Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation.

Thomas Andrew Kean

Ph D

University of Edinburgh

1988

I declare that that this thesis has been composed by myself and that the work reported within is entirely my own except where clearly indicated otherwise in the text.

T. Kean.

Thomas Kean.

Abstract

At present there are two main paradigms for computation: interpretation of a data stream representing a program by a processing unit (software) and an interconnection of active logic elements (hardware). While both systems can (given reasonable definitions) be shown to be equivalent in terms of which functions they can compute they have radically different properties: hardware can potentially provide a much higher performance implementation of a single simple algorithm whereas software can implement a wide variety of extremely complex algorithms. Here we will consider a third paradigm for computation - configurable hardware - in which the interconnection between the active logic elements is dependant on a control store. This paradigm can potentially offer many of the performance advantages of hardware while retaining much of the flexibility of software.

This thesis examines the general properties of configurable systems and examples of previous designs in order to develop a new cellular-array architecture called Configurable Array Logic (CAL). The implementation of this architecture a related statically-programmed system, the Configurable Logic Array (CLA) in VLSI are discussed. The potential of the CAL system for implementation using Wafer Scale Integration is considered. The CAD system which would be required to allow algorithms expressed in normal programming languages to be implemented on the cellular architecture is discussed and the tools developed during the course of the project are covered. Four example designs using the system are presented: a digital stopwatch, a Data Encryption Standard (DES) encryptor, a unit to compute the 3-4 distance transform (which is used in image pattern matching) and a sketch of a system using configurable logic to implement cellular-automaton models for fluid-flow simulations. Finally, methods of extending the Configurable Logic architecture to allow more complex computations to be performed and other directions for further research are discussed.

Table of Contents

Chapter 1

Introduction.

At present there are two main methods of implementing algorithms: interpretation of a data stream representing a program by an active processing unit (software) and interconnection of active logic elements (hardware). In one case the computation performed is dependant on data stored in memory and in the other on the interconnection between a set of physical devices (transistors). Both paradigms can be shown (given reasonable definitions) to be essentially equivalent in terms of the functions they can compute (see, for example, [Savage76]). In this chapter we will examine the strengths and weaknesses of these paradigms and make the case for a third paradigm: Configurable Hardware in which the interconnection between active logic elements (and hence the function computed) is dependant on a control store.

1.1 Software.

The traditional computer architecture consists of a complex processor connected to a very large memory containing both program code and data. This architecture is extremely powerful and has allowed the development of today's complex information processing systems. The use of a single processor, however, results in a fundamental limitation on the performance of such computer systems since there are physical (light-speed) limitations on the speed of individual components. These

limitations are starting to be approached by current technologies and 'parallel' architectures with multiple processors which avoid this bottleneck by performing many operations at the same time are becoming important.

Let us consider the potential performance of a software implementation of an algorithm versus a hardware one: we can see many areas in which the hardware implementation is guaranteed to be faster and more efficient.

1. Instruction Fetch/Decode. In a hardware system the function is fixed so no time or area is wasted storing, fetching or decoding instructions.

2. Word Length. In a hardware implementation operations on single bits can use bit wide units and operations on 64 bit words 64 bit wide units: in software both operations would probably use 32 bit wide units resulting in an area penalty in the first case and a time penalty in the second.

3. Calculation Efficiency. In a hardware implementation calculations can be done using tailored hardware units which use the most efficient possible gate level implementation of a function: in software computations must use a series of predefined instructions. For example, a five way comparator could be implemented as a single hardware unit whereas a software system would usually need to use 4 two way comparison operations. Similarly, a general purpose software processor will often have a large amount of hardware to support instructions not required by the current computation (e.g. floating point units in a text processing application).

4. Memory Size. General purpose computers must provide memory systems large enough to support the algorithm with the worst case memory requirements: thus most algorithms will not make full use of the available memory. Hardware systems, on the other hand, can provide exactly the memory required by the current algorithm.

All of these problems are readily surmountable in a single processor system where it is economic to provide the extra resources: they are much more significant

in a parallel computer system where as well as providing as much memory and computation as possible in each processor node one also wishes to provide as many nodes as possible. There does not seem to be any reasonable way of getting round this problem and all current systems which provide a large number of conventional processing nodes are in the supercomputer price range.

1.2 Hardware.

In recent years many hardware implementations of important algorithms have been suggested (see, for example, the bibliography in [Wolfram86]). These take advantage of the factors mentioned above to provide huge speedups over conventional serial and parallel computers. Direct hardware implementation of realistic algorithms has only become feasible with the advent of Very Large Scale Integrated (VLSI) circuit technology and the recent advances in CAD tools which have made possible the Application Specific Integrated Circuit (ASIC). In this section, therefore, we will consider hardware as being synonomous with VLSI circuits.

Despite the enormous performance advantages of hardware over software these systems are not without problems.

1. ASIC's are hard to design well. To get maximum utilisation of the silicon area available a detailed knowledge of VLSI design techniques is required. Array structures designed at the mask level can sometimes provide huge savings.

2. ASIC's are expensive to build. ASIC's are very expensive in low volumes because of the high capital cost of processing equipment and the relatively large per-design cost caused by different handling requirements and mask-making. Even when direct-write electron-beam machines are used to eliminate mask-making large amounts of computer time are required to determine patterning information and the expensive electron-beam machine is tied up for a considerable time on each chip. This makes prototyping ASIC's costly.

3. ASIC's designs are hard to verify. The complexity of ASIC designs means that a huge amount of CPU time is necessary to simulate them at gate or transistor level on a serial computer. Catalogue part designs are usually prototyped rather than simulated at low level. Simulation times on conventional computers cannot be expected to improve with technology since, in general, the technology being used to develop a new ASIC (and hence its complexity) will be ahead of the technology used in the computer it is simulated on.

4. ASIC's are hard to test. When designing with catalogue parts the engineer expects the manufacturer to test the parts before they are shipped: he can assume that the parts he uses are all good. In most cases the problem of testing for chip production faults can be ignored.

   Every new ASIC design, on the other hand, must have a set of test vectors determined for it: this usually involves fault simulation to test that a particular set of test vectors exercises all internal nodes in the design. Determination of appropriate test vectors requires detailed knowledge of the design and so it must be done by the systems engineer: this is an additional and time consuming task. Automatic test pattern generation programs coupled with a suitable design methodology can ease this problem but they are not a complete solution. Such programs work with gate-level descriptions rather than mask level ones and so must still be used by the system designer. Efficient automatic test pattern generation is only possible for certain classes of sub-components, notably combinational logic blocks, the systems engineer must integrate automatically generated and manually generated patterns for subcomponents into a test set for the whole design.

5. ASIC's are hard to change. ASIC designs are hard to change because every change, however slight, involves redoing the verification steps, determining a new set of test vectors and producing a new design for fabrication. Incremental design systems could reduce the verification and test pattern generation overhead but there is a strong motivation to rerun these steps completely to make absolutely certain the design is correct before fabrication. Fabri-

cation itself will take several weeks and several thousand dollars. Changing a software design, on the other hand, is as easy as altering data within the computer's memory - although, in many cases test examples will also be run before the altered software is 'released' to outside users.

6. ASIC's are outside your control. With a catalogue part system the development engineer has all the parts he needs in store before he starts. If he is unsure about how a subsystem will behave he can breadboard it and see what happens. 'Suck it and See' is preferable to formal verification for the average systems engineer. There are sound reasons for this preference: you get results faster and the results are more meaningful. Building a system and watching it work gives much more confidence than looking at simulation results on a computer. When an ASIC is to be used as a subcomponent in a larger system the problems are worse because it may be impossible to get a good simulation model of the behaviour of the surrounding system.

The problems discussed above all stem from three basic properties of ASIC technology:

1. The customisation of ASIC's is static and involves expensive processing.

2. The structure of ASIC's is irregular and problem specific.

3. ASIC's have a small range of application.

It is important to note, however, that at the present state of VLSI technology none of the above problems are insuperable with proper CAD tools and ASIC technology has allowed the development of many extremely complex new products. The question is whether ASIC technology can continue to provide access to the considerable potential benefits of VLSI as device sizes continue to get smaller and we move towards Wafer Scale Integrated systems or whether a new design style will be required.

1.3 Configurable Hardware.

In this section we will consider a third paradigm for computation: configurable hardware. In this system the interconnection of active logical elements is determined by a control store. This potentially provides much of the performance of dedicated hardware with some of the flexibility of software.

1. Instruction Fetch/Decode. In a configurable hardware system the 'instruction' word which determines the interconnection of logic elements is normally loaded before the computation starts and remains constant throughout the computation. Thus, in most cases, there is no instruction fetch overhead during the computation phase.

2. Word Length / Calculation Efficiency. Like conventional hardware configurable hardware can use units tailored exactly to the computation required.

3. Pipelining. Configurable hardware systems can take advantage of pipelining and other parallelism available in the algorithm at the bit level often resulting in very large performance gains over conventional computers.

4. Reusable. Unlike traditional hardware configurable hardware can be reconfigured an unlimited number of times to compute different functions. This flexibility is not as great as that of software since the functions which can be computed will be heavily constrained by the amount of configurable hardware available and the input/output connections to it which are both fixed for a given system. The additional flexibility will, however, make it economic to provide relatively large amounts of configurable hardware within a computer system since the cost can be spread over many different applications.

Naturally, the 'hybrid' configurable hardware technology has also many disadvantages.

1. Speed. Configurable Hardware will always be considerably slower than conventional hardware because of delays introduced by the switching system. Often this drawback can be reduced since it will be economic to use better processing technology in the re-usable configurable system. Increased area can usually be traded for speed, for example, by providing extra parallelism.

2. Area. The overhead of the control store and switching circuits means that a large increase in area over conventional hardware is unavoidable. This overhead will be especially large in memories which are normally implemented as array structures designed at the mask level.

3. Flexibility. Configurable Hardware cannot approach the flexibility of conventional processors for complex computations. In general, it is most suitable for simple 'inner-loop' computations with high repitition counts which could be considered for implementation in conventional hardware.

Because of these drawbacks Configurable Hardware is not seen as a replacement for either conventional software or hardware but as an alternative system which could potentially implement some important applications more efficiently than either of the previous techniques. Three particular target application areas have been identified: any one of these would, in itself, justify the development of the Configurable Logic system.

1.3.1 EPLD Replacement.

At the moment there is a very large market for Electrically Programmable Logic Devices (EPLD's) in board level systems. These devices can simplify design, reduce package count (by replacing several small TTL devices) and provide extra flexibility by allowing changes in circuits to be made without changing Printed Circuit Board (PCB) layouts. As time progresses the fraction of systems implemented using programmable devices is increasing.

At present, this market is dominated by two-level AND/OR Programmable Logic Array (PLA) devices although some more general gate-array like architec-

n	*Num. Inputs*	*Num. Outputs*	*Single PLA*	*Many PLA's*
1	2	2	3	3
2	4	3	11	6
3	6	4	37	9
4	8	5	75	12

Table 1–1: Number of Product Terms vs Number of Inputs in Adder.

tures are emerging [Xilinx86]. Two level architectures are fundamentally inappropriate for implementing complex systems: as device densities increase all that you can do is increase the number of inputs, outputs and product terms available. Table 1–1 shows how the number of product terms (after minimisation) increases with the number of inputs in an adder when it is implemented as a chain of one bit adders and as a single n bit adder: clearly simply increasing the size of a single array is not a practical way of building large adders. Instead, the function must be partitioned into many different functional blocks: often designed using different techniques. This requires a more general programmable structure.

1.3.2 Prototype System for ASIC's.

Systems designed using the configurable architecture developed in this thesis can also be mapped efficiently to both semi-custom and full-custom silicon implementations. The dynamically programmable technology provides an ideal breadboarding system for designs which will eventually be implemented in silicon, allowing prototype designs to be tried out within the target system (although clock rates may have to be reduced). Directly implementing the function using dynamically programmed devices provides logic simulation at speeds which even the most expensive hardware accelerators cannot approach at relatively low cost.

1.3.3 Algorithm Machine.

This application is more of an exciting possibility than a proven capability. It will be shown (in Chapter 5) that a huge array of configurable cells can be built using wafer scale integration. Eight wafers of configurable chips designed using $1\mu m$ rules would give a 1M (2^{20}) cell array. The question of 'virtualising' such arrays to provide many potentially very large 'logical' arrays from a single physical array will be addressed in Chapter 8. What can be done with this amount of logic? Hopefully, systolic algorithms for applications like sorting, priority queues, searching and signal processing can be mapped onto the cells. This seems to make much more sense than having different special purpose processors for each application where hardware acceleration is required.

1.4 Overview of Thesis.

1. Chapter One. This chapter has covered the basic ideas behind configurable hardware and suggested target applications for the configurable architecture to be developed in the rest of this thesis.

2. Chapter Two. This chapter takes a 'high level' look at the resources available to implement configurable hardware and the different ways in which they can be utilised. Metrics are developed for measuring the efficiency of architectures and the limitations of the current two-level logic EPLD's are examined in detail. The effect on utilisation of increasing the generality of the basic cells in Programmable Logic Arrays is examined to motivate the search for more flexible architectures. Finally, a discussion of timing disciplines for use in programmable systems is presented.

3. Chapter Three. This chapter concentrates on the design of flexible configurable systems. A mapping of the design space is set out and new cellular architecture called Configurable Array Logic (CAL) is proposed. This map-

ping of the design space is then used to compare the present design with several important earlier designs.

4. Chapter Four. This chapter deals with the VLSI implementation of the cellular architecture presented in Chapter 3. The design of the control store and switching struture are developed in detail with several possible implementations being considered. Design and performance figures for leaf cells designed to implement CAL are given as well as possible improvements to the design. The way in which the design will scale with improving technology is discussed.

5. Chapter Five. This chapter will discuss specific implementations of the configurable logic architecture in VLSI. Firstly, a chip implemented using the leaf cell layouts of Chapter 4 is presented and the methods adopted to verify the design and algorithms to test fabricated chips are discussed. Secondly, a statically programmed version in which the cellular array is configured by the second metal mask (the CLA) is dealt with. Thirdly, a proposed extension of the VLSI implementation of CAL described in Chapter 4 to Wafer Scale Integration is covered.

6. Chapter Six. This chapter deals with CAD tools for the Configurable Logic system: the CAD system required to convert conventional high level language programs into configurable logic designs is mapped out and potential problems discussed. The purpose of this chapter is to show that algorithms developed for silicon compiler systems can readily be adapted for cellular systems and that it should be practical to integrate support for configurable logic into an existing silicon compiler environment. CAD tools developed during the course of the project to support the design examples in chapter 7 are also covered. Two major tools are treated in detail: a logic synthesis program and a channel router.

7. Chapter Seven. This chapter covers examples of the application of the configurable logic technology. Four examples are covered: a digital stopwatch

chip, a Data Encryption Standard (DES) [NBS77] encryptor, a unit to compute the 3-4 distance transform (which occurs in image pattern matching) and a design sketch for a system to implement cellular-automata models used in fluid-flow simulations. The first example is typical of the sort of application in which CAL could be used as an EPLD replacement and has been implemented using several different technologies. Area comparisons are given to show where configurable logic fits into the design space. The second example is a much larger system which requires many CAL chips built into a larger array at board level. This example illustrates the applicability of CAL to the prototyping of ASIC's or as an accelerator for conventional computers. The third example is intended to illustrate the use of configurable logic to accelerate 'inner-loop' computations in conventional algorithms. The fourth example illustrates how configurable logic can be used to implement a particular cellular-automata model: cellular-automata are becoming increasingly accepted by the physics community as models for a wide range of phenomena.

8. Chapter Eight. This chapter draws together the research presented in the preceding chapters and sets out possible directions for further research. A design for a 'virtualised' configurable logic system is outlined.

Chapter 2

Principles of Configurable Systems.

This chapter develops a gate level model of configurable systems. Based on this model some fundamental measures of the efficiency of configurable systems are presented.

We then turn our attention to the central design goal of our architecture: the efficient realisation of ASIC size systems and the implications of the generality of this goal on implementations are examined.

The chapter concludes with a discussion of timing disciplines for such programmable systems. This is an area which has received almost no attention in the past.

2.1 Circuit Model.

At this point it is advantageous to have a more formal model of circuits. This will allow fundamental bounds on the cost of implementation to be found. The question of at what level to model the circuit is a difficult one. There are two main options: gate level and switch (transistor) level. The second option is nearer the implementation and reflects the fact that the configuration circuitry and the logic circuitry are both made up of switches. It has, however, some important disadvantages.

1. Convenience. Most systems engineers prefer to design using logic gates rather than switches and all configurable systems to date operate at this level of abstraction. This has the consequence that the functions realised are over $\{0,1\}$ instead of $\{0,1,Z\}$. At this level of abstraction wires have a direction and a single source.

2. Directions of Signal Flow. When designing configurable systems it is important to take account of the fact that the transistors which implement the switches are far from being ideal components. It is reasonable to impose an additional rule that no path may go through more than a small number of switches without buffering to restore logic levels. The introduction of buffering forces a direction on wires, this can be ignored in a gate level abstraction but the buffers would need to be explicitly modelled at the switch level.

3. Implementation Tricks. As we will see in Chapters 3 and 4 there are many important low-level techniques which can be used to increase the efficiency of implementations. At first glance it would seem that switch level models would be a better choice to describe such circuit structures. This is not the case since they often rely on additional properties of physical implementations such as resistance and capacitance and require analogue rather than digital techniques to model properly. Simple equivalent gate level circuits can be found and it is much easier to develop a clean and self-consistent gate level model of general hardware than it is to develop a switch level one.

2.1.1 Definition of Circuit.

We are considering a circuit as being an interconnection of gates, without loss of generality we can assume that all the gates have two inputs and one output. These gates implement a function g chosen from a basis set $\Omega = \{g : \{0,1\}^2 \rightarrow \{0,1\}\}$.

A c gate circuit C can be modelled as a graph $C = (G, W)$ where the gate set G is an enumerated set of two input gates (vertices) and the wire set W is a set

of connections (edges). The gate functions g are chosen from the basis set Ω so $G = \{(i, g_i) : g_i \in \Omega, 0 \leq i < c\}$.

