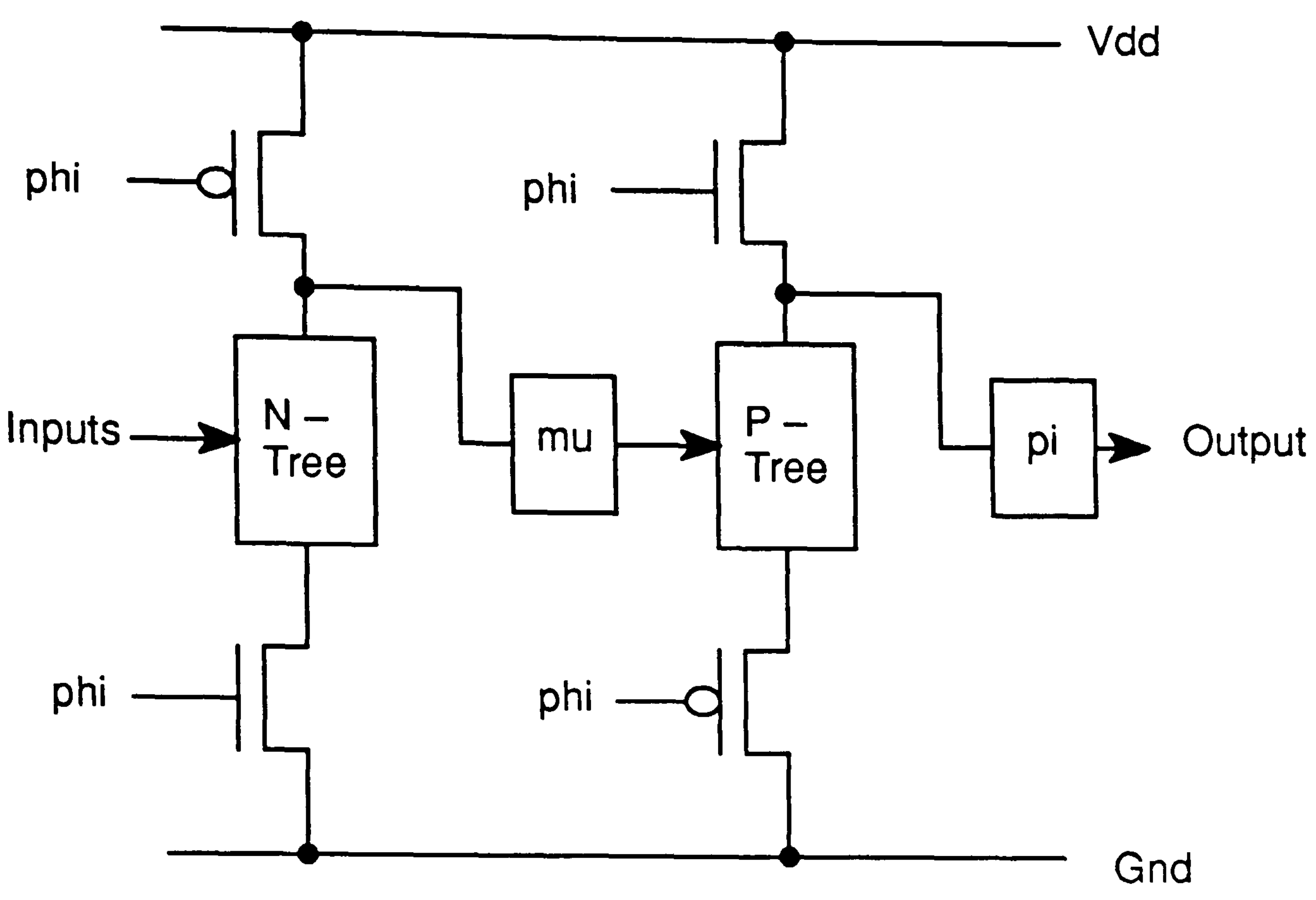
Dynamic versions of these latches are shown in Figure 3.6, each latch needing only five transistors. As only a single pass transistor is used in the latch, the transistors of the inverter should be ratioed to overcome the effects of the threshold drop. The logic value is inverted through each latch, though this can be easily compensated for by the logic blocks.



Mu and Pi Latches (Dynamic Forms)

Figure 3.6

To obtain the benefits of greater integration, dynamic logic is used, this is shown in Figure 3.7 . Each stage precharges whilst the next stage is evaluating with the roles reversed in the next half clock cycle. Other schemes such as *Domino* or *Complementary Tree* may also be used[71] with this single phase clocking scheme.



Dynamic Logic with Single–Phase clocking scheme

Figure 3.7

As there is only a single clock line, the problem of clock skew between phases has been eliminated, but there remains a race problem to be aware of. With a slow rising clock edge, both *mu* and *pi* latches may be on, allowing logic values to race through the pipeline, destroying stored values. To circumvent this problem, fast clock edges should be ensured. This can be done with sufficient clock buffering within the circuit or else more complex latches incorporating suitable switching characteristics can be used.

A further advantage of the scheme is that a greater clocking rate can be sustained as it is unnecessary to increase the clock period to prevent phases overlapping as in a multiphase clocking scheme.

In summary, the single phase clocking scheme, compared with the two phase scheme has advantages :–

– only a single global clock line needs to be distributed throughout the circuit, leading to simpler and more area efficient clock routing.

– skew between multiphase clock lines is eliminated as only a single clock line is used.

– clock generation is greatly simplified as there is no need to generate non–overlapping clock phases.

– a greater clock speed is possible as an increase in clock period to prevent multiphase overlap is unnecessary.

but it has the disadvantages of :–

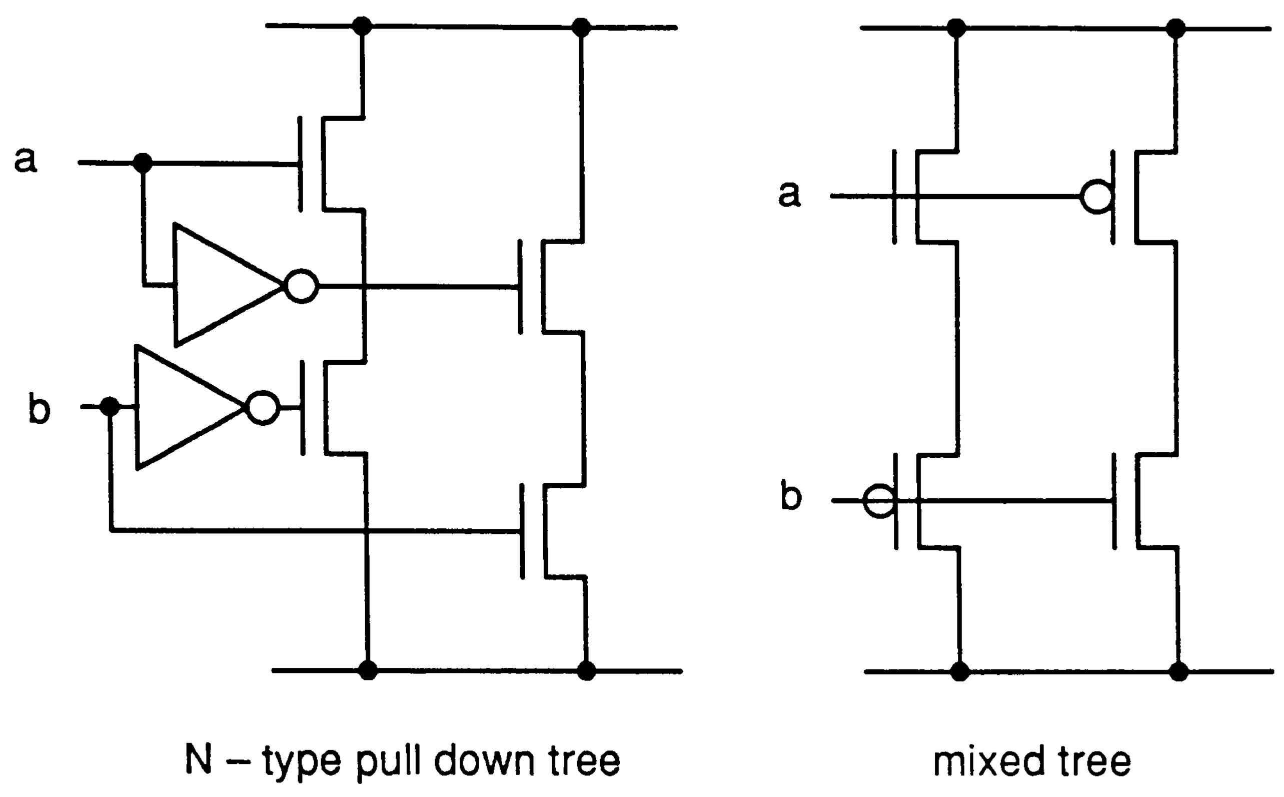
– two forms of latches are needed, raising circuit complexity.

– fast clock edges should be ensured to prevent race–through conditions.

3.3. Mixed Tree Logic

With conventional logic evaluation trees, composed completely of n–type transistors or p–type transistors, inverting of an input cannot be obtained within the tree. This requires an inversion to take place in a previous stage leading to extra logic and extra transistors.

In the mixed tree approach, both n–type and p–type transistors are used in the logic evaluation tree, whether it is a pull up or pull down tree. This gives a compact and flexible logic design style as only the single input, without its inverse, needs to be routed. A p–type transistor acts as an n–type but with its gate signal inverted, and similarly but complementarily for the n–type transistors. Thus suitable use of both p–type and n–type transistors rapidly within the tree can eliminate many of the otherwise necessary inversions. Figure 3.8 shows a two input exclusive–or gate with the conventional n–type pull down tree with a mixed tree implementation of the same function.

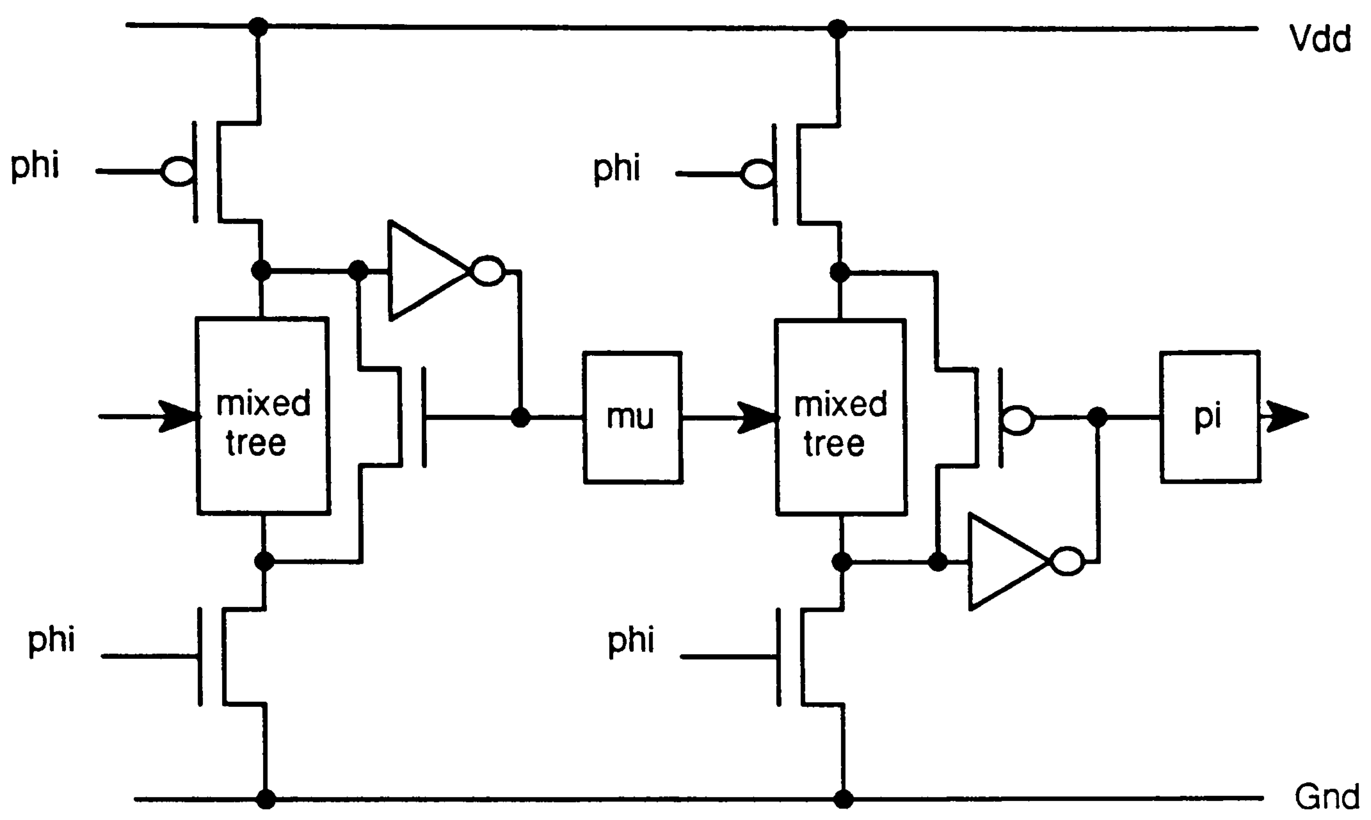


Tree for exclusive–or using n–type and mixed pulldowns

Figure 3.8

A problem arises using p–type transistors as pull–downs and n–type transistors as pull–ups since a gate threshold voltage drop is lost using the transistors in this way. By using an inverter with positive feedback to sense logic transitions and amplify the effect, the full rail logic swing can be reestablised. This also helps speed the logic transition. The insertion of the inverter is shown in Figure 3.9, where the mixed tree approach is merged with the single phase clocking scheme to gain benefits from both concepts.

Both the single phase and mixed tree schemes have been verified as practical design styles. We have two versions of the same six bit multiplier circuit, one the gate–matrix evaluation vehicle mentioned earlier, the other using the tiling scheme described in chapter 2, which have been fabricated and tested demonstrating the design strategies. In both cases the multipliers have worked as expected from logic simulation.

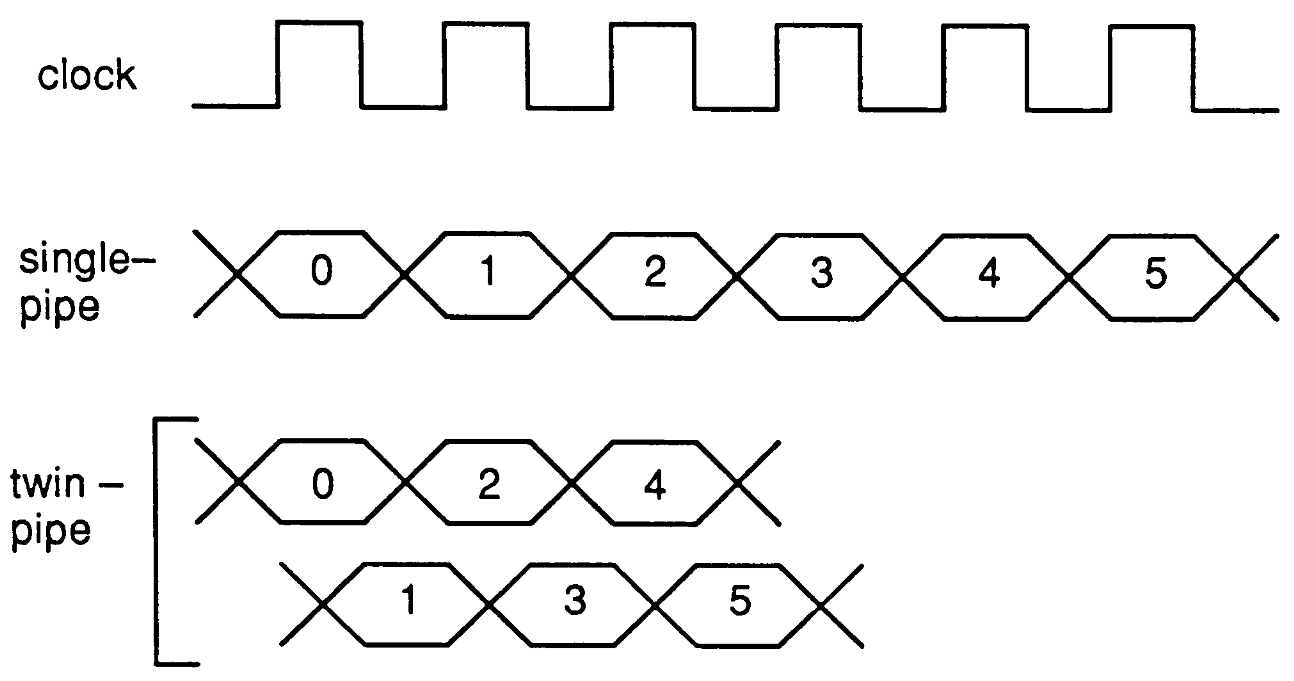


Sense Circuit to improve Mixed–Tree Performance

Figure 3.9

3.4. Twin Pipe Performance Enhancement

The twin–pipe scheme[72] doubles the throughput of conventional bit serial architectures implemented using a single pipe scheme. The technique has been described in an earlier independent paper[73] where it was used to increase the throughput of a multiplier, a similar scheme requiring three phases is also described. Two wires are needed to carry the twin pipe data representation. Timing of the twin pipe representation is shown in Figure 3.10. The method requires a clocking scheme where the clock period can be split into at least two distinct time intervals.

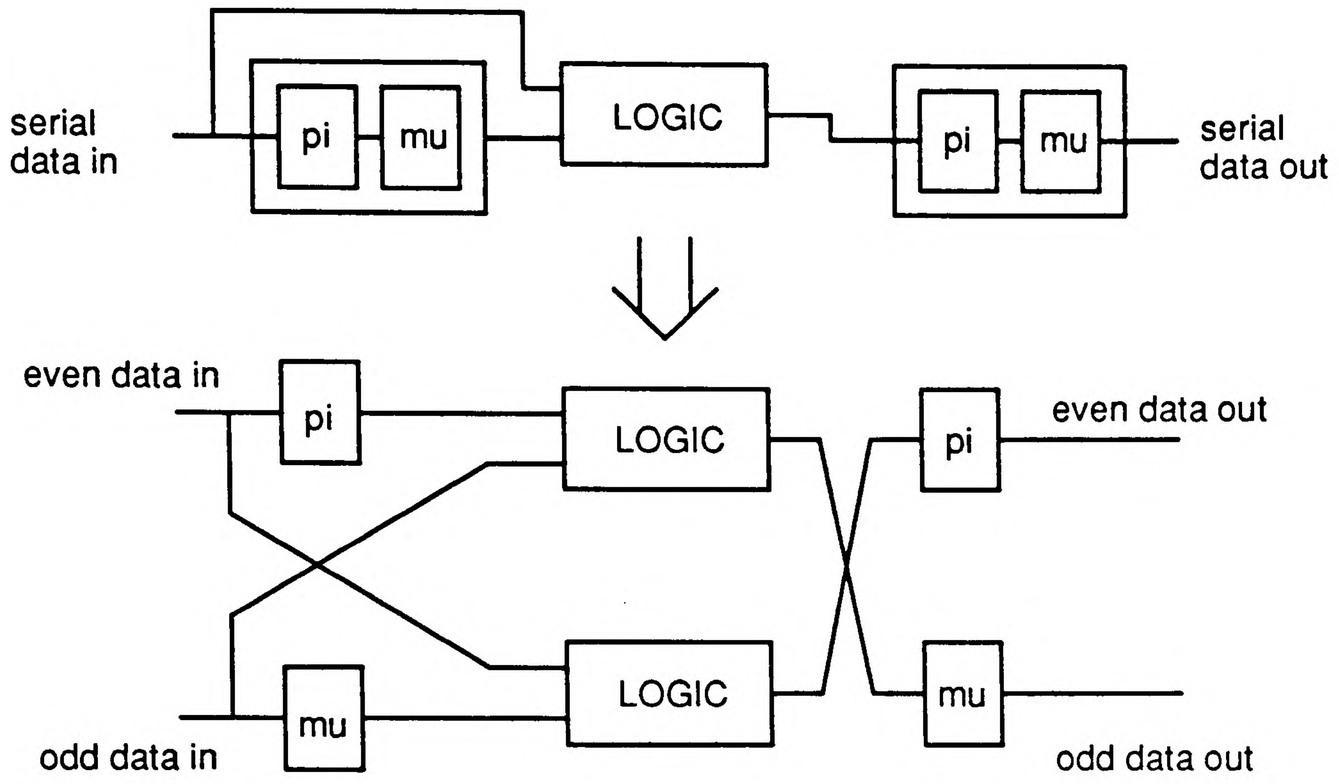


Timing with Twin–Pipe Data Representation

Figure 3.10

A full latch delays its input by a full clock cycle, in the twin pipe scheme each full latch is split into two half latches. The single phase clocking scheme described earlier is highly suited to this method as the *mu* and *pi* half latches are already present with logic between latches operating in different halves of the clock cycle.

The data *pipe* is split into two separate pipes servicing the odd and even data bits separately. The two pipes for the odd and even data bits are processed in a single clock cycle, the even bit being processed one half cycle out of phase with the odd bit as in Figure 3.10. Figure 3.11 shows how a single pipe can be split into two pipes for a section of latched logic in a pipe line for use with the twin pipe scheme.



Splitting single–pipe logic to form twin–pipe logic

Figure 3.11

As each full latch is split into two half latches, storage costs, when compared with a single pipe, are kept constant. It would appear that for logic, the area cost is doubled in this scheme. In many cases this is not the case as due to the asymmetry between odd and even bits normally present in processing serial data, it is possible to reduce the logic in many cases. An asymmetry arises naturally when using the two's complement data representations, as the most significant bit (MSB) is treated differently from the remaining bits.

In the case of a parallel–in, serial–out shift register, the twin pipe implementation requires fewer transistors than its single pipe counterpart[72] yet has twice the throughput.

A further application of the twin pipe scheme is to increase the dynamic range in parts of a bit serial system. The use of both single pipe and twin pipe architectures may realise n–bit processing in the single pipe section and $2n$–bit processing in the twin pipe section. This gives the ability to realise different wordlengths within a bit serial system, thus allowing the dynamic range to be increased in error sensitive computations.

In summary the characteristic features of the twin pipe scheme are :–

– doubles the throughput of conventional bit serial architectures without doubling the circuit area.

– a two wire representation of data is used.

– even data bits and odd data bits are processed in different clock phases and in different circuits (pipes).

– storage costs are the same as with a single pipe scheme as a single full latch is equivalent to two half latches.

– the cost of logic is less than double that in a single pipe scheme when asymmetries in the processing of even and odd data bits can be exploited.

3.5. Multiplier Architecture

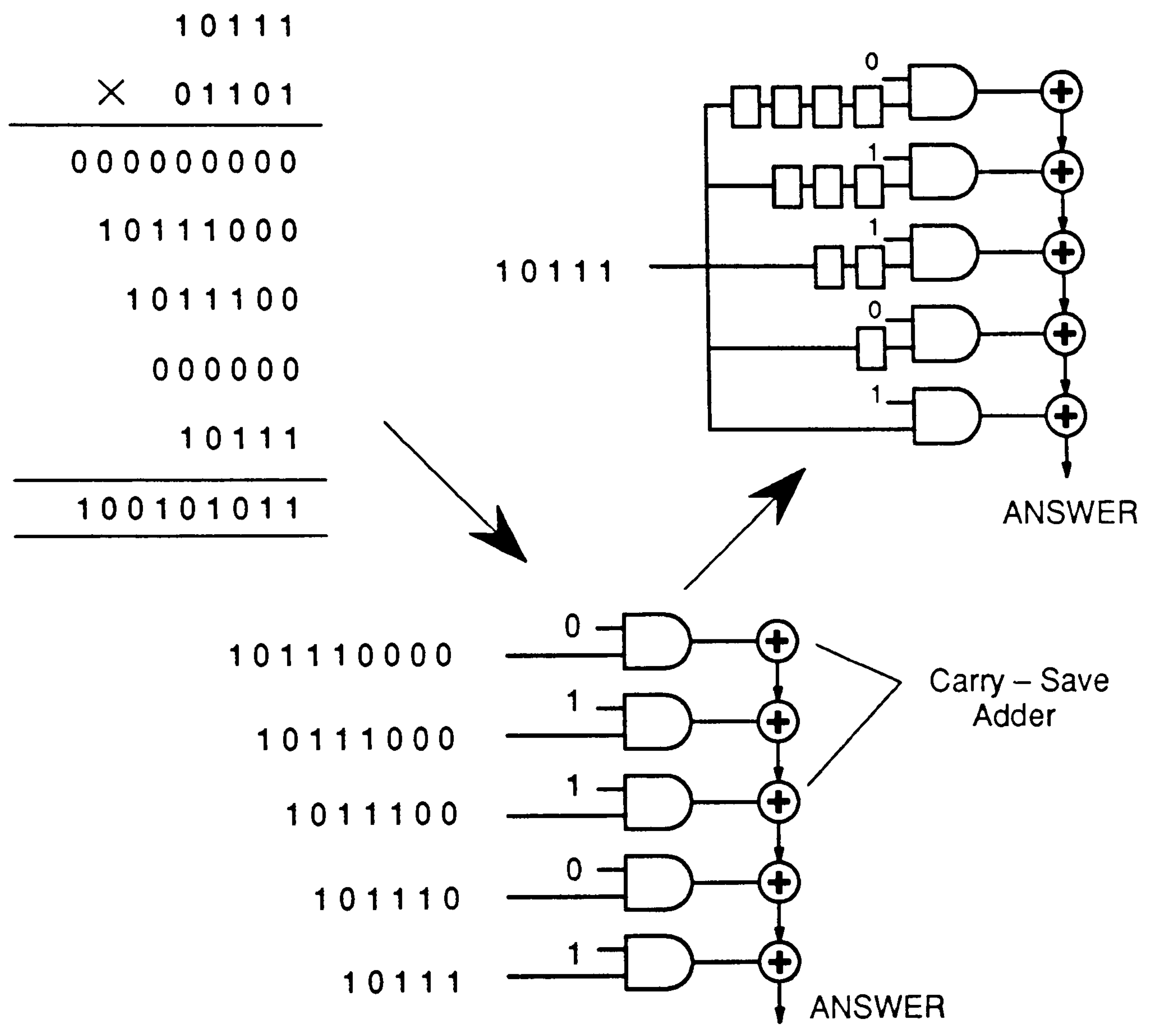
A fully bit serial multiplier was described by Lyon[8] in his paper, which utilised Booth's recoding and was capable of multiplying bit serial numbers using the two's complement data representation. The work on multiplier architectures to perform complex multiplication[74], [75] and compute inner products of vectors has progressed at Edinburgh.

This section describes the serial/parallel multiplier architecture as used in the layout test vehicle.

Multiplication Algorithm

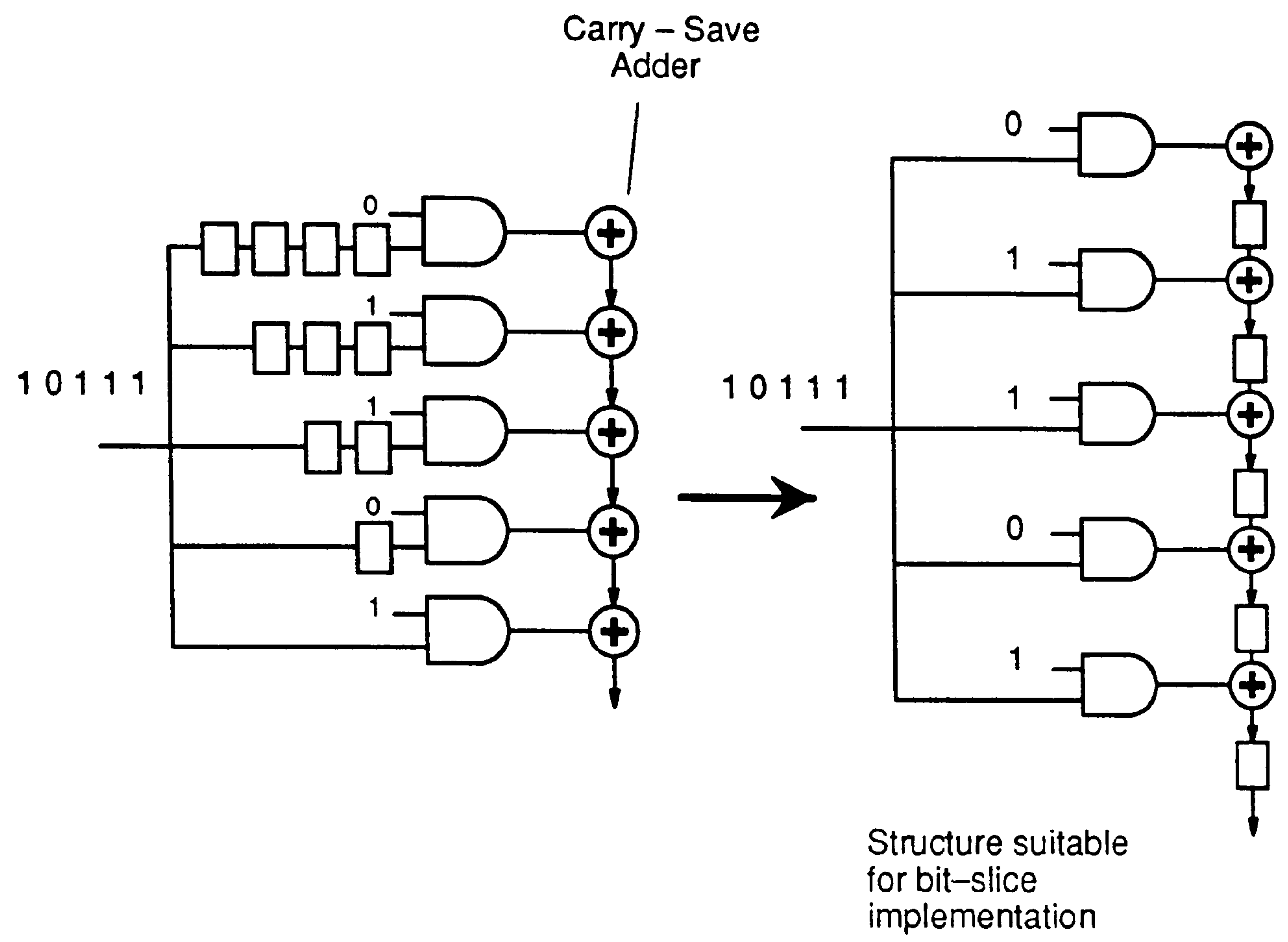
The algorithm used in this multiplier is based on the add–shift scheme of conventional multiplication. This is illustrated in Figure 3.12, data bits are multiplied with a single bit of the coefficient (logical and), shifted then summed together to form the product.

For positive data and coefficient, Figure 3.13 shows how the shifting can be placed in the carry save adder chain making the scheme suitable for bit–slice partitioning. In this case the data is input in serial format, whilst the coefficient is applied to the multiplier in a parallel format. This illustrates the algorithm, but in a practical multiplier, the carry save adders must all be cleared for the next data word, this is accomplished by 'anding' the inputs with the notlsb signal, if done for two inputs of the carry save adder, no carry signal will be generated and stored, to interfere with the next summation operation.



Conventional Multiplication

Figure 3.12



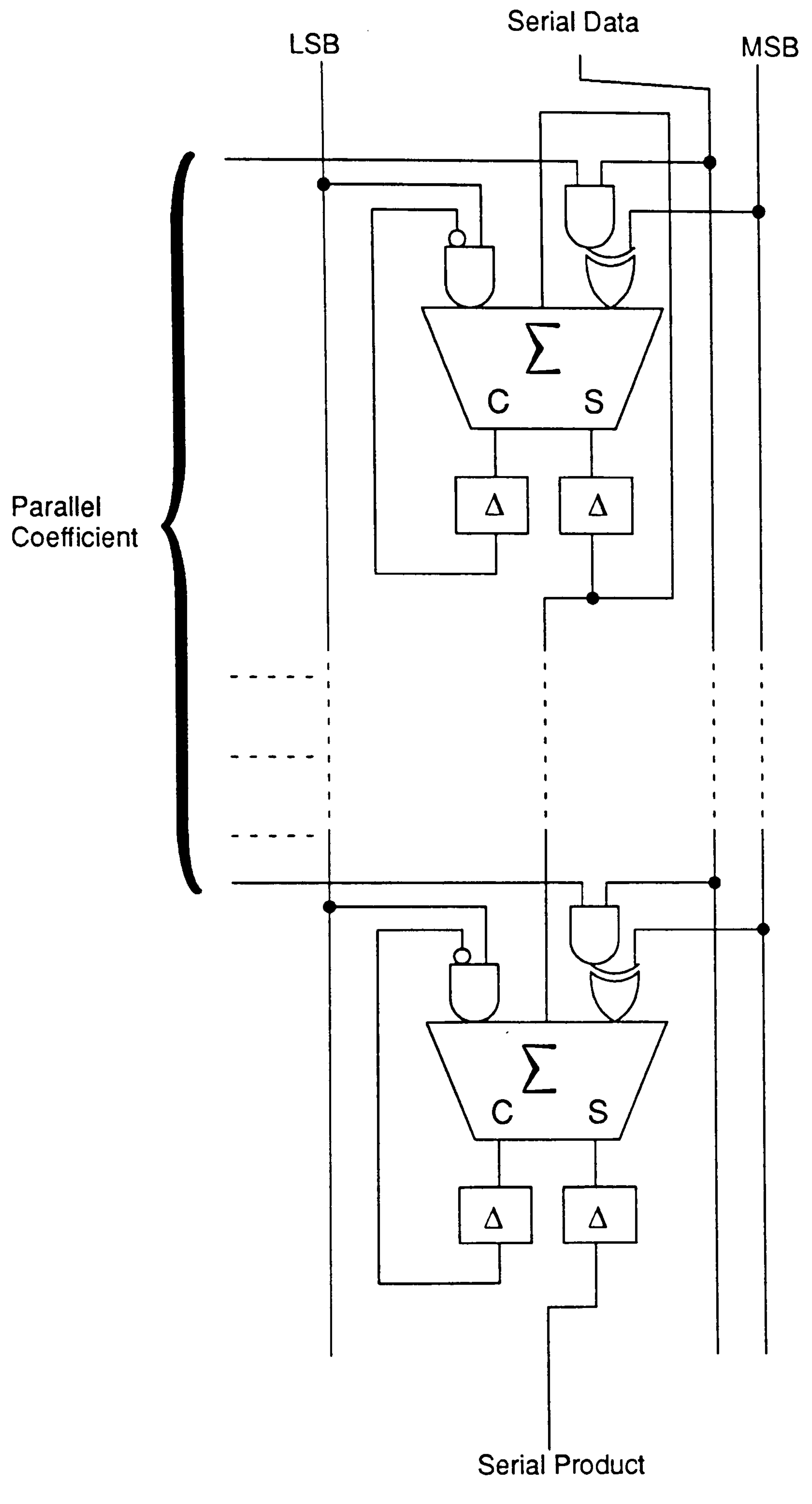
Shifts placed in carry save adder chain

Figure 3.13

For two's complement operation, treatment of the most significant bit of the data and coefficient word is different. We require a subtraction of the coefficient at msb time dependent on the data. This is easily accomplished by producing the two's complement of the coefficient at msb time and adding. The partial products in the carry save accumulator also need to be sign extended as they are in the two's complement form. Both operations can be accomplished by inverting the coefficient at msb time and also providing a feedback loop for the sum output at the first stage. Figure 3.14 shows the multiplier with these additions.

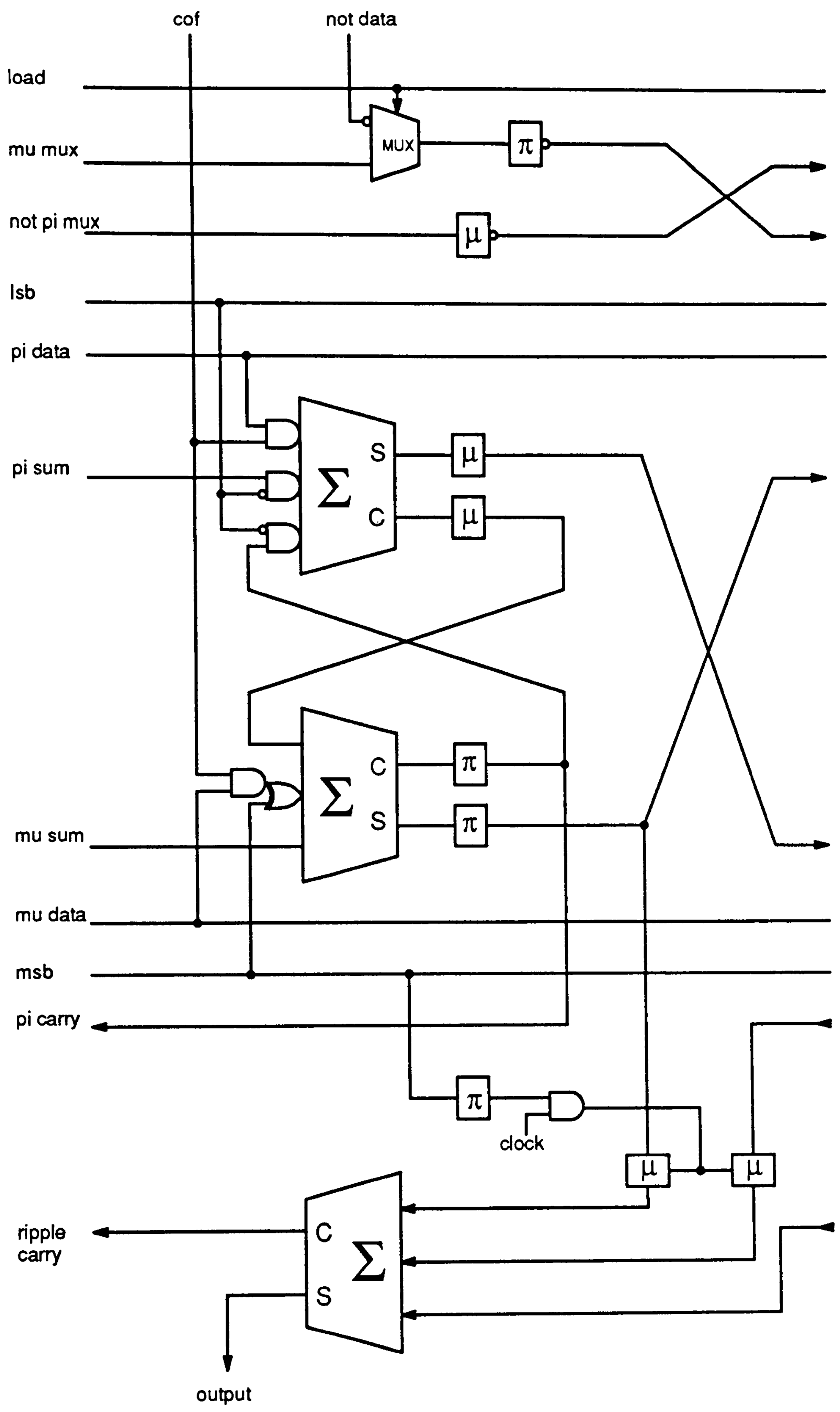
Finally to obtain twin pipe operation, the odd and even pipes need to be separated. This is shown in Figure 3.15, the logic is not entirely duplicated as clearing at lsb time is needed only on the pipe with bits which are active in the same half phase. The same is true about the inversion at msb time. A shift register is also shown to convert a parallel data word into a serial data stream. Another feature to increase the throughput, as the product is read out in a parallel format is the ripple carry adder which performs the final summation in a single clock period. The time for the propagation of the carry is not critical as the ripple adder is not loaded until a whole new data word has been read.

The final multiplier has a very high performance and was the design which was laid out and fabricated using the gate matrix style. The same multiplier architecture was used in another multiplier layout using the tiling scheme described in the previous chapter. For the device fabricated, a six bit multiplier was built, capable of performing the six by six bit product in three full clock cycles (i.e. six half phases) and was found to operate at over 10MHz, the testing and results are discussed in a later chapter.



Two's Complement Operation

Figure 3.14



Twin–Pipe Multiplier

Figure 3.15

Chapter 4

The Generation of Transistor Networks and Gate Matrix Layout

4.1. Introduction

The chosen layout style for the new cell library of FIRST is gate matrix. The automatic production of this type of symbolic layout is the goal described in this chapter. Before symbolic layout can be produced, the transistor nets for the logic must be generated. Using a netlist supplied by the user places the burden of logic design on him/her and complicates the symbolic layout procedure as properties of the netlist cannot be relied upon.

Two programs which perform these jobs are described. The first is MOSYN[76] which generates a transistor net to perform a boolean function from its boolean description. The second is MOSLAY[77] which generates from the transistor net a piece of symbolic layout in the gate matrix style. It uses properties of the network generated by MOSYN so that it is not a general tool capable of transforming an arbitrary transistor net into its equivalent gate matrix symbolic layout.

This chapter includes a description on the use of the *MOSYN* suite of programs for generating transistor networks and producing symbolic layout compatible with gate–matrix.

It describes how the gate–matrix layout can be fleshed to produce process dependent mask data in the CIF format through the use of the program MOSFLESH.

4.2. Introduction to the MOS Suite of Programs

This set of programs allows the transformation from a logic description of a logic block to layout information of the circuit in the CIF format. The set of programs consist of the following :–

MOSYN – transistor net generation program described below.

MOSLAY – generates sparse symbolic layout from transistor nets generated by MOSYN.

MOSCOM – Compacts the sparse symbolic layout generated by MOSLAY.

MOSFLESH – Converts symbolic (process rule independent) layout to CIF. The CIF which is produced is process rule dependent, the information being stored in a library of microcells.

The use of these programs may be made clearer through an example. The following is an example of a two input multiplexor. Figure 4.1 shows the MOSYN input file for this example.

```
; 2 input multiplexor
; Selects either A or B inputs
; depending on the select input
; output is mux
*function mux
C1: or(and(A,select),and(B,not(select)))
else C0;
```

MOSYN input file

Figure 4.1

After processing through MOSYN has completed, a transistor net is generated. In this case six nets are generated. From the index numbers, described in the previous section, we find little to choose between them. Figure 4.2 shows a typical generated transistor net.

The transistor net is described using a set of equations representing the connections between nodes of the circuit. The left hand side of each equation is the name of a node, on the right hand side is a set of FETs to which that node is connected. A term such as [A]B represents a nMOS FET with gate connected to node A and source connected to node B. A term such as {A}B is similar but applies to a pMOS FET. If an input variable is prefixed with a '*' then the complement of that variable is indicated.

```
; ****************************************
; Function : mux(A,select,B)
;  Logic        : 00011101
; ****************************************
; (0) 14 FETs 4 Width 4 Chan. H=2 pFET=7 nFET=7 B=7 C=8
; Input Order(0): A select B
mux = N0
 N0 = [N12]N11 + [N9]N6 + {N12}P5 + {select}P4
 N12 = [A]Gnd + {A}Vdd
 N11 = [select]Gnd
 N9 = [select]Gnd + {select}Vdd
 N6 = [N7]Gnd
 N7 = [B]Gnd + {B}Vdd
 P5 = {N9}Vdd
 P4 = {N7}Vdd
```

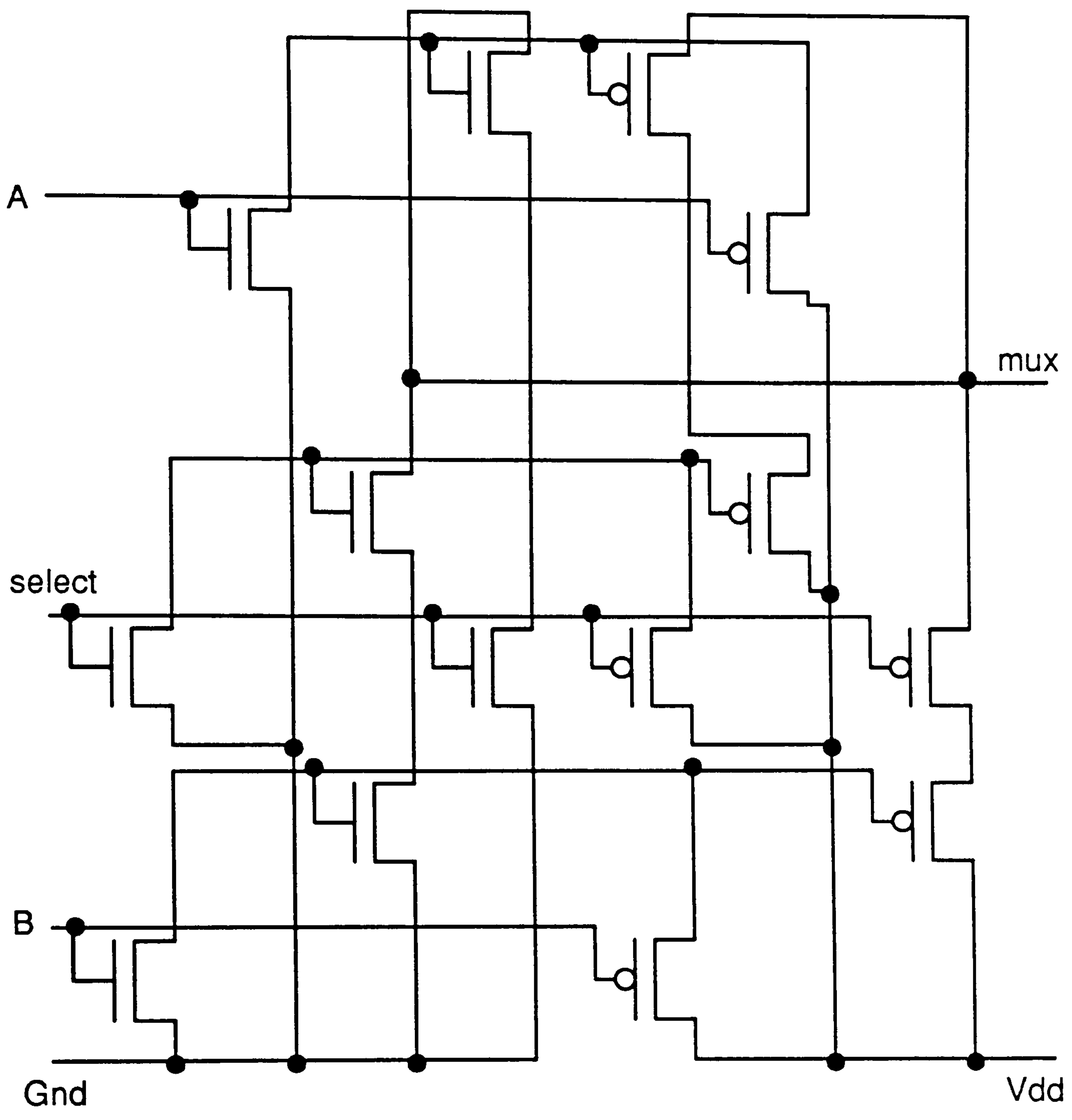
Typical Transistor Net output from MOSYN

Figure 4.2

Once this net has been generated in this format, we may input it into the program MOSLAY. This program can be used to produce transistor circuit diagrams of which Figure 4.3 is an example. MOSLAY also has the capability to produce symbolic layout. The layout produced is sparse and requires use of the compaction program MOSCOM before it can be used.

For this example, the symbolic output, the output obtained by using MOSLAY and MOSCOM to compact it, is shown in Figure 4.4 .

Symbolic layout of this form has no dependency on process design rules and such layout could be stored in a library from which mask data can be easily generated. The program which incorporates the process dependent features of layout is the program MOSFLESH. It takes symbolic data of the above form and produces the process dependent CIF output from it. Figure 4.5 shows a plot of fleshed out symbolic data performed by the program MOSFLESH.



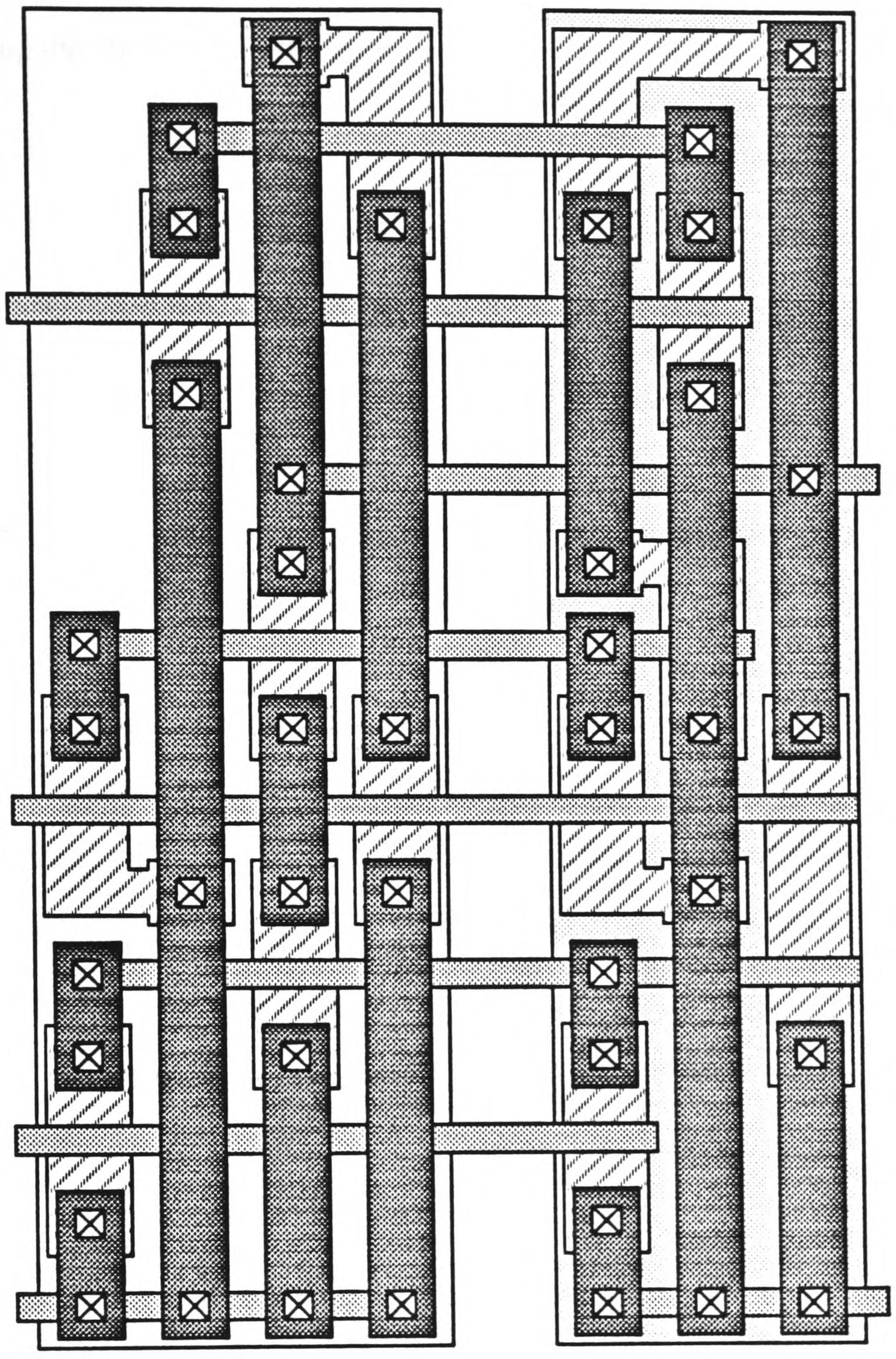
MOSLAY Transistor Circuit Diagram

Figure 4.3

```
;    'M ' ... metal
;    'X ' ... contact
;    ' N' ... nMOS FET
;    ' P' ... pMOS FET
;    ' *' ... diffusion
;    ' =' ... polysilicon
      :                              :
      :          X* * *   * * * *X*  :
      :          M    *   *      M   :
      :      X= =M= = N = P =X=  M   :
      :      M   M    *   *  M   M   :
      :      X*  M   X*  X*  X*  M   :
      :       *  M   M   M    *  M   :
A     : = = = N =M= =M= =M= = P  M   :
      :       *  M   M   M    *  M   :
      :      X*  M   M   M   X*  M   :
      :      M   M   M   M   M   M   :
      :      M   X= =M= =M= =M= =X= =:mux
      :      M   M   M   M   M   M   :
      :      M   X*  M   X* *M*  M   :
      :      M    *  M       M*  M   :
      :  X= =M= = N =M= =X= =MP  M   :
      :  M   M    *  M   M   M*  M   :
      :  X*  M   X*  X*  X*  X*  X*  :
      :   *  M   M    *   *  M    *  :
select: = N =M= =M= = N = P =M= = P  :
      :   *  M   M    *   *  M    *  :
      :   * *X*  X*  X*   * *X*   *  :
      :      M    *  M       M    *  :
      :  X= =M= = N =M= =X= =M= = P  :
      :  M   M    *  M   M   M    *  :
      :  X*  M   X*  M   X*  M   X*  :
      :   *  M   M   M    *  M   M   :
B     : = N =M= =M= =M= = P  M   M   :
      :   *  M   M   M    *  M   M   :
      :  X*  M   M   M   X*  M   M   :
      :  M   M   M   M   M   M   M   :
Gnd   : =X= =X= =X= =X=  X= =X= =X= =:Vdd
      :                              :
; Layout size: 7 x 16
; FETs / Area: 14 / 112 = 12.50 [%]
; Total memory allocated: 791 [Byte]
```

Symbolic Layout of Multiplexor

Figure 4.4



Fleshed Mask Data from Symbolic Layout

Figure 4.5

4.3. Transistor Net Generation – MOSYN

The *MOSYN* suite of programs [78] constitute very useful tools for the production of symbolic layout from an initial logic description of the behaviour of a block. The program *MOSYN* [76] can take input specifications of circuit behaviour in three major forms :–

(a) Truth Table – The logical behaviour is described by a sequence of 1's and 0's making up the truth table of the logical function,

(b) Cube Notation – This is similar to the truth table description but allows the inclusion of don't care states,

(c) Functional Notation – The description of logical behaviour is given as an equivalent Boolean function.

From such a description of the required logical behaviour *MOSYN* will generate a number of transistor nets. The nets differ by having different orderings of input variables. The number of transistor nets generated is normally (i.e. without other options used) equal to the number of permutations of the inputs. Thus if we wished to obtain a transistor net for a gate with say, three inputs, *MOSYN* would generate 6 = factorial(3) nets. *MOSYN* will not produce superfluous nets when there is symmetry between some of the inputs, i.e. there is no change in logical behaviour if these inputs are swapped. This is the case with a completely symmetric function such as an 'exclusive–or' gate or an 'and' gate in which case only a single transistor net would be output from the program.

From the nets generated, to help the designer choose the one which suits his/her needs best, *MOSYN* generates index numbers. The numbers produced by the program as it stands are :–

(a) FET's – The number of FET's in the transistor net,

(b) Width – The maximum number of FET's which share a common input variable. This gives an estimate of the width of the net when symbolic layout is produced using the other programs,

(c) Chan – Number of wiring channels required for connecting FET's. This gives an estimate of the number of metal lines required in conventional gate–matrix layout,

(d) H – A number related to the height of the transistor net tree,

(e) pFET's – The number of p–type FET's in the generated transistor net,

(f) nFET's – The number of n–type FET's in the generated transistor net,

(g) pGATE – Generated with dual rail logic (i.e. using true and complement signals). Number of uncomplemented signals needed as inputs to transistor gates,

(h) nGATE – similar to pGATE but number of complemented signals,

(i) Balancing numbers B & C

5. B – Maximum number of a particular type of FET, i.e. max of pFET & nFET,

6. C – Maximum of the number of complemented or uncomplemented signals needed as inputs to transistor gates, i.e. max of pGATE and nGATE.

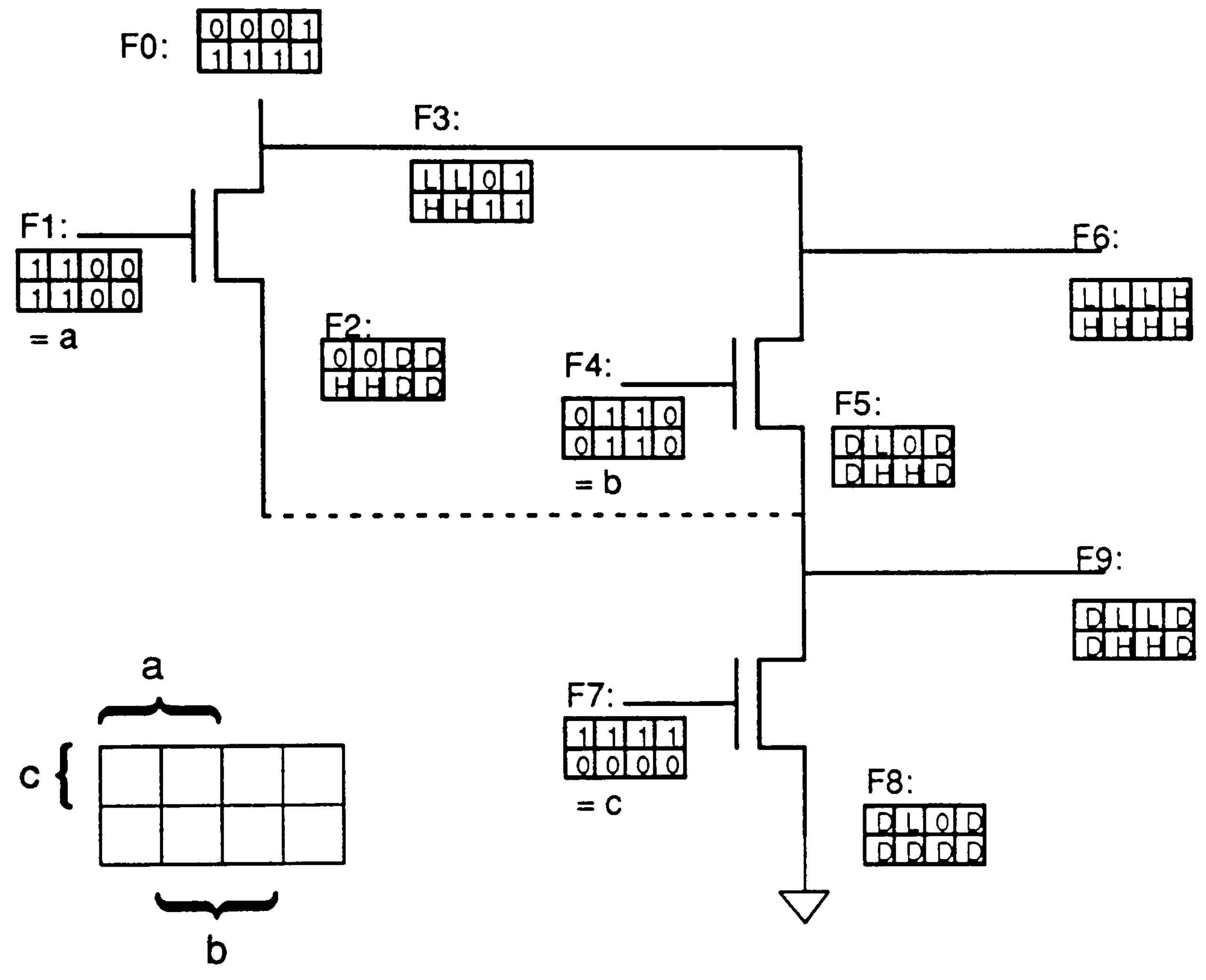
With these indices a designer may choose the net which best suits his/her needs, satisfying the criteria of minimum area, connectability with other blocks, optimal circuit performance or whatever.

Algorithm used for Transistor Net Synthesis

The transistor net generation software MOSYN is an important basis for the work described in later chapters. It forms part of the automatic cell generation software and allows low level blocks to be synthesised from logic descriptions. This netlist can then be used to form symbolic layout for these small logic blocks. The algorithm used in MOSYN is outlined briefly for completeness, the description below is essentially that given in[79]. The details of the algorithm are given in [80].

The transistor network synthesis algorithm employed in MOSYN is a hierarchical three–way decomposition and reduction of a logic function using a seven valued logic system. The seven logic values are in the set { 0, 1, L, H, Y, D, Z }. '0', '1' represent full logic low and full logic high, i.e. 0V & 5V in a CMOS process. 'Z' represents the high impedance state. The more unfamiliar values of 'L' and 'H' represent a lower logic level and a higher logic level, i.e. any voltage level below (above) a logic threshold including the high impedance state, this accounts for a '0' or '1' with a threshold drop, after passing through a 'p' or 'n' –type transistor. 'Y' and 'D' represent a strictly isolated state and the don't care state respectively.

The values '0' and '1' are termed 'n–' and 'p–essential' values and are realised in a MOS circuit by nMOS and pMOS FETs. The remainder are 'empty values'. A seven–valued boolean function will be called empty if it only realises empty values. The values {0, L, Y, D} and {1, H, Y, D} are compatible value sets and a function realising only values from one set can be realised physically by a connection to ground (logical 0) or power (logical 1) respectively. A function which realises incompatible values such as {1, L} will need to be decomposed into simpler functions which can be realised.



MOSYN Synthesis Example

Figure 4.6

Figure 4.6 shows an example of the synthesis process in MOSYN. A transistor net which performs the function $\overline{(a+b)c}$ is generated for a static cMOS circuit. The function at each circuit node is represented in a Karnaugh map form. The process starts by decomposing the root function, in this case F0 $= \overline{(a+b)c}$ into three functions; a gate function F1, a terminal function F2 and a wired function F3. The gate function is determined using an 'ordered literal' strategy, where the first 'essential literal', when one can be found, is selected to form the gate function, from a predetermined input variable order. A literal is taken to be an input variable or its complement. An essential literal is defined for a given function as follows:

> A literal 'x' is n–essential (p–essential) when the logic value for x=1 (x=0) is '0' ('1') and the logic value for x=0 (x=1) is incompatible with '0' ('1') for at least one input variable combination.

The problem then becomes one of assigning the terminal and wired functions. If the transistor in the decomposition is an nFET (pFET), n–essential (p–essential) values should be shared by the terminal and wired functions with any p–essential (n–essential) values being assigned to the wired function. The sharing scheme employed in MOSYN is as follows:

> Essential values which can not be propagated to the node which provides the source of those values are assigned to the wired function and, after the synthesis of the wired circuit portion is finished, essential values which have not been realised by the wired circuit are assigned to the terminal function.

Each essential value is thus assigned to and realised by the wired and/or terminal circuits. When an essential value (0/1) does not need to be assigned to a function, it is converted to its related empty value (L/H). Other empty values are assigned to both the terminal and wired function. When a value cannot propagate to the terminal function source node, because the transistor is off, the function is assigned don't–cares 'D'.

The process is repeated recursively for the wired and terminal functions, but attempting to complete the wired function before starting on the terminal function. In this way a tree–like transistor network results which performs the boolean function specified by the original root function.

During the process decomposition of a function can be stopped when the function :

(a) is empty, i.e. has values only in the set {Z, L, H, Y, D},

(b) can be realised by a constant value '0' or '1',

(c) is an input variable (if steering mode logic is required),

(d) there is connection to a circuit node which has already been synthesised.

In the example of Figure 4.6, an input order {a, b, c} has been used. Inputs 'a' and 'b' are equivalent and swapping their order has no effect on the circuit topology. The ordering can be specified by the user, though the number of transistors used for different orderings may differ. Decomposed functions F6 and F9 are terminated because they are empty, function F8 is realised by '0' (Gnd), functions F1, F4 and F7 are realised by input variables, and function F2 is realised by a connection to F5.

Producing Symbolic Layout from the Transistor Net

The most time consuming task in the problem of laying out a transistor network in gate matrix[38] is that of the assignment of gate positions, i.e. polysilicon columns. The circuit topologies synthesised by MOSYN have the advantage that the input order list is already defined. The simplest and most efficient assignment of input variables to polysilicon rows is to use the same order as the input order list. Thus the assignment of polysilicon columns is a straightforward task. The routing of the metal and diffusion lines is more complex and require routing procedures.

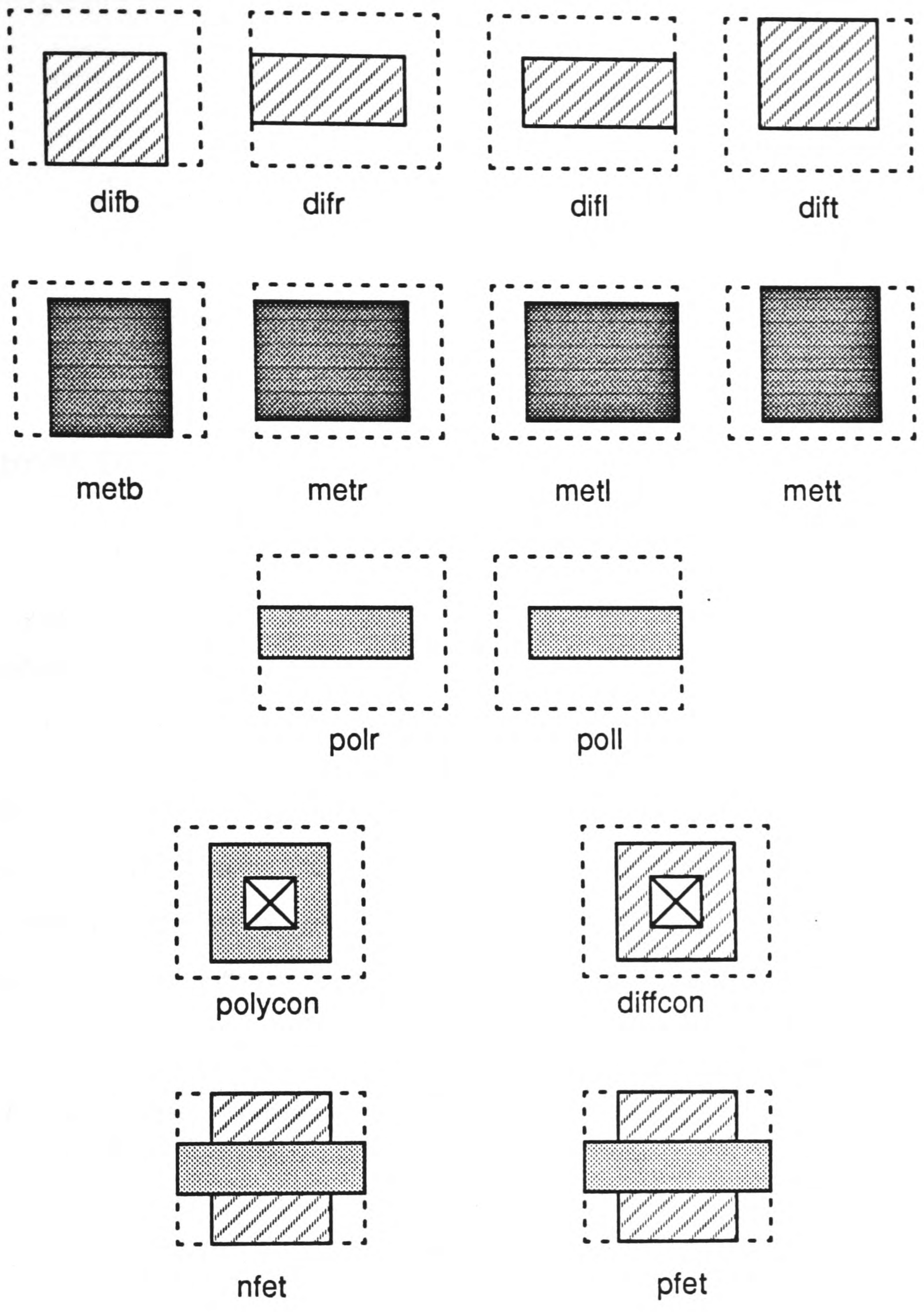
These functions are performed by the program MOSLAY[81] which produces symbolic layout from the transistor net. An example of the symbolic layout produced is that shown in Figure 4.4 .

4.4. Producing CIF from Symbolic Layout

The program MOSFLESH performs the conversion of symbolic layout to CIF by referencing a microcell library. Every basic element on the symbolic layout e.g. transistor, contact, connection, is replaced by its corresponding instance in the microcell library. The program operates by reading in the file containing the symbolic gate matrix layout and substitutes the corresponding microcells. It needs to read in the four adjacent positions in order to determine which microcell to use. Figure 4.7 shows a typical microcell library for MCE 3 micron envelope process rules. Examination of Figure 4.5 shows how these microcells are combined to produce CIF, the microcells have been merged together in the plot.

This has the important advantage that layout for new processes can be achieved quickly by changing this base of microcells. They incorporate most of the process design rules in an easily understood form, although they are not sufficient to contain all the process rule information. Information referring to placement of substrate contacts, separation of n–transistors from p–transistors is not represented.

The place for this further design rule information is seen to be one place higher in the hierarchy, when logic blocks are connected to form slices. Substrate contacts can then be placed within power rails, and logic blocks have to be split (providing the separation between n–type and p–type transistors) when fitting communicating signals.



Example Microcell Library

Figure 4.7

Chapter 5

Layout and Testing of Manually Laid Multiplier

5.1. Introduction

In order to test our proposed layout strategy, a six–bit serial/parallel multiplier was laid out using the gate matrix layout methodology to serve as a test vehicle. This was seen as appropriate as multiplication is a necessary operation in signal processing circuits and the complexity of this element is typical of those in the intended operator library.

This chapter covers the test results obtained from the fabricated multiplier. The tests were performed with the Tektronix DAS for functionality testing and also with a TTL test board for speed testing. The chapter then goes on to describe the modifications made to correct the obvious errors and the refabrication of the multipliers.

5.2. Manual Layout of the Multiplier

The layout started with a logical description of the multiplier bit–slice, in the form of a block diagram. Figure 5.1 shows the logic diagram partitioned into blocks. The stages involved in the layout process were :–

– partitioning of circuit

– transistor net generation

– slice floorplanning

– hand layout of logic blocks

– logic simulation of extracted layout

– interconnection into bit–slice

This layout route is better expressed by the flowchart of Figure 5.2, than by the above implied linear sequence. When layout preceded manually, mistakes were often made which had to be picked up from the logic simulation of the relevant section. Such logic block elements had to be relaid and refitted into the final bit slice. Iteration in the layout loop also occurs in trying to optimise logic blocks to gain a more area efficient layout.

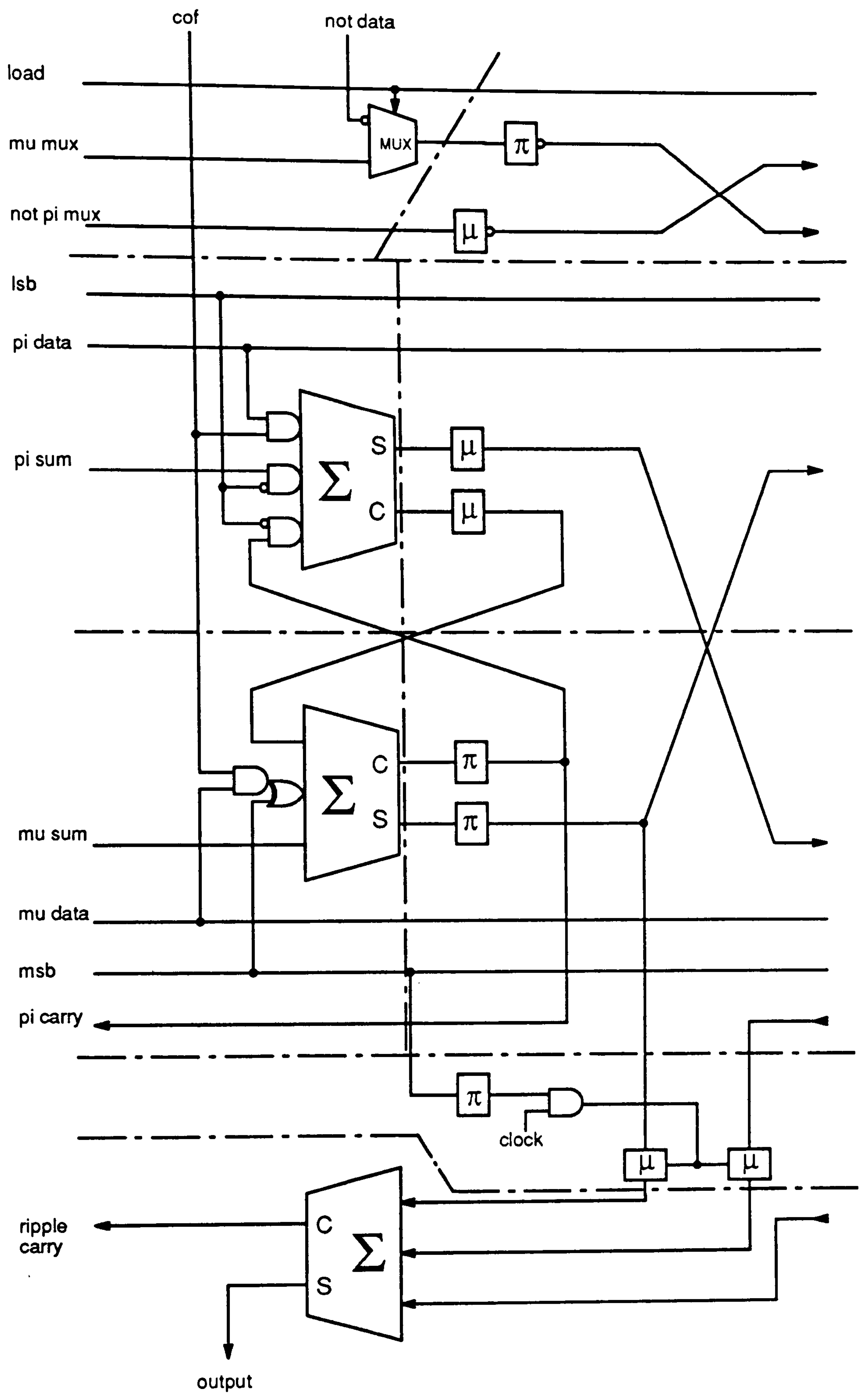
Partioning of the multiplier, as shown in Figure 5.1, collects areas of logic together which form strongly interconnected blocks. This is important as inefficient partioning can lead to inefficient layout.

The transistor nets for each logic block, obtained from partioning the multiplier logic were generated with *MOSYN* described in the previous chapter. The input files to the *MOSYN* program and the transistor nets used for the layout of the multiplier are included in Appendix 3. Transistor nets were selected on the basis of the fewest number of transistors in the net. The latches used were the dynamic single phase ones described in Chapter 2.

To ensure that the blocks laid out by hand would be connectable within the bit–slice, some initial preplanning of the layout was done, slice floorplanning. Care was needed, to allow interconnection of the manually laid blocks as only the second level metal was left available for interconnection. By using gate matrix this meant fixing some polysilicon lines in the blocks, to prevent unmanageable signal crossovers.

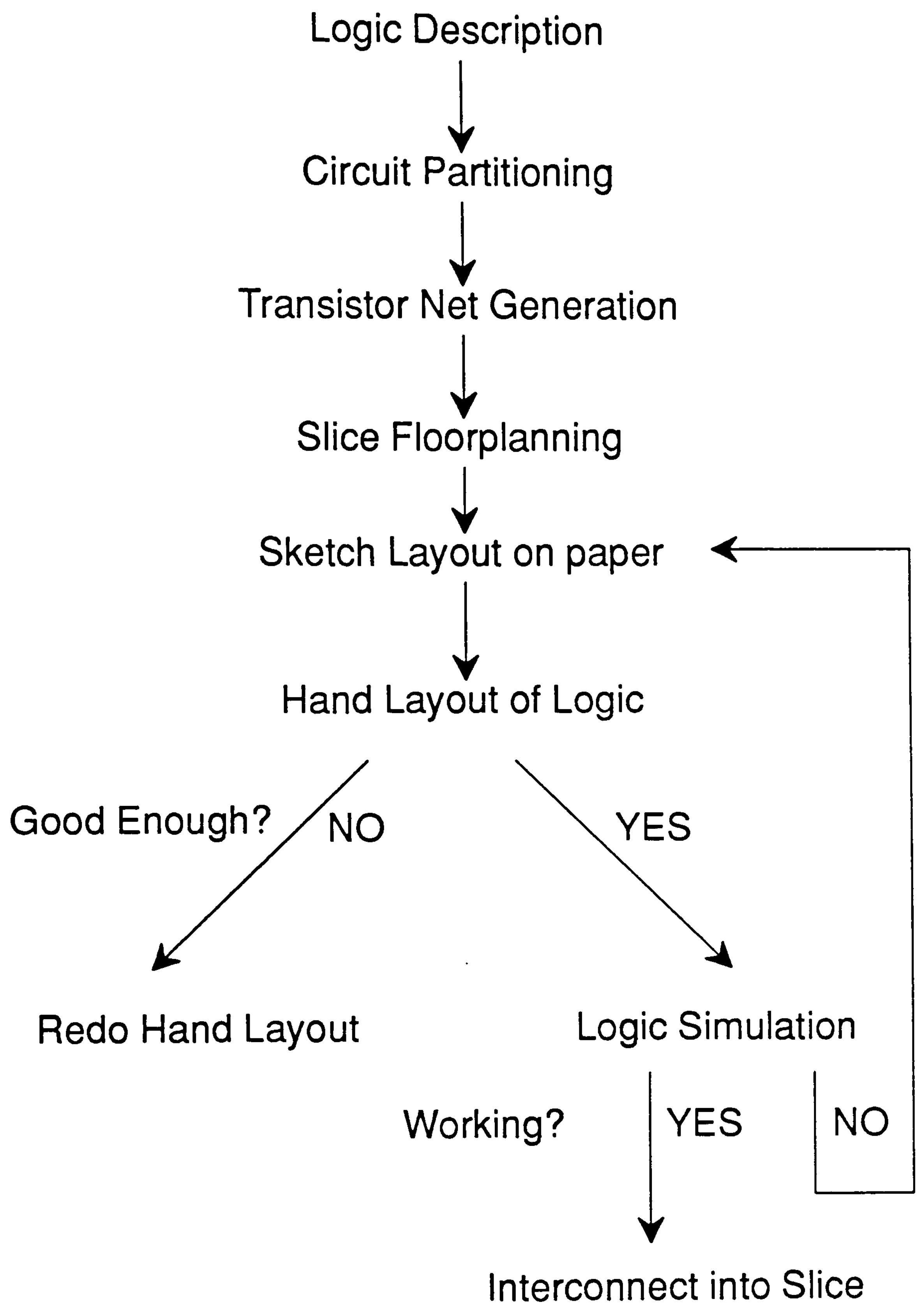
Layout commenced, with the positions of polysilicon lines within blocks being fixed, using the gate matrix style. The process was found to be very systematic and mistakes could be corrected rapidly.

A plot of the final CIF file for the manually laid multiplier is shown in Figure 5.3 .



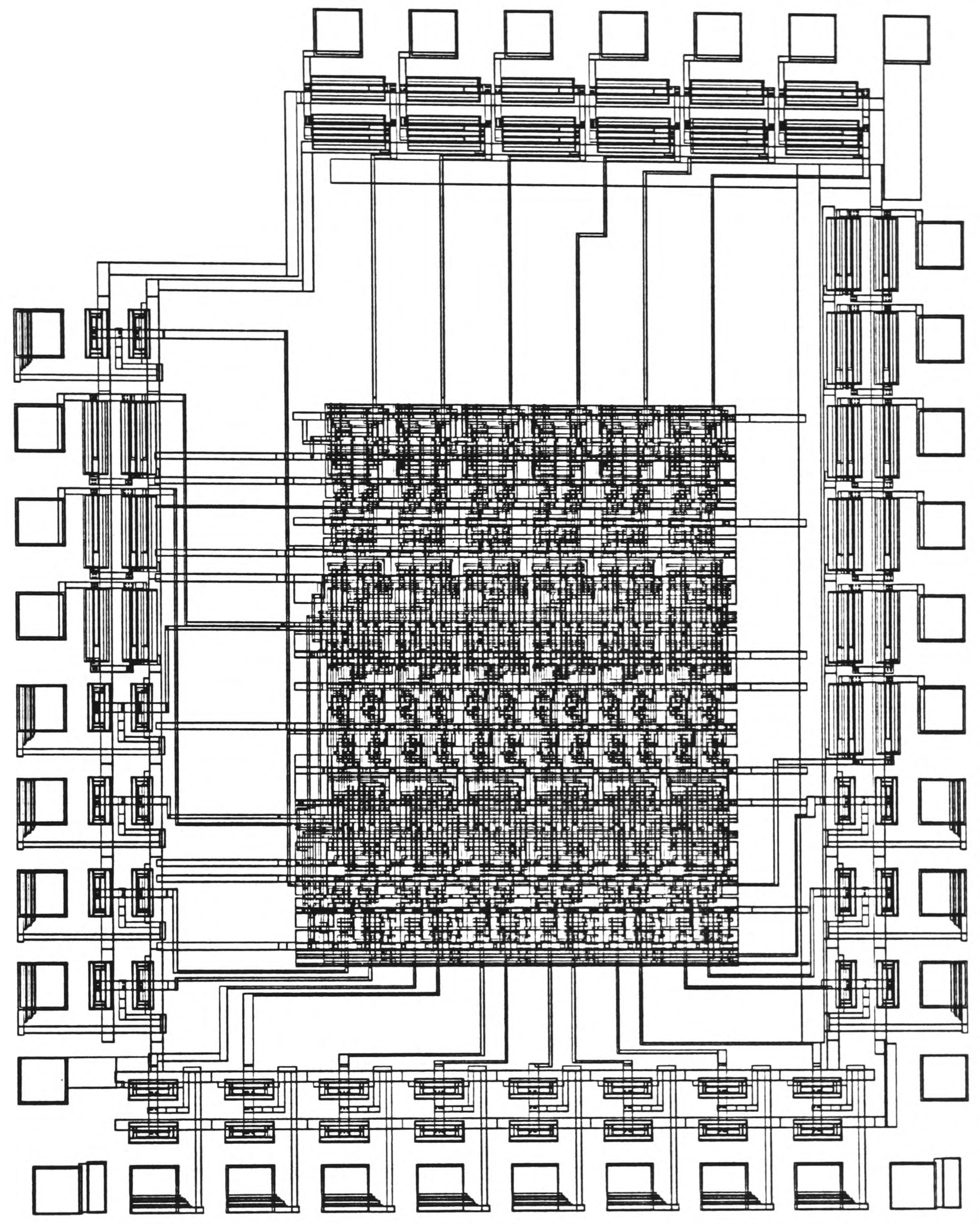
Partitioned Logic of the Multiplier Slice

Figure 5.1



Flowchart of Manual Layout Route

Figure 5.2



Plot of Multiplier containing 6 Slices

Figure 5.3

Deviations from Standard Gate–Matrix

To achieve a higher density in layout, some deviations from normal gate matrix were made. These changes may be summarised as:

1. Contacts from first layer metal to polysilicon and diffusion were not allowed to be adjacent.
2. Some use of second layer metal was allowed within the circuit block.
3. Shifting of contacts from their normal central grid positions to alleviate design rule errors.

These modifications arose from performing the layout by hand, where optimisations could be seen. Adhering to rules 1 and 3 above, enabled a seven by eleven micron grid to be used to lay out microcell elements without introducing design rule errors. If these rules were relaxed then an eleven by eleven micron grid would be needed. These rules make automatic synthesis of layout difficult and they are not incorporated into our automated layout software where we have used the increased grid size to overcome design rule errors. Rule 2 above was found convenient in hand layout of the block and allowed a smaller block to be produced. It is useful to ban the use of second layer metal within a logic block so that it can be used purely for interconnecting blocks.