The wire set W is more complicated: each member of this set is a tuple (source,sink) representing a connection between gate outputs (or circuit inputs) and a gate input or circuit output. A source may take part in several such tuples (since a gate output will often drive several gate inputs) but a sink may take part in at most one tuple (since connecting several sources together is not allowed in a gate level model). We can, therefore, specify the wire set W by enumerating the sources for every gate input. If we label each gate using the integers from 1 to c then we can label all possible sources by using $1 \ldots c$ to represent the output of the corresponding gate. Similarly the sinks can be labelled using integers from $1 \ldots 2c + 1$ (since there are two inputs per gate). Thus $W = \{(i, j) : 0 \leq i \leq c, 1 \leq j < 2c\}$, where no two members of W have the same sink j. Unconnected sinks or sinks which are circuit inputs have no corresponding entry in W. An alternative way to model the wire set W would be as an array containing the sources for each sink with a special *nil* value for unconnected sinks.

There are several points worth making about this model of circuits.

1. Inputs and Outputs. This model does not represent circuit inputs and outputs in any way: circuits are treated as interconnections of gates. Some gate inputs and outputs may be unconnected. The reason for not treating inputs and outputs explicitly is to provide a notion of c gate circuit: if the n inputs and m outputs were modelled then we would need to think about (c,n,m) circuits and the size of the circuit could be increased arbitrarily just by adding more feed through connections between inputs and outputs. A model which dealt with inputs and outputs explicitly might allow a more complete analysis but would complicate the mathematics. Note that we do not require the graph to be connected, i.e. a circuit can be composed of several unrelated subcircuits.

2. Wire Model. The definition of edge nodes prevents gate outputs being con-

nected together. A single wire in the physical circuit connecting a gate output to many gate inputs is modelled by many edges in the graph.

3. Two Input Gates. In physical circuits gates with a fairly arbitrary number of inputs are possible, however the area of the gate increases linearly with the number of inputs. An s input gate can always be replaced by a network of 2 input gates so there is a reasonable algorithm for converting physical circuits into our model. Counting two input gates is a better cost metric than counting gates with arbitrary numbers of inputs since it directly reflects the area required.

Equivalence of Circuits. In simple terms two circuits are equivalent if the same gate functions are connected in the same pattern. Specifying this formally is complicated by the fact that the labellings of gates used in the circuits may well be different: for example a NAND gate labelled 3 in one circuit may be labelled 16 in the other and a wire from the output of that gate could be (3,25) in the first circuit and (16,30) in the second. If the two circuits are equivalent then if we replace all the 16's if the second circuit with 3's and so on for the other gate numbers then we will end up with two identically labelled graphs. This will *not* happen if the two circuits are not equivalent - for example if there was an extra connection from the NAND gate in the second circuit.

More formally, two circuits $C_1 = (G_1, W_1)$ and $C_2 = (G_2, W_2)$ are equivalent if and only if there exists a one to one and onto mapping μ between gate labels in C_1 and gate labels in C_2 such that $G_1 = G_2'$ and $W_1 = W_2'$ where $G_2' = \{(\mu(i), g_i)\}$ and $W_2' = \{(\mu(i), \mu(j div 2) + (j mod 2))\}$.

Containment of Circuits. A circuit is contained within another circuit if it is equivalent to part of the second circuit. More formally, circuit C_2 is contained in circuit C_1 if and only if there is a one to one (but not necessarily onto) mapping between labels in C_2 and labels in C_1 such that $G_2' \subseteq G_1$ and $W_2' \subseteq W_1$ where $G_2' = \{(\mu(i), g_i)\}$ and $W_2' = \{(\mu(i), \mu(j div 2) + (j mod 2))\}$.

Subtraction of Circuits. Based on the idea of containment above we can define what it means to subtract a subcircuit from a larger circuit, informally we delete all the gates and wires corresponding to the subtracted circuit. If C_2 is contained in C_1 then $C_1 - C_2 = (G_1 - G_2', W_1 - W_2')$ where $G_2' = \{\mu(i), g_i\}$ and $W_2' = \{(\mu(i), \mu(j div 2) + (j mod 2))\}$.

Composition of Circuits. A circuit C is composed of circuits $C_1, C_2, C_3, \ldots, C_n$ if and only if $((((C - C_1) - C_2) - C_3) - \ldots - C_n) = (\emptyset, \emptyset)$.

2.1.2 Definition of Configurable Circuit.

A configurable circuit PC is modelled as the quadruple (G,S,W,V) where G is a set of gates as above S is an enumerated set of selectors (or multiplexors) $s_r(x_0, \ldots, x_n, i) = x_i, 0 \leq i < n, n = 2^r$,$W$ is a set of wires as above except that selectors can now be used as sources or sinks and V is a control vector which selects the sources chosen by the selectors. Every bit of V is connected to exactly one selector control input.

Definition of Emulation. A configurable circuit PC with control vector V emulates a circuit C if when all the selectors in PC are removed and all edges in PC with a selector as source are replaced by edges with the selected input as source then the circuit C is contained in the resulting circuit. Note that the circuits do not have to be equivalent because not all the resources in PC need be used.

2.1.3 Definition of c-CLM.

A c gate Configurable Logic Module c-CLM is a programmable circuit which, according to a control vector V will emulate any c gate circuit as defined above.

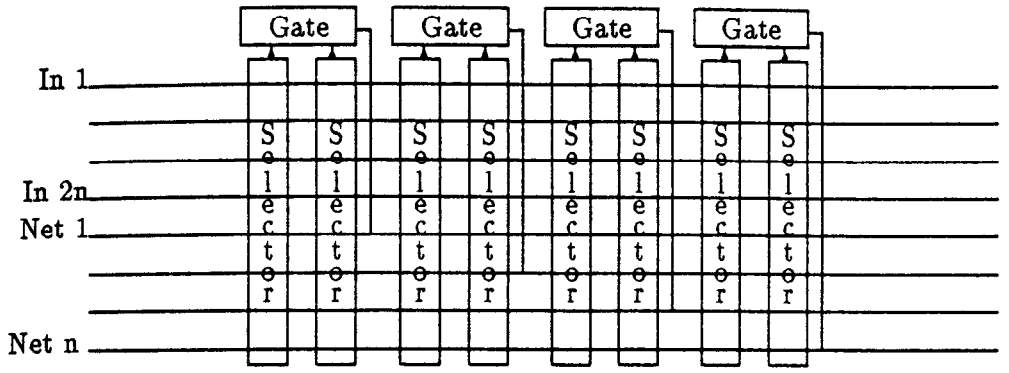


Figure 2–1: Architecture which meets lower bound.

2.1.4 Lower Bounds.

Gates. Obviously, a c-CLM must contain at least c gates.

Inputs and Outputs. A c-CLM must contain at least $2c$ inputs and c outputs (consider a circuit where every gate computed a function of two different input variables).

Memory. If we count the number of c gate circuits then we can easily determine the minimum amount of memory needed to control a c-CLM.

From the definition of circuit it follows that each sink is connected to at most one source: there are $2c + 1$ possible sinks ($2c$ gate inputs or unconnected) and $c + 1$ possible sources (c gate outputs or unconnected) so there are $(2c + 1)^{c+1}$ possible interconnections. We must also consider the number of possible functions which each gate can perform $|\Omega|$, giving a total number of circuits $|\Omega|(2c + 1)^{c+1}$.

This implies that at least

$$\lg(|\Omega|(2c + 1)^{c+1}) = \lg(|\Omega|) + (c + 1)(\lg(2c + 1)) = O(c \lg(c))$$

bits of RAM will be required to control a c-CLM.

Surprisingly, perhaps, there is an architecture which obtains this lower bound (Figure 2–1). This architecture is not practically interesting for circuits with large numbers of gates since it requires global wiring for each net.

2.1.5 Circuits and Functions.

The function performed by a circuit can be represented as $f : \{0,1\}^n \rightarrow \{0,1\}^m$, where n is the number of inputs and m is the number of outputs. In order to decide what function a circuit computes one must identify input and output terminals on the circuit. There are now $2c + m + 1$ (unconnected outputs are still possible) sinks and $c+n+1$ possible sources giving $|\Omega|(2c+m+1)^{c+n+1}$ possibilities. Many of these possibilities may compute the same function. Determining whether two distinct circuits compute the same function is a difficult problem.

It may seem more natural to count distinct functions rather than distinct circuits but there are good reasons for our choice.

1. Different Realisations are Useful. In a general configurable device it is desirable to be able to implement the same function in several different ways: for example there are many possible implementations of the addition function and designers will want to select one with appropriate delay and area for their system. This is to be contrasted with previous architectures where the configurable system was intended to implement one block of random logic: here multiple realisations of a given function would be considered an efficiency overhead.

2. Number of Boolean Functions. There are 2^{2^n} combinational functions $f : \{0,1\}^n \rightarrow \{0,1\}$ and $(2^m)^{2^n}$ functions $f : \{0,1\}^n \rightarrow \{0,1\}^m$. The explosive growth in the number of functions means that *almost all* n input combinational functions require $O(2^n)$ gates to implement [Savage76,Shannon49]. This implies that defining CLM's in terms of computing all functions of n variables would place circuits of very different complexity in the same cat-

egory. Counting gates gives a much better indication of device size than counting input variables.

3. Sequential Circuits. The general interconnection of gates permitted by our circuit model allows latches to be built, therefore some of the circuit permutations correspond to sequential rather than combinational logic. This makes counting functions impossible (since such a circuit could emulate a Turing machine with sequential logic providing the 'tape').

2.2 Efficiency.

The most important measure of the cost of a one bit wide configurable circuit is the number of bits of control store. There are two important reasons for this: firstly the control store is likely to account for a high percentage of the total area and secondly the size of the data-path will normally scale almost exactly with the number of bits of control store. This is because the width of the control vector means that there is no possibility of implementing the two separately with interconnecting wires: they must be intermingled in the same structure so that each bit of control store is situated close to the switches it controls.

2.2.1 Efficiency of Configurable Circuits.

We wish to define efficiency in terms of the 'value' we get from each bit of the control store: for maximum efficiency we would expect each permutation of the control vector V to result in a different circuit being emulated. We can formalise the idea of different circuits as follows.

Definition of Distinct. Two permutations of a configurable circuit with control vectors V_1 and V_2 are distinct if and only if the two emulated circuits C_1 and C_2 are not equivalent.

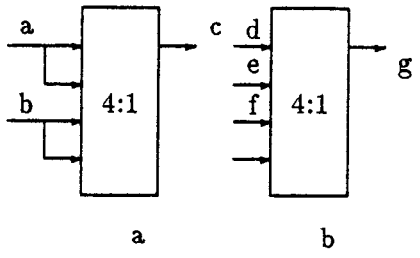


Figure 2–2: Wasted Signals.

Thus, we define the efficiency E of a configurable circuit which realises n distinct circuits with a control vector of length $|V|$ as

$$E = \frac{\lg(n)}{|V|}.$$

This gives $E = 1$ when every permutation of V has a corresponding distinct realisable circuit.

The purpose of this metric is to detect wasted control signals, figure 2–2 shows examples of wasted signals. In figure 2–2a control signals are wasted by having the same data signal on more than one input of a multiplexor: a 2:1 multiplexor with only 1 control line could realise as many distinct configurations. In figure 2–2b one of the permutations is wasted by having an unconnected input to the multiplexor.

This definition of efficiency is intuitive but to make it useful for comparing different configurable logic designs some method of computing it efficiently must be available. The obvious method of generating all possible control signal permutations and comparing the resulting circuits with each other is totally impractical for larger circuits. We need some method of taking advantage of the regularity present in actual designs to reduce the amount of computation. We would also like an efficiency figure for an architecture rather than an efficiency for a particular fixed size circuit.

Let us consider programmable circuit PC as being composed of identical programmable subcircuits or 'cells' $pc_0, \ldots, pc_{n-1}$. Let $n(pc_i)$ be the number of circuits

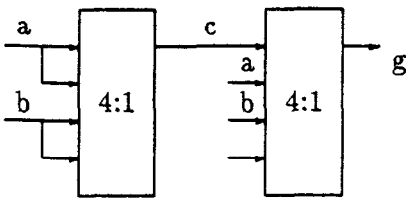


Figure 2–3: Composition Can Reduce Efficiency.

emulated by pc_i then the efficiency of PC is

$$E(PC) \leq \frac{\sum \lg(n(pc_i))}{\sum |V_i|}.$$

Discussion. Consider the number of control signals to PC, $\sum_{i=0}^{n-1} |V_i|$ (if control signals were shared between pc_i and pc_j then we would consider them to be a single subcircuit). It remains to show that the number of emulatable circuits is not increased by any fixed composition. When the number of emulatable circuits was calculated for the subcircuits $pc_1, \ldots, pc_n$ then the input terminals were unknown and assumed to be on different nets. Here, some of them will be on known nets possibly the same as other terminals of the subcircuit: therefore the efficiency could be smaller but since there are no more unknowns it cannot be larger. Figure 2–3 shows an example of a circuit whose efficiency is less than that of its subcircuits.

Architectural Efficiency. The efficiency of a regular array architecture $E(A)$ can be defined as the limit of array efficiency as the array size tends to infinity (figure 2–4). Thus

$$E(1) \geq E(4) \geq E(9) \geq \ldots \geq E(A).$$

Cell Efficiency. In many of the practical architectures we will be concerned with the programmable structure is composed of an array of identical cells. In this case the efficiency of the basic cell $E(1)$ in figure 2–4 is a very important parameter since it is relatively easy to calculate and is a reasonable predictor of architectural efficiency.

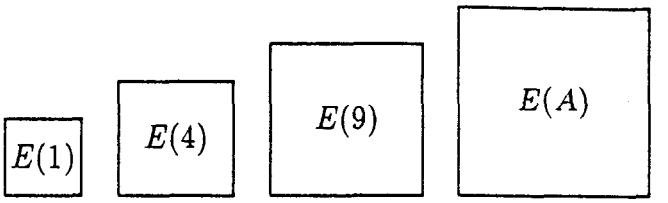


Figure 2–4: Architectural Efficiency.

2.2.2 Functional Efficiency.

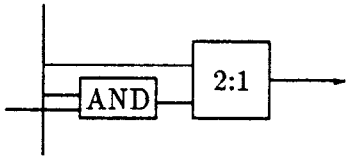
Many programmable logic architectures have been designed to implement irregular combinational functions such as those found in control stores. For such architectures it is more appropriate to measure efficiency in terms of the number of distinct functions realisable f as

$$E = \frac{\lg(f)}{|V|}.$$

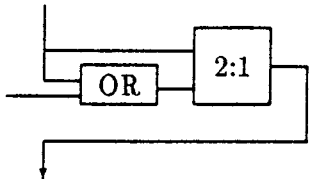
These architectures usually have limited connection patterns and choices of gate types so counting the number of functions implemented is sometimes feasible.

2.2.3 Utilisation.

As well as having a measure of how efficient a programmable architecture is in its utilisation of control store it is tempting to measure the overhead in implementing a particular circuit on a certain programmable structure. This sort of figure is very easy to obtain since one just needs to add up all the resources used and divide by the resources provided. Utilisation figures can be very misleading since bad designs which use unnecessary resources make the programmable architecture look better by increasing utilisation. Similarly, poor designs of programmable architectures will often force roundabout wiring paths (using more selectors) and have a higher utilisation than good designs. For these reasons utilisation figures must be treated with care. One important utilisation statistic is the cell utilisation which is the ratio of cells used in a particular function to cells provided.



AND Plane.



OR Plane.

Figure 2–5: FPLA Cell Designs.

2.3 Design Goals.

Before we go any further it is important to have a clear idea of what range of application we want our configurable system to have. The central design aim is to produce a configurable system which can emulate any system realisable using an ASIC with only a 'small' increase in area and reduction in speed. Let us consider exactly what this entails: the functionality of the ASIC can be described as a function $f : \{0,1\}^n \rightarrow \{0,1\}^m$. This immediately suggests an implementation: any boolean function can be implemented using a Programmable Logic Array (PLA) and there is a dynamically configurable equivalent technology the Field Programmable Logic Array (FPLA). We will consider FPLA's to be composed of an array of two kinds of small flexible cell illustrated in figure 2–5. The first kind makes up the AND plane and the second the OR plane. This is a fairly good model for RAM based programmable PLA's but is significantly different from the normal fuse based FPLA's where wires are undirected and very high fan in gates are available.

One important property of nearly all ASIC designs is that they are composed of many functional blocks. Assume there are s blocks, each of these also realises a function

$$g_i : \{0,1\}^{n_i} \rightarrow \{0,1\}^{m_i}, 0 \leq i < s.$$

Each of these functional blocks may in turn be composed of other functional blocks and the ASIC function f is a hierarchical composition of these subfunctions.

Let us consider the implications of this for an FPLA implementation. Define the area function $A(f)$ of an FPLA implementation of f to be the number of cells used, this gives $A(f) = p(f)(n + m)$ where $p(f)$ is the number of product terms (because the FPLA is a rectangular array (n+m) cells wide and p(f) cells high). Let us assume that the subfunctions g_i are disjoint (i.e. share no inputs or outputs). The area for separate implementation is

$$\sum_{i=0}^{s-1}((n_{g_i} + m_{g_i})p(g_i))$$

whereas the area for a single logic block is

$$A(f) = (\sum_{i=0}^{s-1}(n_{g_i} + m_{g_i}))(\sum_{i=0}^{s-1}p(g_i))$$

This is illustrated in figure 2–6. To simplify the analysis let us assume the subfunctions are the same size so $n_{g_i} + m_{g_i} = a$ and $p(g_i) = b$ for all $g_i, 0 \leq i < s$. Then the area of a separate implementation over the area of a composite implementation is

$$\frac{\sum A(g_i)}{A(f)} = \frac{s(ab)}{(sa)(sb)} = \frac{1}{s}$$

Thus the area of a separate implementation increases linearly with the number of functions s whereas the area of a composite implementation increases as s^2 [Wood79]. It should be noted that this is an average case analysis: it is easy to find values which are better or worse.

Obviously, the analysis is incomplete: several other important factors must be considered.

1. Overhead Circuitry. Each PLA will have some associated overhead circuits at the edges: the costs for this scale better with the single array implementation.

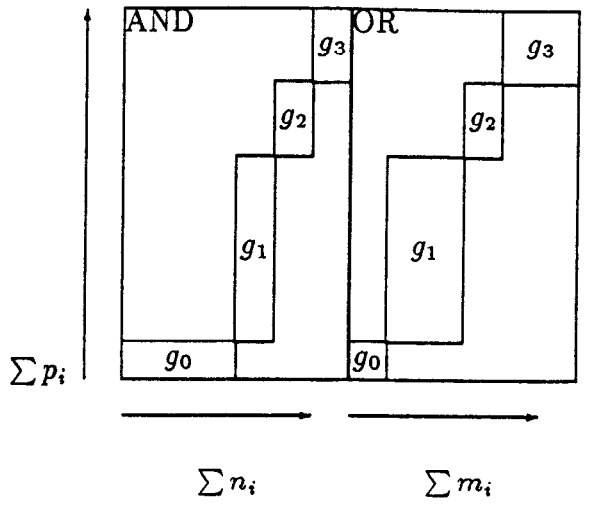


Figure 2–6: Area of Disjoint Functions Implemented in Single Array.

2. Functions are not completely disjoint. In an actual ASIC the functions implemented by separate subunits will not be completely disjoint: they may use outputs from other subunits as inputs or share inputs with other subunits. Subfunctions can themselves be decomposed hierarchically.

3. Logic Minimization. Logic minimization algorithms have large computational complexity and much better results can be obtained for small subunits than for a single large logic block.

4. Concurrency. Decomposing the function into many logic blocks provides the potential for concurrent operation of blocks and pipelining between them. The delay through a small block is likely to be lower than the delay through a large one after some critical size is exceeded.

These considerations imply that if we wish to build ASIC size systems it is essential to be able to implement subfunctions using separate blocks. This has important consequences for the structure of such a device.

2.3.1 Device Structure.

We have seen that in order to build a reconfigurable chip capable of efficiently implementing whole systems several logic blocks will have to be available on the chip. Some means of connecting these together to form the complete function will be required. We have a set of s subfunctions $g_i, 0 \leq i < s$ where s is unknown, $p(g_i)$ is unknown m_i is unknown and n_i is unknown. In the most general case we cannot say that $n_i \leq n$ and $m_i \leq m$ since the subfunctions will not be totally disjoint: it is very common to find subunits in an ASIC with more inputs and outputs than the chip itself. A first attempt at a solution would be to fix a maximum value of s,S and put S identical logic blocks with fixed numbers of inputs,outputs and product terms N,M and P on a chip with some interconnection structure between them. If we split our function f into subfunctions which do not exceed any of these limits then all is well although a potentially large amount of space will be wasted by choosing worst case values of N, M, P (figure 2–7). We can calculate the expected utilisation of each subunit for the case $P = 2^N$: if we wish to ensure that all N input functions can be realised then this is the decision we would have to take. FPLA's with all 2^N product terms provided are known as PROM's.

Utilisation of PROM. The average number of cells required in a PROM is

$$\frac{(\sum_{n=1}^{N}(\sum_{m=1}^{M}(n+m)2^n))}{MN} = \frac{2^N(M+2N-1)-M+1}{N}$$

(because there are MN possible sizes of PROM which fit into a $2^N \times (M+N)$ array with $2^N(M+N)$ cells) giving an average utilisation of

$$U = \frac{2^N(M+2N-1)-M+1}{N(N+M)2^N} = \frac{1}{N} + \frac{N-1}{N(N+M)} - \frac{M-1}{N2^N(N+M)} \simeq \frac{1}{N} + \frac{1}{N+M}.$$

This assumes that subfunctions with n inputs require 2^n product terms i.e. they are implemented using smaller PROMS. We can see that as N gets large on average only about $\frac{1}{N}$ of the PROM will be used: even this is a considerable overestimate because most n input subfunctions will require much less that 2^n product terms. The conclusion is obvious: large PROMS should be avoided at all costs. This suggests that we examine the consequences of having subfunctions which overflow fixed size subunits.

Consequences of Overflow. Suppose that one of the subfunctions g_i does not fit - there are several ways in which this could happen.

1. Not Enough Outputs. This situation can be solved by using more than one array and duplicating the inputs and product terms. This imposes a considerable overhead.

2. Not Enough Inputs. Suppose there were t inputs too many. Then the problem can be solved using a $t+1$ input decoder and 2^{t+1} subunits. The decoder could be implemented using another subunit but for even moderate size t there is a good chance of requiring too many outputs (since the number of outputs grows as 2^t). The overhead here is the decoder and the extra wiring.

3. Not Enough Product Terms. This is the most likely situation. Many functions have large numbers of inputs and outputs but in the worst case 2^n product terms will be required. This situation can be worked round by using two arrays duplicating the inputs and OR'ing the outputs. The overhead here is in the wiring necessary to move inputs and sub-outputs around the chip.

4. Not enough Subunits. In this case multiple subfunctions can be combined into a single array. This has the overhead discussed in the last section.

We can see, therefore, that the provision of many smaller fixed sized PLA units to implement subfunctions does not provide the required efficiency to implement large ASIC sized systems of a programmable structure.

2.3.2 Utilisation and Cell Generality.

In this section we will investigate the effect of increasing the generality of the basic cell on the utilisation of the array.

Row Folding. Suppose you had a single cell that could either form part of an AND plane or part of an OR plane. If you built your S subunits from these cells

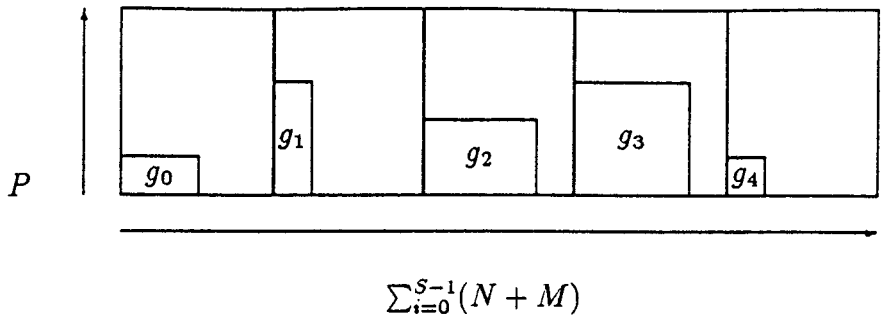


Figure 2–7: Stage 0.

you would no longer have to fix N and M but only $N + M$. Suppose further that all S subunits were abutting each other. With such flexible cells the point at which one array ended and another started would no longer need to be fixed. We would not have to fix N,M and P for each g but only $\sum_{i=0}^{S-1}(N + M)$ and p (figure 2–8).

Column Folding. If we assume that the cells can accept input and output signals from both directions then the arrangement shown in figure 2–9 would be possible. Here two subunits can share cells used in product terms. If the subunits are arranged so that units with large numbers of product terms share with units with small numbers the area is greatly reduced. Column folding can also be useful when only one function is being implemented, especially when row folding is also available.

Rotational Symmetry. The cell still does not cope well with situations where the aspect ratio of some subfunctions forces large wastage of product terms. If the cells can also form arrays in the orthogonal direction, then we can use layouts like figure 2–10 saving many cells. We now have a situation where only the dimensions x and y of an array of flexible cells need be specified in advance.

Internal Routing. The architecture presented up till now still has one flaw: all routing happens at the edge of the chip. Signals will have to travel quite large distances between subunits around the edge and the requirement that all routing

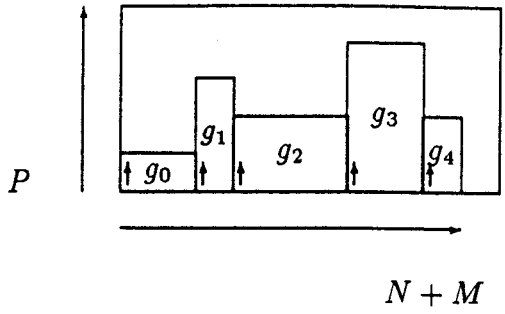


Figure 2–8: Row Folding.

is done at the chip edge does not allow full use of the central area. The cells have been given fairly flexible routing abilities to support the previous steps and with some slight modifications could implement the wiring areas themselves. This allows layouts such as figure 2–11.

General Gates. The final step would be to note that because we now have a flexible routing structure in each cell we are no longer limited to rigid PLA like structures. Gate networks with multiple levels of logic are possible. As well as possibly reducing gate count general networks of gates allow more control over subunit aspect ratio. PLA structures have a fixed aspect ratio which as n becomes large approaches a long thin rectangle (because the number of product terms grows very quickly with the number of inputs). Control over aspect ratio of subunits can result in more efficient floorplans. Even if we keep to PLA like structures the more flexible cells give us much more freedom about where inputs and outputs enter and leave the array, which can reduce the size of wiring areas. If we add some additional logic functions as well as AND and OR to each cell then in some important cases the number of gates required to implement a function can be dramatically reduced: for example the parity function requires $O(n)$ gates if XOR is available but $O(n^2)$ using only AND,OR and INVERT [Savage76].

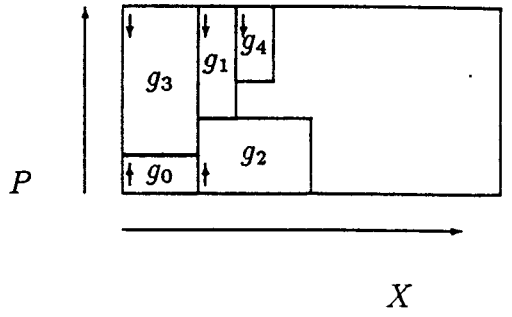


Figure 2–9: Column Folding.

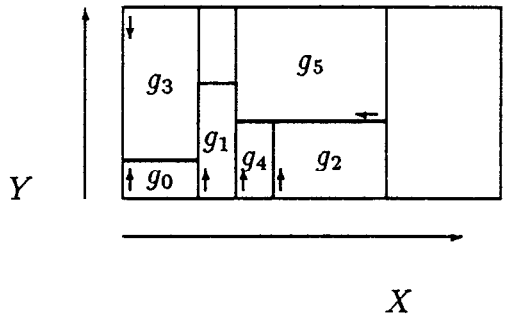


Figure 2–10: Rotational Symmetry.

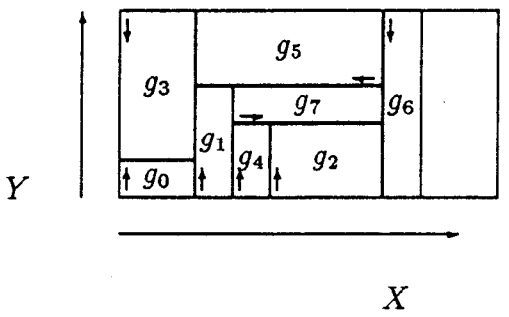


Figure 2–11: Internal Routing.

Stage.	RAM Cells.
FPLA	1 AND 1 OR
Row Folding	2
Column Folding	4
Rotational Symmetry	7
Internal Routing	10
Functional Symmetry	12
General Function	16

Table 2–1: RAM Cell Cost of Increased Generality.

2.3.3 Costs and Benefits.

All this flexibility in the cells has a cost. Table 2–1 shows how the number of bits of RAM required to control the basic cell increases with its generality through these stages. The area of one of these cells will be k times as great as the area of the simple FPLA cell. The increase k will be quite large - maybe as much as 20 - but it is *constant.* The overheads with using a fixed cell vary with the size of the problem and the particular function being implemented. It is easy to find particular examples where this tradeoff is highly advantageous. In the previous section we showed that the ratio of the area of a single array implementation to a multiple array implementation of a function with multiple disjoint subfunctions was about s where s was the number of sub-functions. This analysis did not take into account the fact that the subfunctions themselves have to be packed into a single large rectangular area. This packing cannot, in general, be done with 100% efficiency but in order to approach this efficiency it is necessary to have flexibility of subunit shape,placement and orientation. In this section we have shown that the potential benefit of separate subunits can be realised by using an array of flexible cells.

2.4 Timing Discipline.

There are three possible timing disciplines worth considering for a configurable system.

2.4.1 Self-Timed.

In this discipline [Seitz80] each cell generates explicit 'go' and 'done' signals which are routed in parallel with the data signals. This discipline is very attractive since it relieves the user of most of the timing problems associated with logic design. It has two major failings.

1. Complexity. The cell complexity is much higher than that of cells using the other two timing methodologies. Cell complexity increases design time and crucially in an array structure layout efficiency (i.e. not only will more logic be required because the function is more complex but it will be implemented less area intensively).

2. Size. Because 'go' and 'done' signals also have to be transported about individual cells are many times larger than cells with the same function implemented in one of the other disciplines. As well as the wiring overhead fairly large 'concensus' gates are required where wires split at multiplexors and where functions are being computed to provide the extra control. A trial layout of a cell design in the initial stages of this project suggested an area overhead of about 3 times for a self timed system over a system with no explicit timing support. This overhead makes the use of self-timed cells unattractive at this time.

2.4.2 Clocked.

In this discipline the actions of individual cells are synchronised to a system clock. If the cells are to be used as 'smart-memory' within a computer system this may

be the technique of choice since it would allow a microprocessor to read and write to internal nodes of a circuit implemented by the cells without disturbing a computation being performed by the cell array. Note that these clocks do not relieve the user of responsibility for clocking the implemented circuit to remove timing hazards within it.

2.4.3 Pipelined.

This is an extension of the clocked design methodology and offers some exciting possibilities. If all transfers within and between cells are synchronised to a single fast system clock and additional store is provided at each selection position then the system can be pipelined at a very low level. The additional store does not imply an unacceptable overhead because storage is available on the gate capacitance of the buffering inverters: only an additional pass-transistor is required.

2.4.4 Unsynchronised.

In this case the user is provided only with logic gates and takes full responsibility for timing himself. This flexibility is necessary for cells which are to be used as PLD's. All previous cell designs have taken this approach.

2.5 Summary.

In this chapter we have examined the design of configurable systems. We have introduced fundamental efficiency measures for comparing such systems. We have also seen the effects of the design specification on the structure of such systems and in particular that large arrays of flexible cells seem to be a very suitable structure. We have described some possible timing schemes for such systems. In the next chapter we will develop this further look at previous designs and introduce the cell design which will be used in the rest of the thesis.

Chapter 3

Design of Configurable Systems.

In the last chapter some fundamental principles of configurable systems were developed and it was shown that if general systems were to be emulated efficiently then large numbers of small flexible units were more suitable than small numbers of large units. This chapter will concentrate on the detailed design of such structures.

At present we are thinking about a configurable system as a set of gates with an associated switching system and control store. We must now consider how to lay out such a system on a planar silicon chip. We introduce the concept of *function unit*: this is a physical realisation of a gate which can implement any function chosen from a basis set Ω. We will call a function unit and 'closely associated' routing circuitry a *cell*. As a first step we will assume that the cells of the system are arranged in a grid as shown in (figure 3–1). This assumption is justified by the practicalities of silicon design - it is almost forced if we wish to use high density arrays for the control memory. At this stage we are making no assumptions about what is contained in each cell (switching and gates) or that all cells are the same. There are three important parameters to consider.

1. Connectivity between Cells.

2. Connectivity within Cells.

3. Function Unit Capabilities.

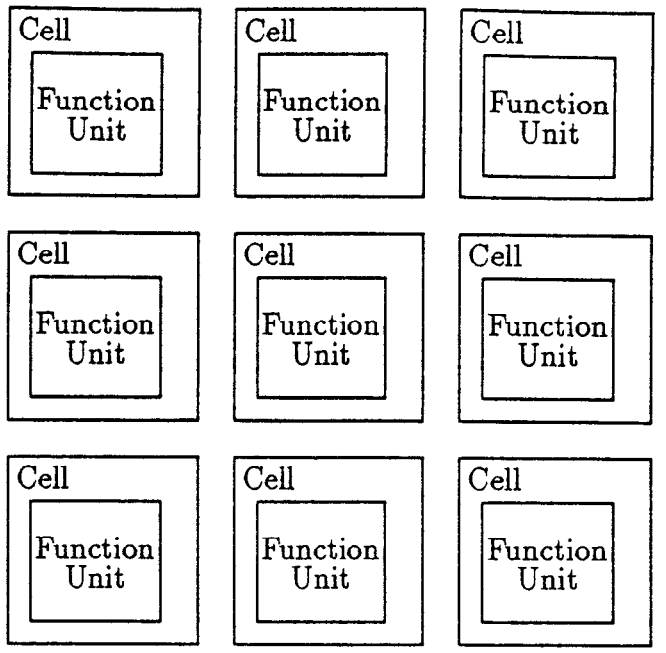


Figure 3–1: Basic Grid of Cells.

It may seem that the distinction between within-cell and between-cell connectivity is arbitrary: after all if we draw the cell boxes big enough then all the routing area is within cells. The distinction is made from the user's perspective where it makes sense to consider long wires as being outside the cells and only interacting with those cells which wish to use them as inputs and outputs.

3.1 Connectivity Between Cells.

This section will present the important design parameters in this area: these will be used in later sections for comparison between different configurable systems.

Directions of Data Flow. If we are interested in emulating general circuits then we can specify the following desirable capability: there should be a path through routing multiplexors between any function unit output and the input of any other (not necessarily adjacent) function unit. We must consider the minimum

number of directions of data flow required to support this capability since each additional direction can be expected to incur overhead in wires and switches.

If we wrap round wires at the edge of the array to form a torus then two orthogonal directions suffice. Obviously, this method often requires wires longer than the manhattan distance between the two cells. If signals running on grid diagonals are available then three directions suffice, again this involves longer than necessary wires. Without diagonals or wrap-round connections then all four grid directions are required: this method can always route in the manhattan distance. Additional directions of data flow potentially reduce wire length. Wiring schemes such as hypercubes which make the array of cells logically n dimensional can be viewed as providing additional directions of data flow.

Regularity. In systems composed of identical cells it is desirable that if the cell box is expanded to include inter-cell connections as well then all the cell boxes can still be identical. This would exclude, for example, systems with global wires along every second column of cells. This property simplifies the user's conceptual model and reduces the complexity of the chip design.

Flexibility. This refers to whether the inter-cell communication forms a fixed pattern or whether it has programmable selectors. Normally, fixed inter-cell connection implies flexible intra-cell connection and vice versa.

Wire Kinds. There are two distinct kinds of connection commonly used.

1. Wired Logic. If a wire can be driven from several sources then the wire can be used to compute a logic function. This can provide a very efficient means of building a very high fan-in gate. It is especially useful in arrays intended to implement a single logic block. The wire can be long and bidirectional because it is unsegmented. This technique relies either on resistance of a pull up device or capacitance in the wire to store charge. In CMOS technology only NOR gates can sensibly be built using wired logic. Inputs and outputs

can easily be inverted so by applying de Morgan's laws you can emulate OR, AND and NAND as well. There is an equivalent circuit in our gate level model in which the single very high fan in wire is replaced by many fan-in two gates.

There are fundamental limits on the size of such wired gates since the RC factor of the wire increases with its length and there occurs a critical point where increasing the size of driver transistors is no longer the best way to reduce the delay [Mead80]. Instead, intermediate buffering must be used: effectively breaking the gate up into several smaller ones.