Substrate contacts were put into the slice by placing them into the power and ground rails. These rails were increased in width to accommodate their presence. As the MCE design rules recommend that the spacing between a transistor and a substrate contact should never exceed a design rule specific distance to avoid latch up, this placed another restriction on the layout of the block. Transistors within the block must never be more than this distance away from a supply rail containing substrate contacts.

Connection of the Blocks into a Slice

Once the individual logic blocks had been laid out manually, in accordance with our slice floorplan, the blocks were connected together. The approach we followed was to use first layer metal in the horizontal direction (i.e. perpendicular to the slice) and second layer metal in the vertical direction. Using polysilicon to join signals between adjacent blocks was avoided as much as possible to avoid long lengths of polysilicon degrading performance.

The completed logic blocks were then divided centrally between the n–type and p–type transistors and the space widened, giving enough space for us to fit all the horizontal first layer metal signals which require contacts to the block. This can be seen on Figure 5.4, the musum block. This wiring scheme is not wasteful of area as the design rules specify a minimum spacing between N and P wells.

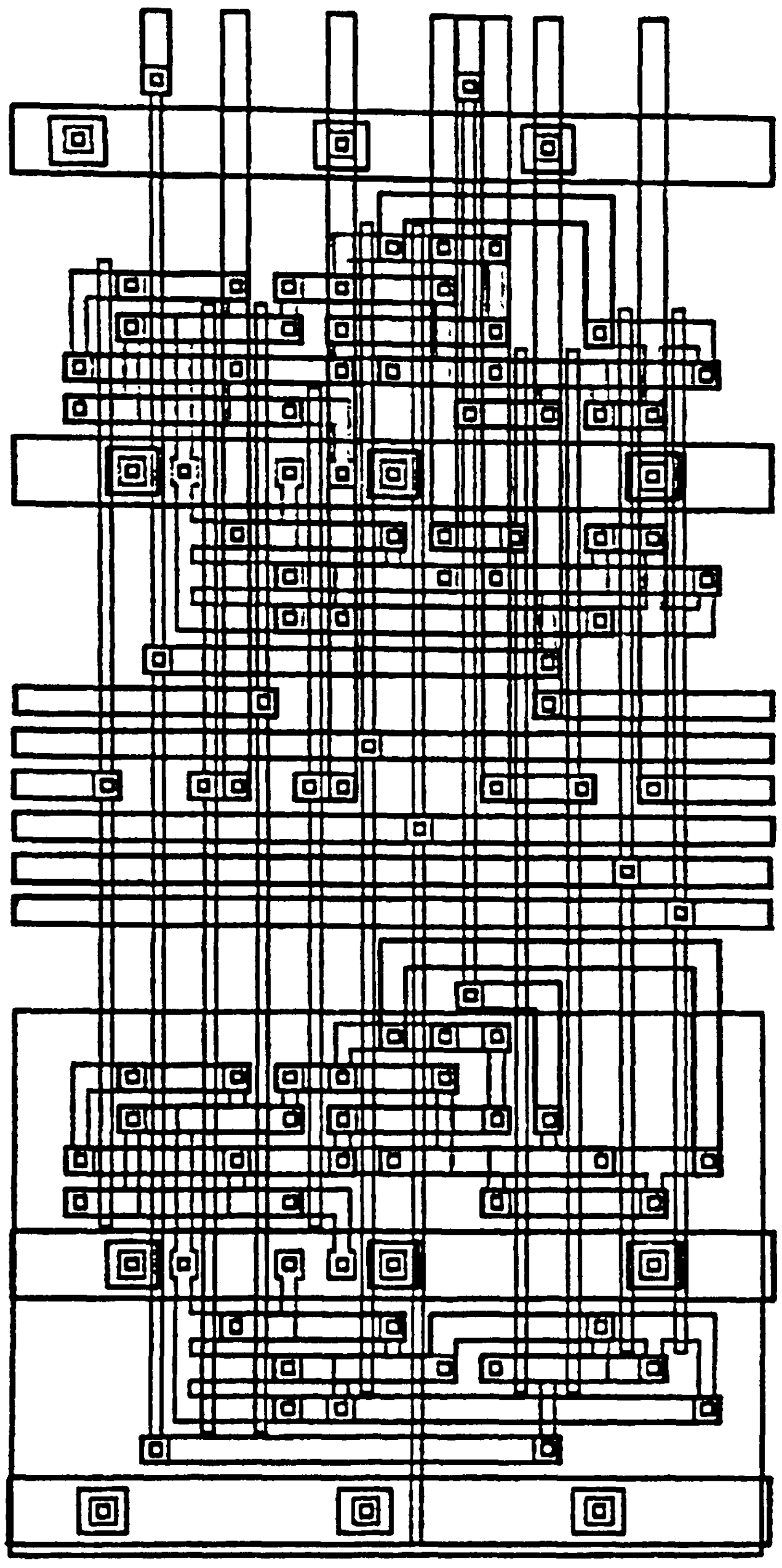
Conclusions from Manual Layout

The multiplier slice contained 175 transistors and was laid in an area of 1565 microns by 180 microns. Using the regular style of gate matrix made the job considerably easier than trying to lay out the slice with no methodology. Some observations on the gate matrix style which are relevant:

1. With the large logic blocks used in the multiplier, long polysilicon lines tended to be produced. This may degrade performance as polysilicon has a high resistivity. The length of these polysilicon lines can be shortened by keeping logic blocks small.

2. In the manual layout of the multiplier, some optimisations were made, described earlier. They would be difficult to implement in a piece of software as they tend to be special cases. These optimisations do not warrant the extra effort required for their inclusion into the layout automation software.

3. The interconnection of blocks was found straightforward. Blocks could be laid out independently then joined together. This is useful for a hierarchical layout design style.



MUSUM block - Hand Layout

Plot of Manually laid MUSUM Block

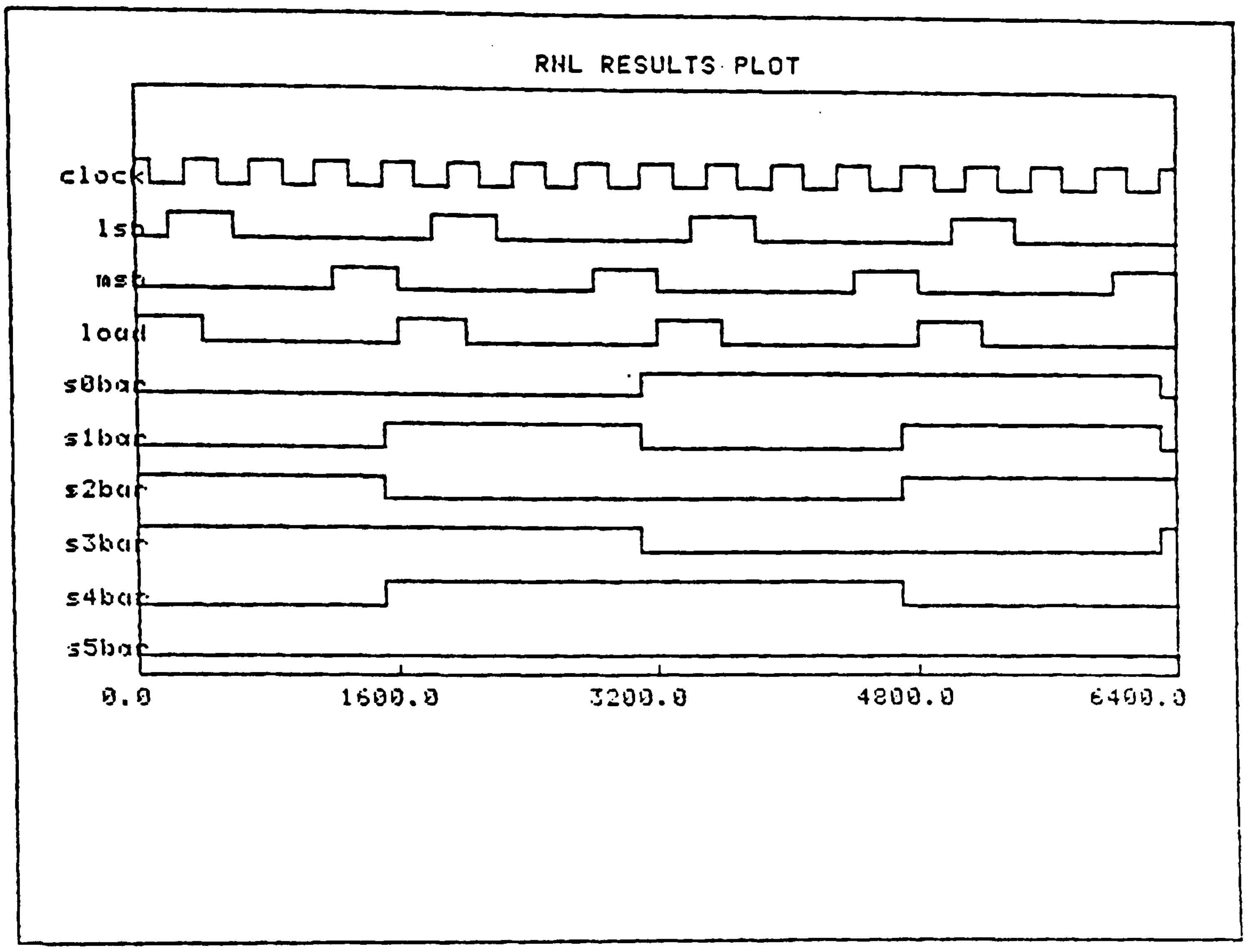
Figure 5.4

5.3. RNL Results

Simulations of the complete chip were performed using the switch level simulator RNL[82]. These results indicated a problem with the extracted circuit. After extensive simulation, the problem was traced to two output pads which were driving into the circuit. These outputs, from the pads, were forcing lows into the desired test points, because of their enhanced drive capability, the floating input of the pad presumably taking a logic high value.

Simulations of the circuits were repeated taking into account the low logic levels at these modes and the results obtained correspond with subsequent electrical testing. The effect of the error was to drive the least significant output bit to a low and also, in some cases, to subtract a one from the answer produced by the remaining bits, because of the inhibited carry from the least significant bit.

To test the circuit a set of input vectors which cause the outputs to change sufficiently (i.e. 2 or 3 output transitions per vector) were generated. These vectors were then used in simulations with RNL to obtain the expected results and were later checked with electrical tests on the fabricated chips. Results from a typical RNL simulation are shown in Figure 5.5, which is in the format of a graphical waveform, showing the results from four test vectors. When testing, sixteen test vectors were used to check the functionality of the multiplier.



RNL Simulation Results

Figure 5.5

5.4. Errors in the Layout of the Multiplier

Two significant errors crept in during the manual layout process. These were :–

– the connection of an output pad, instead of the input to an internal signal,

– the clock line being routed in polysilicon, thus limiting the maximum clock rate possible.

The second error was not critical to functionality, but would make our performance estimates less reliable as speed could be limited by slow clock rise and fall times rather than logic delays. The first error was more critical to circuit operation and did affect the functionality of our multiplier. An output pad connected to an internal test point to observe the serial partial product sum of the last slice was incorrectly wired. This point was connected to the output of the pad instead of its input, driving a logic value into the test point.

Resimulation of the circuit, modelling the effect of this error by holding that test node at a constant logical low produced results with which we could compare with the fabricated chip.

With the circuit simulator used, RNL this incorrectly wired pad became a source of X state or unknown logic values, which the simulator propagated throughout the circuit. With this X state no useful results could be obtained so it was necessary to fix this troublesome node. Correcting this was accomplished by locating the node and then setting it to a zero logic state and resimulating. The results from this simulation provided sensible results affecting only the least significant bit of the answer. Further tests were compared with these results.

5.5. Electrical Tests

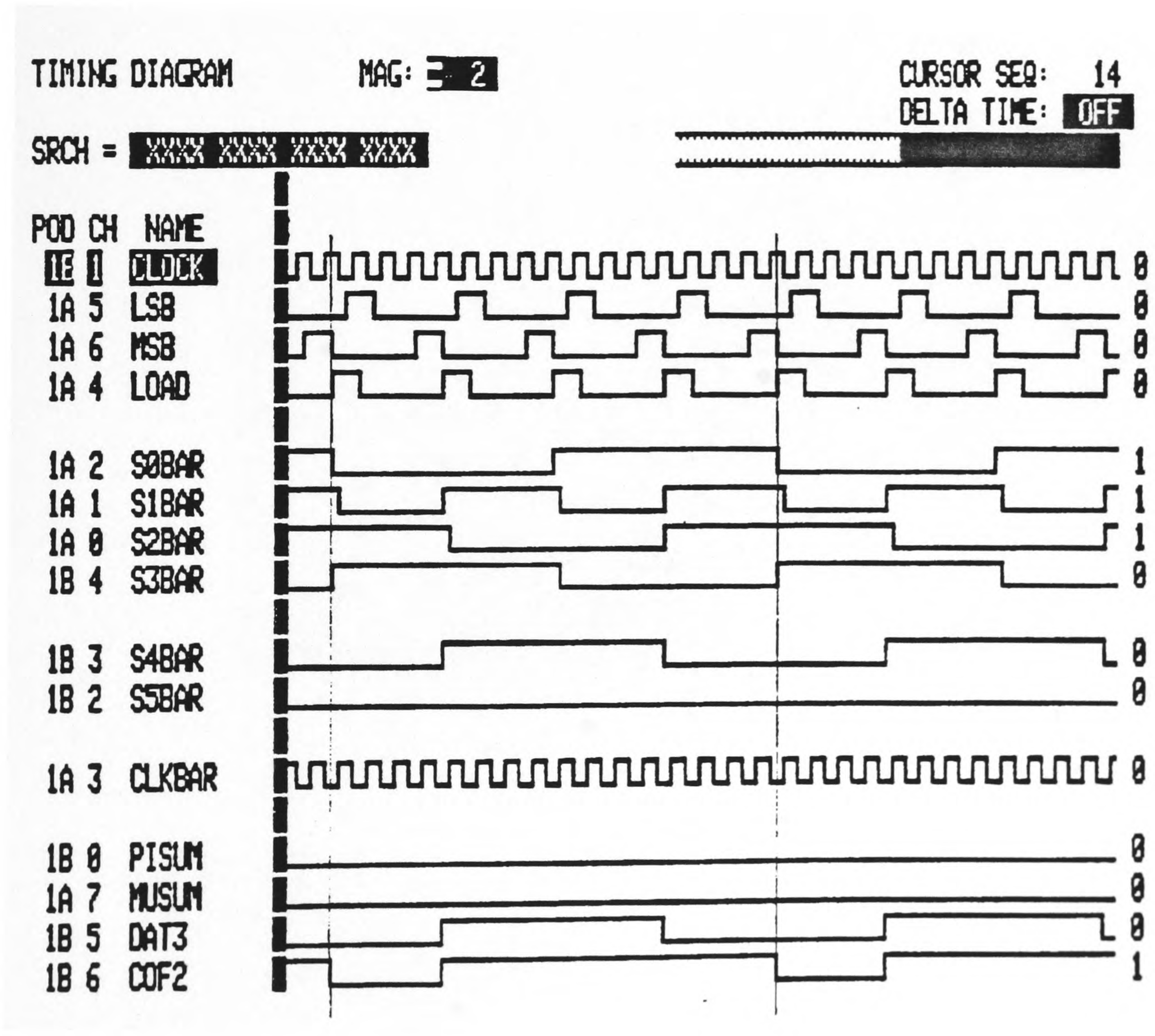
The electrical tests were performed using a Tektronix Digital Analysis System (DAS 9100), capable of performing the tests up to a maximum clock rate of 10MHz. From the twenty chips fabricated six were found to be functional. A chip passed the test if the RNL plot was found to be identical with that obtained from the DAS, given that the timings might differ slightly, it failed otherwise. To exhaustively test the 6 bit by 6 bit multiplier, $2^6 * 2^6 = 4096$ test vector inputs would have been required. This would have been quite an unmanageable number of vectors to input into the DAS 9100 and then visually check against the rnl plots, so only 16 test vectors were used. It is possible that a faulty chip could have passed, but this is unlikely even with this small number of test vectors, as the input vectors were chosen to give a high number of transitions on the outputs to show multiplier activity and as the bit–serial multiplier is composed of 6 regular repeating blocks, any error should have been easily visible. The criticism against this *ad hoc* approach to testing remains, and something similar to the self–test strategy in [83] would have been a preferable way to test the multipliers, but wasn't available in our timescales.

Chips were found functional in the range 1Hz to 10MHz using the Tektronix DAS. It was felt desirable to find the limiting speed for the chips, as an estimation of the performance of these chips. With this purpose in mind, a small TTL test circuit was constructed to provide the correct control signals to the chip under test. The details of the test circuit are given in Appendix 2.

It was found, from the working chips, that operation until 11MHz was possible, with a few operating until 12MHz. From the working chips all worked to 1Hz, with the exception of a single chip which drew paradoxically larger currents at low frequencies.

The chips worked as expected from RNL simulations when the incorrectly wired output pads were taken into account.

The output from the DAS for the data shown in Figure 5.5, is shown in Figure 5.6 .

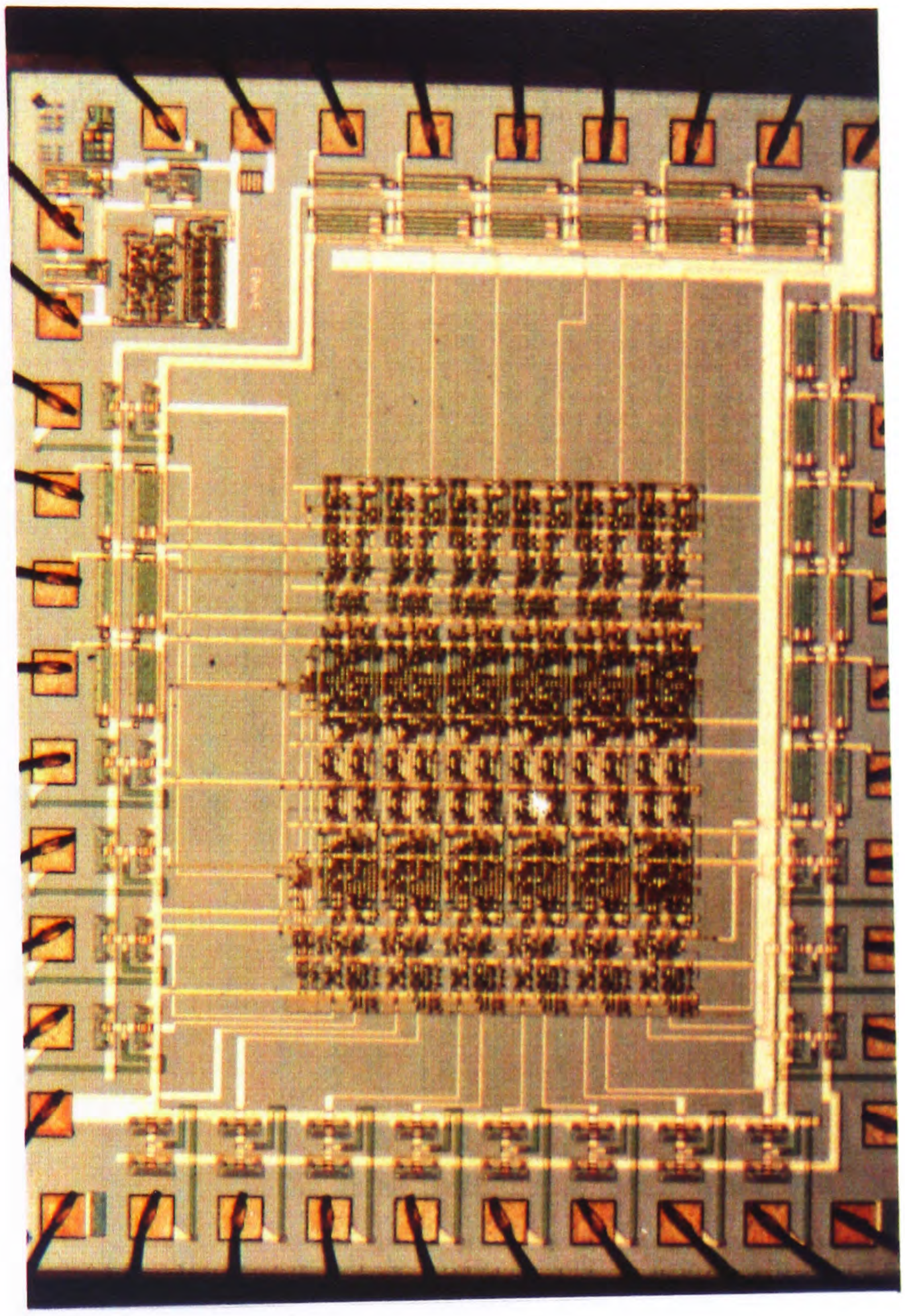


Electrical Test Results using Tektronix DAS

Figure 5.6

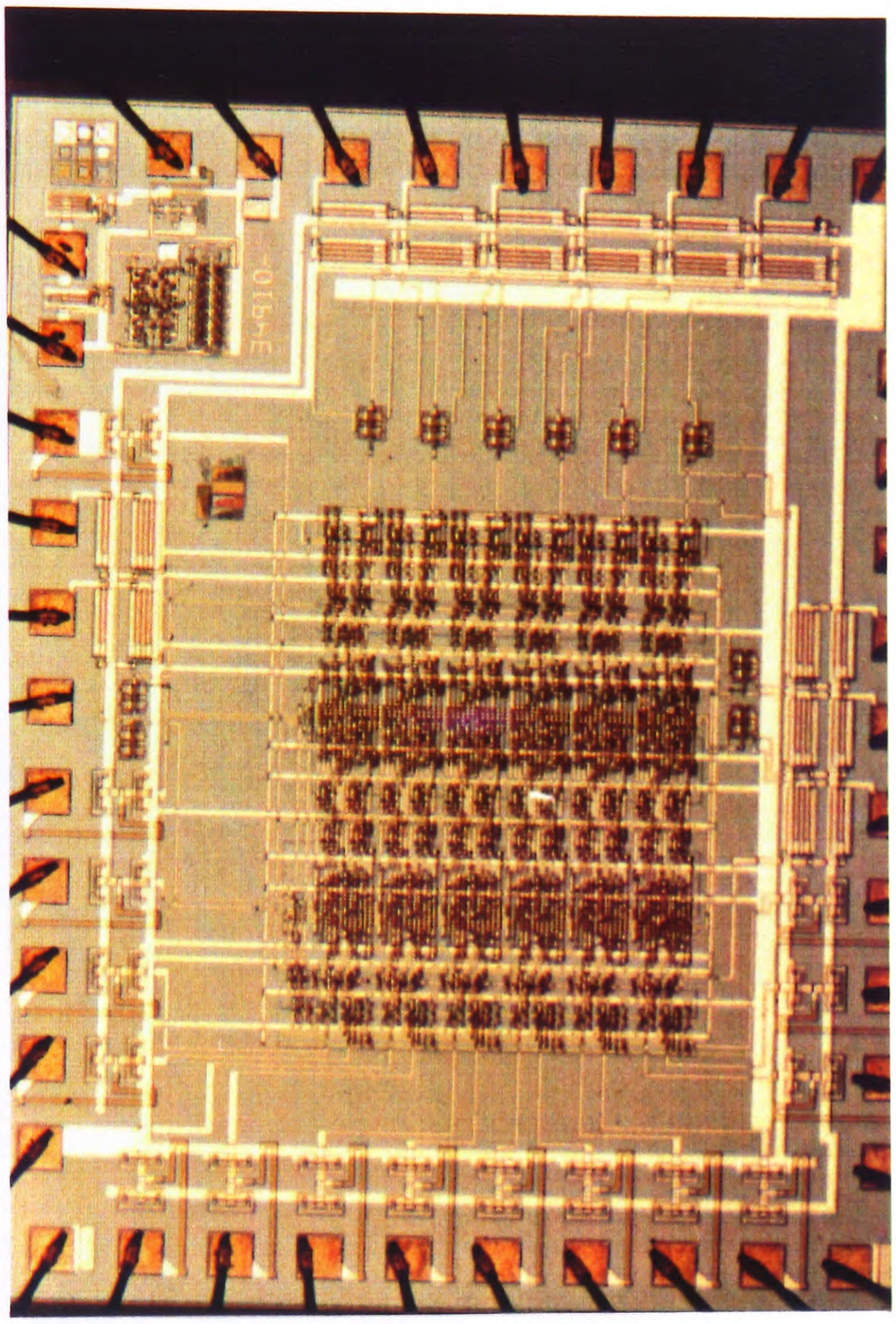
Corrected Versions of Multipliers

The multipliers were corrected[84] and refabricated. They were found to be working when the same testing procedure was applied to them. Photographs of each of the multipliers, Figure 5.7 and Figure 5.8 are given at the end of this chapter. Also included are photographs of a zoom–in of the multiplier bit–slices, Figure 5.9, and a version of the multiplier implemented using the tiling scheme, Figure 5.10, which was tested in an identical fashion as it is functionally equivalent.



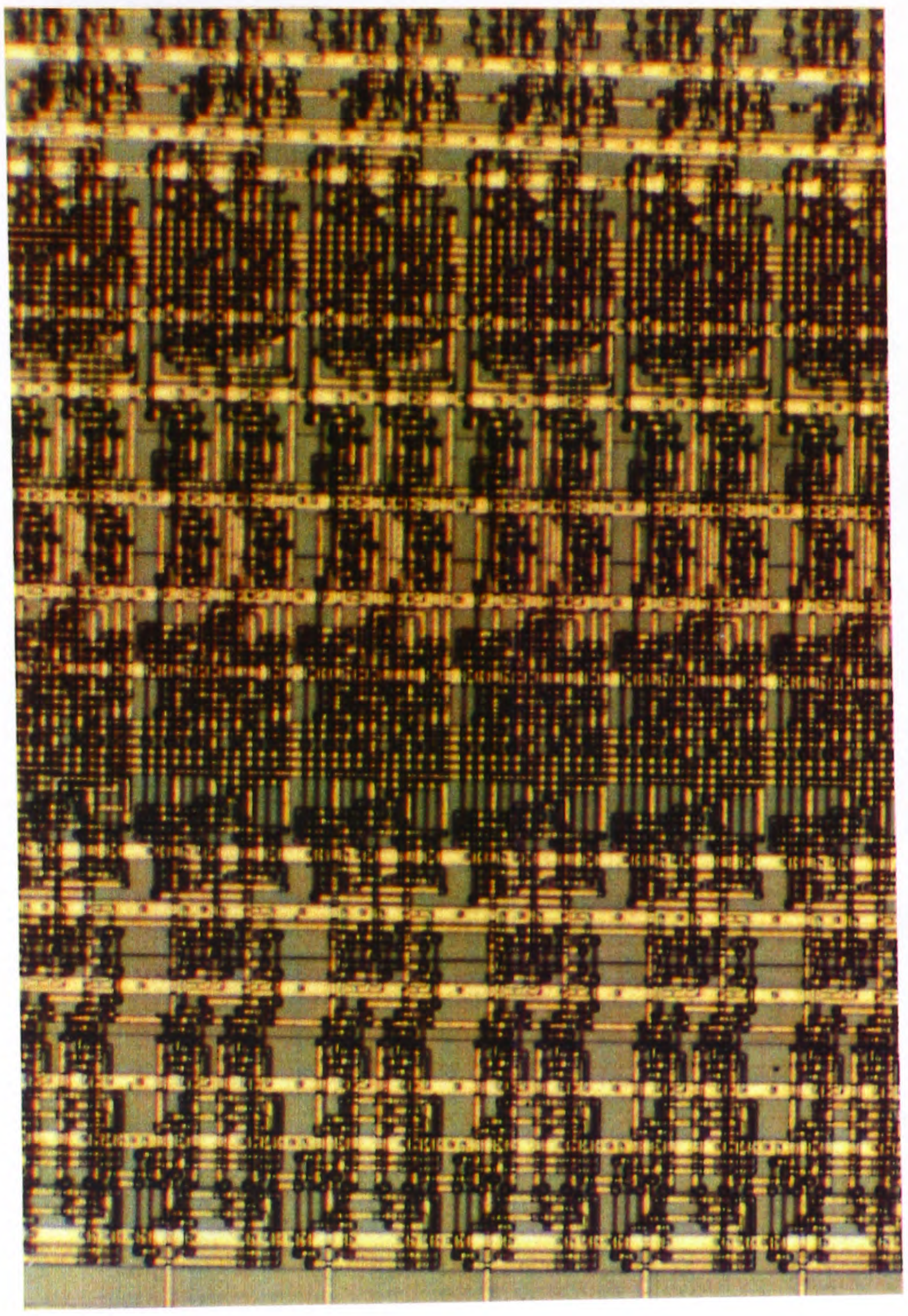
Photograph of Multiplier

Figure 5.7



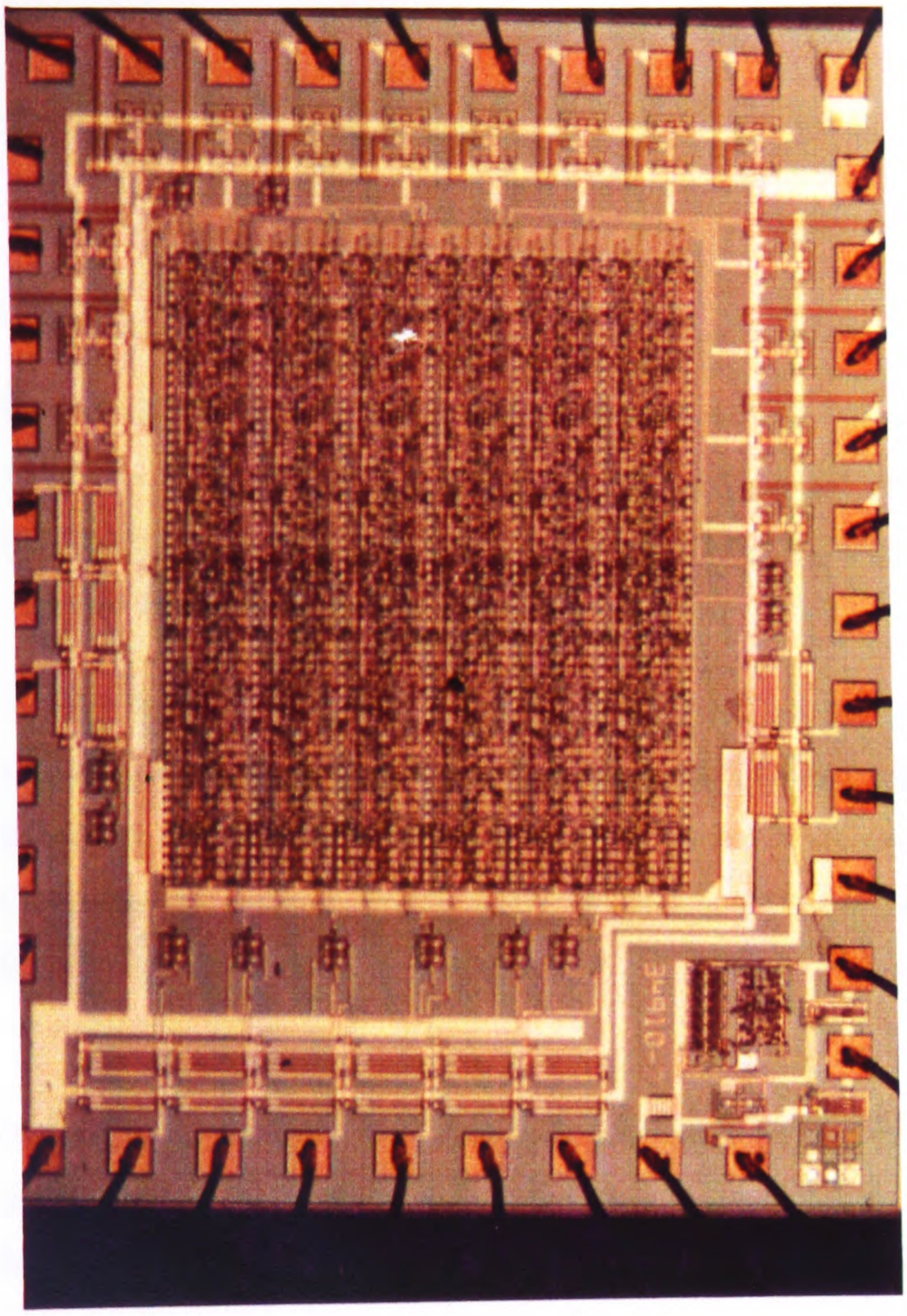
Photograph of Corrected Multiplier

Figure 5.8



Photograph of Bit–Slices

Figure 5.9



Photograph of Tiling Multiplier

Figure 5.10

Chapter 6

SECOND

6.1. Introduction

The major limitation of FIRST is its reliance on an underlying hardware function cell library. These hand laid pieces of logic provide dense efficient layout and high performance, but also place restrictions on the user of the software. The user is restrained to the process rules used by the cell library and also to only those functions given in the library.

In the earlier chapter on the MOS software suite, we have seen that it is possible to produce layout of a user specified logic block in a symbolic form which does not limit the designer to a single set of process rules and allows the adoption of new process rules as they become available. The software generates the transistor network layout from user specifications of the logic blocks, allowing the inclusion into the cell library of new elements as needed in a process independent form.

This chapter describes an extension to FIRST as a trial system which uses the MOS software as a basis on which to build a piece of software which generates elements of the cell library to FIRST automatically. The software has been named *SECOND*[85], [86], [87], (Synthesis of Elementary Cell ON Demand) and gives FIRST the extra flexibility provided by automatic generation of transistor network layout from a user specification in a process independent form.

6.2. Goals

With *SECOND*, we wish to achieve a full–span structural silicon compiler, that is a compiler capable of the complete transformation from a structural specification of an integrated system to a usable silicon layout without being dependent on an intermediate cell library, fixed to a single set of process rules. The software has the capability of generating the cell library components automatically in a process free form, using the gate matrix layout methodology.

A compiler such as FIRST is capable of the transformation from structural specification to layout, but it requires the complete manual layout of a new cell library to be used with a new process rule set. The new software, by generating layout automatically from process rule information gives FIRST enhanced features of flexibility and portability. Portability is achieved through the generation of gate matrix, a process independent symbolic representation, which can be fleshed out to the required process rules. This allows the update of the library to a new process by the alteration of a small information base. Flexibility is obtained by giving the designer the capability of designing systems requiring components not already available in the cell library, as the ability to specify such components will be present.

Low level errors, that is errors in the layout and interconnection of individual transistors are prevented as these functions are carried out by known working pieces of software. High level errors still need to be dealt with, and they can be through simulation of the system with a high level simulator. FIRST provides such a simulator for use with its PRIMITIVE library, but such a simulator would have to be altered to allow the simulation of systems incorporating user specified components.

6.3. Half Span Structural Compilation

Before describing the procedure for assembly in SECOND, a review of the assembly procedure in FIRST is useful.

The interconnection of PRIMITIVEs in FIRST is specified by a high level structural language. The invocation and interconnection of these PRIMITIVEs are specified directly by the language allowing the specification of a signal processing system as a flowgraph of computational elements. To each computational element there corresponds a PRIMITIVE which carries out the function required.

The PRIMITIVEs used in FIRST are parameterisable, allowing PRIMITIVEs of differing wordlengths or latency characteristics to be specified. Parametrisation of the PRIMTIVEs is achieved by constructing the elements from a fixed set of manually designed leaf cells which are bound to a particular set of process rules. The construction of each of the PRIMITIVEs from its corresponding leaf cells is carried out by software assembly procedures particular to each PRIMITIVE.

FIRST differentiates between different levels of complexity by maintaining a typed hierarchy as illustrated in Figure 6.1, each level having a physical significance. The levels indicated in lower case levels are notional and are used for design management only.

```
  SYSTEM
 subsystem
   CHIP
 operator
 PRIMITIVE
```

Hierarchy used in FIRST

Figure 6.1

The construction of an object obeys the rules :–

– Physical objects can only contain objects from the next lowest physical level.

– Notional objects can contain objects of equal or lower levels, but no lower than the next physical level.

The manifestation of these rules is seen in the assembly procedures. At each level the problems of placement and routing are particular to that level, hence the assembly procedures are independent. In FIRST only assembly of the CHIP level takes place as assembly of the SYSTEM level was never implemented. Commercial board layout packages are already available which can perform this function. It is still useful to distinguish between levels of hierarchy as they take a greater significance when they are extended.

Assembly at the CHIP level is undertaken by place and route software, described in chapter 1, which takes as its input a flattened netlist of PRIMITIVEs produced after processing the FIRST language description. The PRIMITIVEs are treated as rectangles with a well defined geometric interface. The system constructed by the connection of these PRIMITIVEs is guaranteed to function because of the electrical conventions used in signaling.

6.4. Full Span Structural Compilation : SECOND

By extending the hierarchy of levels present in FIRST, we can extend the silicon compilation theme down to the generation of symbolic layout. In this manner the software structure of FIRST is built upon without being radically altered. More complex bit–serial PRIMITIVEs, of which the multiplier described is typical, we have found can be built up from bit–slices.

At the lowest level in the hierarchy are the symbolic layout blocks generated with the MOSLAY program. These blocks form a process rule independent base which are assembled together to form a complete system. The assembly procedure must also be process independent to maintain this capability with all generated layout. These elements we term BLOCKs and use them as a level of hierarchy in our full span compilation scheme. As each of these BLOCKs is built out of the smaller microcell units, the lowest level in the hierarchy is the level of the microcells which we term DEVICEs.

We assemble BLOCKs into bit–slices, which cascade together to form a PRIMITIVE of FIRST of the required wordlength, where the size of the PRIMITIVE is determined by the wordlength. The level of hierarchy of the bit–slices forms the next level in our expanded hierarchy, which we term the SLICE level.

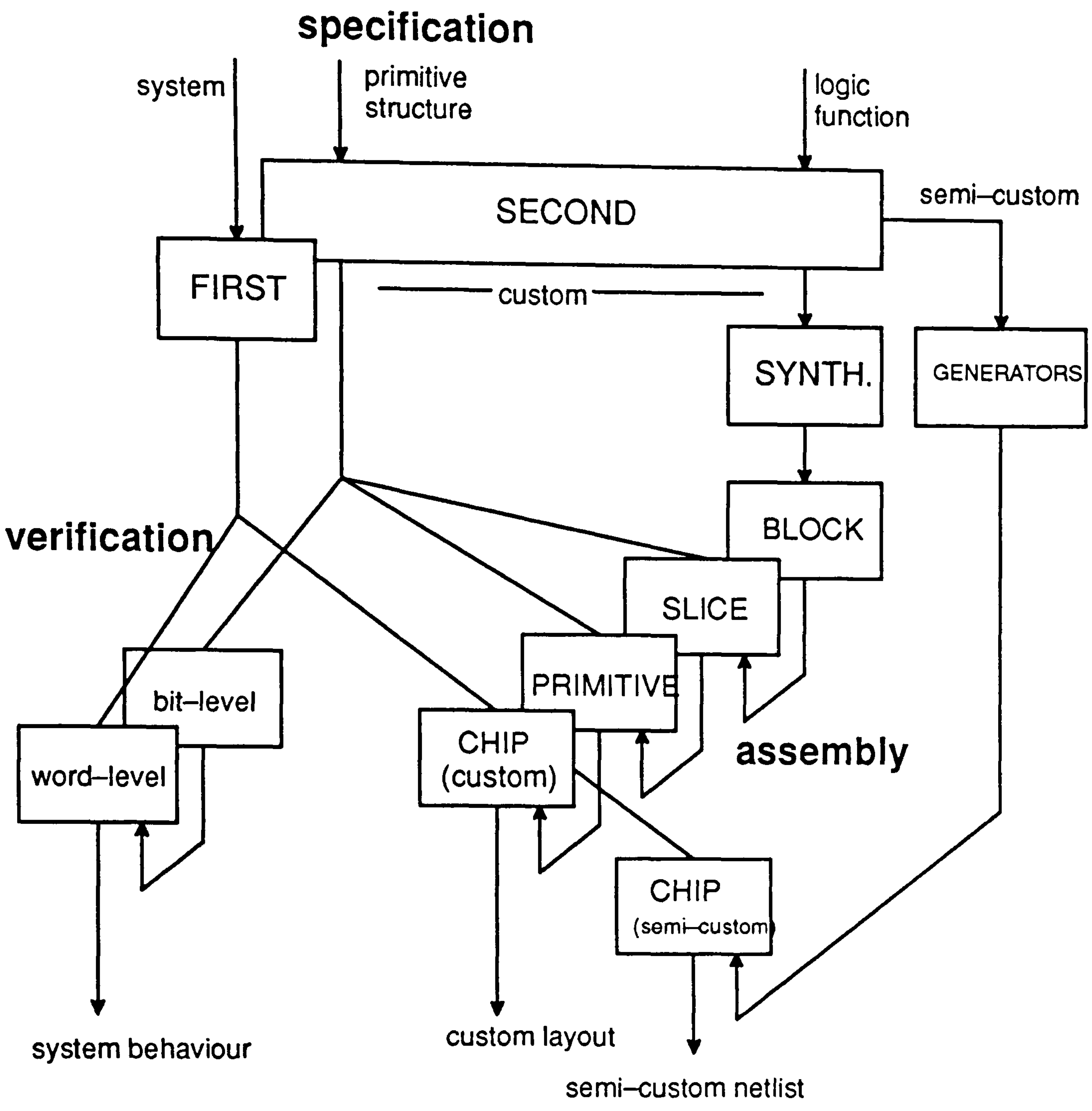
Figure 6.2 shows the extended hierarchy used in SECOND.

```
SYSTEM
subsystem
CHIP
operator
PRIMITIVE
subprimitive
SLICE
subslice
BLOCK
subblock
DEVICE
```

Extended Hierarchy of SECOND

Figure 6.2

Construction of each level in the hierarchy has its own assembly procedure as before, but now the hierarchy extends right down to symbolic layout allowing process tolerance to be achieved. The relationship of SECOND to FIRST is illustrated by Figure 6.3, showing hierarchy and domains of each piece of software. A path is shown which allows the netlist for a semi–custom implementation. A gate netlist also allows portability between processes, the problem of layout becoming the responsibility of the gate array vendor. The process of generating such a netlist will be described in the next chapter.