2. Normal Wires. By normal wires we mean wires in which the control system makes sure that only one source drives the wire at a time. If the wire is long and has n possible connections to gate outputs then this poses a problem. In theory only $\lceil \lg(n) \rceil$ bits of RAM are necessary to select the source for the wire: however the distribution of the sources in space means that either n bits of RAM are required, each bit situated adjacent to the switch it controls, or at least $\lceil \lg(n) \rceil$ control wires must travel with the data wire. The solution which uses additional RAM cells is more area efficient when sources are widely separated.

3.1.1 Nearest Neighbour Connections.

In this case wires connect only adjacent cells, figure 3–2 shows some examples of nearest neighbour schemes. Here all the wires are short and thus sources for every wire are available close to each other allowing maximum efficiency in source selection. Nearest neighbour connections suffer from long propagation delays when long wires are built up from many small segments.

3.1.2 Nearest Neighbour with Longer Busses.

In this design additional longer wires are added to reduce the propagation delay problem. These longer wires can be divided into three classes.

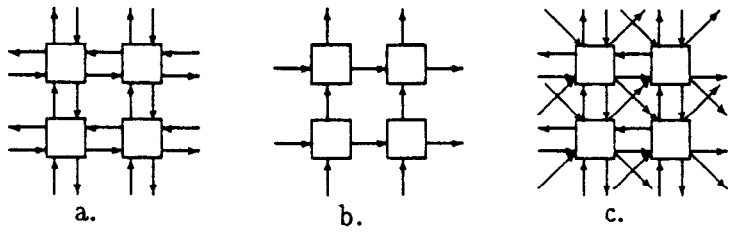


Figure 3–2: Nearest Neighbour Connection Schemes.

1. Global Inputs and Outputs. These are signals which are fed to every cell in the array or which can be driven by every cell in the array. Global input signals are cheap and can be useful for critical signals such as clocks. Global output signals are very expensive: every cell needs one bit of RAM per signal to specify whether it is to drive it (there are far too many cells for routing the control signals alongside the wire to be practical). Global output signals can be useful, for example to allow the output of internal cells to be monitored for testing/debugging purposes. Usually wired logic is used for global output signals: this is the most efficient implementation and the logic function is often useful in itself.

2. Array Crossing. These are signals which cross from one side of an array to the other. Normally these signals run along rows or columns of the array but Shoup [Shoup70] has pointed out advantages to running them at an angle. These signals can be very useful in medium sized arrays where the whole array is devoted to one function: they become less useful as the array size is increased. Again input signals are cheap but output signals are expensive. Normally, wired logic is used for output signals.

3. Medium Range. These are signals which travel further than just a cell's nearest neighbour but not across the whole array. These signals are moderately expensive because of the source selection problem but their greatest fault is that they make the array less regular and complicate the process of

design using the array. The justification for including such signals would be the decrease in propagation delay caused by having longer unswitched paths.

3.1.3 Wiring Channels.

In this section we will consider a switching system based on the traditional pattern of function units and wiring channels used in gate arrays (figure 3–3). Figure 3–4 shows a 'close up' of the channel: wires travel horizontally on one 'layer' and vertically on another. At the 'X' points wires can be broken or contacts can be placed to link the two layers.

The advantages of this structure in silicon come from the fact that the 'switches' are free. They represent only the presence or absence of a wire or a contact hole in an insulating layer: closed switches contribute minimal additional resistance to the circuit. It is possible, therefore, to have long, possibly bidirectional, wires for low propagation delay or lots of short wires. Bidirectionality is only useful if switches as well as logic gates can be implemented by the function unit.

The goal of the switches in a programmable system is to implement the most general connectivity possible between the function unit inputs and outputs while still being reasonably small. There is no reason to suppose that a direct copy of the gate array wiring channel structure is the best way to achieve this. Firstly, let us consider the most general system. We will consider bidirectional wires: the number of connection patterns is the number of different ways of putting n objects into n sets some of which can be empty. The number of ways of putting n objects into p sets none of which are empty is Stirling's second coefficient $\{{n \atop p}\}$ [Knuth73], so the figure of interest is $\sum_{k=1}^{n}\{{n \atop k}\}$. We can calculate this numerically for the 4 by 5 wire example in the figure - this is of interest because switching boxes of this size are used in one of the commercial EPLD's - the result is about 6.82×10^{10}. This is just less than 2^{36} so, theoretically, as few as 36 bits of RAM could suffice to control it. Larger switch matrices are likely to require internal buffering to overcome the combined resistance of series transistor switches.

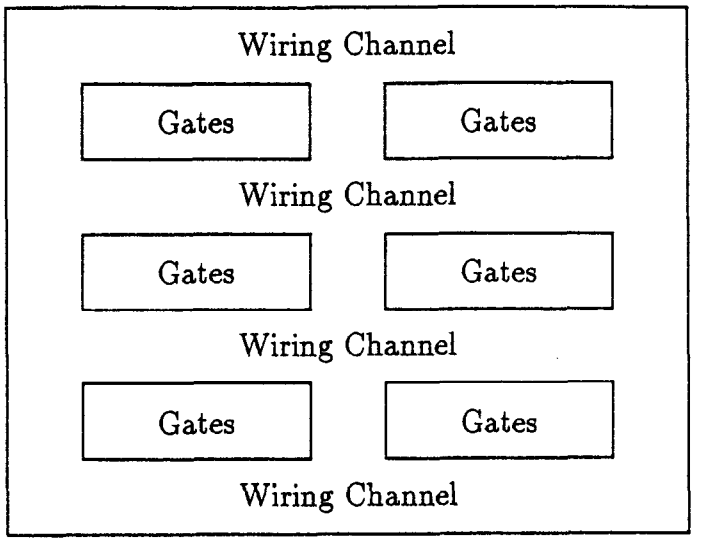


Figure 3–3: Basic Gate Array Floorplan.

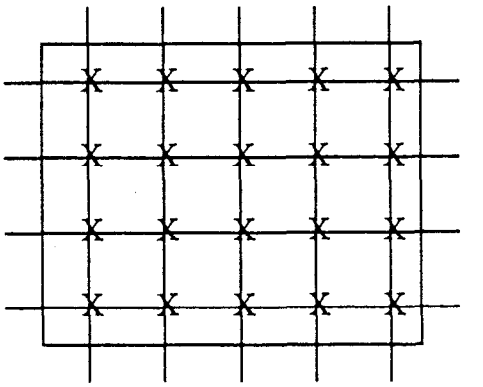


Figure 3–4: Detail of Wiring Channel Structure.

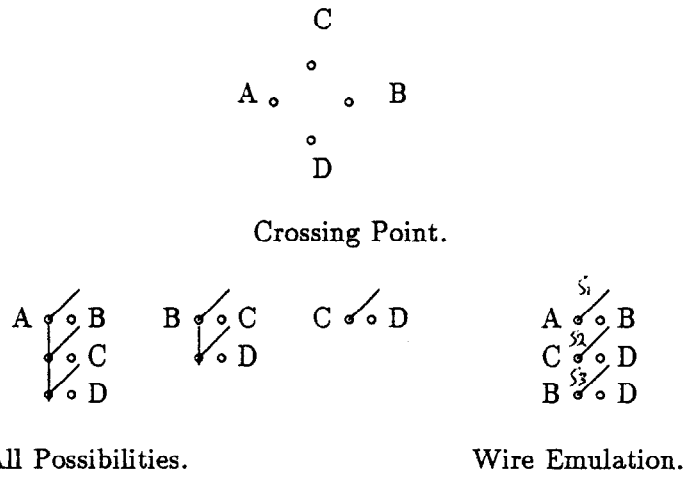


Figure 3–5: Wire Crossing Switching Functions.

Let us consider the implementation of such a switch using a wiring channel like structure. We assume that the switch is composed of identical blocks situated at the 'X' points on figure 3–3. Two reasonable designs for these blocks are given in figure 3–5. In both cases one RAM cell per switch would be required. The first uses three switches and is intended to emulate the wiring channel structure exactly: S_1 and S_2 represent possible cuts on the two 'wiring layers' and S_3 a possible contact between them. The second structure provides all possible connections between the inputs to the block. These structures can only provide a small fraction of the possible permutations because of 'blocking' within the switch, for example in figure 3–6 the connection from A to B cannot be made. Despite this, in the three switch case 60 RAM cells are required and in the 6 switch case (which is less susceptible to blocking) 120: both of these figures are well above the information theoretic bound for the general switch.

This switch model has assumed that the wires are bidirectional: this does not mean that user designs can contain bidirectional wires only that the direction of a particular wire is determined by the control store. If we avoid specifying

the direction of a wire until the chip is programmed then circuits of the form of figure 3–7 will have to be used (analogue techniques which attempt to 'sense' the direction of signal flow would probably require more space and would certainly be slower and less reliable). Depending on path length through the switch matrix, performance goals and process parameters switchable buffers could be required on all matrix inputs, all matrix outputs or both inputs and outputs. In effect we have traded a lot of control store for fewer wires. A system which supported true bidirectionality would have to have an additional user generated control signal associated with each bidirectional wire to specify its direction to the buffer circuits.

Now let us consider single direction wires: if there are n inputs and m outputs then there are m^n possibilities. With 18 inputs and 18 outputs (as in a 4x5 switch with each bidirectional wire replaced by two single direction wires) we would have about 3.93×10^{22} possibilities. These switching patterns can be realised using m, n : 1 multiplexors controlled by $m\lceil \lg n \rceil = 18 \times 5 = 90$ bits of RAM. Multiplexors can be laid out very efficiently and have the important property that the number of switches any signal must pass through is limited and is the same for all permutations: in channel structures some paths might have to 'snake' around to avoid blocking. In fact, it might well be necessary to forbid such snaking for electrical reasons (since every switch will have considerable resistance and buffering within the switch matrix is undesirable), further reducing the number of implementable permutations.

The above argument has demonstrated that wiring channel like switch matrices are an extremely inefficient structure for programmable parts which use high impedance transistor switches controlled by relatively large RAM memory cells. Far greater efficiency can be achieved by fixing the direction of wires and using multiplexors. Multiplexor based designs are also easier to use since all permutations are provided and there is no need to worry about blocking.

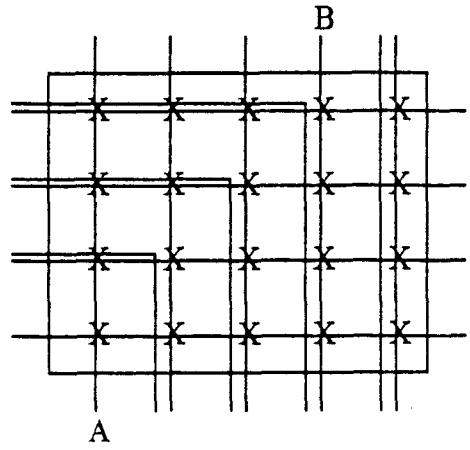


Figure 3–6: Example of Blocking.

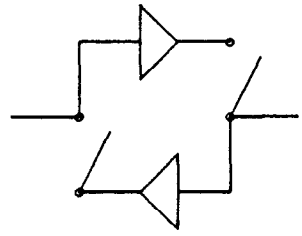


Figure 3–7: Buffering for Bidirectional Wire.

3.2 Function Unit Design.

The first thing to consider when designing a function unit for programmable cells is how many inputs it will have. We can assume that the function unit will have exactly one output and consider cells which have several output functions as having several function units.

3.2.1 Combinational Complexity.

At this point it is worth quoting some important results from combinational complexity theory which will help in the design of function units. This discussion is a very condensed version of the treatment in [Savage76]. The computation performed by any computing system can be represented as a function $f : \{0,1\}^n \rightarrow \{0,1\}^m$. The combinational complexity with fan out s of f over the basis set Ω (consisting of switching functions with fan-in r) is denoted $C_{s,\Omega}(f)$ or $C_s(f)$ where Ω is understood, is the minimum number of computation steps required to compute f with a chain over Ω with fan-out s and with data-set $\tau = \{x_1, x_2, \ldots, x_n, 0, 1\}$. If such a chain does not exist $C_{s,\Omega}$ is not defined. The best way to think of this is as a gate count where Ω is a set of basic gates with the same fan-in r. Source nodes (circuit inputs) are allowed unlimited fan-out and can be connected to anything in τ.

Two special cases of $C_s(f)$ are of interest $C_1(f)$ is denoted $L(f)$ and called the formula complexity $C_\infty(f)$ is denoted $C(f)$ and called the combinational complexity of f.

Another complexity measure is the delay complexity $D_\Omega(f)$: this measures the minimum depth of a chain of gates chosen from Ω which implements function f.

Important Results. Let Ω be a complete basis and let $f : \{0,1\}^n \rightarrow \{0,1\}^m$. Then $C_s(f)$ is defined for all integers $s \geq 1$ and

$$C_\infty(f) \leq C_{s+1}(f) \leq C_s(f) \leq C_1 f \tag{3.1}$$

Let Ω be a complete basis (i.e. all possible functions can be realised by connecting gates chosen from Ω) of fan-in r and let $f : \{0,1\}^n \rightarrow \{0,1\}^m$. Then there is a constant l_I peculiar to the basis with $1 \leq l_I \leq 2$, such that for $s \geq 2$

$$C_s(f) \leq \left(1 + l_I\left(\frac{r-1}{s-1}\right)\right) C_\infty(f) + \frac{ml_I}{s-1} \tag{3.2}$$

Thus $C_s(f)$ is of the same order as $C(f)$ for $s \geq 2$.

Let $f : \{0,1\}^n \rightarrow \{0,1\}$ be a boolean function and let Ω and $\acute{\Omega}$ be two complete bases of fan-in r and $\acute{r}$ respectively. Furthermore, let C_0 and D_0 be two constants defined by

$$C_0 = \max C_{\acute{\Omega}(h)}, D_0 = \max D_{\acute{\Omega}}(h), h \in \Omega$$

$$C_{\acute{\Omega}}(f) \leq C_0 C_\Omega(f), D_{\acute{\Omega}}(f) \leq D_0 D_\Omega(f) \tag{3.3}$$

From these results three conclusions can be drawn.

1. Fan-In. The fan-in must be at least 2 (at least two operands are required to allow computation) but the number of gates required to implement certain functions will decrease with greater fan-ins.

2. Fan-Out. The fan out must be at least two to ensure the combinational complexity is of the same order as C_∞ (by 3.2), greater fan-outs could decrease the number of gates required to implement certain functions (by 3.1).

3. Completeness. The basis Ω must be complete to allow all functions f to be implemented but if additional functions above those required for completeness are provided the number of gates required to implement certain functions may be reduced (by 3.3).

All of these conclusions are moderately obvious but it is good that they can be formally verified. Increasing any of the quantities of fan-in, fan-out and number of functions increases the area of the function unit and the tradeoff between number of function units and function unit complexity is central to the design of configurable systems.

3.2.2 Common Approaches.

Three philosophies of function unit design have emerged:

1. Provide All Functions of n variables. This is only sensible for small n (since 2^n bits of control store are required to select between the 2^{2^n} possibilities) but most cell designs have few inputs anyway so this is not a great problem. Special regular layouts of units capable of realising all functions of n variables are possible. Also, because the number of such functions is a power of two there is no wasted control store. This choice results in cells which are extremely easy to use: users are never in doubt about whether a particular function can be implemented.

2. Provide Special Building Functions. It has been noticed that some functions are much better than others for composing to produce larger functions. In some sense these functions can be viewed as orthogonal to each other and work has been done to find suitable sets of functions [Shoup70]. This approach seems attractive when you count costs in terms of gates but in VLSI topological considerations are dominant: similarly there is no reason why the number of orthogonal functions should be a power of 2 so control store might not be used with maximum efficiency. The system is harder to use in manual designs because you must remember what functions are available. This technique is important in structures with limited routing facilities where there is sometimes no benefit in providing all possible functions (e.g. the cutpoint array architecture dealt with in section 3.5.3).

3. One Function. In this approach a single 'good' function such as NOR or NAND is provided. This reduces cell size because no store is required to select cell function. It is also easy to remember what functions are available! Several cells will be required to implement functions which could be realised by one more complex cell.

All of these function unit designs are suitable for use with automatic logic-synthesis algorithms but method 2 is generally predicated on a particular synthesis

algorithm and is not as suitable as the others for hand design. Method 1 offers the greatest flexibility in algorithm selection at some cost in efficiency if only one synthesis technique is of interest.

3.2.3 Number of Function Units.

There must be at least one function unit but having several function units could allow useful multiple output functions such as adders to be implemented within one cell. Multiple function units within one cell may clash with the basic idea of cellular systems but it is an important technique because in some cases efficiencies in implementation can result. There are three main design possibilities.

One Function Unit. This is the obvious choice and results in a clear user model of the cell. It results in a large array of small cells rather than a small array of large cells.

Many Identical Function Units. Assuming that all function units are identical it can be seen that adding an extra function unit will greatly increase the cell area - not only because of its own size but also because of the extra complication in the routing system. One reason for having many identical units is that it is possible to compose them in a cell which could, for example, perform any two functions of three variables or any single function of four variables.

Many Different Function Units. This probably makes more sense than having many identical function units. One could envisage a cell in which there were four function units each capable of implementing 4 of the possible functions of two boolean variables. This would require 8 control signals. One cell would often be able to compute several useful functions simultaneously. The major arguments against having several different functions are that it increases the perceived complexity of the system to users and that it is unclear whether design techniques could be devised to take advantage of the additional capability.

3.2.4 Number Of Inputs.

There must be at least two inputs to allow computation: after this the number of inputs is influenced by two factors: the routing capabilities and the choice of implementable functions. With nearest neighbour connections only it is hard to get more than two or three operands to a given cell so there is little point in having more than three function block inputs. If it is intended to implement all functions of the input variables then the control store required increases very quickly.

3.2.5 Implementing Function Units.

Three methods of implementing general functions according to a control store have been considered.

1. Lookup Table. The idea behind this is that the RAM itself is used as a lookup table: this method is used in the Xilinx design. If a function has n inputs then it can be described using a 2^n row truth table and implemented using a 2^n bit lookup table. The data inputs drive a selector which chooses which of the RAM cells provides the output.

   The implementation of this method is tricky because it involves selectively connecting the outputs of a RAM to a capacitive load which is in an unknown state. There is the inherent potential to accidentally write the RAM cell. Either great care must be taken with the design or intervening buffers must be used. Solutions which involve changing the sizes of RAM cell transistors must take into account normal read and write of the cell from peripheral logic.

2. Choosing Inputs. This method relies on the fact that a function over $\{x_0, \ldots, x_{n-1}\}$ can be implemented using a function with a larger number of inputs $\{y_0, \ldots, y_{m-1}\}$ and a mapping function which connects y inputs to selected x inputs or constant inputs (1 and 0) [Preparata71,Chen82]. The original reason for developing this technique was that the connection function could be implemented using wires in a gate array, however it can also be implemented

advantageously using multiplexors. This technique has the property that some functions will be calculated faster than others.