Software Domains/Architecture of FIRST and SECOND

Figure 6.3

6.5. Software Structure of SECOND

The software of SECOND covers the generation of layout for BLOCKs from their logical descriptions, to their assembly into SLICEs and PRIMITIVEs for inclusion into FIRST.

A rewrite of FIRST giving it a better floorplanner and router was also envisaged but the software never reached this stage. A trial system was implemented, writing the software for each of the major pieces but these pieces were never assembled together to form a complete system. This decision was taken after comparing results from SECOND with those of a commercial system.

Synthesis starts with the logical descriptions of logic blocks which are transformed into equivalent symbolic layout using the transistor net synthesis software MOSYN described earlier. SECOND either directs synthesis of these BLOCKs or selects the appropriate component from a library of symbolic layout. Construction of these BLOCKs to form a SLICE is undertaken by *slicesyn*[88] which orders BLOCKs and produces routing between the BLOCKs.

Once a SLICE has been constructed, SLICEs can be abutted as alignment of signals is performed by *slicesyn* to form a complete PRIMITIVE. The PRIMITIVEs can then be used with FIRST to provide the required components for the system. Figure 6.4 shows the flow of assembly for the software.

```
                Logic Partioning
       Order terminals for each logic BLOCK
Stretch BLOCKs to equalise widths and align terminals
         Linear Placement of BLOCKs within SLICE
            Routing of BLOCKs to form SLICE
         Abutment of SLICEs to form PRIMITIVE
                Use within FIRST
```

Flow of Assembly

Figure 6.4

Some PRIMITIVEs, such as the adder are composed out of a single SLICE as the width of the adder is independent of wordlength, but those whose width is dependent on wordlength are constructed out of multiple SLICEs.

6.6. Design Capture

Specification of the system components had started to be input in a graphical fashion. The member of the group who worked on this piece of software left during the phase of the project so that the software was never completed.

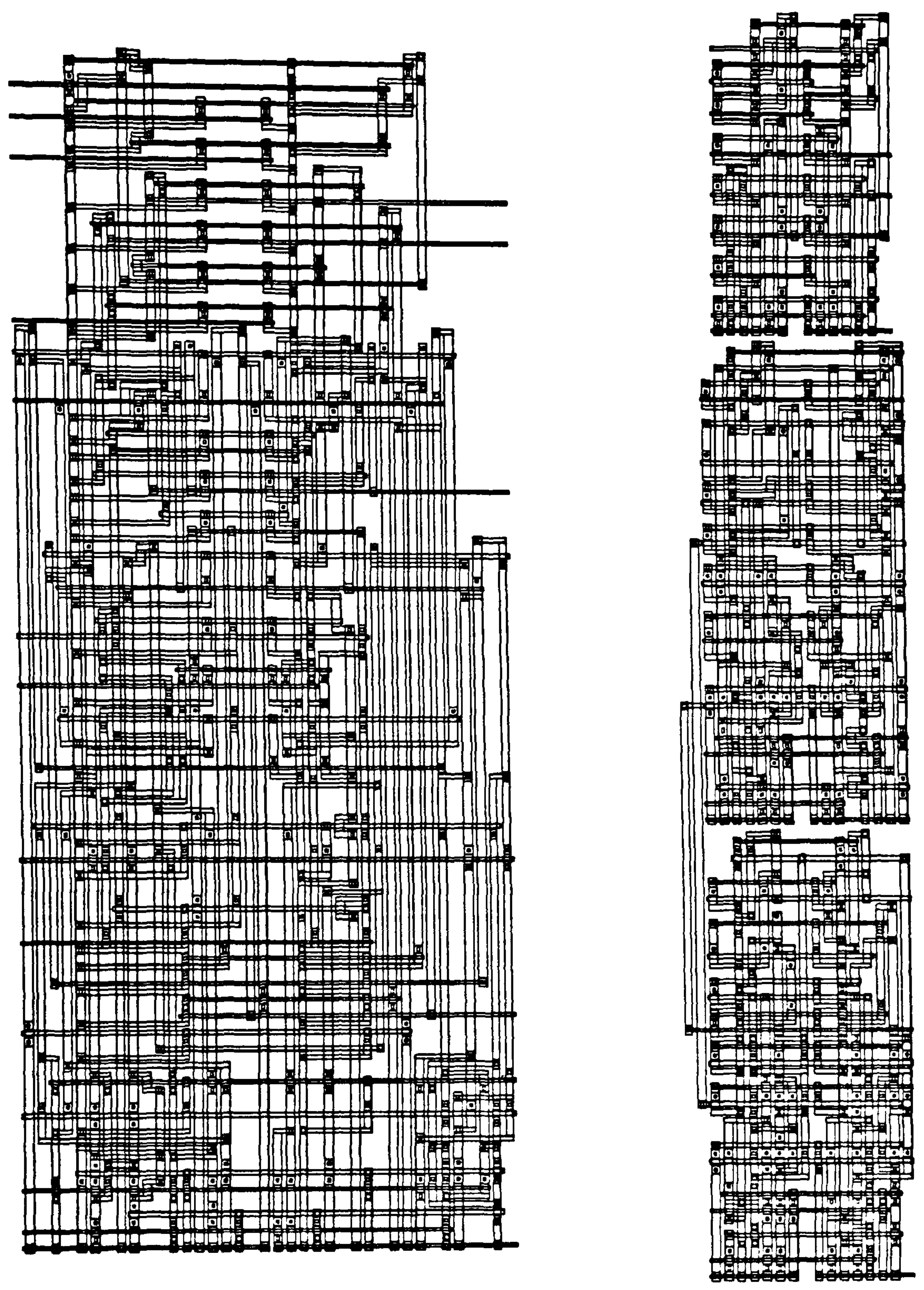
The purpose of the software was to provide a description of the logic up until the SLICE level through a graphical specification. This has the advantage of being more familiar to designers than a specification in a language although a language description is just as powerful.

The software was called VX[89] and written in LISP [90] and provided the logic specification of a SLICE including its hierarchical details in the form of a list. LISP was seen as a useful language in the software development, it was used in the writing of much of the software for SECOND, as it has simple input and output facilities and can operate on symbolic data directly.

6.7. Logic Partioning

The partioning of logic can decide the size and compactness of generated logic blocks and hence the SLICE. Dense strongly interconnected pieces of logic should be together and split into smaller blocks if too complex. Inefficiencies in the layout can arise if the generated BLOCKs are too large. As the complexity of a BLOCK depends on the number of inputs to it, the number of inputs to a BLOCK should be kept at about five or six. Figure 6.5 shows the effect of partitioning the logic of the multiplier and integrating the generated blocks into a slice structure rather than synthesising it as one large block. In the generation of the SLICE, the width of BLOCKs are made equal so that it is important to keep the width of the largest BLOCK minimal.

The partioning of logic is a difficult problem to perform automatically, so that partioning of the logic is taken from the specification given by the user. Using the hierarchical specification of logic in the users description of the SLICE is the simplest method of obtaining partioning information. Closely coupled portions of circuitry distinguished by large numbers of interconnections will be kept together unless the user partitions the circuit badly in their specification. It has the advantage of simplicity, but the disadvantage that different specifications of an otherwise identical (logically) circuit can give rise to different layouts depending on the user specification.

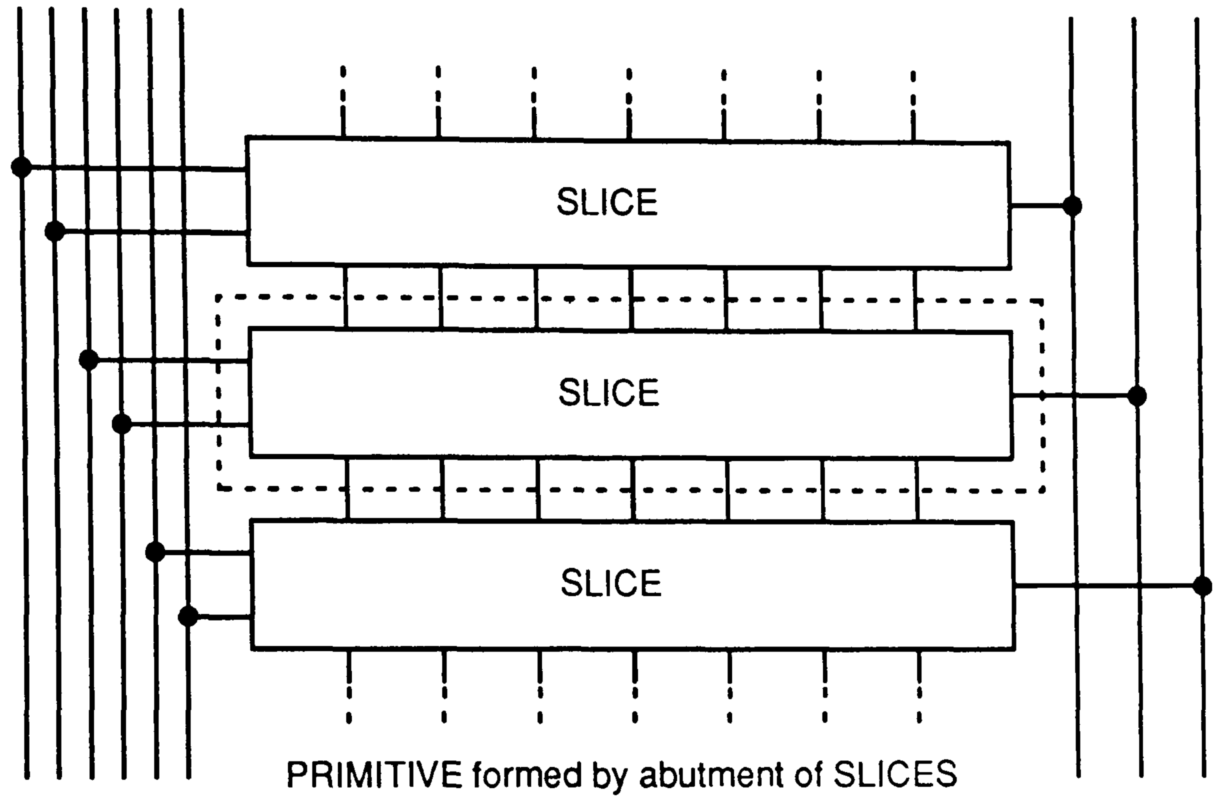


Effect of Partitioning a BLOCK

Figure 6.5

6.8. SLICE Construction

The construction of SLICEs forms an important stage in the synthesis of the final layout. As SLICEs are meant to join together by abutment, signals between SLICEs must be placed so that they align correctly. Power, ground and clock rails can run perpendicular across the SLICE providing common amenities. We place substrate contacts within the power and ground rails, widening them for this purpose. The PRIMITIVE produced takes the form shown in Figure 6.6, interconnection between SLICEs is taken care of by the alignment of signals.



Simplified Structure of PRIMITIVE

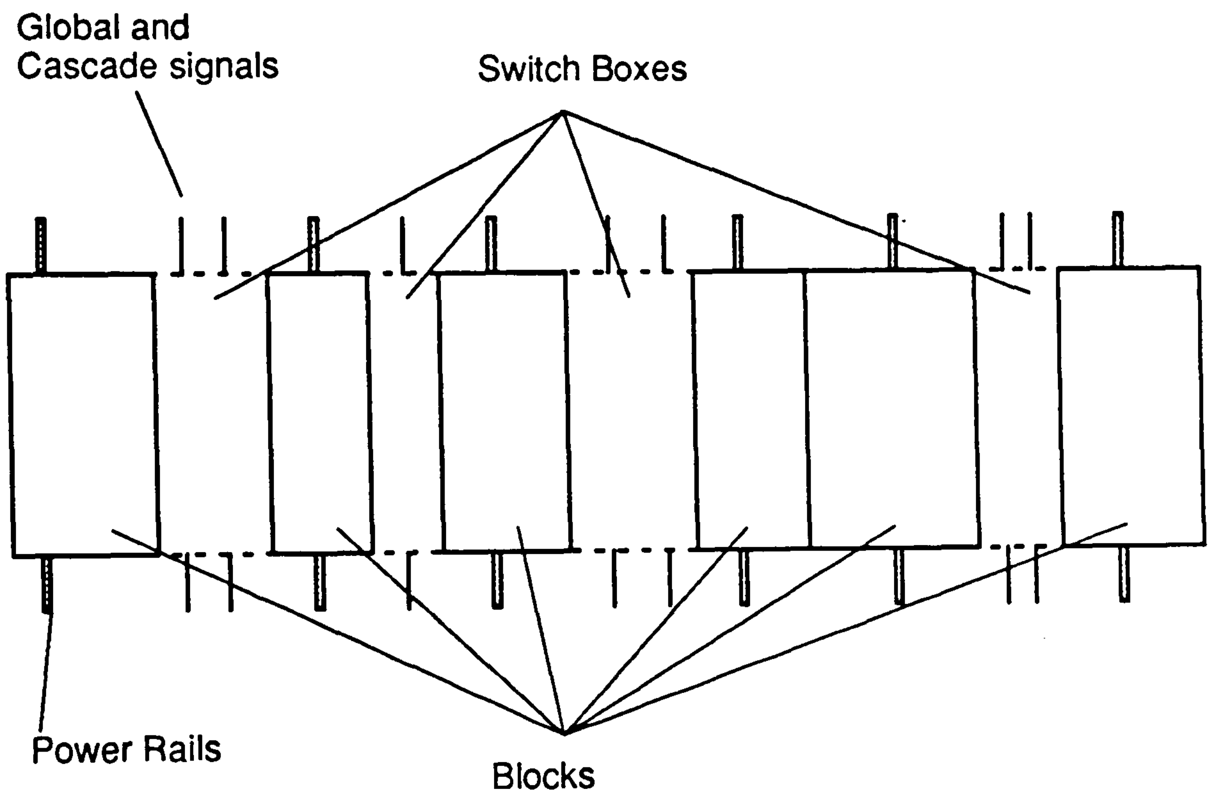
Figure 6.6

A SLICE consists of several BLOCKs. The BLOCKs are laid linearly in a line with routing channels between each BLOCK, as shown in Figure 6.7, the routing channels allow connections to take place between BLOCKs and also SLICE to SLICE signals to be routed. In the routing channel, second layer metal is only used in the direction along the slice, and only for communication between the BLOCKs of the SLICE, first layer metal is restricted to run in the direction across the slice.

The layout produced by MOSYN and MOSLAY uses only first layer metal, which is very useful as this allows second layer metal to run over BLOCKs and gives us a method by which we can link signals which communicate between the BLOCKs. Thus we have the two following restrictions:

(a) SLICE assembly takes place with a two level metal process.

(b) Second layer metal should not be used within the BLOCK, as it is used as a connecting layer for signals between BLOCKs.



Structure of SLICE

Figure 6.7

The ordering of BLOCKs within a SLICE uses a heuristic min–cut procedure. A connection graph for the BLOCK to BLOCK signals is constructed and the vertices of this graph are ordered to give a minimum cut (i.e. the number of edges which need to be broken to divide the graph) between nodes. The ordering of the nodes determines the ordering of the BLOCKs.

The assembly of the SLICE[91] then takes the route shown in Figure 6.8 .

```
          Decide a terminal ordering for each BLOCK
           Synthesise transistor net (using MOSYN)
               Form symbolic layout (subBLOCK)
             Stretch subBLOCKs to equalise widths
  Connect latches if single-phase/mixed-tree logic used --
           the equalised subBLOCKs are our BLOCKs
                      Place the BLOCKS
                   Route between the BLOCKs
```

SLICE Assembly

Figure 6.8

It is useful to handle the layout image as symbolic layout during this assembly flow. This allows us to incorporate imported symbolic layout into the scheme, these predefined pieces then do not need recompilation when changing technology. SubBLOCK stretching introduces gaps into a BLOCK but aligns signals, enabling the routing channels to be narrower. As the subBLOCKs vary in width, there is little penalty in performing this operation.

Three types of signals are found within a SLICE :

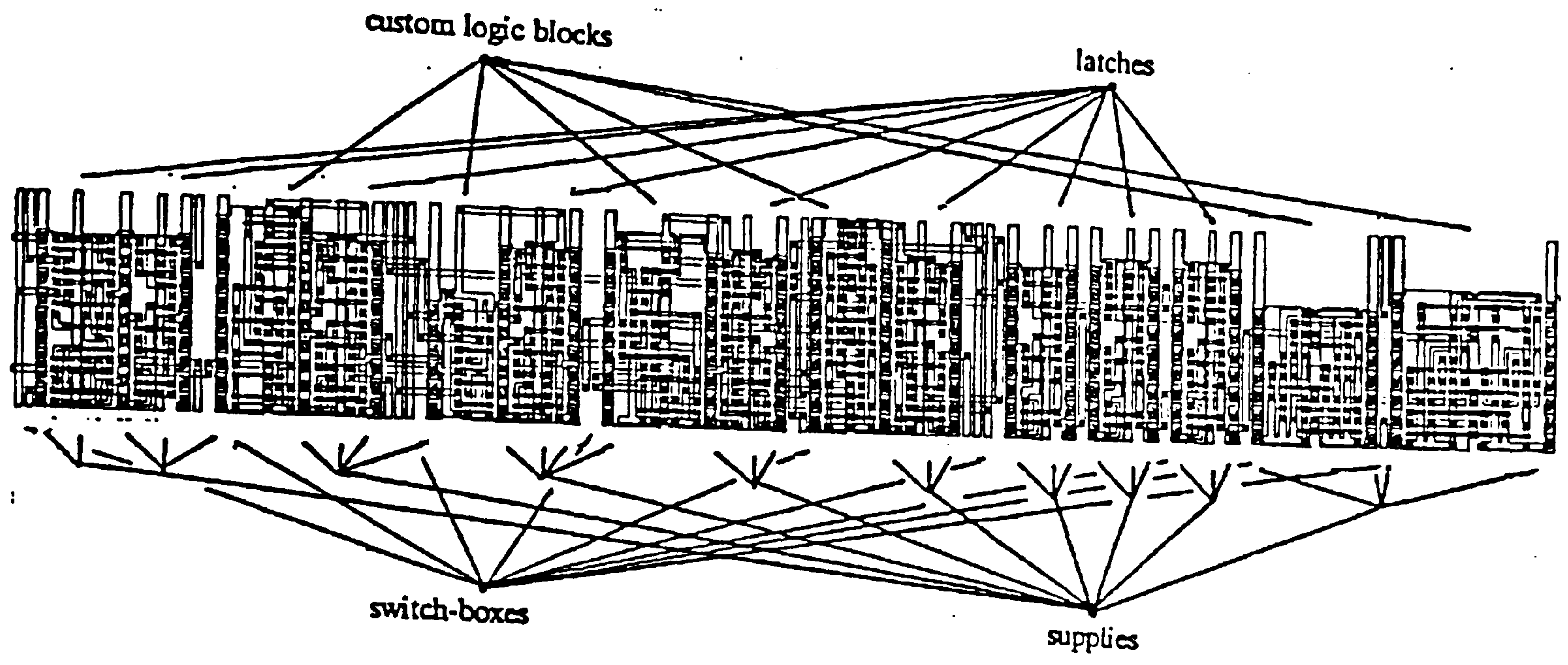
– Global signals, such as the clock which require routing to BLOCKs but otherwise pass through the routing channel.

– Cascade signals, which form the abutting connections between SLICEs.

– BLOCK to BLOCK signals.

The routing of these signals is discussed later in this chapter.

Figure 6.9 shows a serial/parallel multiplier SLICE, which has been fleshed out in 3 μm, MCE (Micro Circuit Engineering) design rules. The slice is a mock up, as although all BLOCKs were synthesised and latches imported from symbolic layout, the placement and routing was performed by hand. It shows the form of a SLICE which we intend.



SLICE mock-up

Figure 6.9

6.9. Incorporating the Mixed-Tree Single Phase Scheme

In chapter 3, we discussed the mixed-tree approach as a technique for reducing the number of transistors in a logic block and hence the area of the block. Both n-type and p-type transistors are used within the same logic evaluation tree, allowing us to perform inversion of an input signal at no extra transistor cost. The penalty of the scheme is the reduction of logic level at the output, with a slowing of the logic evaluation. To overcome this problem sense circuitry, described in chapter 3, is affixed to the output of the mixed tree gate to restore logic levels and increase logic transition.

The single phase clocking scheme described in the chapter is also a useful technique that we would wish to incorporate into the BLOCK generation software, this requires the inclusion of the two single phase latches, *mu* and *pi* forms. The latches include ratioed p-type and n-type transistors in their reduced component count dynamic versions. It is important that they are represented in a symbolic form otherwise, otherwise tailoring these latches to specific design rules through manual layout would lose the process independence of the software.

The layout of the sense circuitry and latches were combined to form a new pair of latches containing the level restoration needed for the reduced transistor, mixed tree approach. Ratioing of transistors in the layout is achieved by parallelling two transistors, as a ratio of two to one is satisfactory for this operation. This translates into a small piece of symbolic layout which we store in a library of components and is appended to generated BLOCKs.

6.10. Routing of BLOCKS

Whilst generating the SLICE, an important operation which must be performed is the routing of signals between the BLOCKs. There are three types of routing signal which need to be taken into account. These are

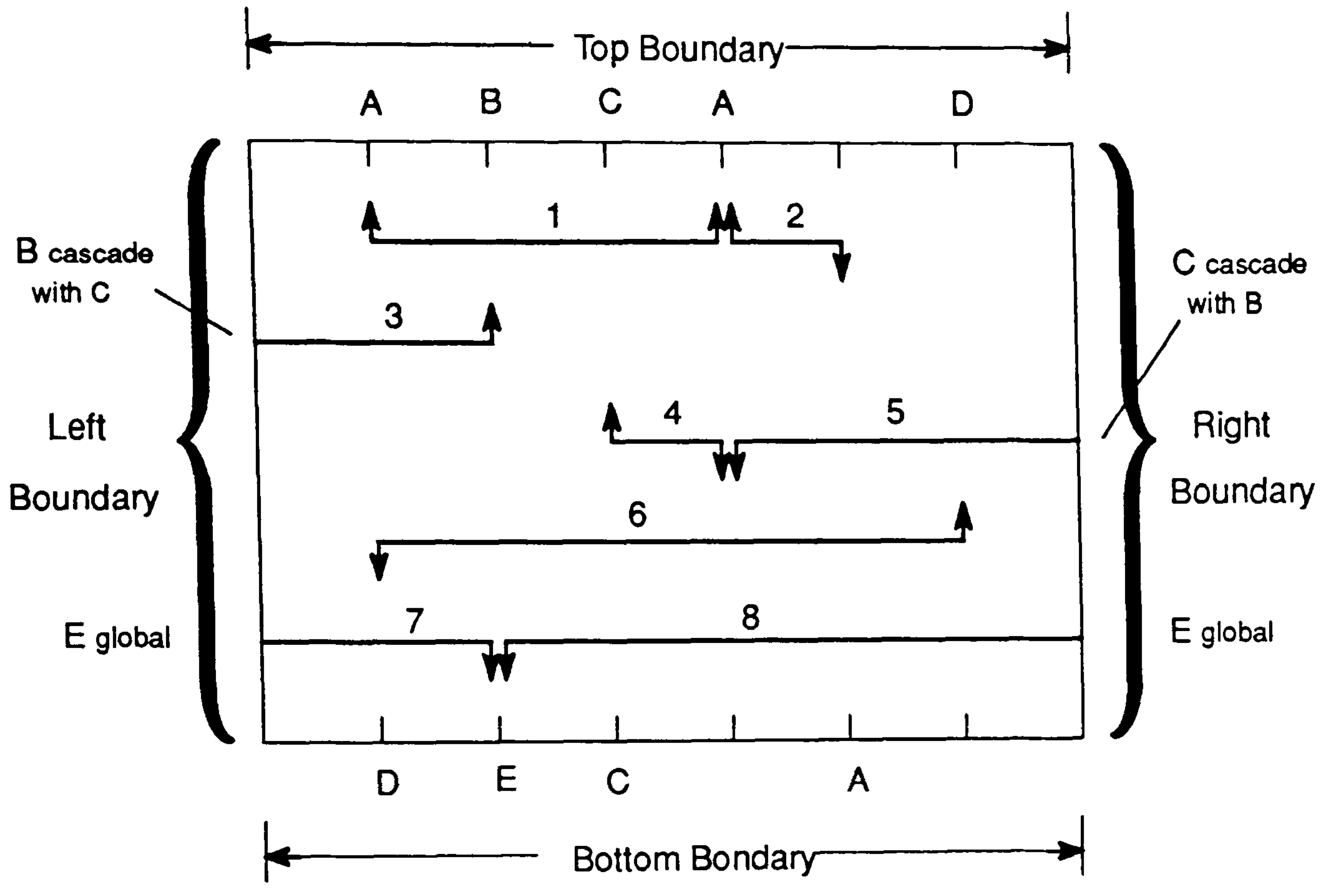
– Global signals, such as the clock, which pass through the inter BLOCK routing channel.

– Cascade signals which appear in pairs, except in the first and last elements of a SLICE, where the output of one SLICE abuts to the input of the adjoining SLICE.

– BLOCK to BLOCK signals within the SLICE.

The objective of a channel router [27], [29], [30], [31] is to complete all the necessary wire connections between specified rows of terminals, minimising the number of horizontal tracks or vias between routing layers or both. In the SLICE architecture, the ends of the channel need to be arranged so that cascade and global signals align when abutting SLICEs, for communication between neighbouring slices. As this is not the conventional channel routing problem, a modified routing algorithm [92] was developed for this case. This section describes that algorithm.

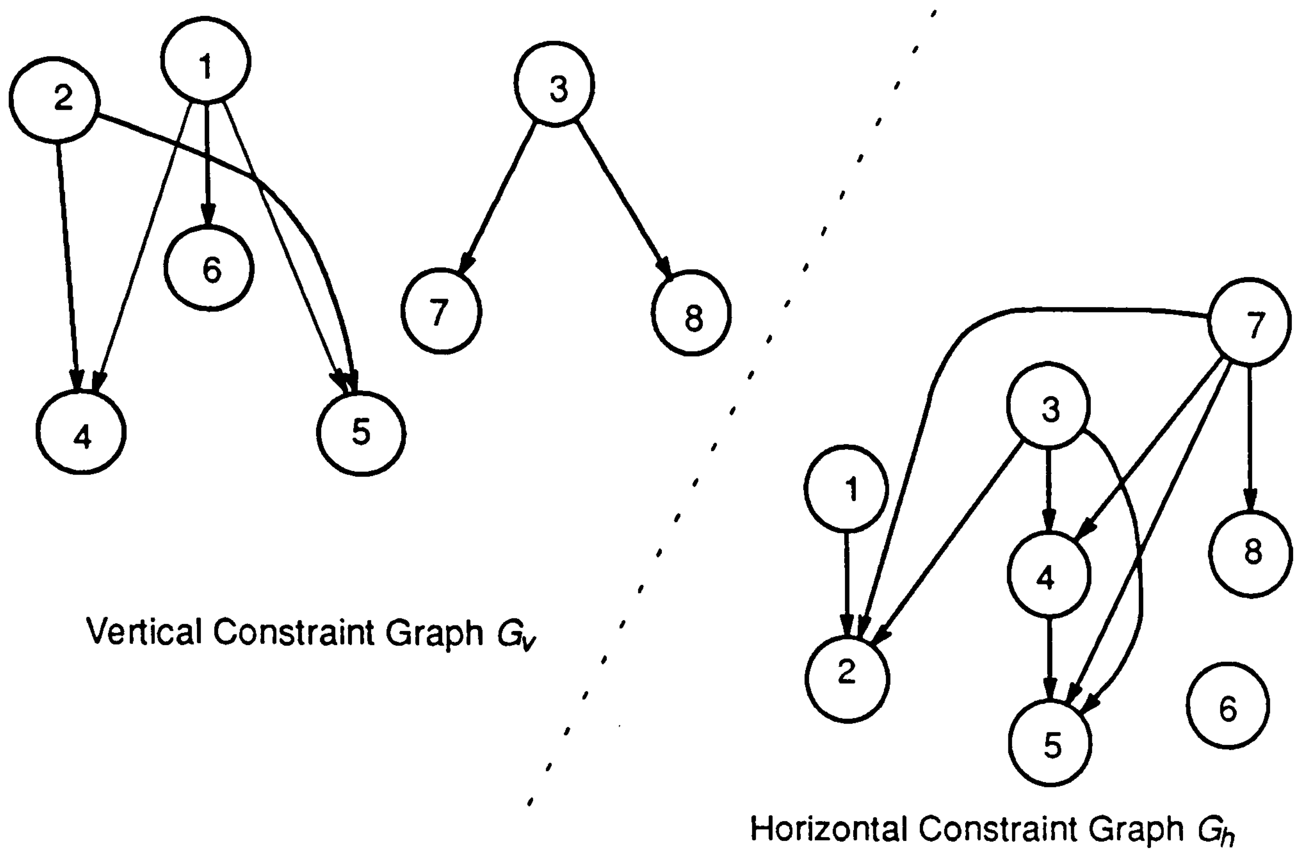
The algorithm is an enhancement of the weighted path algorithm for channel routing[93] to suit our case where global and cascade signals are included. In the normal channel routing problem, signals on the top boundary of a channel need to be routed to signals on the lower boundary. For our case we also need to be able to route the global and cascade signals, Figure 6.10. The wire carrying a signal will be termed net, using graph theory terminology.



Routing Channel between BLOCKs

Figure 6.10

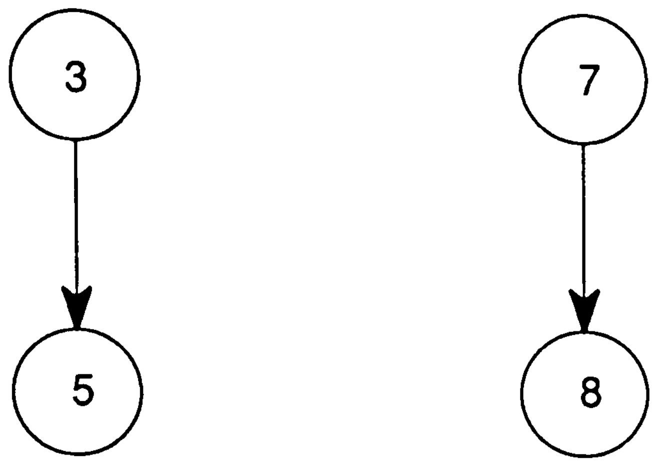
As with the weighted path algorithm all nets will be assumed to be two terminal nets as any multiterminal net can be decomposed into two terminal nets. When the nets are assigned to tracks, neither horizontal nor vertical segments of the nets must overlap. This information can be represented as directed graphs termed the horizontal and vertical constraint graphs, where the nodes of the graph correspond to nets which need to be routed. In the horizontal constraint graph (G_h), if the rightmost terminal of net 'a' is to the left of the leftmost terminal of net 'b', then a directed path from the node representing 'a' is made to the node representing 'b'. In the vertical constraint graph (G_v), if net 'a' must be placed above net 'b', then a directed path from node 'a' to node 'b' exists on the graph. Vertical and horizontal constraint graphs for our example routing problem are shown in Figure 6.11.


Vertical Constraint Graph G_v
Horizontal Constraint Graph G_h
1
2
3
4
5
6
7
8

Vertical & Horizontal Constraint Graphs

Figure 6.11

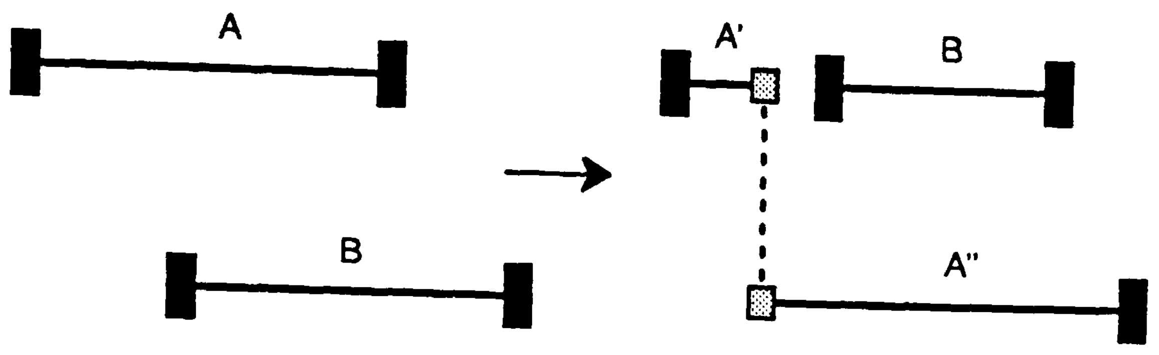
In addition, we must include the global signals which run straight through the channel, but also connect to either the top or lower boundaries of the channel and the cascade signals where a signal entering on one side must connect to a cascade signal on the other side of the channel. These conditions we call 'track constraints' and are represented by a directed graph, Figure 6.12.



Track constraints

Figure 6.12

The vertical constraint graph may contain a cyclical constraint, in which case an exhaustive algorithm for determining dog–leg points is used to remove these cycles. This is described later. When two two–terminal nets of cascade pairs overlap, these nets cannot be routed in the same track. In such a case one of the two must be split into nets as shown in Figure 6.13, so as to place the cascade nets in the same track for connection by abutment. The splitting point needs to be selected carefully to avoid generating a cycle in the vertical constraint graph.



Splitting of overlapping Cascade Nets

Figure 6.13

After these preliminaries, the algorithm then follows the steps:

1. A set S consisting of all source nodes in G_v is selected as the set of candidate nets to be routed.

2. A horizontal constraint graph, G_{hs}, is produced for nodes contained only in the set S, edges in this graph can be obtained from the full horizontal constraint graph G_h.

3. A weight W_i is associated with each node i in the set S, according to the formula,

$$W_i = 1000 \times \left\lceil \frac{V_{di}}{M} \right\rceil + 20 \times N \times H_{ci}$$

where V_{di}, the vertical depth of node i, is the node count associated with the longest path originated from that node, M is the maximum vertical depth for the nodes in the set S, N is the number of nodes in the set S, H_{ci}, the horizontal overlapping count, is the number of overlapping horizontal segments associated with the corresponding net. The value of $\left\lceil \frac{V_{di}}{M} \right\rceil$ is either 0 or 1 as the brackets denote the round to integer operation. This heuristic weight describes the importance of each node.

4. Weights of paths starting on source nodes in the track constraint graph and finishing on sink nodes are evaluated, unless no more nodes are left remaining on this graph in which case the subhorizontal constraint graph G_h is used. The path with the largest weight is chosen for routing.

5. The nodes on the selected path are removed from the vertical constraint graph G_v and from the track constraint graph.

6. The procedure is repeated until the reduced vertical constraint graph is empty at which point the channel has been routed.

The algorithm cannot cope with the problem of a cyclic constraint in the vertical constraint graph. In this case a 'dog–leg' must be introduced into a path in order that the channel can be routed. A dog–leg splits one two–terminal net into two two–terminal nets.

The method used to determine the point at which to split a net, to introduce a dog–leg when it is required, is exhaustive. This is justifiable in our case as the size of the channel to be routed is quite small. The algorithm for breaking cycles in the vertical constraint graph is shown in Figure 6.14 .

```
Count the cycles
Do (until all cycles have been removed)
Begin
  Get a cycle
  Do (until there are fewer cycles or all nodes
                       in the cycle have been selected)
  Begin
    Get a node in the cycle
    Create a candidates-list for the node
    Do (until there are fewer cycles or all columns
              in the candidates-list have been selected)
```

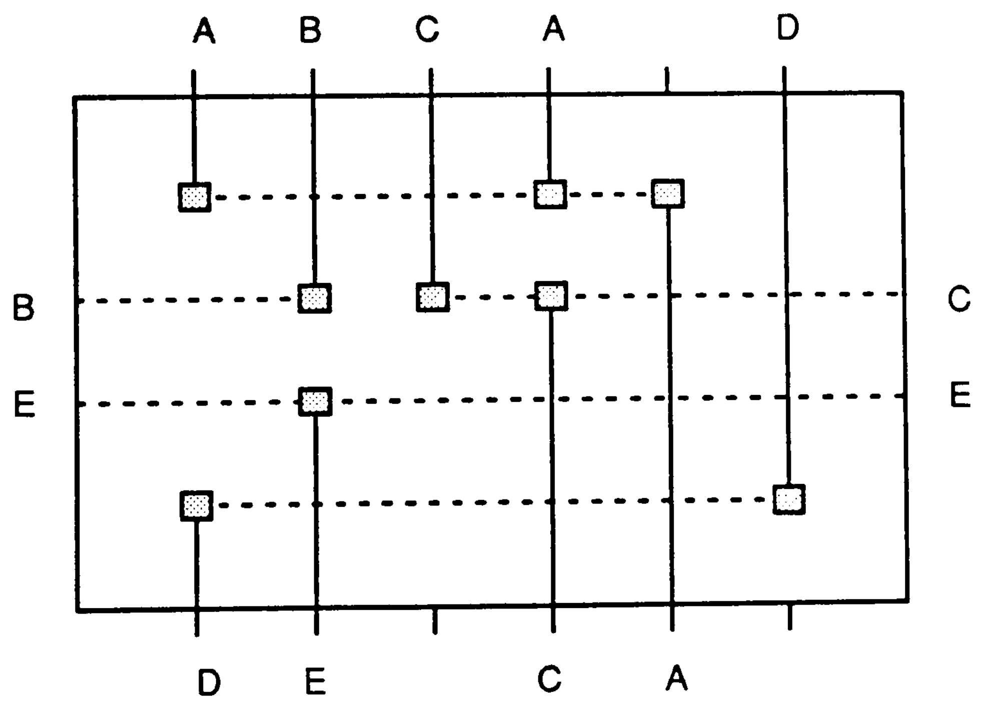
```
      Begin
        Select a column in the candidates-list
                                            as dog-leg point
        Split the net into two using this dog-leg point
        Generate a new vertical constraint graph
        Count the number of cycles
      End
    End
  End
```

Exhaustive Algorithm for breaking cyclic constraints

Figure 6.14

Candidate columns are chosen from the columns not containing terminals of the net. They are chosen in the order of increasing distance from the terminal columns to choose a short detour route caused by the dog–leg.

After dog–legs have been introduced, the above algorithm can be applied to route the channel. Figure 6.15 shows the result applied to our example channel.



Routed Channel

Figure 6.15

6.11. Transistor Net Synthesis

As an alternative to MOSYN's transistor net synthesis algorithm, another algorithm[94] was tried. The incentive to try something else was provided by some observations on MOSYN :–

– to find an optimal net, the algorithm of MOSYN needs to generate transistor logic trees for all distinct orderings of input terminals.

– the algorithm was not designed to generate efficient mixed–tree networks where well–crossings are a major factor to logic block area.

Although inspired by the above considerations, the algorithm proved to be not as flexible as that used in MOSYN. The algorithm is described here as it was intended to be adopted in the SECOND software to produce mixed trees. The description is essentially that in [94].

The algorithm starts with the on–set of the logic function that we wish to generate a transistor net for and will be illustrated with the use of an example. The example is a sum function block with a gated input:

xor(A,or(not(B),C),D)

The on–set for the function is determined, Figure 6.16(a), where each column corresponds to a member of the on set and each row to a circuit input. We try to reduce the number of columns to a minimum by combining columns, introducing the don't care 'X–state' where only one variable differs between the two columns. The process corresponds to finding the minterm expansion of a logical expression, as in the Quine–McCluskey algorithm[67]. This produces a canonic reduced form, shown in Figure 6.16(b), for our example. Experience with MOSYN has shown that 'bunching' placing equivalent inputs next to each other produces better results, than without the bunching, but if we require a specific input order it may not be possible to bunch equivalent inputs. Let us say, in our example, that we want to retain the order A–B–C–D, thereby preventing the bunching of the equivalent inputs A–D.

A	0	0	0	0	1	1	1	1
B	0	0	1	1	0	0	1	1
C	0	1	0	1	0	1	0	1
D	0	0	1	0	1	1	0	1

ON – set

$\longrightarrow$

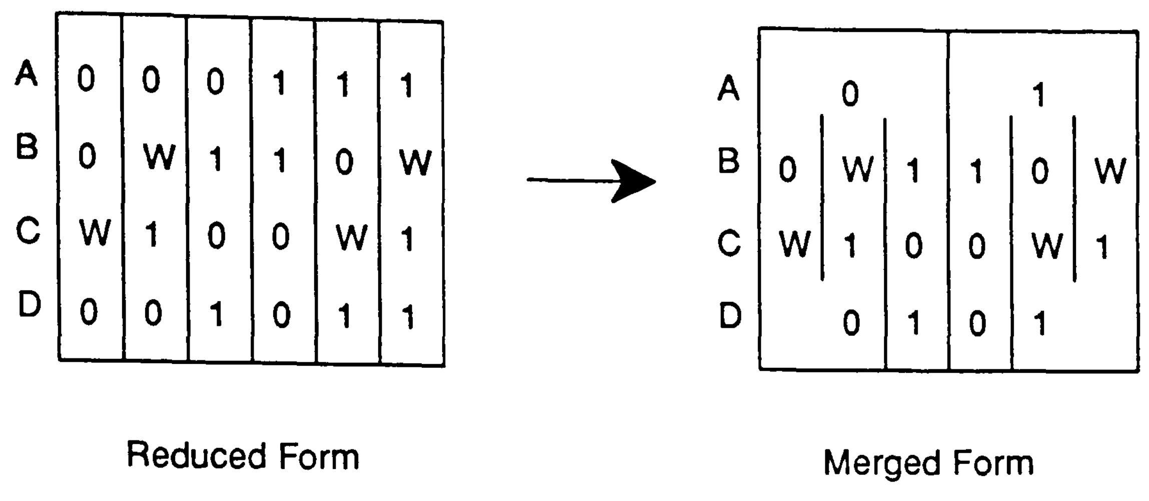
A	0	0	0	1	1	1
B	0	W	1	1	0	W
C	W	1	0	0	W	1
D	0	0	1	0	1	1

Reduced Form

On–set and reduced form

Figure 6.16

Another optimisation can be achieved by lowering of some partitions to allow connection between columns above the partition. The resulting table is shown in Figure 6.17(a). We can continue the process by noting that symbols which look ahead (i.e. downwards to the same pattern) may merge. Merging is equivalent to raising partitions, allowing horizontal connection between columns below the partition. Unfortunately, indiscriminately performing this operation can introduce unwanted conduction paths (sneak–paths), so that a check must be made that the merge does not introduce such a path. All such paths are disallowed, although some of these paths may not affect the logic. The final form of table is shown in Figure 6.17(b) and has 14 FETs.