3. Obvious Method. The programmable unit can be specified as a logic function of the control and data inputs and an easily derived truth table. Standard logic synthesis techniques can be applied to produce an implementation. This method is perhaps most suitable for the less regular building-block function design.

3.2.6 Sequential Function.

As well as having a combinational logic function most programmable cells have a sequential function in each cell. There are three possible decisions which will be investigated here:

1. No Sequential Function. A sequential function is not strictly necessary since flip-flops could be constructed by wiring up gates in adjacent cells using the routing function. There are two compelling reasons for including a sequential function within the basic cell.

   (a) Efficiency. Store is so common in digital hardware it is necessary to be able to implement it cheaply: using multiple cells to implement simple registers would pose an unacceptable overhead in many systems.

   (b) Metastability. Metastability or synchroniser failure is a fundamental problem in digital systems [Seitz80] and especially in programmable logic (see, for example [Xilinx86],[MMI84]). If a data input and a clock signal to a latch change almost simultaneously the output can enter a metastable state in which it is not at a valid logic level. Normally this state will be left very quickly but in theory it can last indefinitely. This can cause failure in the system containing the latch. The probability of entering a metastable state is proportional to the delay in the latch feedback loop. This is very small if the latch is implemented carefully

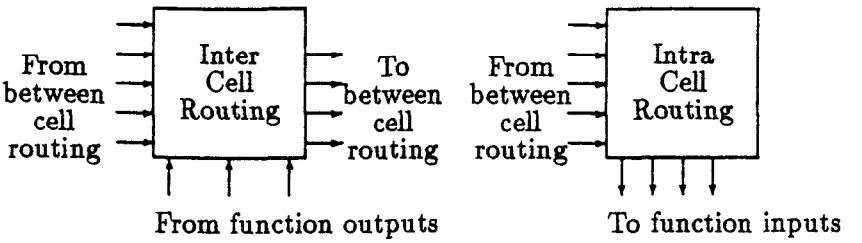


Figure 3–8: Within Cell Routing.

at the mask level as a cell function but if multiple cells are used routing delays will increase the loop delay to a level at which the risk of metastability may be unacceptable in some applications.

2. Simple Latch. A standard small unit such as a D or RS latch is provided from which larger units can be composed. This has the advantage of keeping the function unit relatively small. Another important reason for choosing simple latches is that they only require two or three input variables (D and Clock or R and S and perhaps Clear). Often, they can be implemented by simply adding feedback to the combinational logic function.

3. Complex Programmable Flip-Flop. A very general sequential function is provided which under RAM control can implement any of the usual catalogue flip-flops (not just latches). This system requires at least four input variables since large 'TTL catalogue' flip-flops have many inputs (J, K, Clear, Preset and Clock). This adds considerably to the complexity of the function unit and routing circuitry. It should be noted that more than the minimum number of RAM cells required to select between the x combinational and y sequential functions are likely to be used because of the difficulty of sharing control store between the two units. On the other hand this technique does allow any flip-flop to be implemented within one cell.

3.3 Within Cell Routing.

The within-cell routing function is shown diagrammatically in figure 3–8. Two kinds of routing can be identified: inter-cell routing which selects outputs to be passed to the between cell routing and intra-cell routing which selects inputs to the function unit.

3.3.1 Inter-Cell Routing.

Several important parameters in the design of this routing can be identified.

Flexibility. The design of this circuitry is largely dependent on the choices made in the design of the between cell routing. In systems with very flexible between cell routing (e.g. wiring channels) the within cell routing can have a fixed structure. The design of inter-cell routing is most important in systems with fixed between cell routing such as nearest neighbour connections.

Number of Inputs and Outputs. The complexity of this function depends on the number of connections to the between cell routing. The output of the cells function unit counts as an input to this routing function.

Permutations. Given a set of inputs and outputs one must make choices about which outputs can be sourced from which inputs. Often some permutations are not useful: for example in a nearest neighbour scheme connecting an input from a neighbouring cell back to the same cell. Reducing the number of permutations supported can obviously reduce the size of the cell but if the resulting routing is asymmetrical the cell becomes hard to use. Reducing the routing available in a single cell often has the unwanted side effect of increasing wire length in systems implemented using the cells. It is worth pointing out in this context that increased utilisation of cell resources does not necessarily imply increased efficiency: in fact

limited permutations can cause 'wandering' wires which increase utilisation but decrease real efficiency. For example, a design in which a wire had to travel through every multiplexor in an array in order to leave on the same cell as it entered would have very high utilisation but very low efficiency.

Symmetry. In a programmable structure which is to implement many separate functions symmetry of routing permutations is very important since it allows sub-designs to be mirrored and rotated to improve the floorplan. In designs which are to implement a single logic block especially those which use a fixed array structure for logic synthesis, symmetry is highly undesirable since it results in redundant routing permutations.

3.3.2 Intra-Cell Routing.

These multiplexors select inputs to the function block from cell inputs (figure 3–8). We will assume that connections from function block outputs to function block inputs, (e.g. for feedback in a latch) are internal to the function block. Three designs will be investigated:

1. Direct Connection to Cell Input. In this case a set of cell inputs are chosen and the function block inputs connected directly to them. Shoup [Shoup70] has pointed out that this will usually cause adjacent cells to be wasted routing inputs to the correct terminal. An important use of this technique is in cells where the function unit is symmetrical and all cell inputs are connected to it.

2. Direct Connection to Cell Output. The trick here is that the output multiplexors are used to select inputs for the function block: thus function block inputs can enter the cell from arbitrary directions. The problem is that cell outputs are being decided by unrelated considerations. It may not be possible, for example, for a user to have a 'bus' wire passing horizontally across a line of cells because one of the outputs needed to make this wire must select

another source. This unnecessary coupling between routing and logic design can be expected to make it harder to write design automation software.

3. Dedicated Multiplexors. This involves having additional multiplexors to select function block inputs. It makes the cell much easier to use at the expensive of an increase in circuit complexity.

3.4 Configurable Logic Design.

This section will present the cell design used in the remainder of this thesis based on the mapping of the design space presented above.

3.4.1 Between Cell Communication.

It was decided that the inter-cell communication should be fixed and all selection should happen in the basic cell itself. This allows the whole programmable structure to be built from a single rectangular repeating unit which makes for a very efficient layout. A nearest neighbour wiring scheme was chosen as the basic communication structure allowing extreme simplicity and regularity. Three additional global signals are routed to every cell: two inputs G_1 and G_2 and an output $FTEST$. The two inputs are intended to be used as clock signals in user designs because clocks are one area where delays through nearest neighbour connections could be a serious problem. This has the important side effects of freeing the user from deciding a strategy for clock distribution and freeing many multiplexors for data signals. The global output signal $FTEST$ is used to allow the value of any function unit to be examined without upsetting the routing multiplexors: it is intended mainly as a debugging aid.

3.4.2 Within-Cell Communication.

Minimisation of cells needed for routing could be achieved by having the maximum routing complexity i.e. a crossbar switch with 6 outputs (N,S,E,W, 2 function

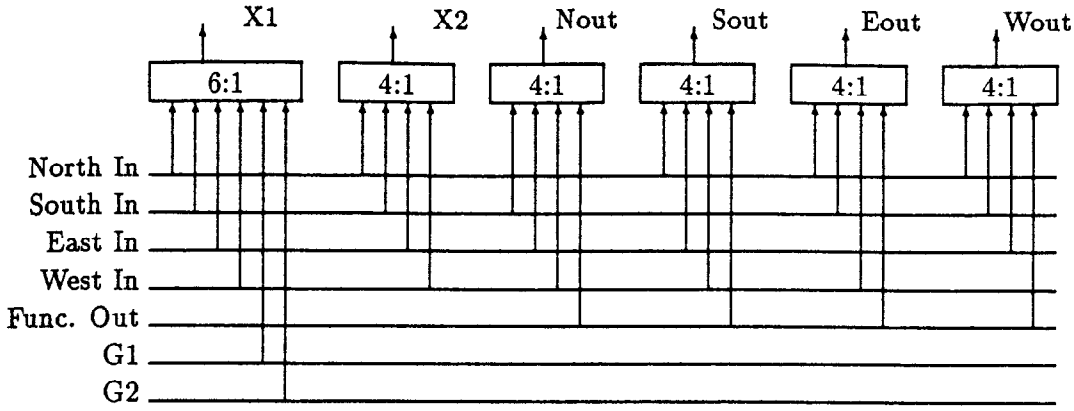


Figure 3–9: Cell Routing.

block inputs (X_1 and X_2)) and 7 inputs (N,S,E,W,G_1,G_2 and function block output). Some permutations, for example, routing North input to North output are not useful since the cell to the North will already have the signal available or be receiving an undefined value (if it has connected South output to South input) so these possibilities are removed. Similarly, there is no need to be able to select the global signals as cell outputs. Removing redundant permutations results in functions which can be realised with 4:1 multiplexors. Four is a 'magic' number here since it is a power of two which means that we get maximum utilisation of control store in such multiplexors.

The routing structure within the cell is shown as figure 3–9. This switching function can be realised by five 4:1 and one 6:1 multiplexor controlled by 13 bits of RAM. Since G_1 and G_2 are intended as clock signals they are only available on one of the function block inputs: this is one of the few cases of asymmetry in the Configurable Logic design.

3.4.3 Function.

There must be at least two inputs and at most six (because the cell itself has only four nearest neighbour and two global inputs). We choose to implement all functions of the input variables to simplify the user interface and minimise

the number of cells required to implement any given function. This means that the number of RAM cells required to control the function unit doubles for every additional input variable: it is unlikely that function units with more than four input variables would be useful even if the cell itself had more than four inputs. The advantages and disadvantages of each choice are outlined below.

1. Two Variables. There are sixteen possible functions of two variables so the cell needs only four bits of RAM to control the function unit. This is consistent with relatively small cells. Several cells will be needed to build larger functions but there are many algorithms for logic synthesis with two input general cells (Chapter 6) and obvious array structures for logic blocks using them. Since there are four inputs to the cell either two must be chosen arbitrarily as function unit inputs (possibly resulting in additional routing and making the cell asymmetrical) or additional routing multiplexors must be provided.

2. Three Variables. There are 256 possible functions of three variables so the cell would require 8 bits of RAM to control the function unit. This still allows reasonably small cells. Many common functions such as three input XOR (used to calculate sum in full adders) can now be implemented within one cell. Three input variable function units have been used in previous designs e.g. the connection machine ALU [Hillis85]. As with two input designs either the cell must be made asymmetrical or additional input-selection multiplexors must be provided. The major disadvantage relative to two-input designs is that getting three inputs into a cell for computation means that they must come from three different directions. It would be hard to find an array layout for logic blocks which got around this problem using only nearest neighbour connections. Given the difficulty of routing three inputs to a cell for computation it is worth considering providing more than one such function unit to allow, for example, both sum and carry to be implemented within the same cell.

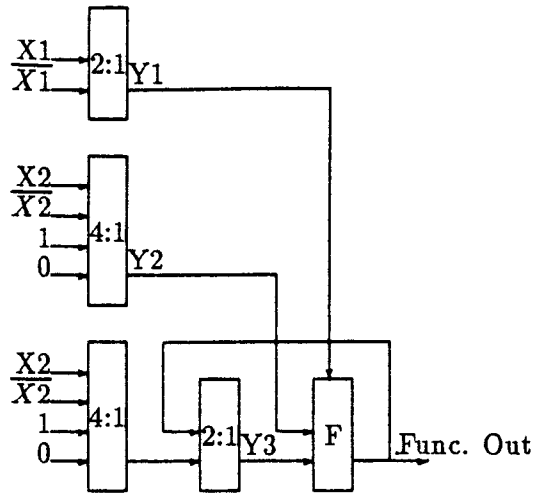


Figure 3–10: CAL Function Block Design.

3. Four Variables. There are 65536 possible functions of four variables so the function unit requires 16 bits of control RAM. This forces us to have large cells. No additional multiplexors are necessary but the problem of routing four inputs to a given cell to get the benefit of its function unit will usually involve wasting many adjacent cells.

Design Decisions. It was decided to use a two input function block since this allowed us to have a very symmetrical but still small basic cell which could make use of previously developed logic synthesis algorithms. The decision to have a two input function unit ruled out the provision of complex flip-flops but it was felt essential to provide at least a simple latch within the basic cell.

A block diagram of the function unit within each cell is given as Figure 3–10. It relies on the fact that all functions of two variables (X1 and X2) can be computed by a 2:1 multiplexor (marked F in the diagram) which selects Y2 if Y1=1 and Y3 if Y1=0 given the inputs shown in Table 3–1 [Chen82]. This technique was chosen because it allows the function unit to be implemented using the same multiplexor designs used in the routing section. This simplifies the VLSI layout and allows us to concentrate on very efficient layout of a single multiplexor and RAM combination leaf-cell.

Number.	*Function*	*Y1*	*Y2*	*Y3*
0	*ZERO*	$x1$	0	0
1	*ONE*	$x1$	1	1
2	$X1$	$x1$	1	0
3	$\overline{X1}$	$x1$	0	1
4	$X2$	$x1$	$x2$	$x2$
5	$\overline{X2}$	$x1$	$\overline{x2}$	$\overline{x2}$
6	$X1 \cdot X2$	$x1$	$x2$	0
7	$X1 \cdot \overline{X2}$	$x1$	$\overline{x2}$	0
8	$\overline{X1} \cdot X2$	$x1$	0	$x2$
9	$\overline{X1} \cdot \overline{X2}$	$x1$	0	$\overline{x2}$
10	$X1 + X2$	$x1$	1	$x2$
11	$X1 + \overline{X2}$	$x1$	1	$\overline{x2}$
12	$\overline{X1} + X2$	$x1$	$x2$	1
13	$\overline{X1} + \overline{X2}$	$x1$	$\overline{x2}$	1
14	$X1 \oplus X2$	$x1$	$\overline{x2}$	$x2$
15	$\overline{X1 \oplus X2}$	$x1$	$x2$	$\overline{x2}$
16	*D Latch*	$x1$=Clk	$x2$=D	*Func. Out*
17	$\overline{D}$ *Latch*	$x1$=Clk	$\overline{x2}$=D	*Func. Out*
18	$D\overline{Clk}$ *Latch*	$\overline{x1}$=Clk	$x2$=D	*Func. Out*
19	$\overline{D}\ \overline{Clk}$ *Latch*	$\overline{x1}$=Clk	$\overline{x2}$=D	*Func. Out*

Table 3–1: CAL Programming Table.

3.4.4 Improvements.

The present design of configurable logic cell is 'minimalist' in as much as no features have been added if their was any doubt as to their utility, even when there was a possibility to implement them using otherwise unused resources. There are two reasons for this: firstly, every additional feature complicates the design and reduces the chances of success on first silicon and secondly, it was felt that it would be better to get some experience with the minimal design before adding extra features. The second point is reinforced by the fact that it would be hard to remove features from the architecture later when user designs could be relying on them. In this section we will consider some possible 'bells-and-whistles' which could be added to the current design, most of these are possible uses for the X1 and Y3 multiplexor permutations not used by the current design.

Global Signals. One area in which the present design may have inadequate provision is in global wiring to allow low-skew distribution of critical control signals. It would be trivial to add another two global inputs using unused inputs on the X1 multiplexor, however, it would probably be better to add row and column array crossing signals to the architecture. To get full benefit from these they should be implemented as wired logic and be able to be driven by cell outputs or pad inputs. This would require two extra RAM cells and some moderately large buffering circuits in each cell.

Extra Functions. Another way of extending the cell would be to use the two extra Y3 multiplexor inputs. These could be connected to global signals or cell inputs. This would allow a few functions of three variables such as the 2:1 multiplexor to be implemented but the function units would no longer be symmetrical with respect to cell inputs i.e. some functions could only be performed using specific cell inputs. Provision of multiplexors would allow efficient implementation of switching structures within user designs.

Another possible extension would be to add two more RAM cells to the basic design to increase the number of functions implementable (the floorplan of the

VLSI implementation favours adding RAM cells in pairs). Adding extra sequential functions such as RS latches or master-slave registers is especially attractive because of the electrical problems associated with multiple cell implementations of these functions.

Application Support. As we discussed in the introduction several important applications for configurable logic are envisioned. Some of these can benefit from extra hardware within the basic cell. For example the current *FTEST* circuitry which allows one function block output to be monitored in real time is very suitable for EPLD applications but in an application where the configurable logic control store was memory mapped on a host computer it would make more sense to make function block outputs available by reading locations in RAM. This facility is extremely easy to implement and is provided in the Xilinx LCA architecture. Another possibility would be to extend the architecture by allowing inputs to the cell array to be provided by writing RAM locations on the chip. This is already possible to a limited extent by programming the function units to implement the constant functions 1 or 0.

3.5 Comparison with Previous Designs.

In this section we will compare the configurable logic cell design with some of the most significant previous designs. We can divide these designs into two classes: simple array structures intended for implementing single logic blocks and more general structures. We will move in approximate order of generality. Firstly we deal with two-level AND-OR type designs: these account for most of the present market for EPLD's [Byte87]. Secondly, we will deal with the cutpoint array: this is very similar to the two level designs except that it uses more general logic gates in the AND plane. Thirdly, Shoup's cell design is important because it was the first reasonable attempt at a higher generality cell - previous designs such as the Wahlstrom array [Shoup70,Wahlstrom67] are grossly inefficient. Finally, the

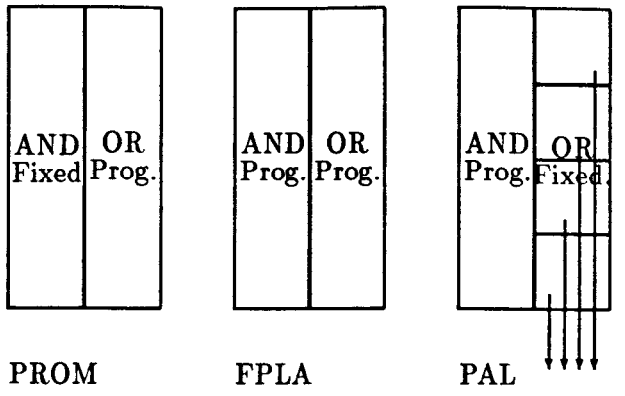


Figure 3–11: PLA Derivatives.

Xilinx system is of interest because it is a commercially available programmable system using up to date technology.

3.5.1 Two Level Designs.

These are what most people think of as programmable logic or EPLD's. Good introductions to design with these parts are given in [MMI84] and [Byte87]. Two level logic is basically simple but has become confusing as company marketing departments invent new acronyms almost every day to persuade customers their products are best. Three classes of programmable two-level logic can be identified (figure 3–11). It is worth pointing out that these designs rely on the use of wired logic for their efficiency: wired logic can be implemented more efficiently in fuse based designs than RAM based designs.

Programmable Read Only Memory (PROM). This is a straightforward derivative of ROM - all minterms are decoded and a programmable OR plane selects which ones to use in each output function. This architecture is general and suitable for irregular functions (such as control stores). It is an 'all functions of n variables architecture' and so cannot be used for functions with large numbers of inputs.

Field Programmable Logic Array (FPLA). This architecture is a development of the PLA: it has a programmable AND plane and a programmable OR plane. The problem with this architecture is that the number of product terms has to be guessed *a priori*; as we have seen it is hard to make good guesses for general functions. In the EPLD marketplace this leads to providing a catalogue of chips with different numbers of inputs, outputs and product terms. The FPLA architecture is the most flexible of the two-level designs and is best suitable for logic blocks with a relatively large number of inputs.

Programmable Array Logic (PAL). This architecture was designed by Monolithic Memories [MMI84] and is composed of a fixed OR plane and a programmable AND plane. The product terms are partitioned into groups, one group for each output variable. Each output is then an OR over a small number of user selectable minterms. This implies that minterm sharing between outputs is no longer possible. This architecture is slightly faster than the FPLA because only one plane is being programmed: this factor is critical in common PAL applications such as memory address decoding. It is a good design for the kind of small functions found as glue logic on printed circuit boards but is totally unsuited for high generality systems. The lack of generality of the architecture is addressed to some extent by providing a catalogue of chips each with different array dimensions and peripheral logic. As well as the bipolar fuse-based PAL devices CMOS EPROM based PAL's are available from Altera [Altera87]. The increased density of CMOS allows much larger numbers of product terms and output groups (or 'macrocells') and flexible I/O units. The Altera CAD system takes advantage of a library of macrocell encodings for standard TTL functions to allow automatic conversion of TTL designs.

Extensions. Four important extensions are often applied to the basic PAL and FPLA designs to increase their utility.

1. Feedback. Implementations of many of the designs above include registers on the array outputs which are fed back to array inputs to allow implementation

of state machines. Use of complex registers such as J-K flip flops rather than simple latches can reduce the number of product terms required [Sobol83]. Any *a priori* division of array inputs and outputs between feedback terms and off-chip inputs and outputs will have adverse consequences on the expected utilisation of the programmable array: this can be offset by providing a catalogue of devices.

2. Decoders. Normally adjacent columns of the AND plane are fed with an input variable and its complement but it is possible to use other functions of the input variables. One important technique is to group two input variables x_1 and x_2 and instead of using $x_1, \overline{x_1}, x_2, \overline{x_2}$ to drive the array use the decoded functions $x_1x_2, \overline{x_1}x_2, x_1\overline{x_2}, \overline{x_1}\overline{x_2}$. This technique can often reduce the number of product terms required. Larger decoders of three or more variables can also be used but this increases the number of columns in the AND plane and can increase the size of the PLA [Fleisher75]. In a programmable structure the need to make *a priori* decisions about the number of decoders and variable grouping decreases the utility of this technique; there is also some additional delay caused by the extra level of logic. None of the common programmable parts provide decoders.

3. Programmable Invert. Sometimes the complement of a function can be implemented with fewer product terms than the function itself (either because fewer product terms are required to cover the minterms or because better sharing with other functions is possible). FPLA chips, therefore, often provide programmable invertors (using XOR gates) on OR plane outputs to allow the logic matrix to implement the complement of the desired function.

4. Programmable I/O Blocks. It is very common to reduce pad costs by providing complex programmable I/O blocks which allow pads to be used as inputs, outputs or Tri-State common inputs and outputs. Often additional programmable latches or flip-flops are included in these blocks to synchronise signals entering and leaving the chip with a system clock.

Timing. It is important to consider the timing properties of two-level designs in some detail especially since the FPLA and PAL designs are often used as asynchronous parts in TTL-type systems with edge triggered clocks. The timing complexity of these systems can undermine the reduction in design times expected from this approach. There are three types of timing problems.

1. Synchroniser Failure. This occurs when a clock signal and a data signal to a latch change at almost the same time forcing the output into a metastable state for an indefinite (but normally very short) period.

2. Logic Hazards. There are two classes of logic hazard (or 'glitch'): static and dynamic. In a static hazard an input variable change which should have no effect on the output causes a temporary change in the output which then returns to its correct value. In a dynamic hazard an input change causes several transitions on an output before it reaches its correct new state. These hazards can be caused by 'changing products' in the AND plane (e.g. in a static hazard when $x = 0$ then $y = 1$ because of product term 10 but when $x = 1$ then $y = 1$ because of product term 15, so when x changes y may well go to 0 for a short period) often they can be eliminated by using large overlapping minterms rather than small disjoint minterms in the logic synthesis. This, of course, complicates the task of the logic designer or the author of the logic synthesis program.

3. Function Hazards. A function hazard occurs because of changes in input variables. Suppose in the design of a state machine two input variables a and b are supposed to go low simultaneously as a result of a state transition. In a physical implementation there may be a slight delay between the two transitions (perhaps because of differences in the length of the feedback wires) resulting in temporary spurious output values.

Use of timing methodologies which would normally be frowned upon is prevalent in PLD designs because of the predictability of the delay through the logic matrix

and the need to 'shoehorn' designs into a standard size chip. From the point of view of timing three classes of design can be identified.

1. Combinational. In this class of design no state is provided on chip. The user must still be aware of timing hazards if the design's outputs are to be used as clock signals for external circuits.

2. Registered Outputs. In this case D type edge triggered flip-flops or latches are placed on array outputs and clocked by an external signal. If off-chip input signals are also latched then this design can be safe from timing hazards. PROM's nearly always use registered outputs since the 'single minterm' covering of output functions means that logic hazards are unavoidable

3. Generated Clocks. In this case clocks or R and S signals for latches can be taken from matrix outputs. This provides extra flexibility for the logic designer but places additional responsibility on him. Logic and function hazards are both very important since 'glitches' in output signals will cause these registers to take spurious values. If a data and a clock signal are taken from adjacent product terms it is not unlikely that they will make a transition almost simultaneously so the synchroniser failure problem must also be considered.

3.5.2 Generalised PLA Architectures.

In this section we will cover some of the attempts which have been made to generalise the basic two level architecture to compete with gate array class systems: all of the methods discussed below were developed for mask-programmed designs but could be adapted for dynamically configurable systems.

IBM High Density PLA. The IBM programmable logic chip [Wood79] attempts to increase the generality of the PLA architecture by incorporating a large amount of programmable circuitry on the periphery of the chip while still keeping the array itself relatively inflexible. This architecture is particularly notable for

the number of patents associated with it. The floorplan is shown in figure 3–12 and offers the following additional capabilities.

1. Two AND Planes. Instead of having a single AND plane there are two: one above and one below the OR plane. This is an attempt to solve the problems of implementing more than one function in a single array without the area overhead of having interleaved AND and OR columns. Obviously it is less flexible than the fully interleaved approach but can be more efficient in particular cases.

2. Segmentation. Both the AND and the OR planes can be segmented at any point on a column. Coupled with allowing inputs and outputs to enter and leave from either side of the array this allows for greatly increased utilisation. The utilisation can be further increased using the programmable interconnect between the AND and OR planes which allows several small functions to be joined together to produce one larger function before being fed into the OR plane. This means that product terms can be built up from two inputs which enter on the same column - one at the top and one at the bottom - thus inputs can share a column if they have only a small number of product terms in common.

3. Input Decoders. There is a complex programmable interconnect scheme at the array periphery to allow inputs to be paired together before being fed to decoders at the array edges. All array inputs are paired and decoded in this manner.

4. Programmable Latches. All output variables are fed through complex programmable latches. These latches allow for programmable inversion of the output variable. Three functions are available J K Master/Slave flip flop, gated latch and AND/polarity hold function. All these latches use two output columns in the PLA, in the first case these are connected to the J and K inputs, in the second case output is used as a clock for a D latch on the second input and in the third case the two outputs are AND'ed together before being passed to a latch controlled by the system clock.

The idea behind using two input flip-flops is that the they 'remember' their current state until explicitly told by a PLA output to update it - state machine designs which use simple D latches require the PLA matrix to update the state vector every clock cycle. This technique is claimed [Sobol83] to make the PLA encoding easier to follow and to reduce PLA real estate by decreasing the number of product terms: it must be remembered that it does require twice as many output terms so the overall area will not necessarily be reduced.

5. Programmable Interconnect. A complex programmable bussing scheme is provided between the edge of the array and the pad ring to allow chip inputs and outputs and PLA feedback terms to be connected in fairly arbitrary patterns to PLA inputs and outputs.

6. Programmable I/O. Special programmable I/O drivers are provided allowing each pad to be used as an input, output or Tri-State input/output.

Hewlett Packard Universal Synchronous Machine (USM). The HP programmable logic chip architecture [Sobol83] is shown in figure 3–13. It offers the following additional capabilities over the PLA.

1. Segmentable AND and OR plane. The design interleaves inputs and outputs (row folding) to allow multiple functions to be implemented efficiently within a single array. Inputs and outputs can enter and leave from the top and the bottom (column folding). Rows and columns of the array can be segmented only at predetermined points. Without segmentation 112 product terms are available; using all 8 row breakpoints 896 are available.

2. Complex Flip-Flops. PLA outputs are latched using complex flip-flops which can implement one of 15 different truth tables, additionally an XOR gate is provided to allow programmable inversion of flip-flop inputs. Like the IBM design each flip-flop is controlled by two PLA output columns.

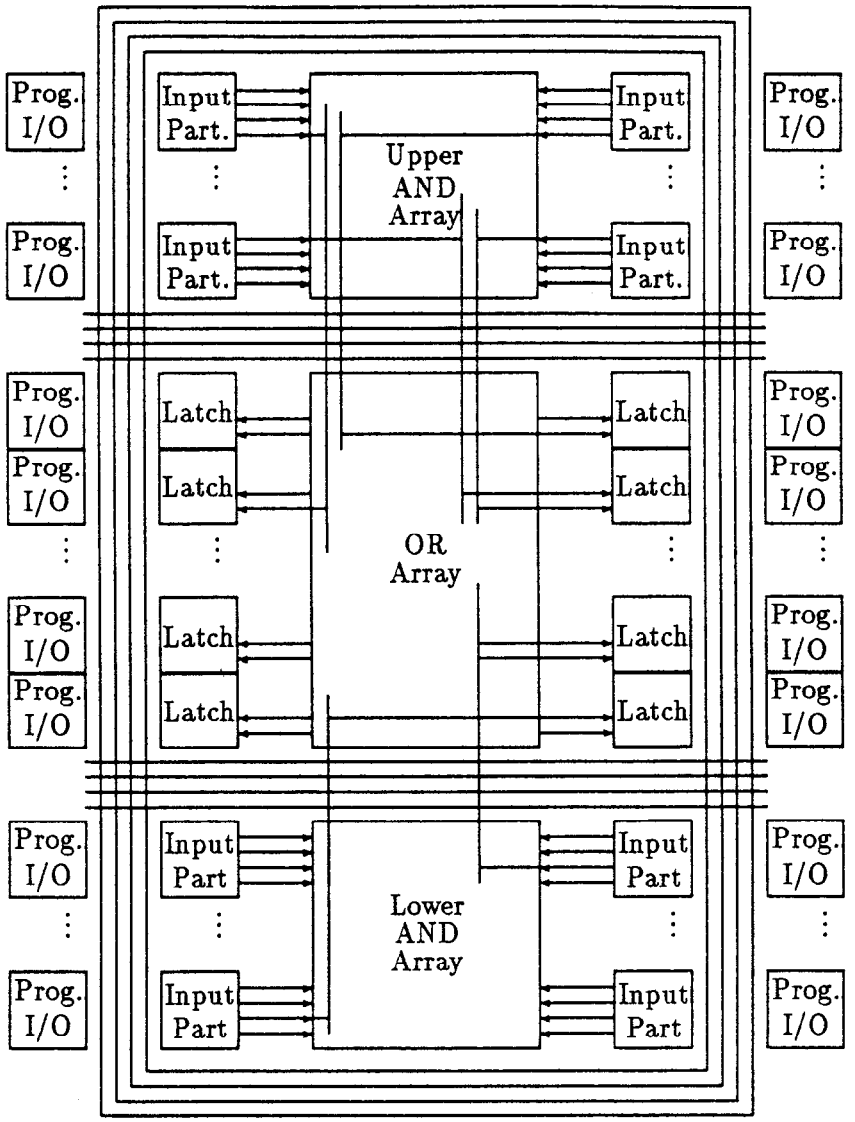


Figure 3–12: The IBM High Density PLA.

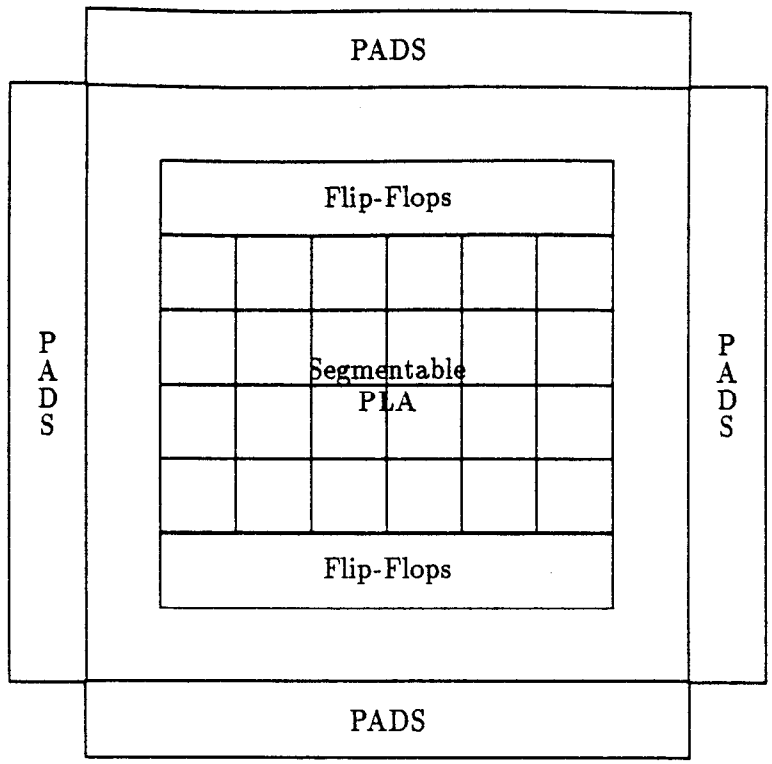


Figure 3–13: The H.P. Universal Synchronous Machine.

3. CAD Support. One of the major claims made for the USM architecture is the ease of mapping state machine designs onto it. To support this application special CAD programs were developed: designers edit the PLA matrix directly rather than specifying logic equations. Presumably this allows much better utilisation of chip resources where multiple functions are implemented within a single segmented array.

Storage Logic Array (SLA). The SLA programmable logic architecture [Patil79] is the most general of the attempts to improve PLA's and takes many of the ideas in the other designs to their logical conclusion. The basic architecture is shown in figure 3–14. Segmentation of both rows and columns is allowed on specified block boundaries - the original paper suggests row segmentation points every 4 columns and column segmentation points every 8 rows. Every few (the paper suggests 2) column segmentation points a latch is provided inside the array. These are RS

latches and require two array output lines to drive them (rather than the conventional D latches clocked by a system clock found at the edge of PLA's). There are breakpoints within the latch design allowing the gates to be used as buffers, invertors or NAND gates instead of the normal latch function. There are obvious potential timing hazards associated with these latches when used as feedback terms since there is no synchronisation to a system clock: [Patil79] claims that because the variance of the delay through two columns is much less than the delay through the fastest column (because the loading and transistor parameters are well matched) race conditions should not occur. A given row or column may be segmented many times (with IBM and HP designs there is no reason to segment columns more than once because there are only two possible input signals). Rows and columns are fully interleaved and signals can enter and leave from the top or bottom of the array. On the diagram 'x' represents a programmable column break point and 'X' a row breakpoint. The normal PLA OR-plane possibilities are available where rows cross latch inputs and the normal PLA AND-plane possibilities where they cross latch outputs.

The multiply segmented architecture and the embedded storage elements allow logic blocks to be embedded in the centre of the array receiving inputs and outputs from other adjacent logic blocks: this is a major increase in generality over the previous designs. Design using SLA's is much more like normal VLSI design with floorplanning becoming an important consideration.

3.5.3 Minnick's Cutpoint Cell.

This design is in the same generality range as the FPLA: like it arrays of these cells were intended to implement single logic blocks. This cell has fixed between cell and fixed within cell communication. The cell schema is shown in figure 3–15: it can implement any of the functions shown in table 3–2. The cell was not intended to be programmed dynamically but by blowing fuses or cutpoints. However, a dynamically programmed version could easily be designed. This array is intended to synthesise a single logic function of n input variables and it performs its task

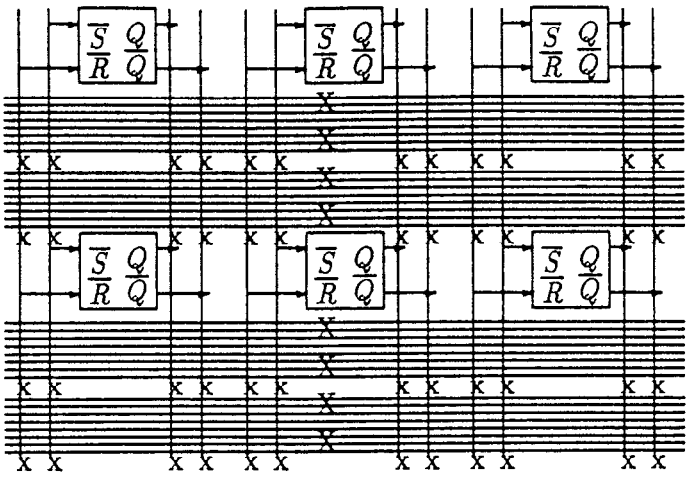


Figure 3–14: The Storage/Logic Array.

very well. It is potentially more efficient than a PLA because the more general gates allow larger functions to be generated in a single product row. Speed will be much lower, however, because signals must propagate through many simple gates rather than through two levels of wired logic.