A	0	0	0	1	1	1
B	0	W	1	1	0	W
C	W	1	0	0	W	1
D	0	0	1	0	1	1

Reduced Form

A		0			1	
B	0	W	1	1	0	W
C	W	1	0	0	W	1
D		0	1	0	1	

Merged Form

Merging Columns

Figure 6.17

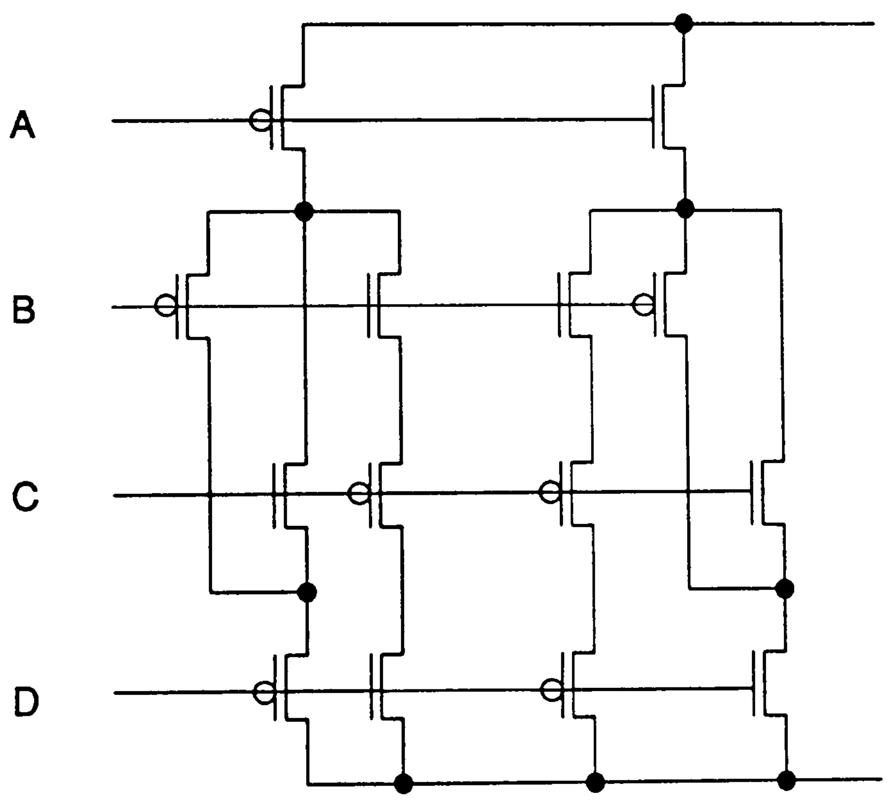
Each symbol has physical significance. In static CMOS, pull–up networks of pFETs are specified by the on–set, and pull–down networks by the off–set. Dynamic mixed–trees (described in chapter 3) allow pFETs and nFETs to coexist in the same discharge network, requiring no complemented signals. The replacements to form the final transistor network is shown in Figure 6.18, and the final transistor network is shown in Figure 6.19.

The method does not always produce optimal networks, MOSYN can perform better than the above algorithm, but the above method is simpler and thus when implemented as code executes faster.

Symbol	MIxed Tree	Static pull-up	Static pull-down
0	pFET	pFET on true	nFET on complement
1	nFET	pFET on complement	nFET on true
W	wire	wire	wire

Physical Replacements for symbols

Figure 6.18



Final Transistor Network

Figure 6.19

Conclusion

An extension to the FIRST silicon compiler SECOND has been described. It allows the use of process rule independent symbolic layout to be used to form final layout. The symbolic layout can be synthesized automatically using MOSYN and MOSLAY and then further software is used to connect these generated BLOCKS together giving FIRST an enhanced flexibility, allowing a user the capability to specify new elements to be incorporated into the existing library.

Primitives are built up from these BLOCKs using a hierarchical assembly procedure, where the hierarchy has been extended from the hierarchy in FIRST to form a full span structural compiler. The algorithms used in this assembly procedure have been described.

Chapter 7

A Comparison of SECOND with Commercial Software

7.1. Introduction

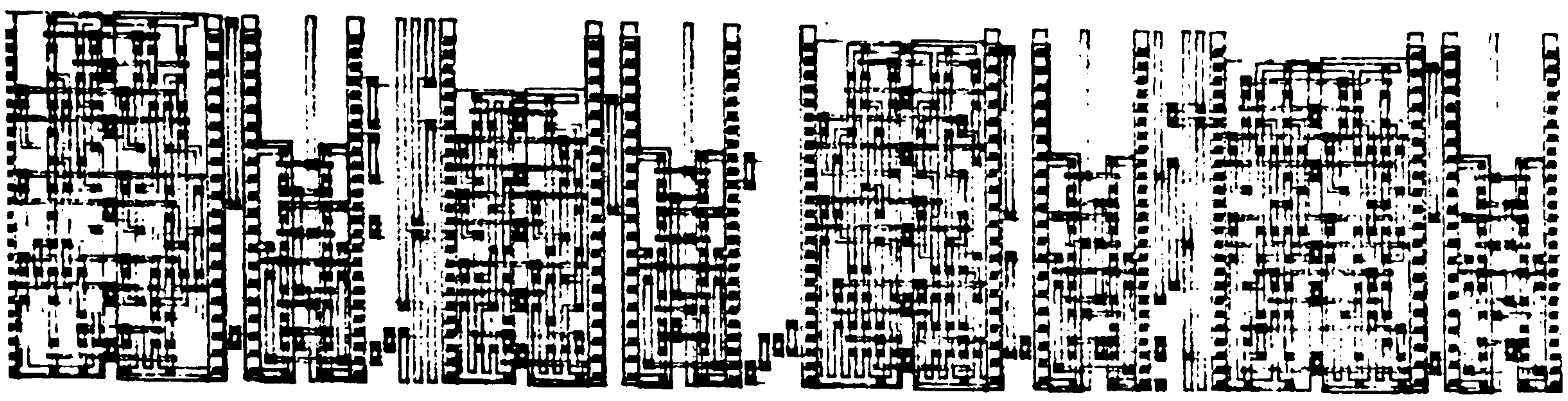
In order to evaluate the quality of the layout generated by the SECOND software, a comparison of the area between layout generated with SECOND and layout generated by a piece of contemporary commercial software was undertaken. The commercial software used for this task was the SOLO 1000 product produced by ES2. A multiplier of equivalent functional specification was synthesized using SECOND and SOLO 1000. The layouts were compared for area usage. Figures for the area usage are contained in [97] and suggestions for possible improvements to the SECOND layout style are contained in [98].

7.2. Area Comparison with Commercial Software

SECOND Layout Style

The layout method in SECOND has been to maintain the hierarchy as specified by the designer and build up a bit–slice structure (a SLICE) which joins with neighbouring elements by abutment eliminating the need for routing channels at the level of the SLICE. The SLICE is built from dense gate–matrix blocks keeping transistor numbers down using dynamic logic and/ or the mixed–tree scheme.

The SLICE consists of static logic BLOCKs (each a single complex gate, combinatorial logic is not cascaded) which are connected to LATCHEs and nearby BLOCKs through switch–boxes which also provide the abutment ports as described in the previous chapter. Figure 7.1 shows a multiplier SLICE produced using SECOND.



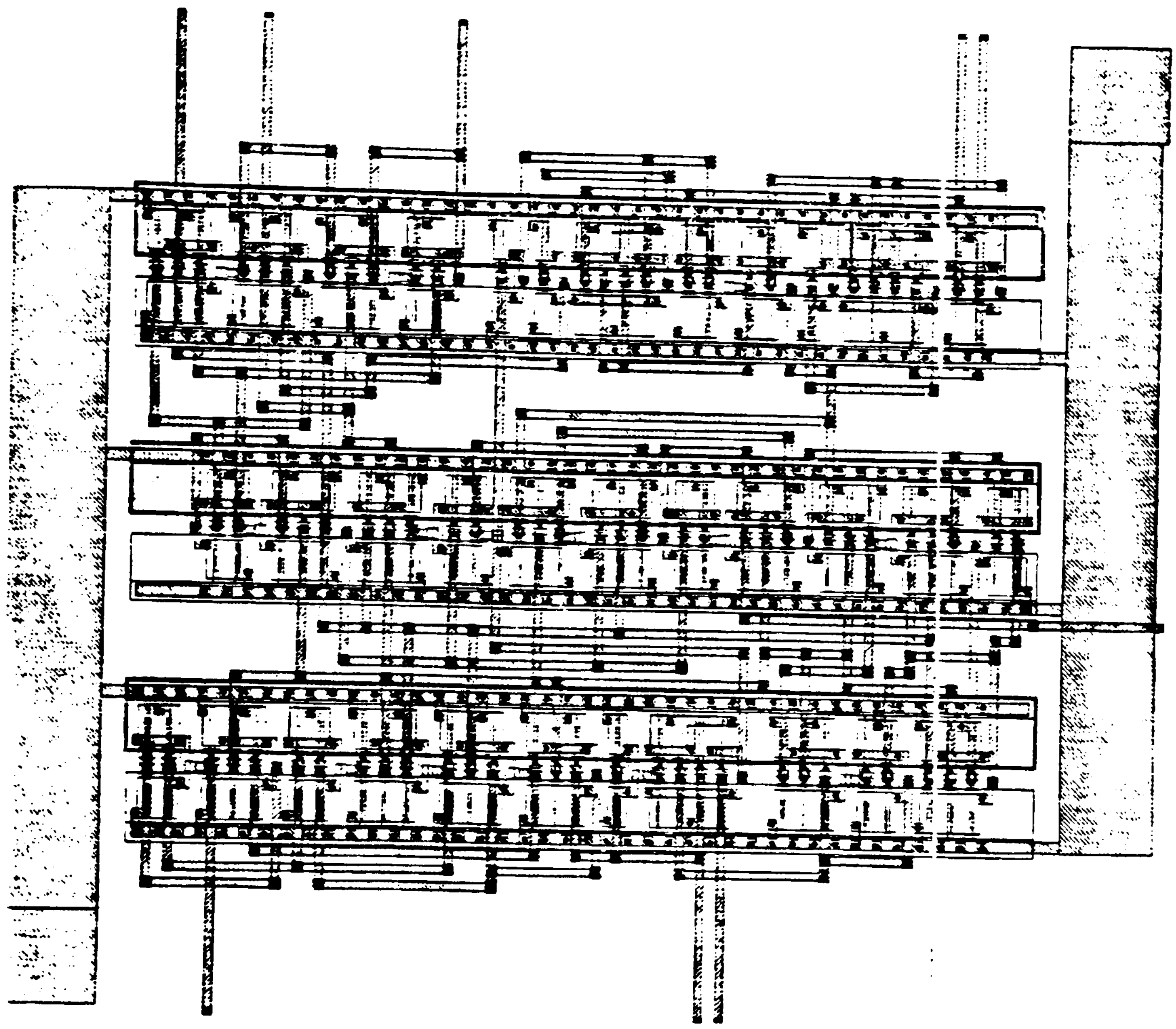
Section of a SECOND Multiplier Slice

Figure 7.1

SOLO Layout Style - Optimised Arrays

Figure 7.2 shows a typical design generated by SOLO 1000. The logic is made up of P-N FET pairs (called stages), wired up to produce NOR or NAND gates. These are cascaded to produce the logic of the desired function. The gates are placed in strips, called functional elements. The interconnection between stages to create the NAND/NOR gates is done within the functional element, the FETs being made large (17:2) to both accommodate this wiring and to increase the speed and drive capability of the gates. Interconnection between the gates and adjacent functional elements takes place in a wiring channel of variable width, which is placed between the functional elements.

The method used in SOLO is to flatten all hierarchy, thus the logic within the functional elements is not in direct relationship to the MODEL description, although the placement of cells is dependent on the system description in MODEL to reduce signal path lengths. A linear placement algorithm is used, followed by a route of cells. This linear placement is broken into sections and bent into a snake-like path for two dimensional placement.



Adder produced by SOLO 1000

Figure 7.2

Area Comparison of Layout Styles

Plots of the SECOND and SOLO 1000 multipliers are given in Figure 7.3 and in Figure 7.4. They use ES2 2 μm CMOS design rules, microcells for this technology are used for the plot generated by SECOND. The area, transistor count and transistor density of the layouts is given in Table 7.1. The SECOND design uses minimum sized transistors. In practice transistor microcells of double width would be used. The figures given in brackets are the area and transistor density using double width microcells in active regions.

	SOLO 1000	*SECOND*
Area	1.23 mm^2	1.08 mm^2 (1.6 mm^2)
Transistor Count	1714	1320
Transistor Count/Area	1360 Trs/mm^2	1222 Trs/mm^2 (625 Trs/mm^2)

Table 7.1

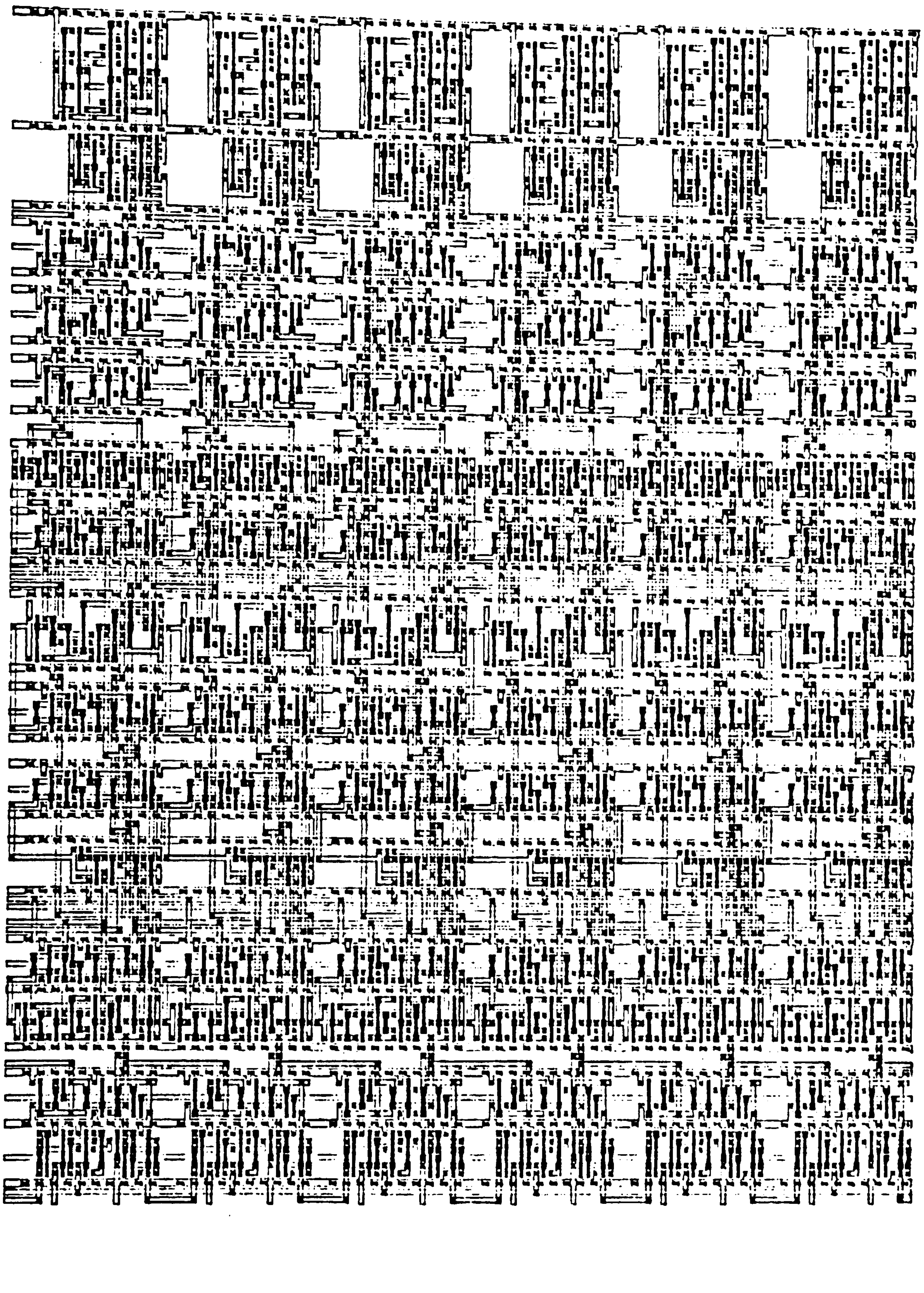
Although the SECOND design is smaller, the SOLO 1000 solution gives a better transistor density. This is a discouraging result considering that SECOND generates dense gate matrix blocks using dynamic logic and the transistor efficient mixed–tree scheme using a bit–slice architecture, whereas SOLO uses fully static logic and a traditional gate array layout, without bit–slice partitioning. SECOND uses fewer transistors and of minimum size, the FETs in the SOLO design are over 4 times bigger. If SECOND was to use static logic, this would increase the area by approximately 35 percent, giving an area (with large transistor microcells) of 2.16 mm^2 and a transistor density of 611.

7.3. Area efficiency in SECOND

SECOND uses silicon area extremely inefficiently when compared to SOLO. The layout generated by SECOND is more regular but it is not more dense. The rigidity of the regular two dimensional mesh of the gate matrix approach, gives a regular appearance, but random and partially regular logic do not map well onto a rigidly regular layout style. The style works well for well structured designs e.g. RAM cells, but for irregular logic, will give rise to many empty holes in the layout, to maintain the regularity.

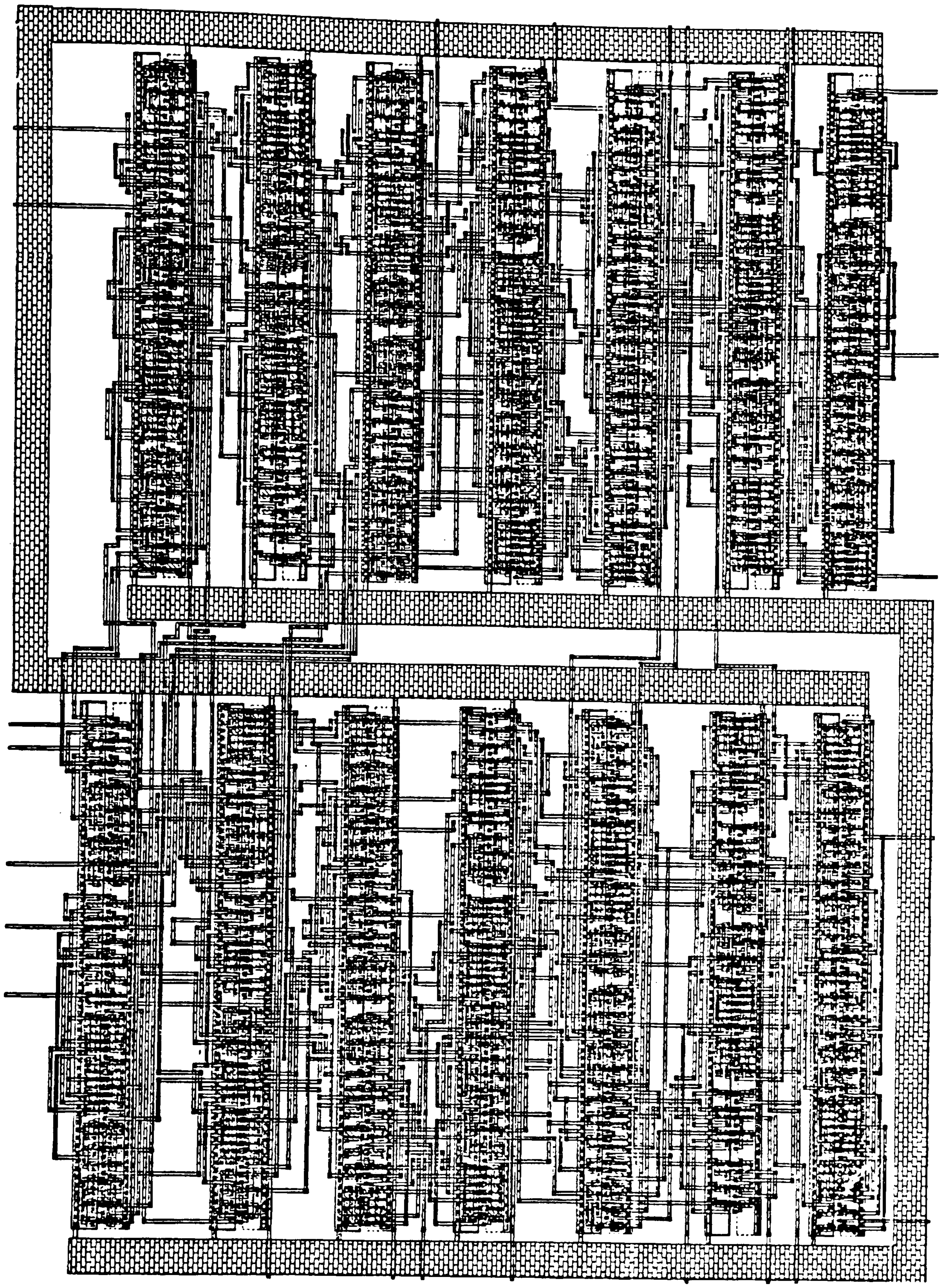
Some further reasons for inefficient use of silicon area are :–

1. Each time a node crosses a gate signal in a SECOND BLOCK, it either forms a FET in diffusion, or it crosses it in metal–1. This involves changing between metal–1 and diffusion layers by placing a contact, thus drain and source areas are not normally shared, and many contacts are needed.



Multiplier produced by SECOND

Figure 7.3



Multiplier produced by SOLO 1000

Figure 7.4

2. In the mixed–tree scheme, transistors are saved as the number of inverse signals can be cut down, but when the logic is laid out, the well separation between P–type and N–type transistors causes extra space in the layout to be needed. Transistors which should form a dense block are forced apart by this.

Area Issues at Hierarchical Levels higher than the SLICE

In SECOND a chunky PRIMITIVE functional block is generated when the SLICEs are abutted. Different PRIMITIVEs (e.g. a multiplier and an adder PRIMITIVE) will have different sizes and a problem arises on how these functional blocks can be connected together. These blocks can be pitch matched to simplify routing, when the PRIMITIVEs have the same number of SLICEs (as the word–size is likely to be constant in a chip), but it is not clear, how these PRIMITIVEs can be placed efficiently on a chip without running into routing and aspect ratio problems. In SOLO, as the hierarchy is removed, the layout spreads evenly allowing an aspect ratio for interconnected PRIMITIVEs to be chosen to fit on a chip, and no area penalty for the routing of the interconnected PRIMITIVEs.

Performance Issues in SECOND and SOLO

No hard simulations to compare the performances of the multipliers produced by SOLO and SECOND were performed, but some performance comparisons can be inferred.

If SOLO and SECOND designs had the same size FETs, then the SECOND design would be expected to operate faster, for two reasons :–

1. Hierarchy of a design is not lost in SECOND, whereas SOLO ignores this, leading to a loss of locality of signals leading to longer connecting wires and hence greater capacitance.

2. As the logic gates in SECOND are not cascaded, being a single complex gate, instead of multi–level logic composed of small gates, SECOND should be faster because of the shorter logic paths.

With a heavily synchronous system, such as a bit–serial processor, clock skew is likely to be a problem. In SECOND the clock lines can be strictly controlled and balanced whereas SOLO randomly routes clock lines without any preference for which branch is more heavily loaded. Heavy simultaneous glitches can be present on the power lines, through the synchronous clocking of large sections of logic. In SECOND, power lines can be sized, whereas SOLO takes no account of such peak activity, although this is not important with most circuits, it may arise with some bit–serial designs.

Possible Improvements to SECOND

More sophisticated software could be used to reduce area usage, some suggestions are :–

– Large microcells are needed to eliminate design rule errors, smaller microcells could be tried, and software which eliminates local design rule errors used.

– Carefully order terminals in the BLOCKs, as this can lead to simple routing channels within the SLICE.

– Use software to minimises the width of the widest BLOCK as the size of this element determines the width of the overall SLICE.

– Use microcells which are smaller in the x–direction for the switch–boxes and N–P well separation.

– Try to match up input orderings to remove dog–legs in the switch–boxes.

7.4. Cell Layout Generation vs. Use of Vendor's Cell Library

As the comparison between layouts generated by SECOND and a gate–array layout shows that using SECOND does not lead to area advantages, an alternative is to use standard gate array software, but supply a vendor specific netlist. The problem of layout is now only the difficulty of generating the vendor specific netlist. Process independence can be achieved if we can generate a suitable netlist, the problem of the actual layout has become the responsibility of the specific vendor. This is the approach adopted in the next chapter.

The problem of clock–skew and noise on power lines is still a problem, but can be overcome with netlist analysis software and the insertion of suitable buffering for clock skew, and power line width adjustment for noise margin problems on power supply.

Conclusion

Through work on a process rule independent full span structural compiler, a multiplier was synthesized. A functionally similar multiplier was synthesized using SOLO and the results compared.

The SECOND software was shown not to improve significantly on the layout of commercial software such as SOLO. A new layout style combined with possibly more sophisticated software would have to be used if continuation on SECOND were to be worthwhile. When tested, the multiplier was found to be non–functional, this was due to errors in the imported latch blocks, which were designed in a hurry to meet processing deadlines. The layout comparisons are still valid as the areas and layout style are still typical.

Using commercial gate array to produce the cell library for FIRST, according to new process rules, is seen as another route to attaining process rule independence. This is seen as a simple and effective method of updating the cell library especially as the custom produced layout from SECOND does not produce competitive layout. This is the basis for the work in the next chapter.

Chapter 8

Gate Level Expansion

8.1. Introduction

A demonstration that components of the FIRST library could be generated automatically from a user description, in a process independent symbolic form was given by the SECOND project described in the previous chapter. The failure of the software to demonstrate conclusively smaller layout than commercially available software, for our multiplier example was a discouraging result, as using available software can achieve similar results.

An alternative to this automation effort which was seen as worthwhile was to used semi–custom processes. With a semi–custom design, the layout and maintenance of the underlying library is the responsibility of the gate–array or cell–library vendor. The gatearray or cell–library vendor usually offer state of the art technologies, where in this case we were targeting at a CMOS silicon on sapphire (SOS) process. To produce the netlist for the vendor requires a piece of software capable of translating the FIRST system description language into the format used by the vendor. In this way we hoped to achieve an alternative method of process–independence which is unobtainable with the FIRST system as it stands.

8.2. Custom vs Semi–custom

The goal of the SECOND project was to achieve process–rule independence by using gate–matrix symbolic layout as the output from the completed compiler. This would be later fleshed out using process design rules inherent in a microcell library as described in the previous

Part of the work covered in this chapter was done under contract for the European Space Agency (ESA).

chapters. The alternative route looked into in this chapter is the generation of a netlist for a semi–custom gate–array or cell library.

In the custom–like approach considered for SECOND, the software has control over placement and thus lengths of interconnections. It has no control over placement in the semi–custom approach. Hence, in the semi–custom approach the clock rate must be relaxed to cope with the delay caused in the transmission of signals.

Custom layout is fast and area efficient, but it requires skill and patience to achieve. In SECOND we hoped to be able to emulate this by producing efficient and optimal (in terms of small logic blocks) layout which would be competitive with custom layout with these features and yet have a high degree of process portability. Semicustom layout is not as fast nor as area–efficient but in terms of the design time and complexity of realisation it is preferable. Software exists which is able to place and route a gate–array or standard cell–library, either fully automatically or else by performing the majority of the task and leaving what remains to a manual expert. We can leave the problem of layout to vendor, and concentrate on the problem of generating universal netlists for the gate–array design.

8.3. Justification for the semi–custom approach

The task of producing a new cell library for the FIRST silicon compiler for each new process is prohibitive given the resources of the university. By producing netlists for gate–arrays/cell libraries, the responsibility for the updating of libraries is shifted to the vendor of the gate–array/cell–library. This allows us to achieve the goal of process portability with the use of the silicon compiler FIRST.

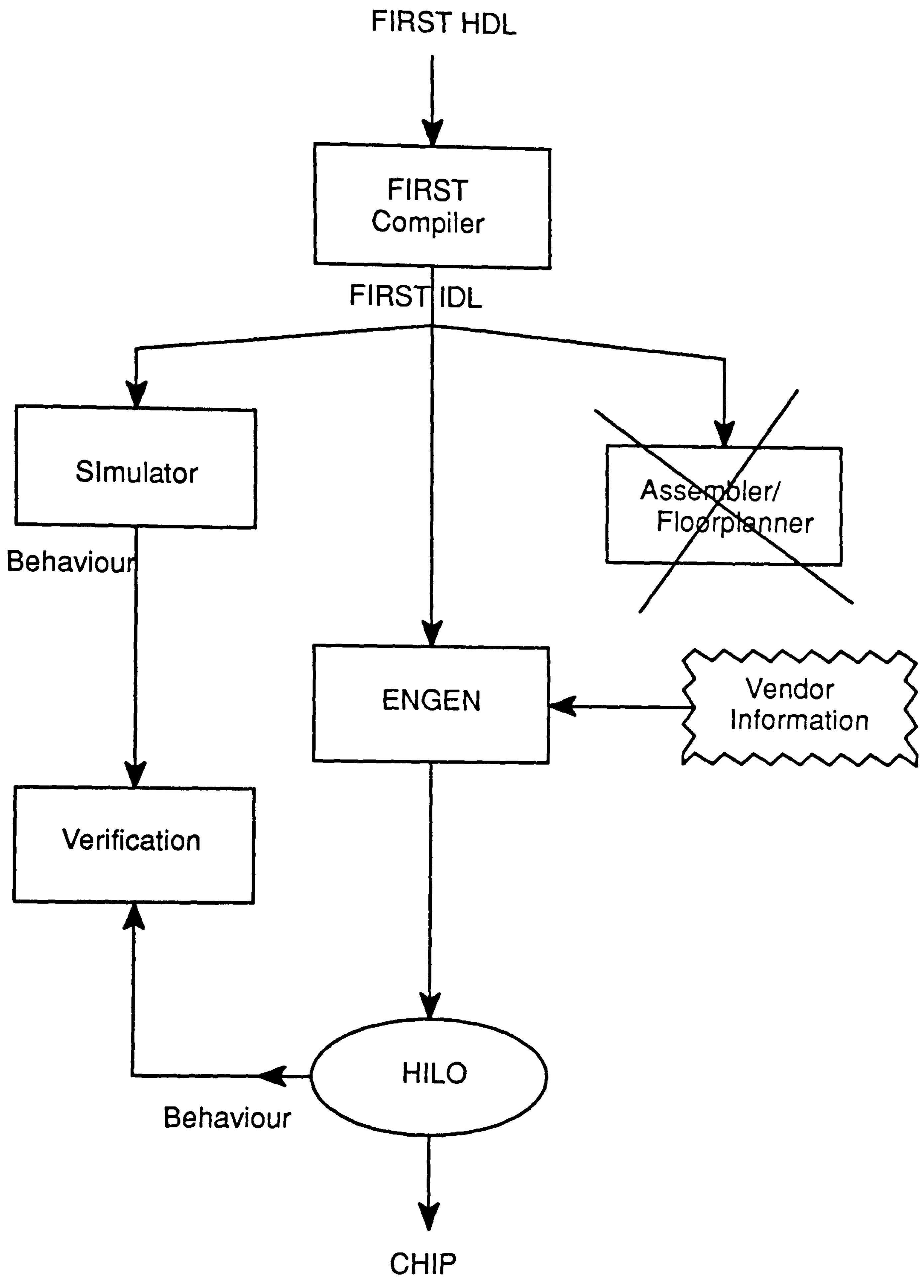
8.4. Extending FIRST for gate netlist generation

The netlist generation software was seen as being an extension to the software of FIRST and to be used in conjunction with it. It would thus be a piece of software to join those pieces comprising FIRST, and extend the software to produce vendor specific netlists. With this in mind, we wanted to use as input to the netlist generation software the intermediate description produced by FIRST. This would then be translated by the netlist expansion software to the required format. This has the advantage of being able to use the simulator of FIRST to carry out a system simulation and then being able to carry out a comparison of output generated from the gate–netlist simulator in order to verify the netlist.

The relationship between the netlist generation software ENGEN and FIRST is shown in Figure 7.1, vendor information must be supplied to the netlist software as we wish the software to be usable with different vendors.

The program started off as a general conversion program [95] of FIRST intermediate code to a netlist for an arbitrary gate–array language. We assume similarities in the languages, so that we can produce each form by only changing the names of netlist delimiters and their ordering. Output using the language NETLIST[96] was sought as this is the netlist description language compatible with the simulator RNL[82] used in the department so could readily be simulated and tested.

In the case where the work was done under ESA contract, portability between vendors was sought, the output netlist was produced in HILO. We looked at two cell–library suppliers, Marconi Electronic Devices Ltd., (MEDL) and the Swedish company ASEA–HAFO, both of which supply CMOS silicon on sapphire cell libraries.

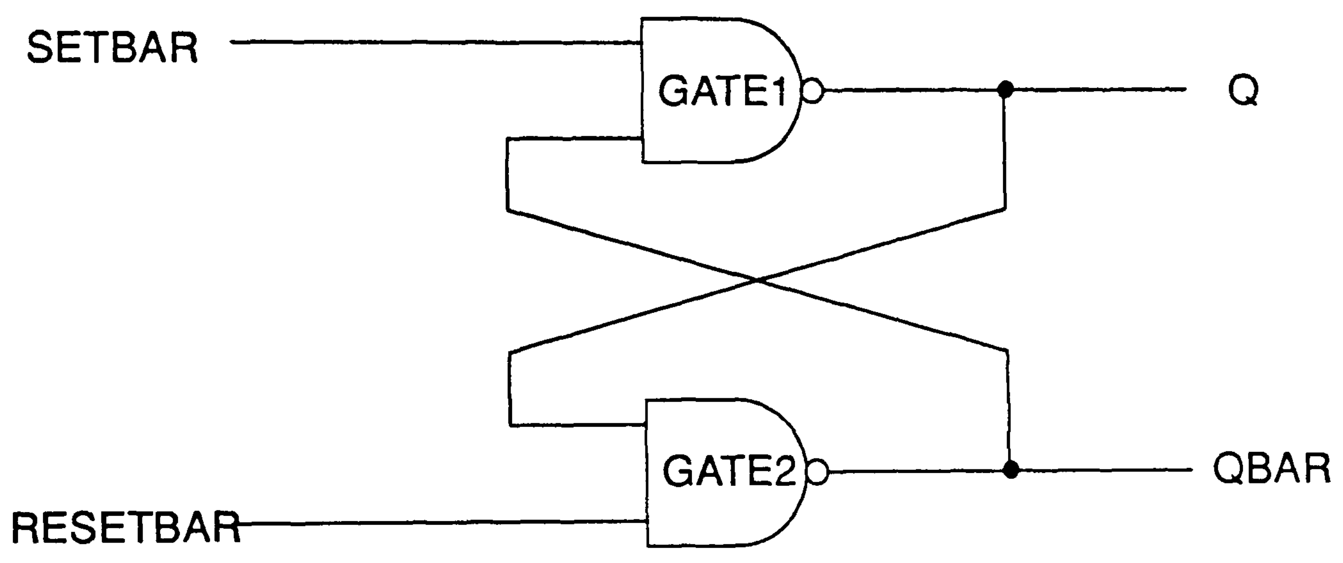


Operational Relationship between FIRST and ENGEN

Figure 8.1

8.5. Overview of HILO

HILO is GENRAD's commercially available functional simulator and fault simulator, although in the work covered in this chapter no use was made of the fault simulator. Designs can be specified hierarchically, and modelling of circuit delays can be done. Circuits can be specified wholly structurally, that is as an interconnection of physical units and was used solely in this capacity in the project. It allows the digital simulation of circuits which are to be integrated onto a chip.



Cross–coupled nand gates

Figure 8.2

In HILO both names of gates and signal nodes are given unique identifying names, but only the signal nodes need to be declared. The signal nodes are declared as WIRE (ordinary connections), UNID (unidirectional signal) and TRI (tri–state signal). For instance the cross–coupled nand gates of Figure 7.2 would have the HILO description of Figure 7.3 .

```
CCT LATCH (Q,SET,RESET)
NAND
      GATE1(Q,QBAR,RESET)
      GATE2(QBAR,Q,SET);
INPUT SET RESET;
UNID Q QBAR.
```

HILO Code for Set–Reset Latch

Figure 8.3

The generation of HILO code is dependent on vendor's names for cells and also the cells supplied. In the example above, the word *NAND* is used to name a two input nand gate, but this will change with different vendors, as will the availability of the cell. In the case of the ubiquitous nand gate, it will certainly be present, but more complex cells such as Full–Adders need not always be present. To get around this, we need some way for the user of the software to supply the names of cells and also indicate their availability in the vendor library.

The use of HILO is a good choice on the account of its wide usage and establishment. It has the undesirable characteristic of not being wholly vendor independent, but this can be overcome by the user of the software supplying vendor names for cells and also their availability.

8.6. Input Format

In order to generate a netlist, the intermediate file from FIRST must be read and then expansion routines called which assemble the desired netlist from smaller elements, the process repeating itself down the hierarchy until the vendors elements are used.

The structure of the FIRST intermediate file

The FIRST system description language consists of object declarations and subsequent instantiations at equal or higher levels of hierarchy. The hierarchy is typed, as PRIMITIVE, CHIP, SYSTEM which have a physical significance in terms of structure and assembly procedure. The FIRST compiler operates on such a system description to produce the intermediate file, which is a netlist for each chip.

The file contains a list of PRIMITIVE and PAD names with the parameters for the PRIMITIVES and the connectivity information. The PADs do not have any parameterisation. A line from the netlist has the following form :–

```
PRIMITIVE-NAME   list of parameters  connecting nodes
```

or in the case of PADS :–

```
PAD        node-name        connecting node-number
```

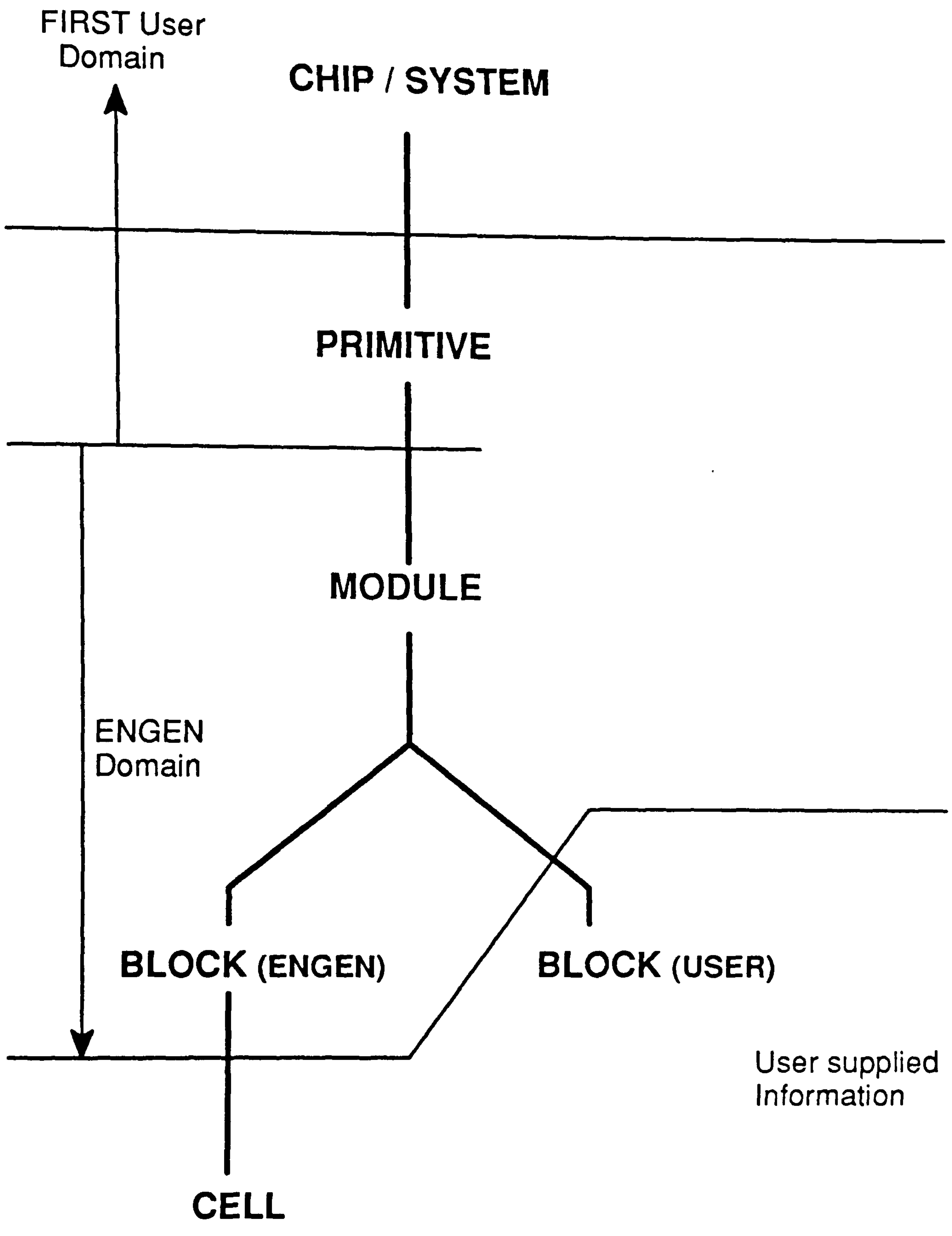
The number of parameters and connecting node–numbers is fixed for each primitive (with the exception of the CONTROLGENERATOR which has a different format in the file). These netlists are *correct by construction*, that is they result from language compilation and are known good syntactically (if not electrically or functionally). This simplifies the reading in of the intermediate file, allowing straightforward code routines to do this job.