Obviously, the design can be generalised to multiple output functions with product sharing in the same way as PLA's. If this is done an interesting situation arises: the ability to synthesise larger functions in the 'AND' plane becomes less useful because it is less likely that a larger function will be usable by several of the output signals. However, if we also use more general gates in the 'OR' plane the number of product terms can be significantly reduced. This situation is considered in more detail in Chapter 6.

3.5.4 Shoup's Control Array Cell.

Figures 3–16, 3–17 and 3–18 show the design of Shoup's cell: arrays of these cells were intended as a replacement for microprogrammed control store. We shall

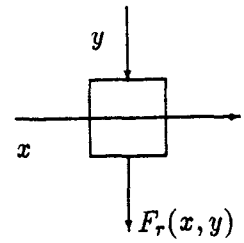


Figure 3–15: Minnick's Cutpoint Cell.

Index r	$F_r(x,y)$
1	$x+y$
2	$\overline{x}+y$
3	$\overline{x}.y$
4	$x.y$
5	$\overline{x} \oplus y$
6	y

Table 3–2: Cutpoint Cell Functions.

consider this cell in some detail: before we begin it is worth mentioning that Shoup considered cell control store to be built up from shift registers constructed using logic gates and multiplexors to be constructed using logic gates. Control store/multiplexor combinations were 'big boxes' on Shoup's diagrams but using the VLSI implementation developed in the next chapter we can consider them to be fairly small ones. This has a strong influence on the tradeoff between flexibility and utilisation in the cell design. Shoup's cell requires 15 control bits.

Inter-Cell Communication. This cell has input and output connections to the four nearest neighbours on a grid. It also has array crossing busses, unusually these run diagonally at 45° to the grid connections along both SW-NE and NW-SE diagonals, one set are inputs and the other set are outputs. This scheme makes every global signal available at one cell in each row and column, after this routing can be done using neighbour connections.

Intra-Cell Communication. The routing function within the cell is rather unusual as well: regularity and symmetry are sacrificed to get maximum utilisation of multiplexors. Four to one multiplexors are used: as we have seen this is an ideal size for routing multiplexors. Signals from north or south can pass through or turn left or right. Signals from east and west can only go straight through or turn right. Shoup claims that this reduced switching function still allows flexible communication between cells: within a single logic block he may be correct but the lack of symmetry means that sub-designs cannot be rotated to aid floorplanning in larger systems.

Function Unit Input Selection. Shoup again attempts to cut down on the number of multiplexors required by using the outputs of the routing multiplexors as sources for a 4 input function block. Not every input to the function block need be significant. Since most cells will not require all four routing multiplexors spare ones can be used to select function block inputs: this will significantly increase multiplexor utilisation. There is one important problem with this approach: it

confuses the allocation of two logically distinct resources. Decisions about which functions a cell performs must now take account of routing decisions and vice-versa. The permutations allowed are fairly restrictive.

Function Unit Design. Shoup's cell and the Configurable Logic design both have a single bit of state but the way it is used is totally different. In Shoup's case the state is used to store the previous value of the function block output and the whole system is synchronised to a single global clock. This allows efficient realisation of the automata used in the control stores this cell was intended to implement. The routing multiplexors are further overloaded by the need to deal with two function output signals.

The combinational functions provided are also radically different from those in the configurable logic design. Firstly, one should note that there is a hidden routing cost in the function unit. Most cells will not want four input AND gates with preset terms inverted (even with a bus input signal it is hard to arrange to route four inputs to the cell) so additional control store is provided to mask out any input: 4 bits are required. This is exactly the amount of control store necessary to provide two 4:1 multiplexors to select function block inputs. Instead of being able to compute any logic function of any two inputs we can compute a specific logic function of up to four inputs and four control bits are saved. The tradeoff is certainly interesting but not as advantageous as it at first appears.

Array Crossing Busses. The output bus in Shoup's cell is a wired AND of all the cell outputs, it is chosen by an additional multiplexor over the neighbour routing outputs. This is another tradeoff of flexibility against multiplexor size: placing the output bus selector there means that a neighbour routing multiplexor will be used up to decide the output bus function. This further constrains the selection of neighbour outputs and cell inputs. The fact that only two neighbour multiplexors can be chosen makes things worse: it would seem to be much better to use a 4:1 multiplexor and select directly from two cell inputs and the sequential and combinational output functions.

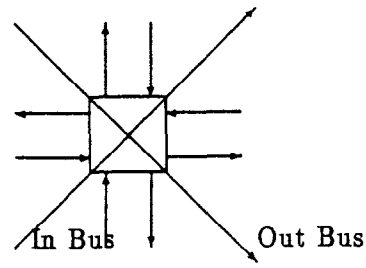


Figure 3–16: Block Diagram of Shoup's Cell.

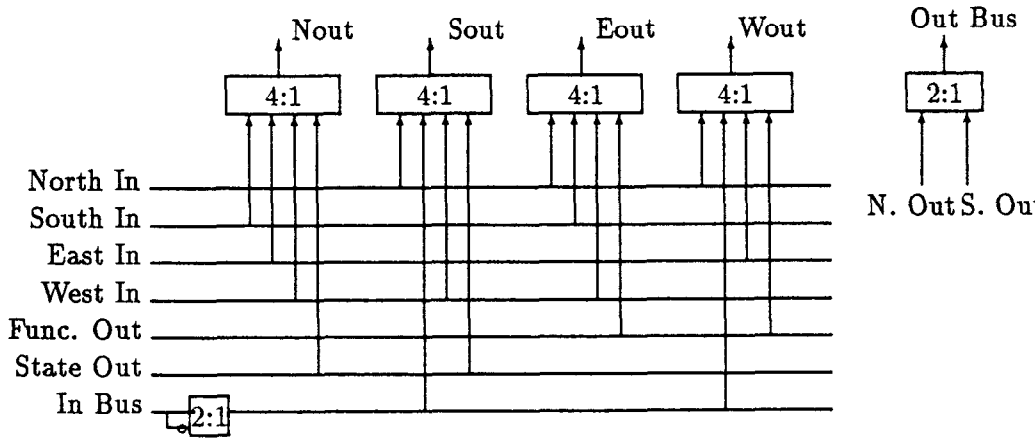


Figure 3–17: Routing Section Design of Shoup's Cell.

Diagramming. It is interesting to note that the usage of multiplexors in Shoup's cell is so convoluted that he does not attempt to diagram the actual paths being implemented within example designs. Instead the numeric values stored in the routing multiplexors are given! The ability to draw easily comprehended diagrams of cell usage in implemented systems is crucial to allow manual design using the system. The situation is made worse by the fact that no algorithm for automatic logic synthesis is provided.

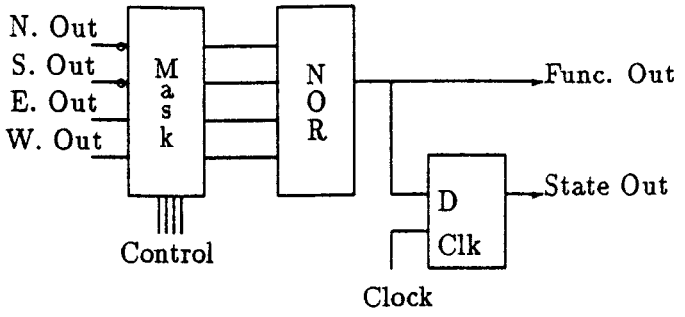


Figure 3–18: Function Unit Design of Shoup's Cell.

3.5.5 The Xilinx Programmable Gate Array.

This is an ambitious system aimed at the EPLD market. It is not clear whether it is intended to be a general system in the sense defined in Chapter 2 but its implementation favours its use to implement subsystems rather than systems. Because of the commercial nature of this design its internal structure has not been made totally public. The documentation concentrates on design using the CAD tools supplied by Xilinx and does not give a complete description of exactly what configuration possibilities exist. The diagrams below are based on those in [Xilinx86] but some changes have been made for clarity; in particular the diagram of the function unit is a merged version of two diagrams in the Xilinx documentation.

Inter Cell Communication. The Xilinx system is notable for having flexible between cell routing and fixed within cell routing. The aim is to emulate the gates-and-wiring-channels structure used in gate arrays. The layout is shown in figure 3–19: note that this does not give the whole story; additional switches are required to connect function block inputs and outputs to the wiring areas.

The wires on the Xilinx chip have their direction fixed by the control store and require the special buffering shown in figure 3–7. Instead of providing these buffers on the edge of each switch matrix the chip is divided up into 9 sections (on a 3x3 grid) and the buffers are placed on section boundaries. This means that

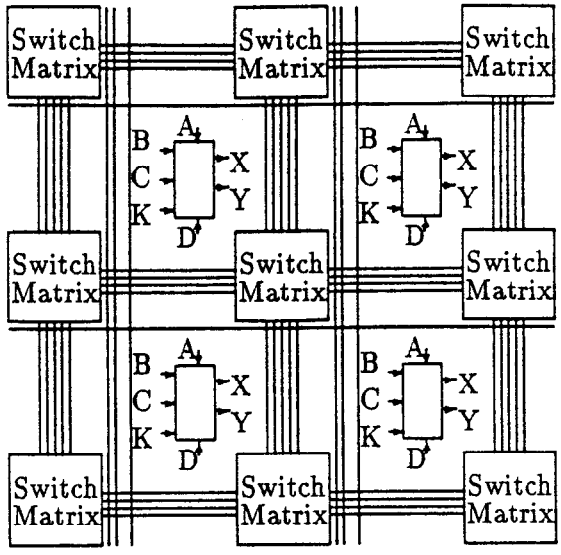


Figure 3–19: Routing Structure of Xilinx Array.

long signal paths within a single section can have insufficient buffering: such paths are flagged by the design automation system. Although this saves control store it reduces the number of routing permutations (since legal switch settings may correspond to under-buffered paths) and makes the routing problem even harder.

The design of the switch blocks is shown in figure 3–20 and table 3–3 shows the implemented connections. The implementation of the subunits is not explained in the Xilinx literature but it is a safe guess that a multiplexor like solution has been adopted because there is no mention of one connection through a subunit preventing another (blocking). This subdivided structure requires less control store than a general switch and supplies many fewer routing possibilities. Bringing signals to a switch matrix does not guarantee that they can be connected, either because there is no path through the switch or because of blocking in internal segments between the subunits. Some paths involve passing through multiple subunits presumably incurring greater delay. The Xilinx documentation suggests that each connection within a subunit involves passing through only a single transistor; this minimises switch resistance but normally involves the use of one RAM cell per switch (Chapter 4). The pass transistors in the switching units are described as

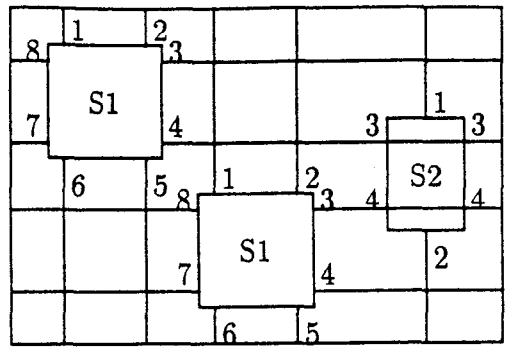


Figure 3–20: Internal Design of Xilinx Switch Matrix.

'special'; presumably processing techniques have been used to reduce V_T and hence the degradation of logic 1's so that fewer buffers are required. This should result in a useful decrease in propagation delay.

Taking account of the fact that the switch which allows $X \rightarrow Y$ also allows $Y \rightarrow X$ 45 RAM cells would be required to implement each matrix.

Within Cell Communication. Here we will consider the selection of sources for the Configurable Logic Block (CLB) inputs. There are three categories of input signal.

1. General Purpose Interconnect. CLB inputs and outputs can be connected to 'adjacent' segments of general interconnect. It is unclear whether every signal can be connected to every adjacent line but we will assume that this is the case.

2. Long Lines. There are four long lines adjacent to each cell.

   (a) Global Long Line. This line is powered by a special buffer and is intended to provide a low skew clock to the whole array. It runs vertically along each column of the array and can be connected to the B or K inputs of any CLB.

Terminal	Possible Connections	Possible Connections
	Large Switch (S1)	Small Switch (S2)
1	3,5,6,7,8	2,3,4
2	3,4,5,6,8	1,3,4
3	1,2,5,7,8	1,2
4	2,5,6,7,8	1,2
5	1,2,3,4,7	*
6	1,2,4,7,8	*
7	1,3,4,5,6	*
8	1,2,3,4,6	*

Table 3–3: Switch Matrix Connections.

(b) Vertical Long Lines. There are two vertical long lines in each column. They can be driven either by a CLB output or an IOB (I/O Block) output. One of these lines can also be driven by a second global signal on a column by column basis and can be connected to the B, C or K inputs of column CLB's. It is not made clear what the potential sources and sinks of the second long line are.

(c) Horizontal Long Line. The documentation does not explicitly state the possible sources and sinks for the horizontal long line. Examination of example diagrams suggests it can be sourced from IOB's on either side of the array and connected to A and D inputs of adjacent cells.

3. Direct Interconnect. In this case connections from cell outputs to adjacent cell inputs are made without passing through the switch matrices. These are intended for high speed local connections. Each CLB's X output may be connected to the C or D inputs of the CLB above or the A and B inputs of the CLB below it. Each CLB's Y output may be connected to the B input of the block on its right. Thus direct interconnect cannot be used to

route signals right to left through the array and there are restrictions on the sequential functions which can be performed using direct interconnect sources (e.g. you cannot use direct interconnect to provide a clock signal for the cell below).

Function Unit Design. The Xilinx function unit is the most complex of those seen to date. The multiplexors are built from trees of 2:1 multiplexors. The combinational section is shown in figure 3–22 it is composed of three main components.

1. Input Selectors. There are two input selection networks capable of selecting any three variables from the five inputs (there are $\binom{5}{3} = 10$ possibilities so 4 bits of RAM are required in each network).

2. Lookup Tables. There are two function units capable of implementing any function of three variables based on a 8 bit RAM lookup table.

3. Combining Functions. On the output of the two function generators is a single multiplexor controlled by the B input which can extend them to implement any function of four variables. This technique relies on Shannon's decomposition $f(x_0, x_1, x_2, x_3) = x_0 f(1, x_1, x_2, x_3) + \overline{x_0} f(0, x_1, x_1, x_3)$, one function unit can implement $g(x_1, x_2, x_3) = f(1, x_1, x_2, x_3)$ and the other $h(x_1, x_2, x_3) = f(0, x_1, x_2, x_3)$ and the multiplexor controlled by x_0 implements $f = x_0 g + \overline{x_0} h$. A single extra RAM cell is required to decide between implementing a single function of 4 variables or two functions of 3 variables. By selecting a different set of three input variables in one function unit from the other a small number of 5 input functions can also be implemented.

The combinational logic requires a total of 25 RAM cells to control it. Large flip-flops can also be implemented using a single cell. The extra programmability in the CLB uses 6 3:1 multiplexors requiring an extra 12 bits of RAM for a total of 37 bits (this assumes the use of the most efficient 'tree' multiplexor design (Chapter 4)).

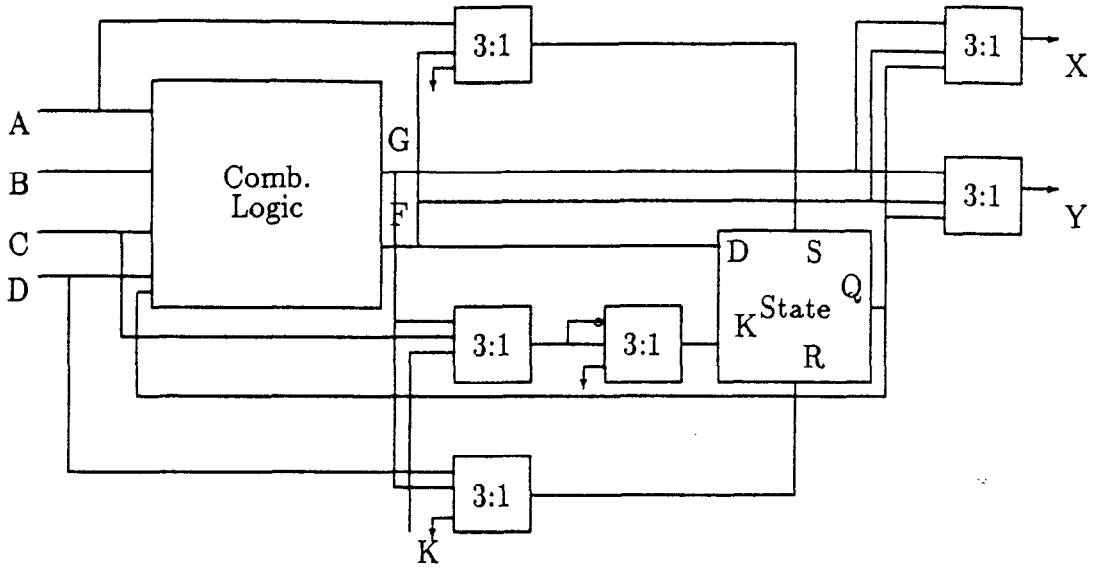


Figure 3–21: Functional Diagram of Xilinx Cell.

Input/Output Blocks. At the edges of the array of cells are special purpose Input/Output Blocks (IOB's), in the XC2064 there are 16 along the top and bottom edges and 13 along the left and right edges. A diagram of the Xilinx I/O block is given as figure 3–23. Each block can be configured for a wide variety of functions: as well as the obvious bidirectional tri-state and normal input and output pads open-collector pads can be produced by using the same signal to control TS and OUT. A global control function allows selection of TTL or CMOS input levels for all pads. Xilinx application notes gives examples where IOB's are used as Schmitt-triggers, oscillators, registers and shift-register counters. The latter applications are presumably important since otherwise a whole CLB would be required to implement each very simple one bit register. Using I/O blocks is not a complete solution and it makes resource allocation decisions even harder.