8.7. ENGEN – Expanded Netlist GENerator

The software for converting the output from the FIRST compiler to HILO, was given the name ENGEN (from Expanded Netlist GENerator). It consists of assembly procedures for the PRIMITIVES written in the programming language C. It also has routines for reading in the file and outputting the file in the form of usable HILO.

Hierarchy in ENGEN

The experience gained in whilst working on SECOND indicated it would be a good idea to maintain the concept of the hierarchy in the netlist generation software. A different hierarchical scheme is used, which should not be confused with that in SECOND. The levels are similar in their functional complexity to keep some compatibility between the two software systems.



Levels of Hierarchy

Figure 8.4

The special purpose hierarchy used is very important to the operation of ENGEN, the levels are illustrated in Figure 7.3, PRIMITIVEs, MODULEs, BLOCKs, CELLs. The MODULE and CELL levels are new levels of hierarchy introduced for the netlist generation. CELLs are basic gates, of complexity lower than the previously introduced BLOCKs, consisting of basic gates, nand, nor and latches forming a universal set from which all systems can be constructed. MODULEs are the largest units which can be specified without including parameterisation, introduced to simplify the netlist generation. The PRIMITIVE is the only parameterisable level and a PRIMITIVE is constructed from a variable number of MODULEs which compose the PRIMTIVE. User parameters are used in determining the number of MODULEs used in PRIMITIVE construction.

The levels beneath the PRIMITIVE are not parameterisable and are fixed in form allowing a recursive hierarchical expansion. Underneath the level of MODULEs are BLOCKs. BLOCKs are kept in two forms, a portable and an optional vendor specific form. The portable form is built from CELLs, which are assumed to be universally available and provide the level of hierarchy where different vendors are liable to use different names for basic elements. The vendor specific form allows the user to optimise the generated netlist, by using including a vendor BLOCK, if it exists, which will be more efficient than constructing it out of the low level BLOCKs.

Portability between vendors

As HILO is not a completely portable language, it contains vendor specific information which must be incorporated into the gate–netlist produced by the software. This comes in the form of names of elements and the connection order of signals to that element. As this can vary from vendor to vendor, some means of specifying this to the software must be made available.

This is achieved by defining a library of elements which we wish to directly incorporate into netlist from the vendors library. This is done at two levels of the hierarchy, the CELL level and BLOCK level. All levels above these are constructed from these elements so that names and signal orderings are known. The list of CELLs, which are used to construct the portable BLOCKs are shown in Table 8.1.

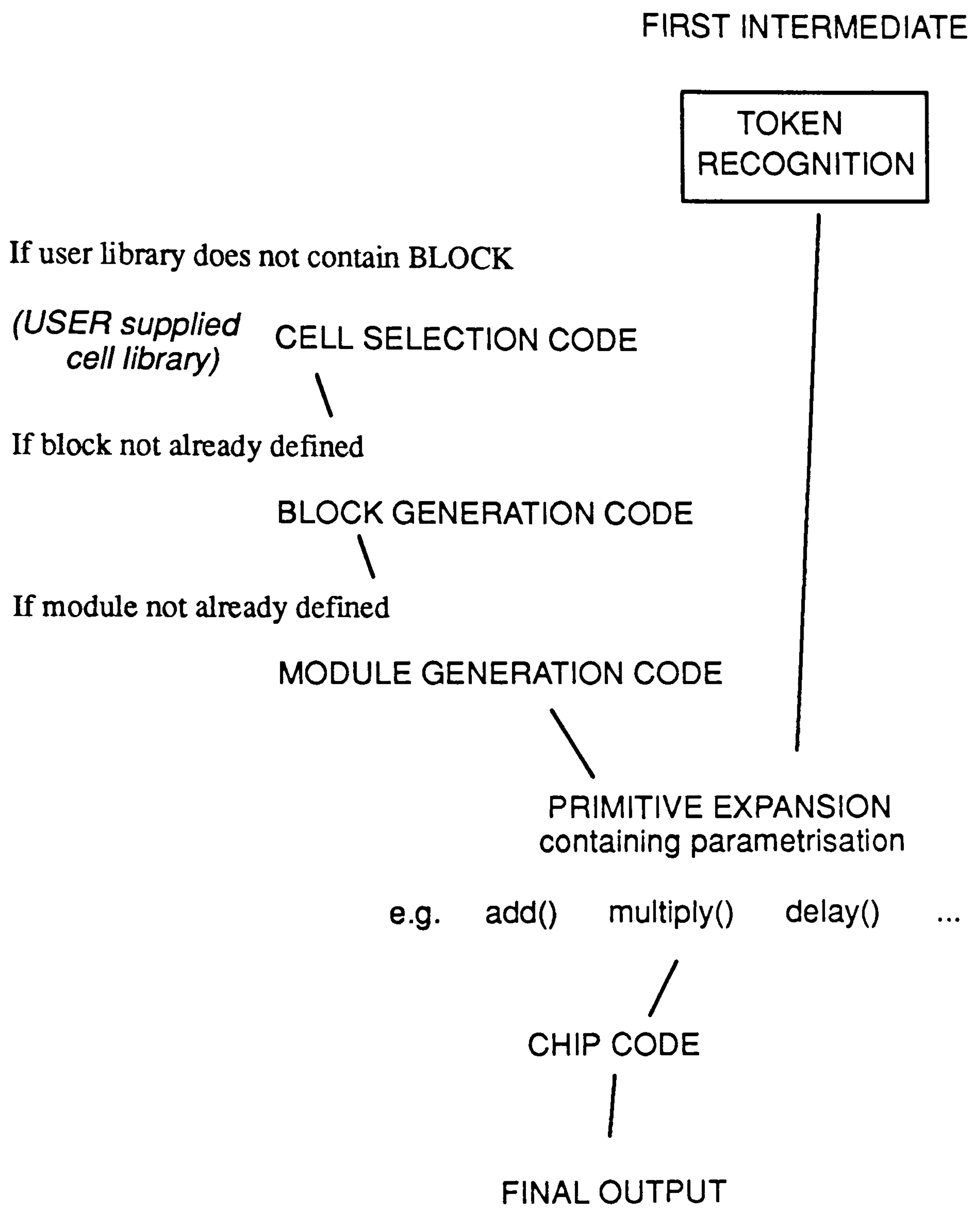
ENGEN CELL–library	
1	2–input nand gate
2	3–input nand gate
3	2–input nor gate
4	3–input nor gate
5	inverter
6	inverting buffer
7	positive edge–triggered D–type flip–flop
8	high–level D–latch (data stable when clock is high)
9	low–level D–latch (data stable when clock is low)
10	direct input pad
11	CMOS compatible output pad

Table 8.1

The vendor names of the CELLs and any BLOCKs (needing flags indicating their existence) are contained in a library file, which the software reads on starting up. Another file contains the definitions for CELLs and BLOCKs for the given vendor. Although these elements are contained in the vendors library, higher levels elements constructed from them assume a specific signal ordering which may be achieved easily by giving a trivial definition of the element with the signal ordering used.

Netlist Generation

The expansion of PRIMITIVEs down to the level of BLOCKs and latches requires that the structure of each PRIMITIVE is captured in the form of a high–level language routine which produces the HDL equivalent of a block diagram of the PRIMITIVE. The process is basically a hierarchical macro expansion of the intermediate file. PRIMITIVEs are divided into MODULEs, with the HILO code produced, MODULEs into BLOCKs and BLOCKs into CELLs. The operation of the program is illustrated in Figure 7.5, which shows how the routines work together. A list of the routines used in the code is given in Appendix 3.


FIRST INTERMEDIATE
TOKEN RECOGNITION
If user library does not contain BLOCK
(USER supplied cell library)
CELL SELECTION CODE
If block not already defined
BLOCK GENERATION CODE
If module not already defined
MODULE GENERATION CODE
PRIMITIVE EXPANSION containing parametrisation
e.g. add() multiply() delay() ...
CHIP CODE
FINAL OUTPUT

Netlist Generation

Figure 8.5

Following is a piece of pseudocode which generates the netlist of a PRIMITIVE. MODULE and BLOCK generation is accomplished in the same manner, but do not contain any parameters, so that the element need only be defined once, and a unique name for each set of different parameters need not be defined.

```
#include <stdio.h>

#include "../includes/name.h"

primcall(par,si,so,ci,co)

int par[[]], si[], so[], ci[], co[];

-- these arrays are as follows :-

--    par[[]] - parameter values

--    si, so, ci, co carry node numbers for

--             serial/control inputs & outputs

{

    char primname[20];


    if(!is_blockdefined(B_BLOCKNAME))

        define_blockname();

    -- etc. for all blocks used.

   if(!is_moduledefined(M_MODULENAME))

        define_modulename();

    -- etc. for all modules used.

    -- Note: if module calls block that block needs to

    --       be defined as above.

   sprintf(primname,"NAME%d%d",par[],par[]);

    -- set up a unique name for each set of parameters.

  if(!is_primdefined(primname)){

        -- primitive body

    }

    -- set up sigin with input signals

    -- set up sigout with output signals

    chip_primcall(primname, NoInputs,

            sigin, NoOutputs, sigout);

}
```

8.8. Physical special cases

Some physical engineering issues arose, separate from the software problems of netlist generation. Three physical objects required special treatment – these are latches, buffers and pads. Strategies used to achieve their incorporation are discussed below.

Latches

All the internal latches are positive edge–triggered D–type flip–flops. Transparent latches (half–latches) are used to pipeline data transfer through pads. For the efficient realisation of these elements, use of the vendor supplied components is advised, and thus it is required by the software.

Buffers

Some signal lines in the generated netlist can be heavily loaded. This is the case with the clock line which must supply all the latches within the circuit and the clocked pads. Other signal lines will not be substantially loaded unless this is especially intended (to minimise hardware) by the designer. For this reason, the clock line and perhaps others need to be buffered.

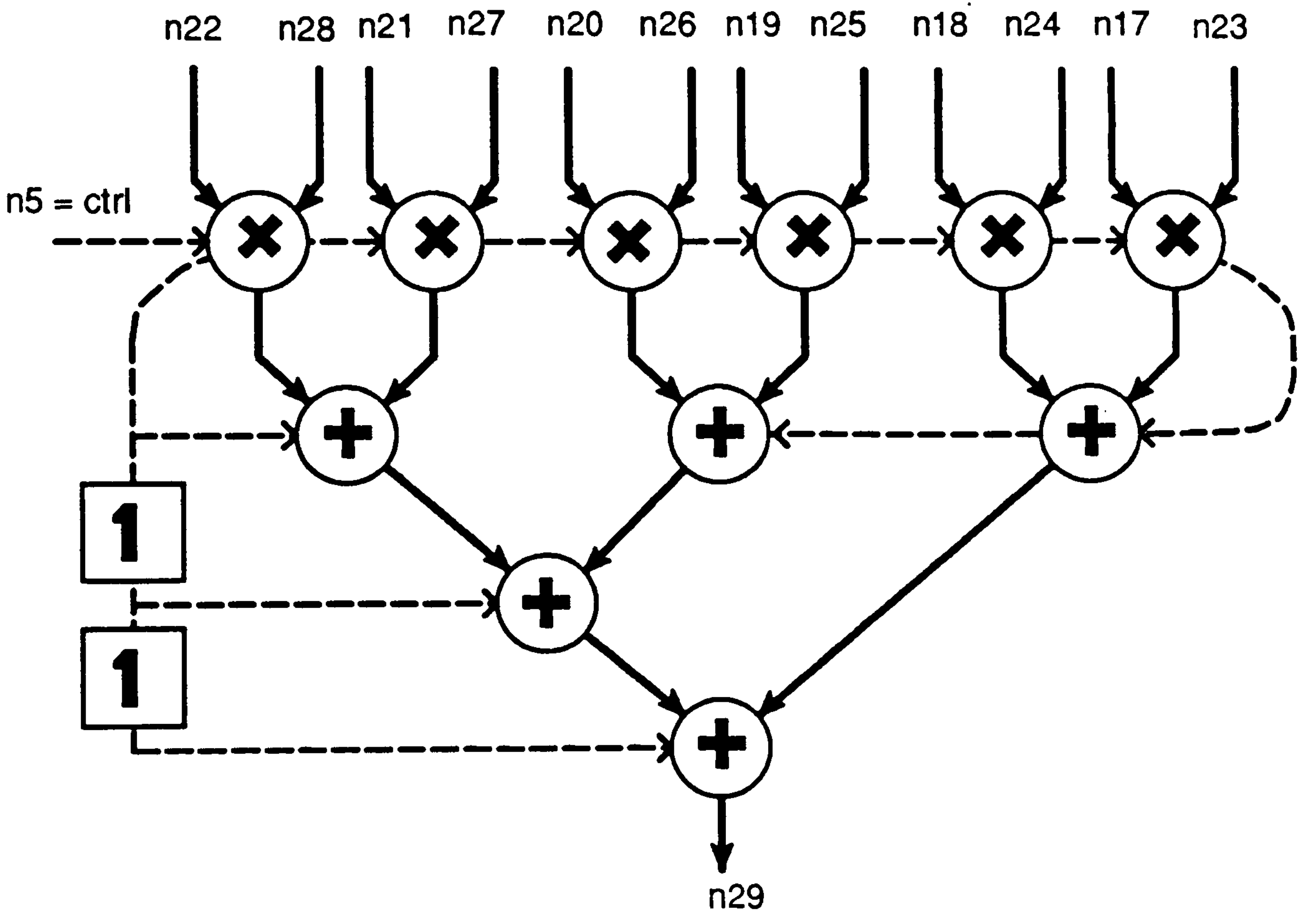
The buffers present a portability problem, as different vendors provide different solutions to the buffering problem. Ideally, the use of a single, parameterisable buffer is preferred, however HILO at present is incapable of this degree of flexibility. For instance of the two CMOS–SOS vendors, which were looked at, MEDL provides only one buffer type, whereas HAFO provides a range of inverters of various drive capability. An alternative approach to the parameterisable buffer, compatible with these libraries is to parallel–up inverters/buffers.

Pads

PADs are treated as special PRIMITIVEs with no parameters. A PAD consists of the appropriate pad from the vendor's library, in the case of an input pad driving a buffered half–latch, and in the case of an output pad driven by a plain half–latch. In this case the BLOCKs for the input–pad and output–pad are the same as the CELLs, as a pad cannot be constructed out of simpler elements. Input and output half latches should be clocked on opposite clock phases, consistent with FIRSTs conventions for off–chip data transfer, and for on–chip data transfer between PRIMITIVEs.

8.9. Example of generated HILO

The software was tested on a simple six point FIR filter. Figure 7.6 shows schematically the construction of the filter.



Schematic of FIR filter

Figure 8.6

The software successfully generated a netlist which was syntactically correct and which we were able to simulate using both an NEC cmos cell library and MEDL's SOS cell library, after only a change in the definitions of the CELLs. The complete netlist generated was over 400 lines long, as it contained the definitions of the PRIMITIVEs and the MODULEs and BLOCKs used. Following is an example of HILO code for the core of the chip at the level of PRIMITIVEs and corresponding to Figure 7.6.

```
****{
CCT CORE (n5,n17,n18,n19,n20,n21,n22,
                              n23,n24,n25,n26,n27,n28,PHI,n29)
**
MULT10800           prim1(PHI,n17,n23,n5,n30,,n6) ;
MULT10800           prim2(PHI,n18,n24,n5,n31,,) ;
MULT10800           prim3(PHI,n19,n25,n5,n32,,) ;
MULT10800           prim4(PHI,n20,n26,n5,n33,,) ;
MULT10800           prim5(PHI,n21,n27,n5,n34,,) ;
MULT10800           prim6(PHI,n22,n28,n5,n35,,n7) ;
ADD01000            prim7(PHI,n30,n31,0,n6,n36,) ;
ADD01000            prim8(PHI,n32,n33,0,n6,n37,) ;
ADD01000            prim9(PHI,n34,n35,0,n7,n38,) ;
ADD01000            prim10(PHI,n36,n37,0,n8,n39,) ;
ADD01010            prim11(PHI,n39,n38,0,n9,n29,) ;
CBITDELAY01         prim12(PHI,n7,n8) ;
CBITDELAY01         prim13(PHI,n8,n9) ;
INPUT n5 n17 n18 n19 n20 n21 n22 n23 n24 n25 n26 n27
      n28 PHI;
UNID n30 n6 n31 n32 n33 n34 n35 n7 n36 n37 n38 n8 n39
     n9 n29.
****}
```

8.10. Performance of the Resulting Netlist

As bit-serial systems have an inherent high level of pipelining and as logical complexity of blocks are low, most nodes in the circuit have a low loading connecting to a small number of known connections, which are kept small through the design of the PRIMITIVEs. Thus most nodes in the circuit have a low capacitive loading allowing high performance.

Unfortunately this is not the case for the CLOCK line, which feeds to each latch in the circuit. Thus the clock line is severely loaded and we have found this to have limited the performance of our generated netlists.

The easiest method to overcome this problem is to buffer the clock line with a single large buffer at the start of the line. The strength of this buffer will vary with the loading on the clock line, to provide sufficient buffering for the load. As parameterisable buffers were not included in the libraries of the vendors we looked at, nor is this facility available in HILO, although vendors supply buffers of various strengths, another method must be sought. Parallelling up of buffers is the simplest scheme though not the most efficient.

A better clock distribution scheme is to cascade buffers[99] sizing them along the cascade. Buffers fan out hierarchically to supply the clock to internal points, but this is a difficult scheme to implement with the method we have chosen to expand the netlist, as it requires a flattened netlist to be performed optimally.

Conclusions

This chapter describes the FIRST to semicustom gate netlist compiler, ENGEN. The software generates HILO compatible output, which may be tailored with little effort to suit particular silicon vendors.

HILO for a specific example was successfully produced and the functionality of the component was verified using random test vectors. Simulation indicated performance at 10MHz, for MEDL's CMOS–SOS library, using the HILO simulator with MEDL's models, an example of the output follows:

```
NAND GATES = 652

NOR GATES = 224

BUF GATES = 459

NOT GATES = 2395

BUFIF1 GATES = 1660

NOTIF0 GATES = 28

TRANIF0 GATES = 1

TRANIF1 GATES = 1


LOADED CIRCUIT USES 2160 KBYTES


            O

            U

            T

 ---TIME--

         0  X

      3401  1

      3507  0

      3701  1

      3907  0

      4001  1

      4107  0

      4301  1

      4607  0

      4701  1


FINISH OF FAULT FREE SIMULATION


FINISH TIME = 5000

401 SECONDS   485266 EVENTS   192 KBYTES
```

Which can be compared with the expected results shown in Figure 7.7, also showing the first three random test vectors used in the simulations. The results were obtained using the FIRST simulator.

```
T
I                          1  1                 3  3                 4
M                          5  6                 1  2                 7
E          3               3  3                 3  3                 3
           0               0  0                 0  0                 0


                8 sign extension bits added to 8 bit
                coefficient for compatibility with
                16 bit data wordlength.
                     /       \              /       \              /       \
C1         0010110111111111  1110111111111111  0010010111111111  ...
C2         1100011000000000  0111110000000000  0000000000000000  ...
C3         1000001000000000  1000001111111111  1001101111111111  ...
C4         0001101000000000  1010101000000000  0100001000000000  ...
C5         0111110000000000  1101000111111111  1101010111111111  ...
C6         0000000111111111  1010110000000000  1010101111111111  ...
           lsb          msb  lsb          msb  lsb          msb

                2 guard sign extension bits added as specified
                on p82 "VLSI signal processing : Denyer & Renshaw"
                          /\                 /\                 /\
D1         0010110111111111  1110111111111111  0010010111111111  ...
D2         1100011000000000  0111110000000000  0000000000000000  ...
D3         1000001000000000  1000001111111111  1001101111111111  ...
D4         0001101000000000  1010101000000000  0100001000000000  ...
D5         0111110000000000  1101000111111111  1101010111111111  ...
D6         0000000111111111  1010110000000000  1010101111111111  ...
           lsb          msb  lsb          msb  lsb          msb

           control signal coincident with LSB
CTRL       1000000000000000  1000000000000000  1000000000000000

           output present after latency of 18 bits
           (1.5*8+2) + (1+1+1) + (1) = 18
           multiplier  adders    pad
OUT        XXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXX  XX10011010011101
```

Expected simulation Results

Figure 8.7

The figure of 10MHz is lower than expected for a silicon on sapphire process, which we attribute to the heavy loading of the clock line. A higher figure is expected if a clock buffering strategy is employed.

The software allows portability between different vendors is achieved by modifying a library of circuit descriptions of eleven CELLs in terms of components supplied by the vendor.

To make better use of the vendor's library, the user can modify elements at the next level of hierarchy, the BLOCK level, but no higher levels are intentionally user alterable.

Chapter 9

Recommendations & Suggestions

9.1. Structural vs. Behavioural Compilation

FIRST is a structural compiler, that is it assembles the logic from a description of how hardware blocks are connected. It derives its function from the functions of the hardware blocks. SECOND which is an enhancement to the FIRST compiler remains a structural compiler, as its input specification is presented to the software via the FIRST structural language interface. The user does not have the facility to write out the algorithm in a behavioural format, that is naive of the final implementation, but this for the type of signal processing system which the compiler addresses itself to is no severe limitation. Each hardware block has a clear behaviour and a system is composed of these blocks in a hierarchical fashion. The behaviour of the overall system is predicatable and defining a system is very much like programming in a higher level language.

Work on making the FIRST language more like a high level programming language was not undertaken in our project. At the University of Hong Kong, work on this has been done[100]. In their work the behaviour can be specified in a manner much more like a conventional programming language. A parser then constructs an equivalent flow graph which can be implemented directly. Also control signals are generated automatically for the system using a graph based algorithm, relieving the user of their specific definition. Although complete behavioural compilation is still far off, the use of language interface which mimics a normal programming language can move us closer to the goal.

Alternative work on compilers for signal processing use a language interface which is modelled much closer to a normal programming language. At the Technische Hochschule Darmstadt, flexibility is added to their compiler by the use of the input language ALGIC[101], [102]. CATHEDRAL uses the language SILAGE [62] as the input description.

It is recommended that some more work on behavioural level compilation be done. This does not need to be very sophisticated before it starts to make the environment for a user much more comfortable, for example, calculation of the control signals in FIRST. The general problem, that of transforming a general programming language specification to silicon artwork for an integrated circuit performing the same function remains very complex. It contains problems which are likely to be unsolvable if we want optimal results. Work on this area is just beginning at Edinburgh University, whilst this thesis is being written by S.A.R.I (Silicon Architectures Research Initiative), a consortium of industrial partners with the University. The language used in this project is ADA, which is transformed into a lower level language such as Praxis's ELLA.

Mixed–Tree Dynamic Logic

The operating speed which was used as a goal for the performance of our generated cell library was set at 50MHz for a state–of–art process. Taking the ES2 2 μm process to be such, we should hopefully achieve a clocking rate of this order for our automatically generated library. Unfortunately the transistor efficient mixed–tree dynamic scheme is not well suited to high speed operation, although for lower performances it is useful as a low area technique.

Simulations on the speed using the circuit simulator SPICE[103] of the mixed–tree dynamic scheme indicated the problems with the scheme. These results indicated that the rise time (or fall time) of the mixed–tree pull–up (or pull–down) network limits the speed of operation of circuits to about a quarter of our intended speed in the worst case, using minimum sized transistors, which is what is generated by our software in SECOND. The sense circuit cannot compensate for this loss in speed as voltages will not have reached the threshold at the point at which it becomes activated. The threshold voltage of the transistor, the point at which the sense circuit becomes activated affects critically the speed, and would thus need careful processing to ensure high speed.

A simple solution to this problem is to use another scheme, perhaps fully static operation, when high speed is critical and to use the dynamic scheme when area and not time is the costliest resource.

9.2. A Bit–Serial Design Frame

This section introduces a design frame which could be used to make bit–serial components more attractive in a bit–parallel world.

Introduction

Bit parallel systems dominate the electronics industry. Most designers of electronic systems are familiar with the parallel approach which has become the VLSI paradigm. Thus many of the more powerful electronic components which are widely available are bit–parallel.

In contrast, bit–serial systems do not have such widespread support, diminishing the appeal to build such designs. Furthermore interfacing between the two worlds of bit–serial/bit–parallel is complicated by the difference of data formats and control signals.

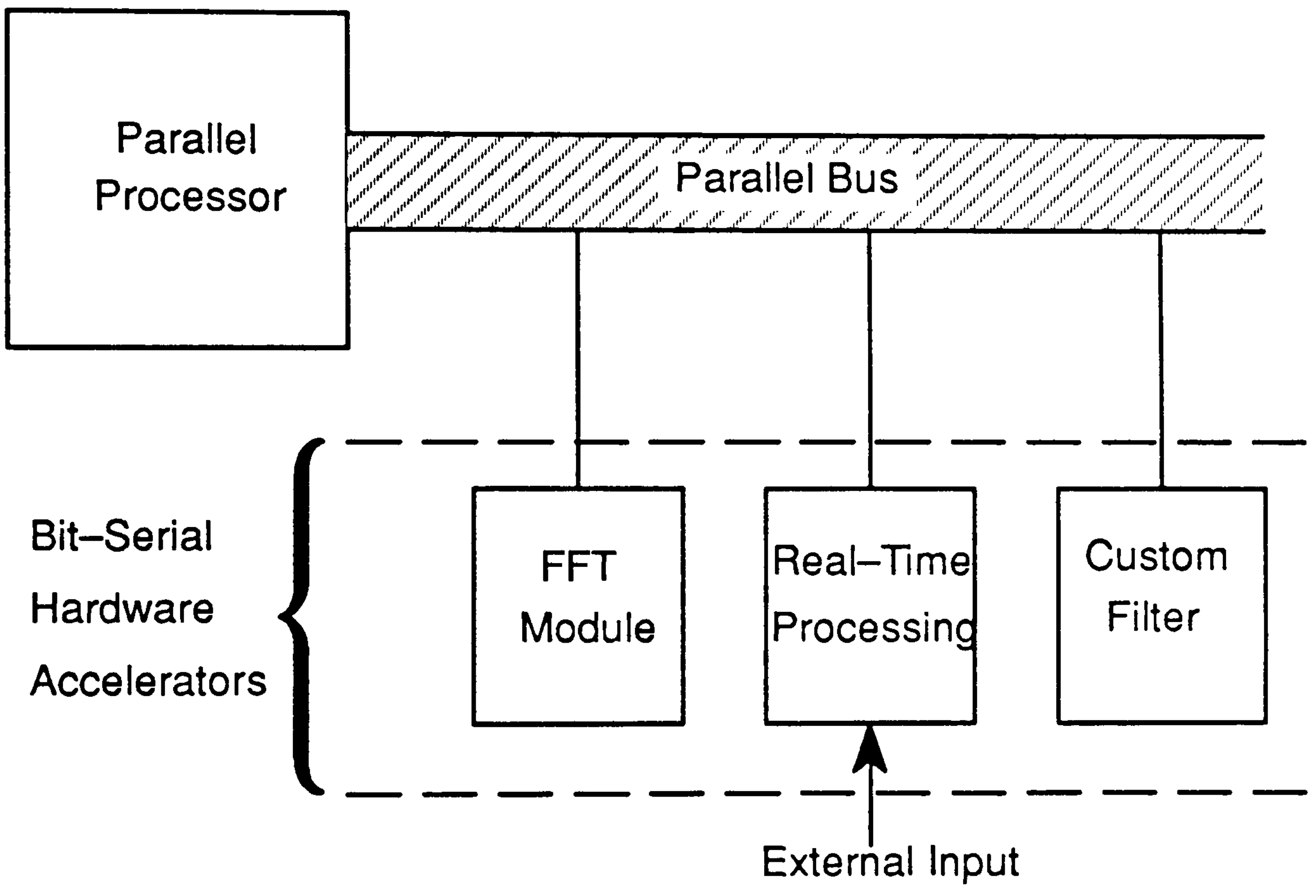
Despite this, bit–serial design has many advantages to offer. Components have a compact size, because only a bit at a time is transferred between elements. Wiring for data communication is reduced, needing only a single wire for communication of a bit–serial signal. Bit–serial systems tend to have a greater processing throughput for an equal area of silicon, because of the reduced data communication overhead, logical depth and the use of highly pipelined logic.

A design frame is an attempt to reduce this separation between the two domains, by providing a usable and understandable interface connecting them. The design frame[104] concept has already been proposed for purely bit–parallel systems. In this version data can be sent and retrieved by parallel circuits exterior to the frame and processed within the frame by a bit–serial processor. The bit–serial component, in effect, appears as a parallel element to the world outside the frame. Thus a designer can enjoy the benefits of bit serial design without having to isolate it from the bit parallel universe.

This is a tentative description of a frame capable of providing such an interface.

Aims

The type of application where the combination of serial and parallel components is felt to be most effective is in systems composed of a general purpose bit parallel processing element communicating with a number of fixed single function serial sub–processes (hardware accelerators). Thus computationally intensive tasks can be delegated out to bit–serial processors optimised in performing that computation under the direction of the main processor. Figure 8.1 is an attempt to illustrate the theme.

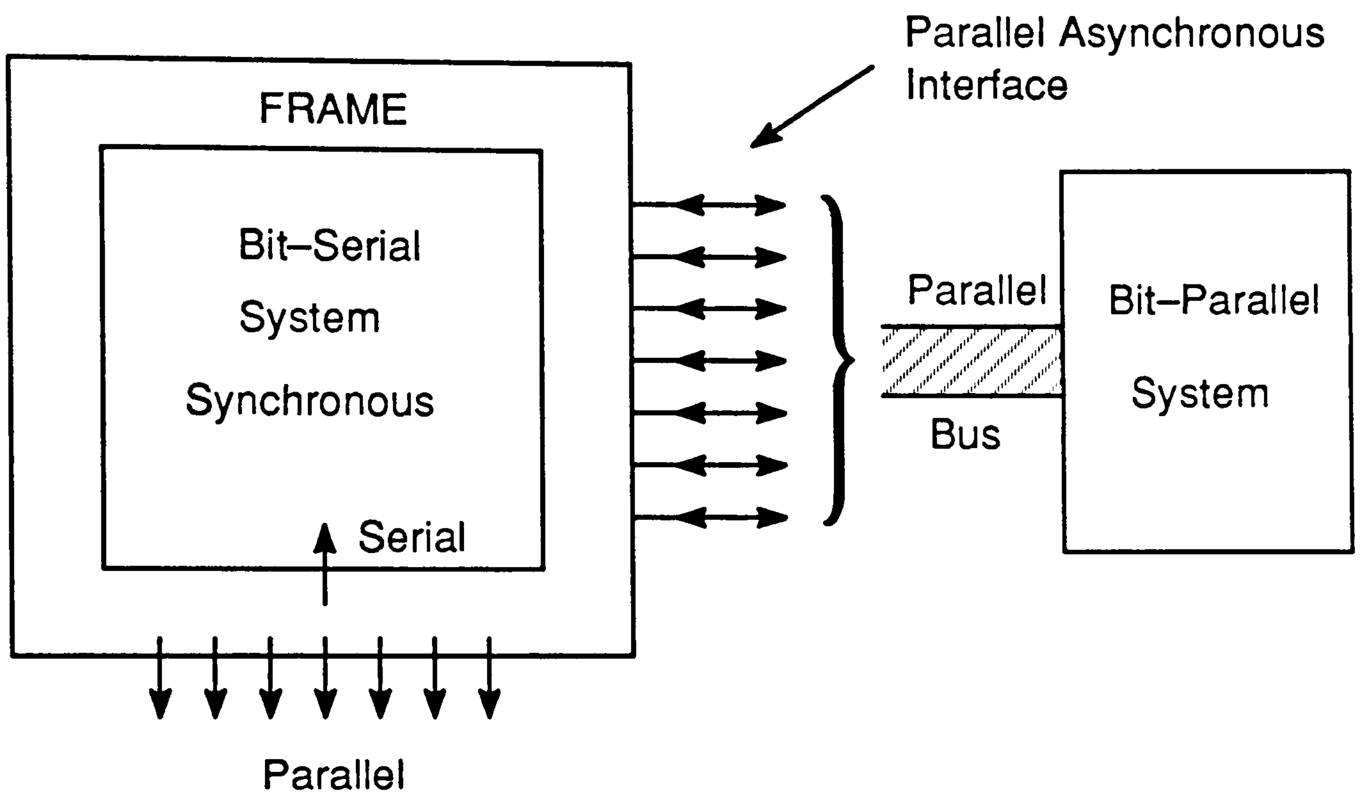


Bit–serial processors connected to Parallel processor

Figure 9.1

In order that the processes can operate at their maximum rates, an asynchronous interface is desirable, since synchronous operation is complicated by the varying time requirements of different operations.

Within the frame all signals should conform to the bit–serial format, although it may be profitable to maintain some signals in their bit–parallel form, e.g. coefficients to multipliers. External to the frame they must be bit–parallel. This is shown in Figure 8.2, which shows the bit–serial field surrounded by the frame.



Use of Frame within a System

Figure 9.2

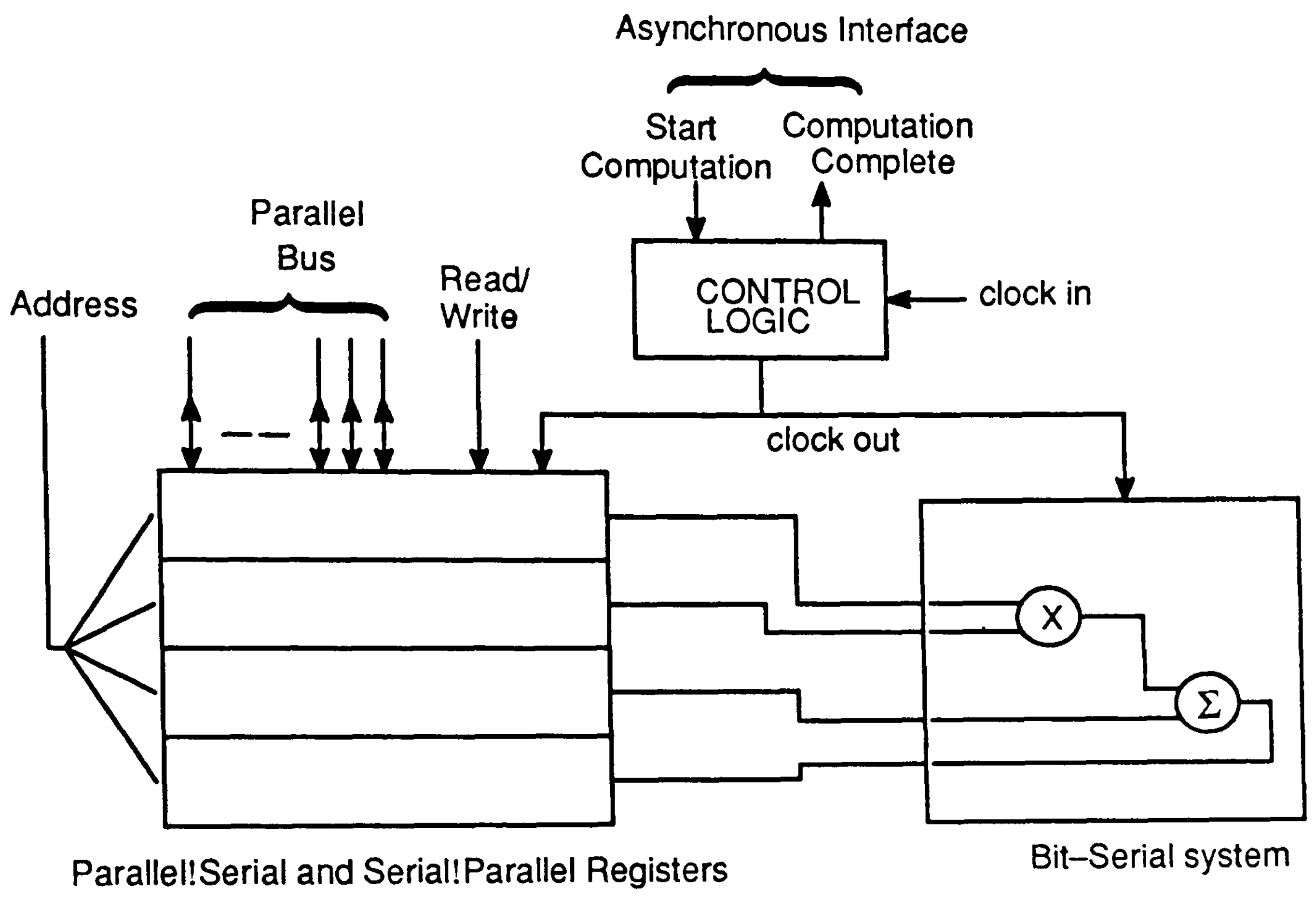
The frame is felt to be a fixed physical entity, within which an arbitrary bit–serial component can be designed. The logic for the design frame could be incorporated with the frame for the input output pads for the circuit. In this case the conversion between formats should be done using only a small amount of extra logic so that plenty of space is left for the bit–serial processor.

Another aim is to keep the interface control as simple as possible, so that using the interface does not introduce the need for extraneous pieces of glue logic. In this case, the interface is accomplished using two signals, a *Start Computation* and a *Computation Complete* signal to initiate and indicate the finish of an operation.

A Frame Implementation

To achieve the aims, we require a means of converting serial to parallel and vice versa, to be able to send and retrieve information from the process. This is done using shift registers to perform this function. They allow loading of data in parallel and shifting out in serial, when used for parallel to serial conversion, and the other way around when used for serial to parallel conversion. When a register must be both readable and writable, for both input and output operations, a register combining both types of conversions must be utilised.

Figure 8.3 shows the method by which the registers connect with the two domains. This also shows how the control signals interface to the circuitry. Their function is to start a fixed number of clock pulses to be sent to the bit–serial hardware and also to the data registers. This allows the registers to clock out their contents to the serial hardware for processing whilst simultaneously clocking in the results of an operation.



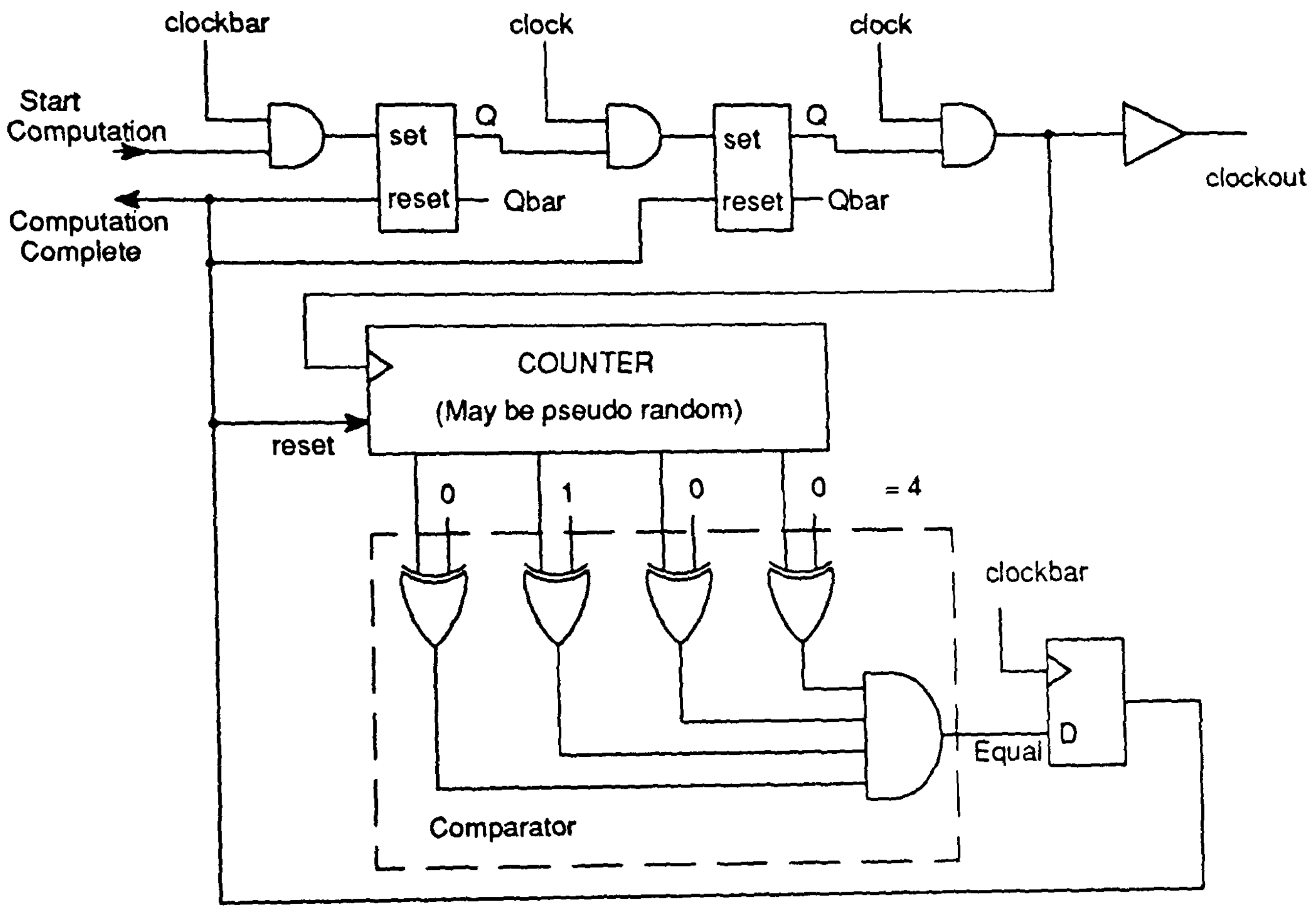
Registers in the Frame

Figure 9.3

The results clocked in will not necessarily be those produced for the data which is clocked in at the time, unless the serial processor has no latency. This means that the results from the serial hardware will be from a previous loading of the input to the serial circuit. This may limit some of the applications of the hardware in applications where latency may be a problem.

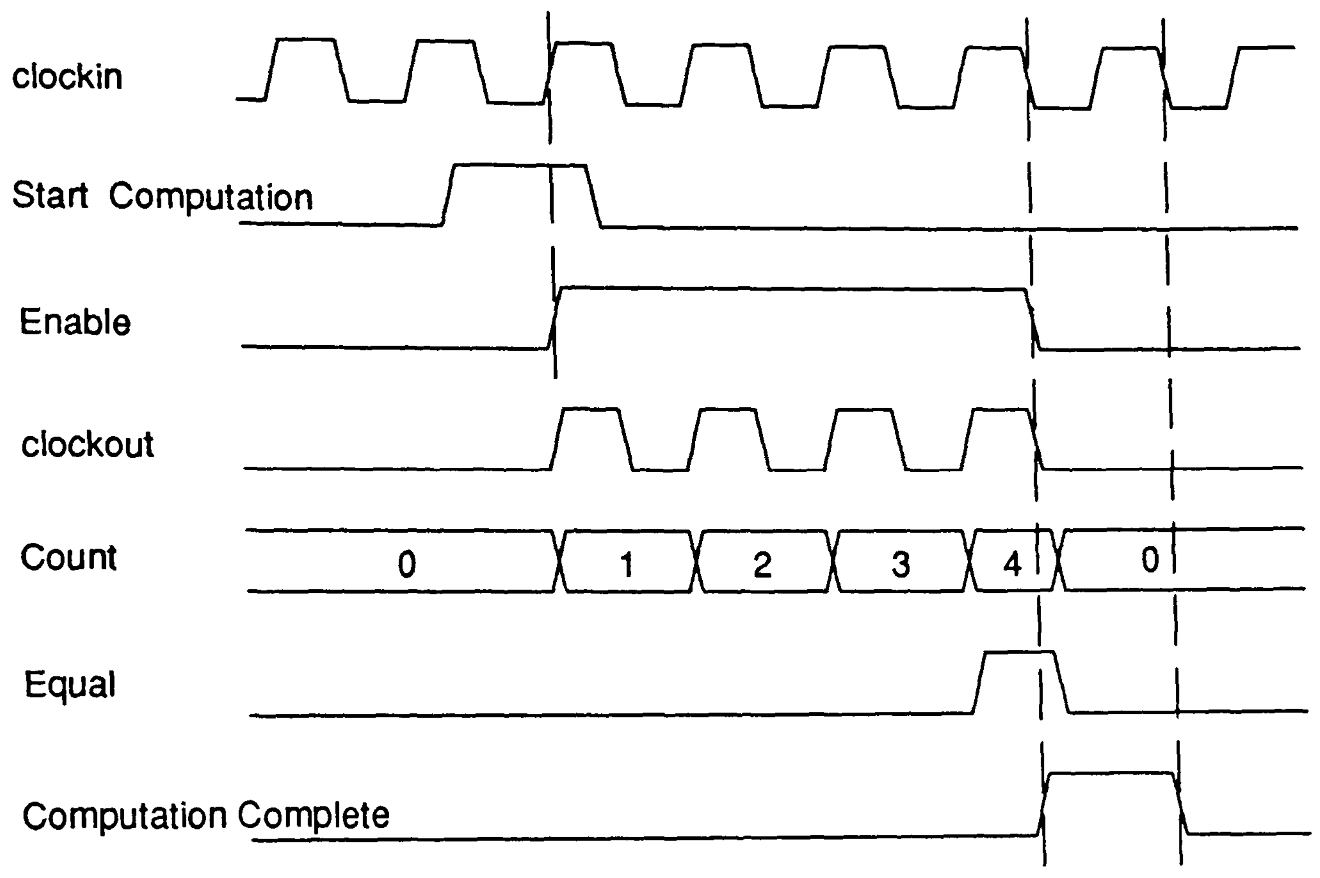
A probable arises if the latency of the serial hardware is not a multiple of the wordsize, meaning that two fractions of words are present in the parallel register, when the main bit parallel processor intends to read the result. The simplest solution to this problem is to put extra delay elements on the serial outputs to make the latency a multiple of wordsize so that data correctly aligns.

To handle the control signals, a piece of control logic is present, Figure 8.4 show a possible logic implementation. This handles the two asynchronous control signals *Start Computation* and *Computation Complete*, and provides as an output a fixed number of maximum rate clock pulses to the fixed function, bit serial accelerator it controls. Figure 8.5 shows the timing of the logic signals.



Control Logic for Frame

Figure 9.4



Signal Timing for Control Logic

Figure 9.5

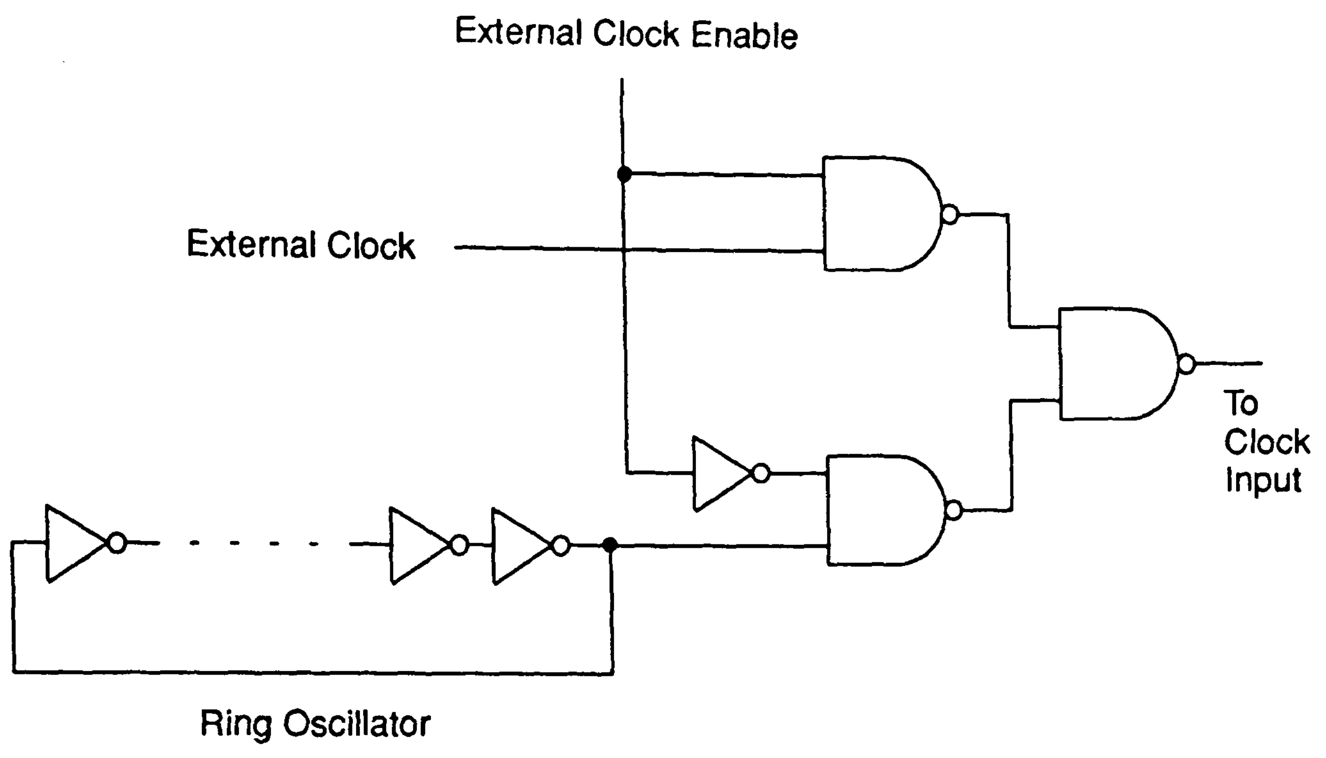
An Internal Clock Option

Supplying the clock externally to the frame may be an inconvenience, in this case an internally generated clock signal may be preferable. Here we use a ring oscillator to generate the clock signal. This clock is fed into the control circuit of Figure 8.4, in exactly the same way as the external clock. It is advisable to have the ability to override this internal clock with an external clock. This is useful because :–

– The internal ring oscillator may not be working.

– The ring oscillator frequency is either to high for the bit–serial circuitry or too low to make optimal use of the circuit.

Figure 8.6 shows a suggested circuit for this internal clock option.



Internal Clock Option

Figure 9.6

Criteria for Frame

This section summarises by looking at some criteria by which the frame implementation may be judged.

Flexibility

The frame is to be used in general purpose systems so must be able to function with a wide range of circuits. This is a feature of the frame discussed, though it is limited to fixed function bit-serial hardware which is able to perform its task in a fixed number of clock cycles. A further restriction on the bit-serial hardware is that it should be static in order that the internal state is not lost whilst waiting for the next piece of data as interchanges cannot be guaranteed at fixed intervals. The first restriction is not serious since this is exactly the type of bit-serial processing element which is produced by the *FIRST* silicon compiler.

Simple Interface

To be usable the frame should not have overly complex signal conventions, otherwise any designer will, rightly, give up and try something else. The control signal conventions utilise only two signals which drive the asynchronous interface extra to the normal read/write and addressing logic required. This keeps the interface at a simple and understandable level.

Low Circuit Overhead

The interfacing frame must not be a major consumer of silicon resources otherwise its uses will be severely limited. With the frame implementation discussed, the main consumer of silicon will be the serial/parallel registers whose number will depend on the application. There are no schemes where the use of such registers can be totally eliminated whilst providing a serial/parallel interface. The area taken by the control logic will be small in comparison.

A simple design frame which can surround a bit–serial system is presented. The logic required is kept minimal whilst allowing it to be easily used in the type of application described.

Chapter 10

Conclusions

The intent of this thesis, that of automatically generating the cell library to FIRST using a design rule independent symbolic layout, has been completed to the point that a SLICE of the multiplier described earlier could be synthesised automatically. The layout is competitive with that produced by commercial software such as SOLO 1000 supplied by ES2. Gate matrix is chosen as a suitable symbolic layout scheme because of its regularity and suitability for automation. Using symbolic layout allows us to change generated layout rapidly for use with new processes.

The work built upon the transistor net synthesis and symbolic layout software MOSYN and MOSLAY, with the aim of giving FIRST both flexibility and portability. To these pieces of software were added programs to flesh the symbolic layout to the silicon mask data format CIF, and to route together these generated BLOCKs into a bit–slice. The software task is kept manageable by introducing some new levels of hierarchy to those used by FIRST. The bit–slice paradigm is used in constructing PRIMITIVEs as this is an efficient scheme and well suited to the bit–serial components used in FIRST. This software which is meant to be used as an extension to FIRST we term SECOND. With it we are able to generate new logic block elements and slices to form new PRIMITIVEs for inclusion in FIRST from a specification in terms of connectivity between BLOCKs and the boolean logic of the BLOCKs.

Novel circuit and architectural techniques, described earlier were used to enhance the performance or area efficiency of our generated circuit elements. These techniques are :

Single Phase Clocking – simplifying clock routing and allowing increased clock rates, Twin–pipe performance enhancement – doubling the throughput of a bit–serial element without a doubling of the area required,

Mixed–Tree Dynamic Logic – a transistor efficient scheme where area may be at a premium, although not recommended where high speeds are needed.

The expected area of our generated layout produced with SECOND when compared with SOLO 1000 was not as small as had been expected. It is much simpler to use a readily available tool such as SOLO 1000 to generate a new cell library for us. A program for converting FIRST code to MODEL (the input language of SOLO 1000) would all that would be required. Justifying the software effort to replicate available and viable alternatives becomes difficult unless a definite increase in efficieny can be demonstrated. It is felt that by using more sophisticated software in SECOND that such an advantage could be demonstrated, though at the expense of a greater software effort.

As an alternative approach to process portability, a semi–custom route was also explored. Netlists for a gate–array vendor are generated from intermediate netlists produced by FIRST. The netlist can be generated for different vendors through the specification of a small universal cell library or larger block elements if these are present in the vendors library. This approach has generated correct netlists for example files, with a six point FIR filter given as an example. The software for performing this task we have called ENGEN, and allows the rapid generation of working netlists in HILO to be produced from FIRST code. Performance of the resulting netlist has been found low through simulation. The source of this poor performance was identified as the heavy loading on the clock line in these generated netlists. By incorporating a clock buffering strategy in the netlists produced should enhance the performance of these netlists.

A suggestion for further research is seen as the behavioural to structural translation of a system specification. Even a small amount of effort in this area is seen as increasing the ease of use of the tools for the user. This aspect is now being covered at the University with industrial collaboration as SARI (Silicon Architectures Research Initiative).

The route to process portability can be achieved using the techniques described in this thesis. Also covered is a bit–serial design frame which allow bit–serial processors to be used with a parallel system. Appendix 1 includes detail of the functional level expansion of an algorithm, not necessarily for digital signal processing, as a microcontroller which was done with collab-oration of Thorn EMI under the CASE award scheme.

References

1. D. L. Johannsen, "Silicon Compilation," PhD Thesis, California Institute of Technology, 1981.

2. D. L. Johannsen, "Bristle Blocks: A Silicon Compiler," *Proc. 16th DA Conf.*, San Diego, 1979.

3. P. B. Denyer and D. Renshaw, *VLSI Signal Processing – A Bit–Serial Approach*, Addison–Wesley, 1985.

4. Neil Bergmann, "A Case Study of the F.I.R.S.T. Silicon Compiler," in *Third Caltech Conf. on VLSI*, ed. Randal Bryant, pp. 413 – 430, Computer Science Press, Pasadena, 21st – 23rd March 1983.

5. J. R. Southard, "MacPitts: An Approach to Silicon Compilation," *Computer*, vol. 16, pp. 74 – 82, December 1983.

6. Leland B. Jackson, James F. Kaiser, and Henry S. McDonald, "An Approach to the Implementation of Digital Filters," *Transactions on Audio and Electroacoustics*, vol. AU–16, no. 3, pp. 413 – 421, IEEE, September 1968.

7. Noble R. Powell and John M. Irwin, "Signal Processing with Bit–Serial Word–Parallel Architectures," *SPIE*, vol. 154, pp. 98 – 104, 1978.

8. R. F. Lyon, "Two's Complement Pipeline Multipliers," *Trans. IEEE*, vol. COM–24, pp. 418 – 425, April 1976.

9. R. F. Lyon, *A Bit–Serial Architectural Methodology for Signal Processing*, VLSI 81, pp. 131 – 140, Academic Press, Proceedings 1st Int. Conf. on VLSI, Edinburgh, 18 – 21 August 1981.

10. P. B. Denyer, D. Renshaw, and N. Bergmann, "A Silicon Compiler for VLSI Signal Processors," *Proc. ESSCIRC'82*, pp. 215 – 218, Brussel, September 1982.

11. P. B. Denyer and D. Renshaw, *VLSI Signal Processing – A Bit–Serial Approach*, pp. 18 – 19, Addison–Wesley, 1985.

12. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice–Hall, 1975.

13. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison–Wesley, 1980.

14. J. A. Newkirk and R. Mathews, *The VLSI Designer's Library*, Addison–Wesley, 1983.

15. Stephen Sunter, "Designing a CMOS standard cell library," in *Proceedings Custom Integrated Circuits Conference*, pp. 237 – 240, IEEE, Portland, Oregon, 4th – 7th May 1987.

16. R. K. Brayton and et al., in *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

17. J. A. Darringer and et al., "LSS: A System for Production Logic Synthesis," *IBM J. Res. Develop.*, vol. 28, pp. 537 – 545, September 1984.

18. David Gregory, Karen Bartlett, Aart de Geus, and Gary Hachtel, "Socrates: A System for Automatically Synthesizing and Optimizing combinational logic," in *Proceedings 23rd Design Automation Conference*, pp. 79 – 8 5, IEEE, 1986.

19. N. H. E. Weste and K. Eshraghian, in *Principles of CMOS VLSI Design*, Addison–Wesley, 1985.

20. Various Authors, "Semi–Custom IC Design and VLSI," in *Proceedings IEE symposium*, ed. P. J. Hicks, Peter Peregrinus Ltd. for IEE, 1983.

21. Michiel A. Beunder, Bernd Hoefflinger, and Juergen P. Kernhof, "New Directions in Semicustom Arrays," *Journal Solid State Circuits*, vol. 23, no. 3, pp. 728 – 735, IEEE, June 1988.

22. Michiel Beunder, Juergen Kernhof, and Bernd Hoefflinger, "The CMOS GATE FOREST: High Performance Semicustom ASIC's for Complex Logic and Memory," in *Digest of Technical Papers, ESSCIRC*, pp. 245 – 248, vd e verlag gmbh., Bad Soden, Federal Rep. of Germany, 23rd – 25th September 1987.

23. Gary Robson, "Automatic Placement and Routing of Gate Arrays," *VLSI Design*, pp. 35 – 43, April 1984.

24. Shigeo Noda, Hitoshi Yoshizawa, Etsuko Fukuda, Haruo Kato, Hiroshi Kawanishi, and Takashi Fujii, "Automatic Layout Algorithms for Function Blocks of CMOS Gate Arrays," in *Proceedings 22nd Design Automation Conferenc e*, pp. 46 – 52, IEEE, 1985.

25. Alfred E. Dunlop and Brian W. Kernighan, "A Procedure for Placement of Standard–Cell VLSI Circuits," *Transactions CAD.*, vol. CAD–4, no. 1, pp. 92 – 98, IEEE, January 1985.

26. Mark R. Hartoog, "Analysis of Placement Procedures for VLSI Standard Cell Layout," in *Proceedings of 23rd Design Automation Conference*, pp. 314 – 319, IEEE, 1986.

27. Takeshi Yoshimura and Ernest S. Kuh, "Efficient Algorithms for Channel Routing," *Transactions CAD.*, vol. CAD–1, no. 1, pp. 25 – 35, IEEE, January 1982.

28. Ash M. Patel, Norman L. Soong, and Robert K. Korn, "Hierarchical VLSI Routing – An Approximate Routing Procedure," *Transactions CAD.*, vol. CAD–4, no. 2, pp. 121 – 126, IEEE, April 1985.

29. James Reed, Alberto Sangiovanni–Vincentelli, and Mauro Santomauro, "A New Symbolic Channel Router: YACR2," *Transactions CAD.*, vol. CAD–4, no. 3, pp. 208 – 219, IEEE, July 1985.

30. Ronald L. Rivest and Charles M. Fiduccia, "A GREEDY Channel Router", in *Proceedings 19th Design Automation Conference*, pp. 418 – 424, IEEE, 1982.

31. David N. Deutsch, "A Dogleg Channel Router", Proceedings 13th Design Automation Conference, pp. 425 – 433, IEEE, 1976.

32. Carl Sechen and Alberto Sangiovanni–Vincentelli, "TimberWolf3.2 : A New Standard Cell Placement and Global Routing Package," in *Proceedings of 23rd Design Automation Conference*, pp. 432 – 439, IEEE, 1986.

33. J. P. Gray, I. Buchanan, and P. S. Robertson, "Designing Gate Arrays Using a Silicon Compiler," *Proc. 19th DA Conf.*, pp. 377 – 383, Las Vegas, 1982.

34. Natalie Royal, John Hunter, and Irene Buchanan, "A Case Study in Process Independence," in *Proceeding 22nd Design Automation Conference*, pp. 591 – 596, IEEE, 23rd – 26th June, 1985.

35. Arnold Weinberger, "Large Scale Integration of MOS Complex Logic: A Layout Method," *Journal Solid State Circuits*, vol. SC–2, no. 4, pp. 182 – 190, IEEE, December 1967.

36. N. H. E. Weste and K. Eshraghian, "Gate–matrix layout," in *Principles of CMOS VLSI Design*, pp. 274 – 278, Addison–Wesley, 1985.

37. Alexander D. Lopez and Hung–Fai S. Law, "A Dense Gate Matrix Layout Method for MOS VLSI," *Journal Solid State Circuits*, vol. SC–15, no. 4, pp. 736 – 740, IEEE, August 1980.

38. Omar Wing, Shuo Huang, and Rui Wang, "Gate Matrix Layout," *Transactions CAD.*, vol. CAD–4, no. 3, pp. 220 – 231, IEEE, July 1985.

39. Hung–Fai Stephen Law, "Gate Matrix: A Practical, Stylized Approach to Symbolic Layout," *VLSI Design*, pp. 49 – 59, September 1983.

40. R. H. Krambeck, D. E. Blahut, HF. S. Law, B. W. Colbry, H. C. So, M. Harrison, and J. Soukup, "Top Down Design of a one chip 32–bit CPU," in *Proceedings of 1st Int. Conf. on VLSI*, ed. J. P. Gray, vol. VLSI 81, pp. 35 – 41, Academic Press, Edinburgh, 18th – 21st August 1981.

41. D.K. Hwang, W.K. Fuchs, and S.M. Kang, "An Efficient Approach to Gate Matrix Layout," in *Digest of Papers, Int. Conf. on Computer–Aided Design, ICCAD*, pp. 312 – 315, IEEE, 1986.

42. H.W. Leong, "A New Algorithm for Gate Matrix Layout," in *Digest of Papers, Int. Conf. on Computer–Aided Design, ICCAD*, pp. 316 – 319, IEEE, 1986.

43. Koh'ichi Nakatani, Takashi Fujii, Tohru Kikuno, and Noriyoshi Yoshida, "A Heuristic Algorithm for Gate Matrix Layout," in *Digest of Papers, Int. Conf. on Computer–Aided Design, I CCAD*, pp. 324 – 327, IEEE, 1986.

44. C. Piguet, E. Dijkstra, and G. Berweiler, "Automatic Generation of CMOS Layout Cells under Topological Constraints," in *Proceedings ESSCIRC 1986*, pp. 68 – 70, Delft, Netherlands, September 1986.

45. Huang Shuo and Omar Wing, "Improved Gate Matrix Layout," in *Digest of Papers, Int. Conf. on Computer–Aided Design, ICCAD*, pp. 320 – 323, IEEE, 1986.

46. R. D. Freeman, S. M. Kang, C. G. Lin–Hendel, and M. L. Newby, "Automated Extraction of SPICE Circuit Models from Symbolic Gate Matrix Layout with Pruning," in *Proceedings 23rd Design Automation Conference*, pp. 4 18 – 424, IEEE, 1986.

47. Sung Mo Kang, R. H. Krambeck, Hung–Fai Stephen Law, and Alexander D. Lopez, "Gate Matrix Layout of Random Control Logic in a 32–bit CMOS CPU Chip Adaptable to Evolving Logic Design," *Transactions CAD.*, vol. CAD– 2, no. 1, pp. 18 – 29, IEEE, January 1983.

48. Neil Weste, "Virtual Grid Symbolic Layout," in *Proceedings 18th Design Automation Conference*, pp. 225 – 233, IEEE, 1981.

49. Neil Weste and Bryan Ackland, "A Pragmatic Approach to Topological Symbolic IC Design," in *Proceedings of 1st Int. Conf. on VLSI*, ed. J. P. Gray, vol. VLSI 81, pp. 117 – 129, Academic Press, Edinburgh, 18th – 21st August 1981.

50. M. Collins Cope, I. P. Reid, K. P. Page and M. J. Jones, "STICKS – A Symbolic Layout System for VLSI design," in *Proceedings of the third SILICON DESIGN Conference*, pp. 381 – 391, Electronic Design Automation Limited, Ed. Dr A. P. Ambler, July 15th – 17th 1986.

51. Neil Bergmann, "Generalised CMOS – A Technology Independent CMOS IC Design Style," in *Proceedings 22nd Design Automation Conference*, pp. 273 – 278, IEEE, 1985.

52. Thaddeus J. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, 1986.

53. Jin Kim and John McDermott, "TALIB : An IC Layout Design Assistant," in *Proceedings of AAAI Conference*, pp. 197 – 201, Washington DC, 1983.

54. Paul W. Kollaritsch and Neil H. E. Weste, "TOPOLOGIZER: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout," *Journal Solid State Circuits*, vol. SC–20, no. 3, pp. 799 – 804, IEEE, June 1985.

55. John Ousterhout, "Editing VLSI Circuits with Caesar," User Manual, University of California, 11th June 1985.

56. J. K. Ousterhout and et al., "Magic: A VLSI Layout System," *Proc. 21st DA Conf.*, pp. 152 – 165, Albuquerque, NM, 1984.

57. Stephen P. Pope and R. W. Brodersen, "Macrocell Design for Concurrent Signal Processing," in *Third Caltech Conf. on VLSI*, ed. Randal Bryant, pp. 395 – 412, Computer Science Press, Pasadena, 21st – 23rd March 1983.

58. Jan M. Rabaey, Stephen P. Pope, and Robert W. Brodersen, "An Integrated Automated Layout Generation System for DSP Circuits," *Transactions CAD.*, vol. CAD–4, no. 3, pp. 285 – 296, IEEE, July 1985.

59. Rajeev Jain and Robert W. Brodersen, "Integration of Algorithms in Silicon," in *Digest of Technical Papers, ESSCIRC*, pp. 239 – 244, vde verlag gmbh., Bad Soden, Federal Rep. of Germany, 23rd – 25th September 1987.

60. B. De Loore, H. De Man, J. Decaluwe, and L. Lauwers, "A Circuit Design and Layout Strategy for Bit–Serial Architectures," in *Digest of Technical Papers, ESSCIRC*, pp. 253 – 256, vde verlag gmbh., Bad Soden, Federal Rep. of Germany, 23rd – 25th September 1987.

61. L. Claesen, F. Catthoor, H. De Man, J. Vandewalle, S. Note, and K. Mertens, "A CAD Environment for the thorough Analysis, Simulation and Characterisation of VLSI implementable DSP Systems," in *Proceedings Int. C onf. on Computer Design, ICCD*, pp. 72 – 75, IEEE, 1986.

62. J. Rabaey, J. Vanhoof, G. Goossens, F. Catthoor, and H. De Man, "CATHEDRAL–II : Computer Aided Synthesis of Digital Signal Processing Systems," in *Proceedings Custom Integrated Circuits Conference*, pp. 157 – 160 , IEEE, Portland, Oregon, 4th – 7th May 1987.

63. H. De Man, J. Rabaey, and P. Six, "CATHEDRAL 2 : A Synthesis and Module Generation System for Multiprocessor Systems on a Chip," *NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design*, IMEC, Belgium, 1986.

64. L. Rijnders, Paul Six, and Hugo J. De Man, "Design of a Process–Tolerant Cell Library for Regular Structures Using Symbolic Layout and Hierarchical Compaction.," *Journal Solid State Circuits*, vol. 23, no. 3, pp. 714 – 721, IEEE, June 1988.

65. Kris Croes, Hugo J. De Man, and Paul Six, "CAMELEON: A Process–Tolerant Symbolic Layout System," *Journal Solid State Circuits*, vol. 23, no. 3, pp. 705 – 713, IEEE, June 1988.

66. C. L. Seitz, "System Timing," in *C. A. Mead & L. A. Conway, "Introduction to VLSI Systems"*, pp. 218 – 262, Addison–Wesley, 1980.

67. Lewin, *Logical Design of Switching Circuits*, Nelson, London, 1968.

68. F. J. Hill and G. R. Petersen, in *Introduction to Switching Theory and Logical Design*, Wiley, 1968.

69. R. H. Krambeck, Charles M. Lee, and Hung–Fai Stephen Law, "High–Speed Compact Circuits with CMOS," *Journal Solid State Circuits*, vol. SC–17, no. 3, pp. 614 – 619, IEEE, June 1982.

70. Nelson F. Goncalves and Hugo J. De Man, "NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures," *Journal Solid State Circuits*, vol. SC–18, no. 3, pp. 261 – 266, IEEE, June 1983.

71. M. S. McGregor, P. B. Denyer, and A. F. Murray, "A Single–Phase Clocking Scheme for CMOS VLSI," in *Proceedings 1987 VLSI Research Conference*, pp. 257 – 271, Palo Alto, CA, March 1987.

72. M. S. McGregor, S. G. Smith, P. B. Denyer, and A. F. Murray, "Serial–Data Computation on Twin Pipelines," *Electronics Letters*, vol. 23, no. 6, pp. 292 – 293, IEE, 12th March 1987.

73. J. B. Gosling, "Design of large high–speed binary multiplier units," in *Proceedings of the IEE*, vol. 118, pp. 499 – 505, IEE, March/April 1971. Mentions a series–parallel multiplier which uses the same principl es as the twin–pipe multiplier.

74. S. G. Smith, "Serial/Parallel Modules for Complex Arithmetic," *Electronics Letters*, vol. 22, no. 23, pp. 1256 – 1257, IEE, 6th November 1986.

75. S. G. Smith, "Efficient Serial/Parallel Inner–Product Computation," *Electronics Letters*, vol. 22, no. 14, pp. 750 – 752, IEE, 3rd July 1986.

76. K. Asada, "MOS Circuit Synthesizer: MOSYN/2.0 User Manual," User Manual, Edinburgh/Tokyo University, 6th March 1986.

77. K. Asada, "MOS Circuit Diagram and Symbolic Layout Generator: MOSLAY/1.0 User Manual," User Manual, Edinburgh/Tokyo University, 18th June 1986.

78. Kunihiro Asada and John Mavor, "A MOS Leaf–cell Generation System from Boolean Expressions," in *Proceedings Custom Integrated Circuits Conference*, pp. 21 – 24, IEEE, Portland, Oregon, 4th – 7th May 1987.

79. K. Asada and J. Mavor, "Area Optimized MOS Circuit Generation using the circuit Synthesis Program MOSYN–2," in *Proceedings ESSCIRC 1986*, pp. 71 – 73, Delft, Netherlands, September 1986.

80. Kunihiro Asada and John Mavor, "MOSYN: A MOS Circuit Synthesis Program Employing 3–Way Decomposition and Reduction based on 7–value Logic," Internal Paper, Edinburgh University, 1986. Submitted to IEEE Trans. CAD.

81. Kunihiro Asada and John Mavor, "A Fully Automated MOS Symbolic Layout and Circuit Diagram Generation from Boolean Expressions," Internal Paper, Edinburgh University, August 1986. Submitted to IEEE Trans. CAD.

82. UW/NW VLSI Consortium, "RNL 4.2 User's Guide," User Manual, University of Washington.

83. Alan F. Murray, Testing and Self–Testing, Chapter 7 in *VLSI Signal Processing – A Bit–Serial Approach*, P. B. Denyer and D. Renshaw, Addison–Wesley, 1985.

84. Kathryn Booth, "The Design of Bit–Serial Twin–Pipe Multipliers using Single Phase CMOS," MSc Project Report: MSP 58, Edinburgh University, September 1987. Covers operation of the serial/parallel multipliers after correction of layout errors.

85. S. G. Smith, P. B. Denyer, D. Renshaw, K. Asada, K. P. Coplan, M. Keightley, and J. I. Mhar, "Full–Span Structural Compilation of DSP Hardware," in *Proceedings Int. Conf. on Acoustics, Speech & Signal Processing, ICASSP*, pp. 495 – 498, IEEE, Dallas, Texas, April 1987.

86. Stewart G. Smith, Michael Keightley, Peter B. Denyer, and Shigenori Nagara, "SECOND: Synthesis of Elementary Circuits on Demand," *Journal Solid State Circuits*, vol. 23, no. 3, pp. 722 – 727, IEEE, June 1988.

87. S. G. Smith, P. B. Denyer, M. Keightley, and S. Nagara, "SECOND: A VLSI Function Library Compiler," in *Digest of Technical Papers, ESSCIRC*, pp. 205 – 208, vde verlag gmbh., Bad Soden, Federal Rep. of Germany, 23rd – 25th September 1987.

88. S. Nagara, "SLICE assembly using "slicesyn"," Internal Report, Edinburgh University, Edinburgh, September 1987.

89. K. Coplan, "VX," User Manual, University of Edinburgh.

90. P. H. Winston and B. K. P. Horn, *LISP*, Addison–Wesley, 1981. Introduction to LISP as a language suitable for data–driven programming.

91. S. Nagara , "Assembly Strategy of a SLICE," Internal Report, Edinburgh University.

92. S. Nagara, S. G. Smith, P. B. Denyer, and M. Keightley, "New Channel Routing Algorithm for Bit–Slice Applications," *Electronics Letters*, vol. 23, no. 25, pp. 1389 – 1391, IEE, 3rd December 1987.

93. S. Y. Boar, C. Y. Tsay, and K. S. Chao, "A Weighted Path Algorithm for Channel Routing," in *Proc. of ISCAS '83*, pp. 1225 – 1228, IEEE, Newport Beach, CA, May 1983.

94. S. G. Smith and M. Keightley, "Method for Synthesis of MOSFET Logic Networks," *Electronics Letters*, vol. 23, no. 17, pp. 902 – 903, IEE, 13th August 1987.

95. Bob Mhar, "A Description of the Program NET," Internal Document, Edinburgh University, July 1987. A short description of routines used in program with the then current cell and block libraries.

96. UW/NW VLSI Consortium, "NETLIST User's Guide," User Manual, University of Washington.

97. Michael Keightley, "Area Comparison of a Multiplier Produced by *SOLO 1000* and *SECOND*," Internal Report, Edinburgh University, September 1987.

98. Michael Keightley, "Future Prospects for *SECOND*," Internal Report, Edinburgh University, October 1987.

99. Carver Mead and Martin Rem, "Minimum Propagation Delays in VLSI," *Journal Solid State Circuits*, vol. SC–17, no. 4, pp. 773 – 775, IEEE, August 1982. Correspodence Section.

100. Y. S. Cheung and S. C. Leung, "A Second Generation Silicon Compiler for Bit–Serial Signal Processing Architecture," in *Proceedings Int. Conf. on Acoustics, Speech & Signal Processing, ICASSP*, pp. 487 – 490, IEEE, Dallas, Texas, April 1987.

101. M. Glesner, J. Schuck, and H. Joepen, "A Flexible Silicon Compiler for Digital Signal Processing Circuits," in *Proceedings Int. Conf. on Computer Design, ICCD*, pp. 845 – 850, IEEE, 1984.

102. H. Joepen and M. Glesner, "Architecture Construction for a General Silicon Compiler System," in *Proceedings Int. Conf. on Computer Design, ICCD*, pp. 312 – 316, IEEE, 1985.

103. S. Nagara, "Performance estimation of mixed–tree dynamic logic in SECOND," Internal Report, Edinburgh University, October 1987.

104. Gaetano Borriello and Randy H. Katz, "DESIGN FRAMES: A New System Integration Methodology," in *1985 Chapel Hill Conference on VLSI*, 1985.

Appendix 1

CASE Work

Work done with Thorn EMI under the SERC CASE award scheme. The author is grateful to Thorn EMI for permission to include this work in my thesis.

A1.1. Function to architecture, through the design of a microcontroller.

Introduction

The controller designer project at Thorn EMI aimed to provide enhanced design capabilities of controllers with substantial user interaction. The tool allows design of physical appearance and functionality of the product under development. As the project now stood, design of the physical appearance is undertaken by use of two programs, an icon editor, used to design small portions of the display, and a level designer, used to piece together individual icons to form the complete form of the porduct being designed.

The functionality of the product can be specified in a graphical manner which produces a C program, containing calls to a library of routines suitable for interactive simulation of the product. The user needs minimal knowledge of C as the specifiable functions are provided by the software library allowing the straightforward use of functions which are appropriate to controllers. The finished program calls the icons which were previously input via the icon editor to give an interactive display of the device as it would work. User interaction is allowed by using the mouse to point and 'press' keyswitches on the controller.

The aim of this report is to describe how the functional specification provided by the user through the use of the function editor can be used to generate a hardware description (HDL) for a device capable of performing the function specified. The task is simplified by the size of the library of statements the user is allowed. The hardware description produced should be in a form suitable for simulation and use with an available compiler to generate a gate–array or standard–cell chip for the product.

The first section will look at three alternative approaches to the problem of producing hardware :–

1. Producing cells for each software statement.

2. Finite State Machine approach.

3. A simplified datapath, i.e. a microcontroller.

A brief description each approach is given. Afterwards the implementation used is considered.

Cell Based approach

Using this scheme, a cell is produced for each program statement. A graph is produced from the program indicating dataflow and cells are placed on the chip corresponding to each of the functions used in the program.

For each element in the dataflow graph a corresponding hardware element is produced. In this manner hardware which duplicates the function of the program specification can easily be produced. The scheme has the advantage of speed but uses large numbers of components, which is wasteful.

In this application speed is not needed because the fastest response need only keep up typically with keypresses on a keypad.

Finite State Machine Approach

This scheme generates a set of pla's which simulate a parser. The input is specified in the form of a BNF* which specifies the actions of the controller dependent on keypress sequences. The details of the method is considered in reference [A1]. Complex function calls can be specified by the input syntax.

The idea relates the keypress sequence and menu choices (with options) to a simple language. Programs such as LEX and YACC on unix systems can convert the BNF language syntax into a program. By using this syntax to specify the functionality of a controller, the means of transforming the specification into other forms exist. Here PLA's are constructed to simulate the specified controller 'language' syntax.

Datapath Approach

This approach builds a simple datapath capable of performing the operations needed for the control application. Mapping code to hardware is straight forward since the process is essentially a compilation. Code for an instruction rom is generated and the rom is implemented. The approach has the advantage that many parts of the datapath are fixed and do not need to be resynthesized at each invocation. The idea is similar to that considered in reference [A2] to produce silicon from a occam program.

A further advantage is gained since, intermediate code is essentially machine code, so that code for a commercial micro–controller, e.g Z8 or 8048, can be produced by as a by product of the process. With these advantages the datapath approach seems the most promising and is the one chosen for further detailing in this report.

A1.2. Requirements of Microcontroller

Before considering HDL generation we should bear in mind the application of the product. The applications in mind are for human interactive controllers. Here key presses are used to activate functions, list menus or other information on a display (typically liquid crystal) providing more user interaction. The shortest events the controller must respond to are likely to be the response on a keypad of a human user. A timer is a source of further events as it can provide useful timing and clock information to a user.

*BNF stands for Backus Naur Form, it is a symbology for specifying programming languages

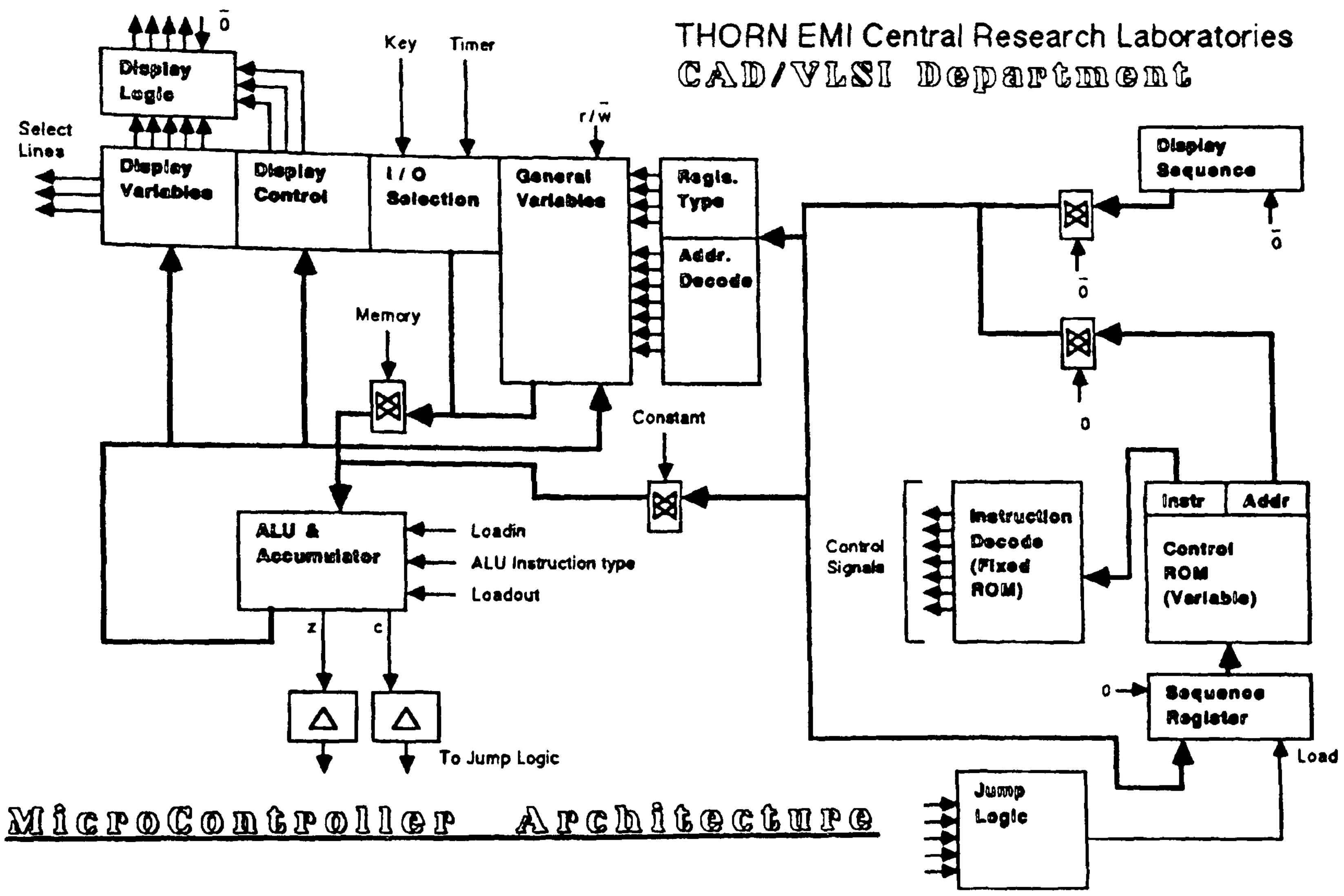
To satisfy these requirements key selection and timer access need to be provided. For liquid crystal display purposes a constant refreshing period is desirable. Since the type of operations required from a key press will be relatively simple a subroutine mechanism on the hardware is not essential, but would be a useful feature.

By using liquid crystal displays multiplexing of the is displays necessary, and if we were to update the display as dictated by the free use of the designers specification, then updates may become erratic on a liquid crystal display, leading to variations in display contrast.

This problem is overcome by refreshing a display element on alternate clock cycles. If the assembly code is used, then interrupts should be used to simulate this operation. This makes display updating and running of the code independent operations.

A1.3. Description of the HDL Generation

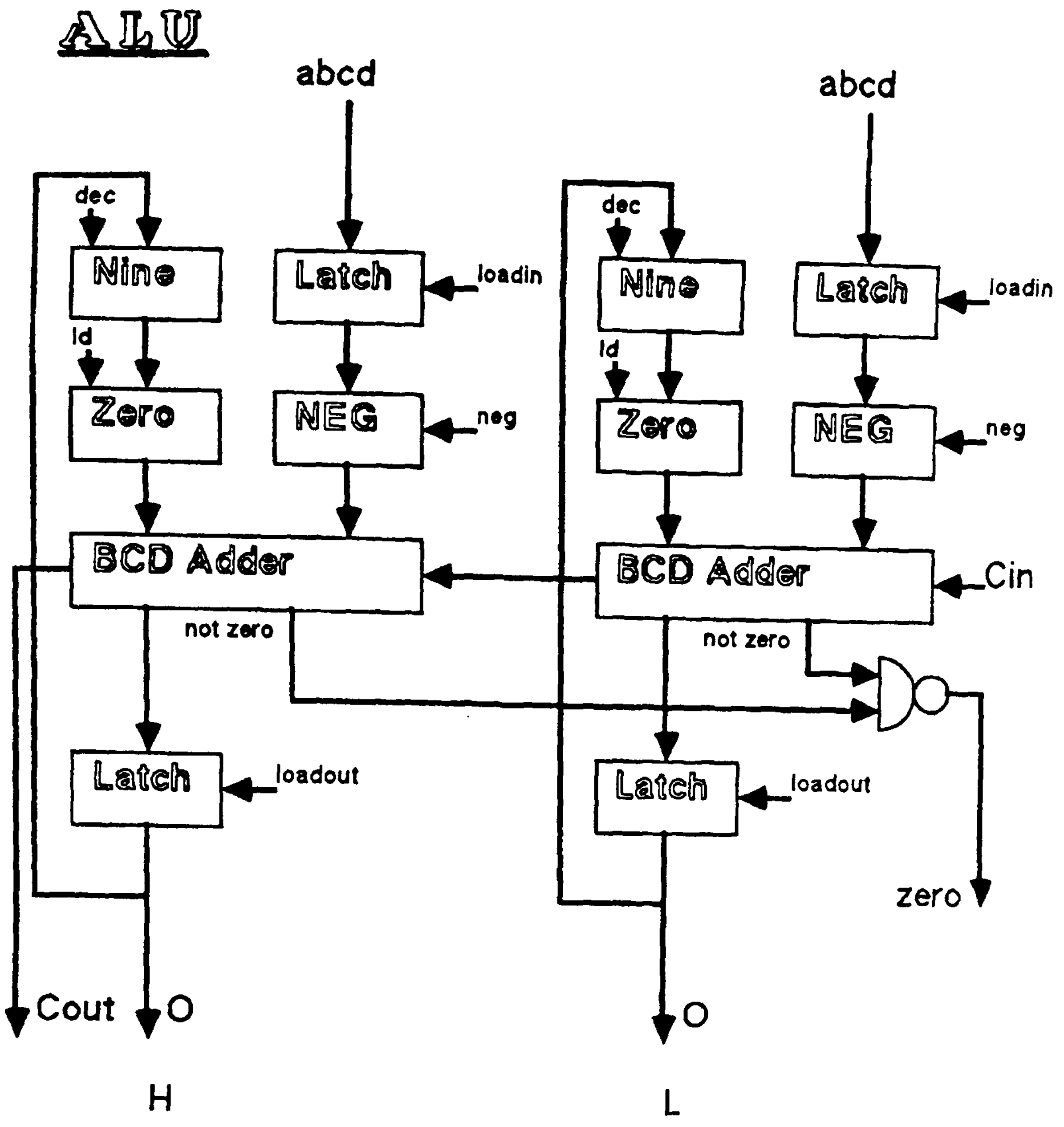
By using a simple microcontroller schema such as shown in Figure A1.1, HDL generation can be simplified.



Microcontroller Architecture

Figure A1.1

This has a fixed architecture for most of its part to carry out the functions required for reading keypresses and displaying output. The HDL for these sections will be fixed. Figure A1.2 shows an ALU which is suitable for use in this microcontroller. Parts of the microcontroller which variable, depending on the controllers application, e.g. ROM, numbers of registers, are straightforward to generate.



THORN EMI Central Research Laboratories
CAD/VLSI Department

ALU for Microcontroller

Figure A1.2

HDL for a device which performs the functions of the users specification can thus be generated. This HDL can be simulated to test for performance and optimisations made before using it with a vendor.

A typical set of instructions which could be executed for this simple microcontroller consist of :-

jp addr	pc = addr : jump

jpz addr	if zero then pc=addr else pc++ : jump if zero
jpnz addr	if zero then pc++ else pc=addr : jump if not zero
jpc addr	if carry then pc=addr else pc++ : jump if carry
jpnc addr	if carry then pc++ else pc=addr : jump if no carry
ldc const	Acc = const : load constant in accumulator
ldm addr	Acc = (addr) : load memory into accumulator
inc addr	Acc = (addr) + 1 : increment contents of memory leaving result in accumulator
dec addr	Acc = (addr) – 1 : decrement contents of memory leaving result in accumulator
add const	Acc = Acc + const : add constant to accumulator
add addr	Acc = Acc + (addr) : add memory to accumulator
sub const	Acc = Acc – const : subtract constant from accumulator
sub addr	Acc = Acc – (addr) : subtract memory from accumulator
cmp const	sets zero if const = Acc : compare constant with accumulator
cmp addr	sets zero if addr = Acc : compare memory with accumulator
sta addr	(addr) = Acc : store accumulator in memory

CodeSegments

The way code is translated from that generated by the users specification is essentially a substitution operation, for each operation or function statement, a sequence of assembly instructions which perform the same function is generated.

Note: pc represents the contents of the sequence register on the diagram, but pc for program counter is used because it's usage is more common

For instance, for the case statement :

```
switch(var){
        case 0:  code0 ; break ;
        case 1:  code1 ; break ;
...
}
```

would be transformed into :–

```
        ldm     var
case0:  cmp     0       ! Check if zero
        jpnz    case1
        code0
        jp      endcase
case1:  cmp     1       ! Check if one
        code1
        jpnz    case2
...
endcase: ..
```

The rest of the code is generated in a similar fashion. This code is used to program the control ROM in the microcontroller.

References

[A1] Custom v.l.s.i. circuits for man–machine interaction: an automatic design method, *K. R. DIMOND & J. BAGHERLI*, Jour. IERE, Vol. 56, No.2,pp. 79–85, Feb 1986.

[A2] Compiling occam into silicon, *David May & Catherine Keane*, Inmos Technical Note 23.

Appendix 2

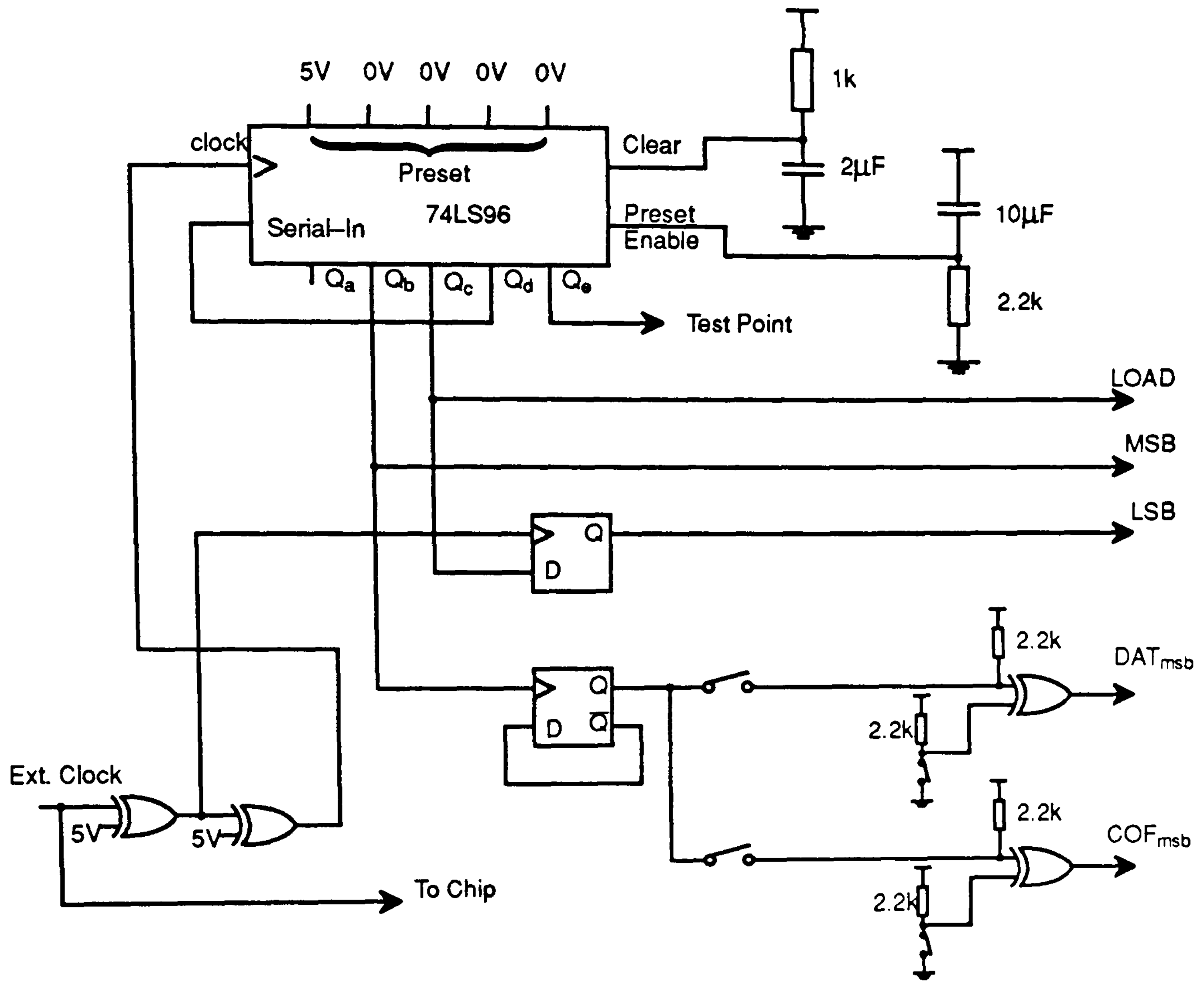
TTL Test Circuit

To test the 6 bit multiplier chips which were fabricated in order to evaluate our layout methodology and the novel circuit techniques at clock speeds greater than 10MHz, a TTL test circuit was constructed. This appendix describes this circuit. The board was designed to test the two pin compatible versions of the six bit multipliers laid out using a gate matrix scheme and the tiling scheme.

The circuit diagram of this test board is shown in Figure A2. The control signals, corresponding to Figure 3.15 to the circuit are produced from a four bit shift register which cycles a single logic one around continously. This provides the *mu* timed control signals. For the *lsb* signal which is a *pi* timed signal, the delay of one half clock cycle is provided by a D–type latch. The shift register powers up in the correct state for control signal generation from the reset provided on power up.

The data and coefficient are set up on Dual In Line (DIL) switches, which would provide unchanging inputs. To be able to at speed changing data inputs are required, for this reason the most significant bits of the data and coefficient bits are toggled on each successive cycle using the MSB comtrol signal.

Outputs can be observed on test points. An external TTL compatible clock must be supplied together with 5V power for the TTL circuit. Power to the device under test can be provided from a separate source if needed (for measuring the current to the chip).



Circuit Diagram of TTL test board

Figure A2.

There are a few observations of non optimality of the test board:

- the shift register is rather sensitive to glitches on the clock input, so care needs to be taken when connecting and disconnecting the clock. Another trouble is the clock cannot be applied whilst the circuit is powered up, since the register will not start in the correct state. To get the shift register in the correct state, the circuit is powered down and up again and the clock reapplied.

- the generated control signals only reach 4V in the high state since they are outputs of TTL gates. This has not been a problem with testing.

- the external clock feeds directly to the device under test, but it also feeds the TTL circuit, hence the external clock must be TTL compatible. Rise and fall times of the clock must be acceptable to the TTL circuit otherwise no control signals are generated.

– Unused pins to the device under test are left floating. This is not regarded as good design practice since the chip under test is a CMOS device, but no problems arose during tests.

Appendix 3

Routines used in ENGEN Code

Overview of Hierarchy

There are five levels of hierarchy recognised by the program these are :–

CELL	The lowest level which is completely user dependent. The names of vendors elements are incorporated in the cells allowing portability.
BLOCK	The blocks are constructed from cells unless the vendor has such an element within the supplied library, in which case the user can use this element by putting in the definition into an information file.
MODULE	These are elements constructed from blocks and/or cells. Modules differ from blocks by being of a complexity great enough that there is little likelihood of their inclusion into a vendors library. They are the largest fixed units, i.e. have no parameterisation.
PRIMITIVE	It is at the level of PRIMITIVE that there is parametrisation. A new primitive definition is constructed for each call of the primitive with a different set of parameters. So that names are not confused, a unique name is produced for primitives with different parameters by using the parameters in the constructed name. Since primitives are bulit up differently depending on the parameterisation, the writing of primitive generating code needs care.
CHIP	This is the top level as far as the program is concerned. The chip is the colection of primitive calls together with the calls to the PADS.

Routines used for writing primitives

This section gives a list with brief descriptions of the routines used for writing block, module, primitive generation code.

is_blockdefined(BLOCKNAME)

Used to check if the block is defined. A block needs only to be defined once so this routine is used for this check. BLOCKNAME is an integer defined in a header file.

block_define(BLOCKNAME)

Precedes the definition of the block BLOCKNAME.

block_signal_in(length, signame)

Specifies the inputs to a block in a block definition, where :-

length:	number of input signals.
signame:	array of integer node numbers corresponding to input signals.

block_signal_out(length, signame)

Specifies the outputs from a block in a block definition, where :-

length:	number of output signals.
signame:	array of integer node numbers corresponding to output signals.

int block_cellcall(name, length_in, sigs_in, length_out, sigs_out)

Used to call up a cell within a block definition, where :-

name	The name of the cell which is defined in a header file.
length_in	Number of input signals to cell.
length_out	Number of output signals from cell.
sigs_n	Array of integer input nodes to cell.
sigs_out	Array of integer output nodes from cell.

int block_blockcall(name, length_in, sigs_in, length_out, sigs_out)

Used to call up a block within a block definition, where :–

name	The name of the block.
length_in	Number of input signals to block.
length_out	Number of output signals from block.
sigs_in	Array of integer input nodes to block.
sigs_out	Array of integer output nodes from block.

block_enddefine(BLOCKNAME)

Ends the definition of the block BLOCKNAME.

block_comment(string)

Used for comments within a block.

is_moduledefined(moduletype)

Check if module has been defined.

module_define(moduletype)

Begins module definition.

module_signal_in(length, signame)

Declare input signals to module.

module_signal_out(length, signame)

Declare output signals from module.

int module_cellcall(name, length_in, sigs_in, length_out, sigs_out)

Call a cell from within a module.

int module_blockcall(name, length_in, sigs_in, length_out, sigs_out)

Call a block from within a module.

module_enddefine(name)

End the definition of a module.

module_comment(string)

Put a comment in the module instantiation.

is_primdefined(primname)

Check if primitive is defined.

prim_define(primname)

Begin definition of a primitive.

prim_signal_in(length, signame)

Declare input signals to primitive.

prim_signal_out(length, signame)

Declare output signals from primitive.

int prim_blockcall(name, length_in, sigs_in, length_out, sigs_out)

Call a block from within a primitive.

int prim_modulecall(name, length_in, sigs_in, length_out, sigs_out)

Call a module from within a primitive.

prim_enddefine(primname)

End the definition of a primitive.

prim_comment(string)

Put a comment within a primitive instantiation.

chip_signal_in(primname, signame, lengthin, nodesin)

Declare an input to the chip.

chip_signal_out(primname, signame, lengthout, nodesout)

Declare an output from the chip.

int chip_primcall(primname, length_in, sigs_in, length_out, sigs_out)

Call a primitive from within the chip.

chip_enddefine()

End the definition of the chip.

chip_comment(string)

Put a comment for the chip.

Appendix 4

Author's Publications

Full-Span Structural Compilation of DSP Hardware

S. G. Smith, P. B. Denyer, D. Renshaw, K. Asada,[†]
K. P. Coplan, M. Keightley and J. I. Mhar

University of Edinburgh
Department of Electrical Engineering
King's Buildings
Mayfield Rd.
Edinburgh EH9 3JL
Scotland

[†]University of Tokyo
Department of Electronics Engineering
7-3-1 Hongo
Bunkyo-Ku
Tokyo 113
Japan

Abstract

Most current silicon compilers rely on an underlying hardware function-library, thereby restricting the user to producing structural assemblies of library components. Although this gives the systems designer the power to implement function directly in silicon, it precludes the use of arbitrary cell functions that might be more suited to his application than the existing set. Moreover, should a new process become available, the cell-library must be redesigned and laid-out according to new rules, by a circuit and layout expert. An advance in structural silicon compilation, *SECOND*, is reported, which promises to impart flexibility and portability to cell-libraries. Through the automatic generation of layout from logical descriptions, cell-libraries may be maintained in process-independent form, and incorporate not only new components, but also new processes with ease. Implementations of user-specified chips may be in custom or semi-custom form - only physical assembly procedures differ.

1. Introduction

Major advances in technology for the fabrication of VLSI integrated circuits have resulted in the potential to realise complex real-time computational algorithms which were previously of only theoretical interest [1]. With the integration of hundreds of thousands of devices on a single silicon surface arises the considerable likelihood of implementation errors - a 'complexity crisis' confronts VLSI designers. To combat complexity, structured design styles and software tools [2] have emerged, culminating in the structural silicon compiler [1]. These compilers guarantee working parts from high-level structural descriptions, through automatic assembly of known-good modules by known-good techniques. Low-level errors are avoided by forcing the designer to follow this route.

2. Half-span structural compilation: *FIRST*

FIRST [1], developed at the University of Edinburgh, is a good example of a current structural silicon compiler. *FIRST* is targetted at digital signal processing applications, realising fixed-function bit-serial flowgraphs as solutions to the real-time computational problems encountered in this field. *FIRST* can generate chip designs in impressively short times *if* the design can be expressed in the language (i.e. using the restricted forms of function available), and *if* performance goals can be met by the underlying technology. The *FIRST* user is presented with a high-level language interface, via which he specifies signal processing systems architectures as flowgraphs of computational elements, partitioned functionally and physically. Underneath the *FIRST* user-interface is a *fixed* library of around twenty parameterisable bit-serial building-blocks, or 'primitives'. These in turn are composed of configurations of hand-crafted 'leaf-cells', bound to a *fixed* technology and maximum clocking rate. Computational and signal bandwidths may be matched through a variety of multiplexing techniques, but the throughput of each primitive element is *fixed*.

FIRST has a 'typed' hierarchy, as shown in Fig. 1(a). Here upper-case letters represent levels with physical significance. Lower-case levels are notional, used for design management only. The construction rules are simple:

1 Physical objects can only contain objects at lower levels, no lower than the next physical level.

2 Notional objects can contain objects at equal or lower levels, no lower than the next physical level.

Each physical level in the hierarchy has an associated assembly methodology. Assembly in *FIRST* is 'blind', i.e. assembly procedures at different physical levels of hierarchy are unconnected. In *FIRST*, SYSTEM assembly was never implemented, as commercial board layout packages abound. However, CHIP interfaces are arranged to guarantee communication at the function library clock rate, up to a predefined fan-out limit.

CHIPs are assembled by place-and-route software, using a simple floorplanner driven by a flattened netlist of PRIMITIVEs resulting from language compilation. The FIRST floorplanner treats PRIMITIVEs merely as rectangles whose behaviour at the interfaces is well-defined. PRIMITIVE interfaces are also arranged to guarantee communication at the function library clock rate, up to a predefined fan-out limit.

PRIMITIVEs are functional elements such as multipliers, serial memories etc., whose flexibility is greatly increased by parameterising many features. Each PRIMITIVE instance is assembled, according to its parameters, using ad hoc composition procedures and hand-crafted layout cells, details of which are hidden from the *FIRST* user. The production of both layout cells and composition procedures is time-consuming. In a sense, yesterday's chip-level problems have become today's cell-level problems. Through automation of cell synthesis and primitive composition, the same benefits may be brought to function library design as are currently brought by silicon compilers to chip design.

3. Full-span structural compilation: *SECOND*

The *FIRST* function library suffers from three instances of inflexibility - *functional, technological* and *operational*. Provision of flexibility in these three areas should greatly enhance the power of *FIRST* users to realise efficient solutions to real-time computational problems. This flexibility should not however re-impose the burden of complexity on the designer. Software tools must manage the increased flexibility, leaving the designer free to reap its benefits.

The development of design automation software to overcome the three inflexibilities is underway. The software, known as '*SECOND*' (*S*ynthesis of *E*lementary *C*ircuits *ON D*emand), allows a library of 'leaf-cells' to be specified in technology-free form (i.e. as small blocks of logic function), along with composition procedures for automatic assembly of functional primitives. The resulting library is *functionally, technologically* and *operationally* flexible.

A typed hierarchy like that of *FIRST* is useful, both for the designer's management of information, and for capturing physical partitioning information. To implement full-span structural compilation, we extend this concept down to the device level (Fig. 1(b)). A secondary software architecture underneath *FIRST*, as shown in Fig. 2, gives the sophisticated user increased power by allowing him direct access to the function library. *SECOND* allows the designer to specify custom primitives, as a hierarchical list of logic blocks and latches. Each physical level of hierarchy has its own assembler (floorplanner), but only at DEVICE level are ad hoc techniques employed. Logic function is only captured at BLOCK level - above BLOCK level we are only concerned with 'boxes and wires'. Logic BLOCKs (or subblocks) are described in terms of functional 'atoms' (AND, NOR etc.), *or* as binary cubes identifying the on-set, off-set and don't cares of the required function.

As well as customised logic blocks, the designer may call on either previously used blocks or pieces of hand-crafted layout. The former facility is analogous to software programmers using libraries of known-good high-level coded subroutines to improve productivity, while the latter facility is analogous to software programmers calling known-good machine-code subroutines to improve performance. Underlining the analogy, the latter comes with a loss of design portability. Latches may be treated as special BLOCKs themselves, rather than being incorporated in BLOCKs. Frequently-used objects such as these may benefit from inclusion in the microcell-library, at the small investment of an expert's time to hand-pack layout in the target technology. Failing this, they may be specified in identical manner to logic blocks. In order to maintain performance, we forbid the cascading of BLOCKs, i.e. BLOCK outputs connect to latches only.

The user of *FIRST* requires expertise in architectures, certainly at the systems level. Should his expertise extend down to the logic gate level, *SECOND* will allow extensions of the basic library in a manner analogous to the production and maintenance of the subroutine libraries produced by software programmers. If not, he will still have the original function library at his disposal, but with the important difference that he will be able to take advantage of improved silicon processing when the opportunity arises.

SECOND promises an effective route from *structural* intent into silicon. The route from *behavioural* intent into silicon - the 'holy grail' of systems designers [3] - is outwith the immediate scope of this project, however a behaviour-to-structure compiler, when it comes, will be able to call on the techniques proposed.

4. Logic and Circuit Synthesis

For the span of silicon compilation to move down from the function library level to the device level, two software tools are required. Logic synthesisers aid the designer in arriving at the optimal realisation of his logic specifications (Boolean equations, binary cubes etc.) Circuit synthesisers take the resulting netlists and produce layout in some tractable design style.

Such tools are beginning to appear in silicon compilation projects. Early logic synthesisers were targetted at simple and well-understood structures such as PLAs, state-machines etc. [4]. Most modern logic synthesisers use heuristics to seek near-optimal solutions, and are thus able to tackle large combinatorial problems. Logic synthesis is as efficient as manual design, but takes a fraction of the time. On the other hand automatic layout is normally less efficient than manual. We aim to combine the short design cycles and confidence of automatic layout with the area-efficiency and performance of structured manual layout.

4.1. The *MOS* suite of programs

Recently *MOSYN* [5] was developed at the University of Edinburgh, in collaboration with the University of Tokyo. This logic synthesis package has been expanded to include the automatic generation of compacted layout. The *MOSYN* user communicates his intent in the form of either truth table notation, binary cube notation or a concise logic-oriented language. In reply, *MOSYN* generates (in fairly brute-force manner) a set of transistor networks which realise the logic specification, returning parameters which help the user to select the optimal choice according to requirements. *MOSLAY* then produces symbolic gate-matrix [6] layout, *MOSCOM* compacts, and finally *MOSFLESH* produces CIF [2] for IC mask preparation. As the area cost of uncompacted gate matrix layout already rivals that of manual layout [6], compaction further ensures layout efficiency.

Only the final fleshing step is concerned with process-dependent information. *MOSFLESH* requires a library of 'microcells' (process-dependent layout elements which correspond to transistors, contacts etc.), but the task of producing these microcells from given design rules is trivial in comparison to redesigning an entire cell library. Fig. 3 shows the *MOSYN* route from input specification to static gate-matrix layout for the even-pipe, μ-timed, gated carry-block of Fig. 5(b) in [7].

4.2. *GATESYN* - a semi-custom BLOCK synthesiser

It is our wish that *SECOND* support not only custom, but semi-custom realisations of user-defined function. *MOSYN* is useful only in the former context. A tool is under development which produces optimised netlists from BLOCK specifications, targetted at semi-custom implementation media such as libraries of standard cells. *GATESYN* is to be compatible with *MOSYN* at the user interface.

5. Functional components of *SECOND*

The purpose of *SECOND* is to allow the specification of PRIMITIVEs, their functional verification, and their implementation in custom or semi-custom form in arbitrary technologies. To this end, a software architecture similar to that of *FIRST* is required (Fig. 2), but with some important differences.

5.1. PRIMITIVE specification (design capture)

Design capture in *FIRST* is via a single, high-level language interface. The *FIRST* user specifies systems as hierarchical lists of functional objects ('boxes and wires'), with PRIMITIVEs at the lowest level of hierarchy. The *SECOND* user, on the other hand, is involved in implementation details somewhat closer to technological interfaces. In particular, he has to support multiple representations of PRIMITIVEs. As PRIMITIVE specification above BLOCK level is still a matter of 'boxes and wires', a *FIRST*-like interface may be extended part of the way down the *SECOND* hierarchy. *VX* is a graphical design tool under development for just such a purpose.

A representational dichotomy occurs at BLOCK level, not just between custom and semi-custom forms, but between operational forms as well [7]. While the custom circuit techniques used in *SECOND* rely on conductance of transistor networks [8], semi-custom versions of logical operators are implemented as collections of logic gates or standard cells. Moreover, 5 representations of a BLOCK must be maintained if *SECOND* is to support both single-pipe and twin-pipe operational modes. These are the single-pipe version, and π- and μ-timed pairs of (reduced) twin-pipe functions [7].

The software architecture of Fig. 2 presents a single interface to the *SECOND* user for BLOCK specification. *PRESYN* takes as input the specification of a BLOCK in single-pipe form, but with identification of even-pipe and odd-pipe control signals [7]. *PRESYN* produces *MOSYN* language listings of the 5 BLOCK representations, which drive *MOSYN* or *GATESYN* to produce custom and semi-custom representations respectively.

PRESYN does not yet have sufficient descriptive power. It is our intention to use the SUBBLOCK level to specify variations between custom and semi-custom assembly. As semi-custom implementations cannot support the performance of custom, we may wish to relax the current restriction on cascading BLOCKs in semi-custom assembly, thus avoiding the duplication of logic function necessary in the custom case. An example of such duplication is in the CSAS [7] BLOCK example cited earlier (Fig. 3). Three AND-gates are functionally duplicated in SUBBLOCKs for both sum and carry formation - in semi-custom, they could be shared (as indeed could some gates in sum and carry SUBBLOCKs). Ideally *GATESYN* should minimise gate count in this manner, without user-intervention.

Parameterisation

As in *FIRST*, the function library may be made more flexible by including families of functionally 'soft' (i.e. programmable at compile-time) PRIMITIVEs such as multipliers and memories, instead of fixed instances of each. The family is identified by name and the individual by parameters.

Bit-serial hardware consists mainly of shift-register structures with built-in function [7]. These can be single-stage structures (e.g. adder), linear arrays (e.g. multiplier) or even two-dimensional arrays (e.g. word-size memory [1]). In the multiple-stage cases a PRIMITIVE consists of repetitions or cascades of arbitrary instances of hardware SLICEs whose internal content and external connectivity are *mostly* identical. Parameterisation of these aspects of SLICEs greatly increases the descriptive power of the design capture medium, allowing one SLICE description to service the entire PRIMITIVE family, thereby maximising designer efficiency. Two high-level programming constructs are useful here. The provision of *repeat* statements is mandatory, allowing variable-length arrays of SLICEs to be specified. The provision of *conditional* statements is also desirable, allowing minor differences between SLICE instantiations to be specified.

Unification

We anticipate the eventual unification of *FIRST* and *SECOND* into a single user interface. Although design capture would then be uniform across the full span of systems design, the less sophisticated user need not deploy *SECOND*, instead using *FIRST* with an existing function library. Once again, we compare this with the novice software programmer who calls standard library subroutines, and the expert who writes his own. Each uses the same design medium, and both benefit from an efficient compilation environment.

5.2. PRIMITIVE verification (behavioural simulation)

An important difference arises when we compare tools required for verification in *FIRST* and *SECOND*. *FIRST* revolves round a floorplanner and function library which guarantees performance of CHIPs and SYSTEMs. Responsibility for this guarantee rests in the first case with the creator of the floorplanning strategy, and in the second with the individual designer of each PRIMITIVE. The user is only responsible for the *functional* correctness of his design - to this end he is provided with a word-level, event driven simulator. The domain of the simulator is sufficiently abstract (and run-times short) to allow algorithmic exploration and verification.

SECOND lets the user don the mantle of responsibility not only for functional correctness of PRIMITIVEs, but also for their *performance*. Thus circuit simulation must be employed at the BLOCK level to ensure that the logical depth between latches never exceeds the target for the particular function library in question. However this is only required in marginal cases, which are highlighted by *MOSYN*'s statistical information output.

Behavioural simulation now spans the full design space. In *FIRST*, a further responsibility of the PRIMITIVE designer is to provide a high-level language coding of the primitive's word-level behaviour. This responsibility passes to the *SECOND* user, as does the responsibility at the lower level of describing logical behaviour of BLOCKs. The latter task is made simple by the fact that BLOCKs are necessarily small objects - logical simulators at higher levels may 'look-up' the verificational truth-tables generated by *MOSYN*. However at present the abstraction of PRIMITIVE behaviour to word-level remains a manual task, and is correspondingly a potential source of errors.

5.3. PRIMITIVE implementation (physical assembly)

At this stage the differences between custom and semi-custom implementations become clearly defined. While three physical hierarchical 'types' are provided for PRIMITIVE specification, these types are only significant in the custom case.

Semi-custom

Semi-custom design descriptions have no need for typed physical levels of hierarchy [9], although hierarchy still has an important part to play in efficient assembly of semi-custom chips. Here the responsibility of *SECOND* is to produce functionally correct netlists for manufacture - assembly is the responsibility of whichever semi-custom IC fabrication house is employed.

Custom

In the custom case, hierarchical typing allows the designer to dictate how the primitive is logically partitioned and physically assembled. In this way he is able to control the floorplanning of the primitive. At the lowest user level, logical BLOCKs are specified in *MOSYN* language, and *MOSYN* returns layout with no regard to external connectivity. SLICEs are assembled by placing blocks, and PRIMITIVEs by stacking slices. If genuine full-span compilation is to be achieved, it is imperative that these tasks be automated.

While the latter task is mostly the fairly straightforward abutment of rectangles, we perceive the former as the single most difficult barrier between half-span and full-span structural compilation. SLICE assembly entails optimal placement and routing (in metal 2) of BLOCKs. BLOCKs and latches may share wells and supplies in a 'back-to-back' manner. Efficient SLICE assembly will require that BLOCK assembly use external connectivity information, compromising the elegance of 'blind' typed assembly. Researchers in the *CATHEDRAL* project propose an interactive approach to module generation [10] - such techniques may find use here.

6. PRIMITIVE design example - a serial/parallel multiplier

To demonstrate the isolated use of *SECOND*, we report the design of a 6-bit serial/parallel (S/P) multiplier for use in a datapath environment. The operation of this multiplier is similar to the fractional S/P multiplier described in [11,7], except that interfaces are bit-parallel. 'Twin-pipe' architecture is employed to boost performance [7], using static logic trees and dynamic latches. Fig. 4(a) shows the layout of a partial slice produced by *SECOND*, and Fig. 4(b) the equivalent module laid out manually in the same gate-matrix style. The 6-bit part occupies an area of 2.2 mm^2 in 3-μm CMOS, and transistor count is 1200. Under test, the part was functional, although inefficiencies in circuit engineering limited the maximum clock rate to 12MHz (24MHz data rate). We anticipate at least a twofold performance increase in the next version, which uses dynamic logic throughout, pipelined buffering of broadcast signals, and no BLOCK cascading.

7. Conclusions

Silicon compilers have been proposed as the answer to the custom IC design 'bottleneck', by giving systems designers access to the skills of circuit and layout experts. However compilers based on function libraries merely move the bottleneck from chip level to cell level. Full-span compilation moves the bottleneck down to the device level. Circuit and layout expertise is only required in the creation of tiny 'microcells' such as transistors and contacts. In the semi-custom implementation route, such details are of no concern.

SECOND (*S*ynthesis of *E*lementary *C*ircuits *ON* *D*emand) is a design capture, simulation and layout package which will eventually extend *FIRST*'s typed hierarchy down to the device level. Instead of the current assembly of *FIRST* primitives using ad hoc composition procedures and hand-crafted cells, *SECOND* supports the logic-level capture of cells and their automatic synthesis and (eventually) placement. The user controls hardware partitioning through hierarchical 'typing' - each hierarchical type corresponds to a physical level with associated assembler/floorplanner. Only at the device level are process-dependent, ad hoc techniques employed.

While the area-cost of chip designs produced by full-span compilation may at times exceed that of manual layout, the reduction in design effort afforded more than compensates for this. In custom realisations, area-efficiency may be improved by identifying distinct levels of hardware hierarchy, and invoking appropriate assembly procedures for each. Semi-custom place and route software may also exploit intelligent partitioning. A sub-optimal design realisation in today's technology may then compare favourably with an efficient realisation in yesterday's, at a fraction of the cost.

References

1. P. B. Denyer and D. Renshaw, *VLSI Signal Processing - A Bit-Serial Approach*, Addison-Wesley (1985).
2. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley (1980).
3. P. B. Denyer, "System Compilation," pp. 3 - 13 in *VLSI Signal Processing, II*, ed. S. Y. Kung, R. E. Owen and J. G. Nash, IEEE Press (1986).
4. R. K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers (1984).
5. K. Asada and J. Mavor, "Area Optimised MOS Circuit Generation Using the Circuit Synthesis Program MOSYN-2," *Proc. ESSCIRC '86*, pp. 71 - 73 (Delft, September 1986).
6. A. D. Lopez and H.-F. S. Law, "A Dense Gate-Matrix Layout Method for MOS VLSI," *Trans. IEEE* ED-27 pp. 1671 - 1675 (August 1980).
7. S. G. Smith, M. S. McGregor, and P. B. Denyer, "Techniques to Increase the Computational Throughput of Bit-Serial Architectures," *these proceedings*, (1987).
8. M. S. McGregor, P. B. Denyer, and A. F. Murray, "A Single-Phase Clocking Scheme for CMOS VLSI," *Proc. 1987 VLSI Research Conf.*, (Palo Alto, CA, March 1987).
9. J. P. Gray, I. Buchanan, and P. S. Robertson, "Designing Gate Arrays Using a Silicon Compiler," *Proc. 19th DA Conf.*, pp. 377 - 383 (Las Vegas, 1982).
10. P. Six, I. Vandeweerd, and H. De Man, "An Interactive Environment for Creating Module Generators," *Proc. ESSCIRC'86*, pp. 65 - 67 (Delft, September 1986).
11. S. G. Smith and P. B. Denyer, "Serial/Parallel Architectures for Area-Efficient Vector Multiplication," *these proceedings*, (1987).

Figure 1: typed hierarchies

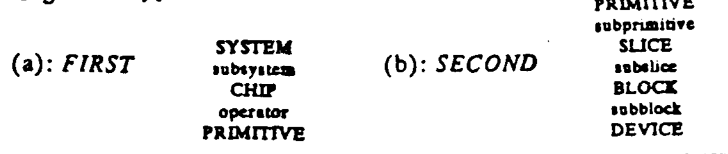
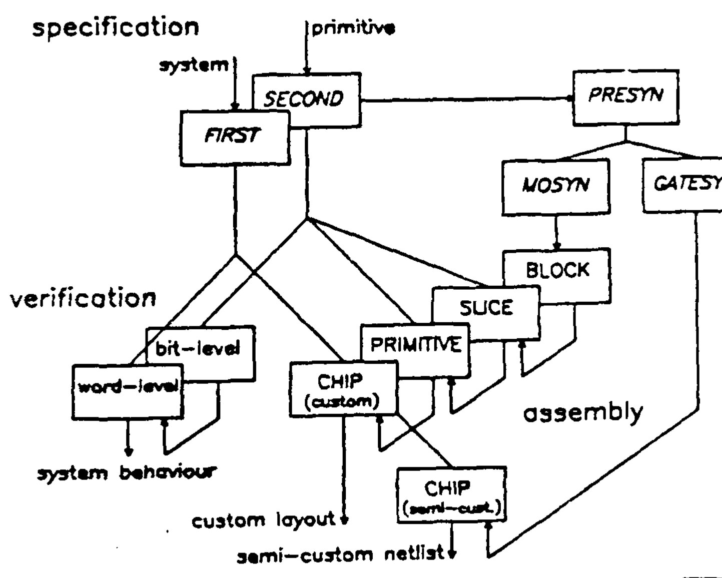


Figure 2: software architecture



```
NAME: lscar
EVEN: lsb
ODD: msb
FUNCTION: major(and(xor(msb,cof),dat),and(b,not(lsb)),and(c,not(lsb)))
```

Figure 3: *MOSYN* design route

```
*function lscarevmu
C1: major(and(cof,dat),and(b,not(lsb)),and(c,not(lsb)))
else C0
```

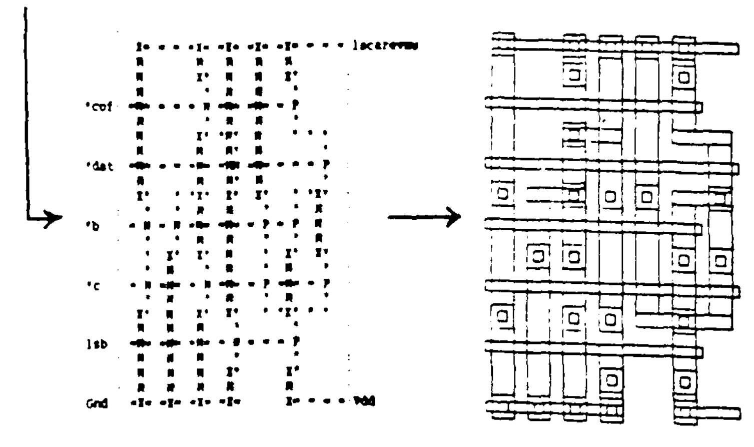
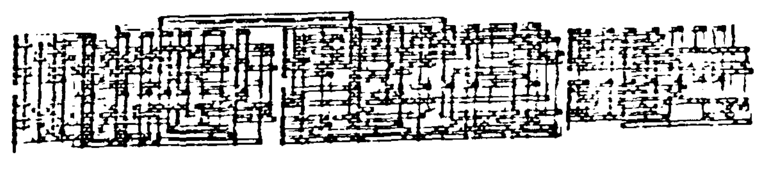
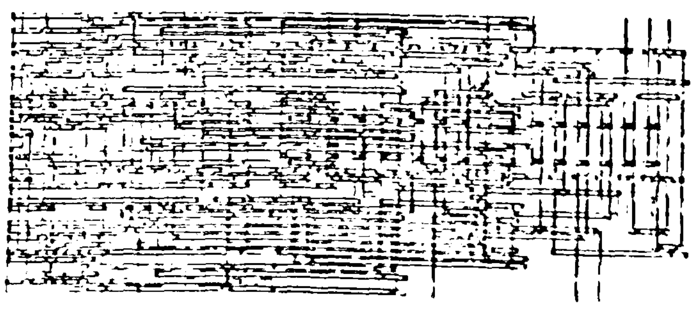


Figure 4: layout


(b) partial slice produced manually, using *MOSYN* blocks


(a) partial slice produced automatically