Control Store. The striking thing about the Xilinx design is just how many RAM cells are needed to control a relatively small array. The XC2064 chip has 64 cells and 58 programmable I/O blocks. The 8x8 array of cells is implanted in

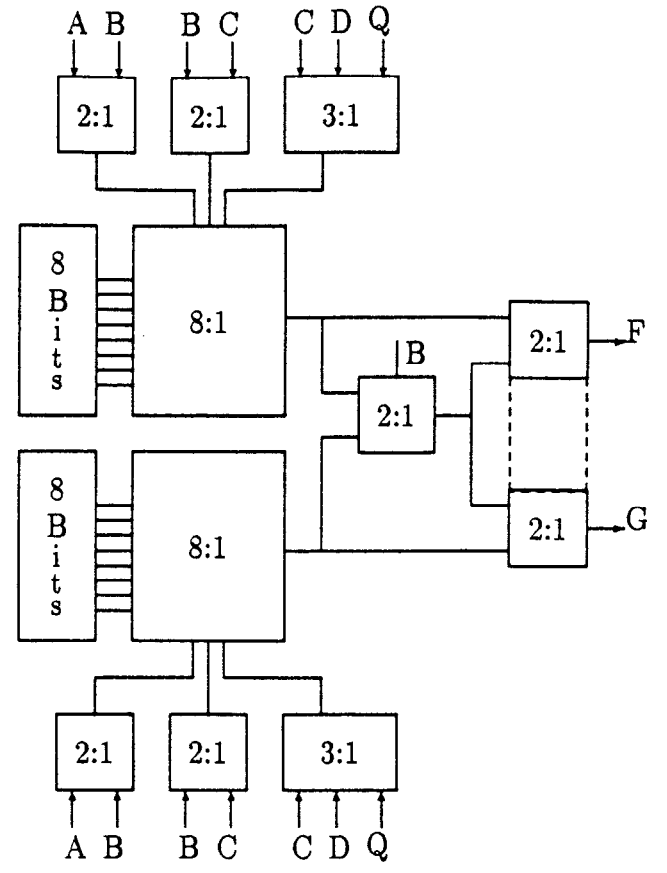


Figure 3–22: Xilinx Combinational Function.

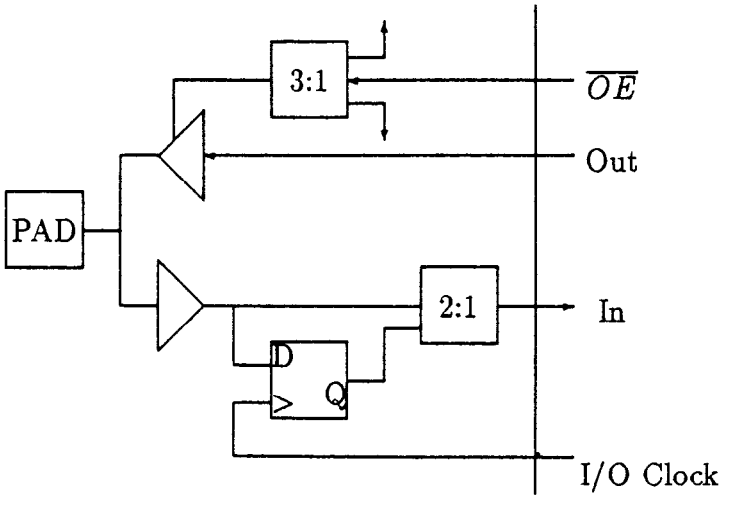


Figure 3–23: Xilinx Input/Output Block.

a 9x9 array of switching units. It requires 12038 bits of control, 290 of these are used in the I/O blocks leaving 11748 for the cells and switching structure. As a comparison a 16x16 CAL chip requires 5120 bits.

Diagramming. Designs done using the Xilinx array are presented in a format similar to printed circuit board layouts. Each block in the array is given a user specified label and the wiring structure between them is diagrammed by drawing the logical connections within the wiring channel area using standard circuit schematic conventions for crossing and connecting wires. Schematic editor like tools are provided for manipulating the designs based on this representation and an automatic router (presumably a maze router) is also available.

Development. Recently, preliminary data sheets for a new family of LCA's the 3000 series have been issued. The basic architecture remains unchanged but the following notable improvements have been made.

1. 5 Variable CLB's. CLB's can now compute any function of 5 variables or two functions of 4. Slightly more complex sequential resources are also provided. Naturally doubling the number of variables doubles the size of the lookup table.

2. Extra Routing Resources. As well as the resources described above array crossing Tri-State lines are provided.

3. Circuit Improvements. The I/O blocks now have additional configuration bits to specify electrical properties like slew rate. The configuration program loading unit has been improved and memory cells can now be cleared automatically on power on. A large number of other minor improvements have been made to programming and I/O capabilities.

4. Array Sizes. The largest LCA now has 320, 5 variable CLB's. This requires 64160 bits of configuration information. Since 5 transistor RAM cells are

used and RAM probably accounts for only about half of the silicon area the mask for this chip must be quite large even with good processing technology.

Overview. The basic idea behind this design is that most common medium sized logic functions should be implementable using one cell. The consequence of this decision is that design using these cells is best done manually so that the system is defined in terms of implementable units. There is a high 'variance' in the efficiency with which different functions can be implemented: for example a seven segment decoder can be implemented using just seven cells (specifying each output as a function of 4 input variables) with extremely high efficiency but a 32 bit latch would require 32 cells and make very little use of the resources provided. The philosophical question is whether high efficiency for restricted problem spaces given a large amount of human effort is more important than ease of design of general systems.

3.5.6 The Actel ACT Series.

Very recently, a new family of programmable gate array devices has been announced [Mohsen88]. These devices are based on a novel 'antifuse' technology (antifuse because 'blowing' the fuse switches it ON) antifuses provide low on-resistance in a very compact switching element, [Mohsen88] claims that the antifuse is about the same size as a via (via's are about $6\mu m$ on a side in $2\mu m$ technology). Naturally, addressing overheads must be added to this but the size will still be very much less than that of the equivalent structure in a RAM and transistor architecture. The Actel antifuse is an important development since previous attempts at such devices capability have required exotic processing or have been unreliable in use. The low $1k\Omega$ on resistance of the antifuse coupled with its small size allows conventional gate-array architectures to be used: the ACT1020 is implemented in $2\mu m$ CMOS, contains 186,000 antifuses and provides roughly 2000 equivalent gates. Larger 6000 gate equivalent devices in $1.5\mu m$ CMOS are also available. The wiring channels in the Actel gate arrays have 25 tracks (compare this with the 5 tracks of the Xilinx architecture): these wide wiring channels

allow most of the logic cells to be used in user designs. Speed and density (gate-count) figures for the Actel device are claimed to be comparable with small mask programmed gate arrays.

The Actel products can be expected to revolutionise the EPLD marketplace where write-once non-volatile systems are preferred: they do not, however, directly compete with the Xilinx or Configurable Logic systems in applications where it is necessary to reprogram the device many times.

3.6 Summary.

We have considered the design of the basic cell in a cellular array architecture and proposed a design which seems to be suitable for our purposes. This design has been compared with some of the most successful previous designs. The perceived advantages of the configurable logic design are its generality, symmetry and simplicity. This allows reasonably efficient implementation of a wide range of architectures. In the remainder of this thesis we will limit ourselves to considering the VLSI implementation and range of application of this design.

Chapter 4

VLSI Implementation.

This chapter will explore the implementation of CAL in VLSI. We will assume the use of a $2\mu m$ n-well CMOS process with two metal layers and one polysilicon layer: this is a typical process for ASIC designs. For an introduction to CMOS design see [Weste85,Glasser85]. General points common to all implementations will be investigated and several designs of key components compared.

4.1 Implementation of Control Store.

Obviously any implementation of a reconfigurable architecture must provide for a control store. One interesting point is that in PLD applications users often view the ability to read the control store as a positive disadvantage because it can reveal the design of their system to competitors: however, write only control memory has serious testability problems. VLSI structures for implementing control stores can be divided into four categories:-

1. Read Only. For example, Read Only Memory arrays. In this application there is no advantage to using read only memory. It is much more efficient to use fixed wires than to use switches controlled by fixed memory. Fixed wires cannot be read back to determine design information.

2. Write Once. For example, PROM arrays. Here the programming is done by blowing fusible links within the array. Again it may well make more

sense to use fusible links instead of pass transistors as switches rather than pass transistors controlled by fusible link memory. The choice is complicated by the need to route high programming voltages to the links without causing latchup in adjacent CMOS circuits and the need to address individual fuses. This technology is less widespread than normal CMOS but it is very important for EPLD applications.

There are several important additional points about fuse based systems which should be made. Firstly, there is the difficulty of testing write-once programmable systems. Users can expect some percentage of delivered chips to be defective and must write their own test vector sets. Secondly, with fuse based systems it is possible to provide a 'programming' fuse which can be blown after successful programming to isolate the control store from the inquisitive. Finally, it is worth mentioning that the design of fuse based systems has totally different ground rules from that of systems based on transistor switches because the impedance of fuses is very much lower. This makes long wires and wiring channel like structures much more attractive.

3. Write Few, Read Many. For example EPROM's and EEPROM's. This kind of control store is highly desirable in EPLD applications for two reasons:-

   (a) User Programmable. Users can reprogram the devices quickly using readily available equipment when designs are changed. Programming does not involve the manufacturer of the device so devices can be supplied 'blank'.

   (b) Non-Volatile. The information in the device is not lost when power is removed. This means that no additional circuitry is required at board level to restore the EPLD program on power up.

   (c) Testable. Since the device can be programmed many times manufacturers can fully test it before shipping.

   (d) Security. EPROM based systems (e.g. [Altera87]) can provide security for user designs using a 'no-read' bit in the control store since such a bit cannot be cleared without erasing the whole device.

Unfortunately this technology has several problems as well: programming uses much higher than normal voltages (about 25V) and special processing steps are required. Mixing these technologies with normal CMOS logic can be expected to considerably reduce the density of the array. EEPROM technology is, however, improving rapidly and it may well become very attractive as a means of implementing control store in the near future.

4. Write Many, Read Many. There are two main classes of read/write memory suggested for use in programmable applications.

   (a) Shift Register Storage. Early cellular designs with writable control stores all used shift registers. There were several good reasons for this: firstly early authors were very interested in the fault tolerance properties of cellular systems. Shift registers can be extremely fault tolerant if means are provided for altering the path through which programming information comes to a cell. A second reason was that shift registers were reasonably area efficient, it is only with MOS technology that shift register storage has become much more expensive than RAM storage. Shift registers are also more suitable than RAM for 'commutative' or self programming systems since control information can be generated in the middle of the configurable array.

   (b) RAM Storage. There is no doubt that RAM is the technology of choice for control stores today. This is because shift registers are necessarily master/slave designs containing two latches whereas RAM cell designs contain only one latch. The layout of RAM cells in array structures is also very efficient in terms of sharing contacts with adjacent cells. RAM designs can trade off area in complex peripheral circuits for very simple and small cells giving very high efficiency in large arrays.

Read/write memory can also implement program security via no-read bits in the configuration data; this is done in some of the Xilinx LCA chips. The value of this protection is questionable, however, since a readable copy of the

programming information must be kept in non-volatile memory somewhere in the system.

4.2 RAM Control Stores.

These are the normal RAM technologies familiar to all computer users. They are ideal for the ASIC prototyping and algorithm implementation applications of configurable logic but not for EPLD applications. All the work to date on configurable logic has concentrated on this kind of control store. The volatility of RAM is a major drawback to its use in EPLD applications since it requires that some form of backing store is provided. There are several ways of reducing this disadvantage.

1. Separate Arrays. Fabricate chips with two memory arrays: one EPROM and one with RAM controlling the configurable architecture. On power up on-chip circuitry automatically programs the RAM from the EPROM array. The advantage of this architecture is that the high voltages needed to program the EPROM can be kept well away from the normal CMOS logic. Both arrays can be dense and only a 'thin' communications path is required between them.

2. Two chips, one package. Place a configurable logic chip and a standard EPROM chip within the same special package. The CAL chip has special circuitry to load its control store automatically on power up. The advantage of this architecture is that the manufacturer of the CAL need not have in-house capability to build EPROMS. Multi-chip packages are now commonly used to provide large RAM arrays with low board real estate costs.

3. Battery Back Up. It is fairly common to use lithium batteries to preserve the state of CMOS logic within computers on power down within circuits such as clocks and 'system-configuration' memories and the same technique could be applied to preserve the configuration of CAL chips. If battery back

up is to be used it may be necessary to separate the power supplies for the RAM and the logical units since the most area-efficient designs of switching systems use pass-transistor logic which has a small static power consumption ('normal' fully complementary CMOS requires negligible static current).

4. Ferric RAM. Recently, systems have been developed which implement non-volatile static RAM cells using ferric storage elements. Apart from being non-volatile these systems have very similar area and performance to standard static RAM technology. As this technology matures it may well become the most attractive choice for EPLD control stores.

This section will pursue the selection of a RAM cell design for the configurable logic architecture in considerable detail. The reason for this attention is that the design of the RAM cell and multiplexor combination is the dominant factor in determining the size of a configurable logic array. Although we have separated the discussion of RAM cell design from the discussion of multiplexor design for convenience they are very tightly coupled and must be considered together when designing a programmable chip. There are two main design parameters: area of the RAM multiplexor combination and delay through the multiplexor. These criteria are significantly different from those in normal RAM cell design because RAM read and write times are not critical and the layout of the cell must allow internal state signals to be brought out to control the multiplexor.

We will now consider the choice of RAM cell design: firstly we will consider the two main classes of RAM cell design static and dynamic and deal with their basic properties, then we will consider a number of candidate RAM cell designs and look at specimen layouts for them. Based on this data we will choose the design for our configurable logic implementation and discuss what improvements could be made as the configurable logic technology matures.

4.2.1 Dynamic Designs.

These are designs in which information is stored as charge on a capacitor, eventually this charge will leak away and so it is necessary periodically to 'refresh' the storage cell by rewriting the information contained within it.

There are two important problems specific to dynamic RAM designs (figure 4–1) in this application.

Refresh Read. All DRAM designs rely on charge sharing between the storage capacitor and the bit lines to read the cell - this degrades the voltage on the storage node. The degradation is almost complete because the capacitance of the bit lines will be hundreds of times greater than the capacitance of the storage nodes in a large array. Special sensing techniques are normally required to detect the difference in bit line voltage caused by the read operation. Thus in DRAM's read is usually destructive and is immediately followed by a write to restore the stored value. Note that a whole column of RAM cells must be refreshed after every read because a whole column of RAM is connected to the bit lines when the corresponding word line goes high.

In the CAL application degradation of the stored voltage during the refresh read is not tolerable because it would result in the multiplexors routing wrongly. It should be noted that the maximum voltage passed by a pass transistor multiplexor is $V_s - V_t \simeq V_s - 0.9V$: it is important that $V_s \simeq 5V$ to ensure good performance and noise immunity in the logic circuits.

Capacitive Coupling. Another important problem is the capacitive coupling between the 'logic' circuitry and the storage nodes (figure 4–1). C_s is the storage capacitor, C_l is a parasitic capacitor associated with the controlled transistor. When the RAM cell is refreshed the value on the logic node is unknown, suppose it is high and the stored value is also high. At some point the value on the logic node may go low, the voltage on the storage node will then be divided across the two capacitors C_l and C_s. Although $C_l \ll C_s$ this is a significant problem

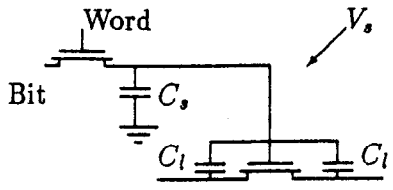


Figure 4–1: Capacitive Coupling Problem.

because the voltage on C_s is also subject to leakage. As pointed out above any decrease in V_s causes a decrease in the maximum voltage passed by the controlled multiplexor. This problem forces more frequent refreshes and further constrains the design space.

Noise Resistance. The resistance of dynamic designs to electrical noise or alpha-particle strikes is much less than that of static designs. In some EPLD applications this may be sufficiently important to preclude their use. The sort of error detection and correction codes often used in computer memory applications cannot be readily applied to a device in which RAM cell output directly controls logic.

Several methods could be used to get around these problems. In an array in which computation was synchronised to a system clock, for example, it would be possible to ensure all gate outputs and array inputs were latched during a refresh. With care this could allow the computation to be restarted after the pass transistor control voltages were restored. Use of circuit techniques (e.g. storing voltages higher that V_{dd} on C_s) to limit the effect of the voltage degradation could also be considered.

4.2.2 Static Designs.

Static RAM (SRAM) designs are different from dynamic designs in that no refresh operation is necessary. The information in the RAM cell is conserved by active elements within it.

Transistor Sizing. It will be noticed that in the trial layouts done below corresponding transistors in different designs have different widths. All the p-type transistors have been drawn at minimum width because they serve only to counteract leakage on the n-type transistor gates, for this reason they can even be replaced by very high resistance undoped polysilicon. Usually the bit-line pass transistors are also drawn minimum width to save space. In 'conservative' cell designs the pull down n-types are ratioed 3:1 over the bit line pass transistors: this removes all possibility of spurious writes and also leads to fast read times. It is also possible to use minimum sized or a 2:1 ratioing of pull downs to save area if more attention is paid to the design of the peripheral reading and writing circuitry. In the candidate designs below 3:1 ratioing has been used wherever it did not cause a large area penalty but where significant space could be saved minimum sized pull downs have been used.

4.2.3 Candidate Designs.

One Transistor Dynamic. This design uses only one transistor and one capacitor and represents the ultimate in high density memory (figure 4–2). It only provides a Q and not a $\overline{Q}$ output and so it cannot be used with many of the multiplexor designs in the next section. This design normally uses special processing to increase the storage capacitance and requires complex peripheral circuits for reading, for this reason it is not suitable for the present CAL chip. The gate capacitance of the controlled multiplexor transistors will provide at least part of the required storage capacitance allowing even greater density.

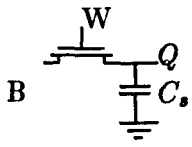


Figure 4–2: One Transistor Dynamic RAM.

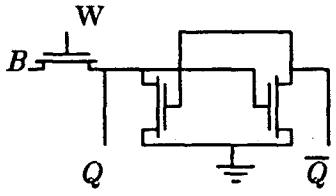


Figure 4–3: Three Transistor Dynamic RAM.

Three Transistor Dynamic. This design (figure 4–3) is an interesting alternative to the three transistor static RAM. It provides the same function in slightly less area and with no special processing but requires more complex peripheral support circuits. The two layouts (figures 4–4, 4–5) illustrate the potential benefits of bringing out only one state signal. The second layout could be done much more compactly if 45° degree lines were available, buried poly-diffusion contacts would also be useful.

Six Transistor Static. This is the textbook static RAM design (figure 4–6). It is electrically simple compared to the other RAM designs and very noise resistant. Resistance to alpha-particle and power supply noise can be critical in EPLD applications. This design is popular in ASIC cell libraries because it requires no special processing or circuit tricks although it is not generally used for commodity RAM chips. Area is quite high because of the two bit lines and p-type transistors. This means using an n-well within the array and the spacing rule between n-wells and n transistors represents a large area overhead. There are two common layouts for this cell with one (figure 4–7) or two word lines (figure 4–8).

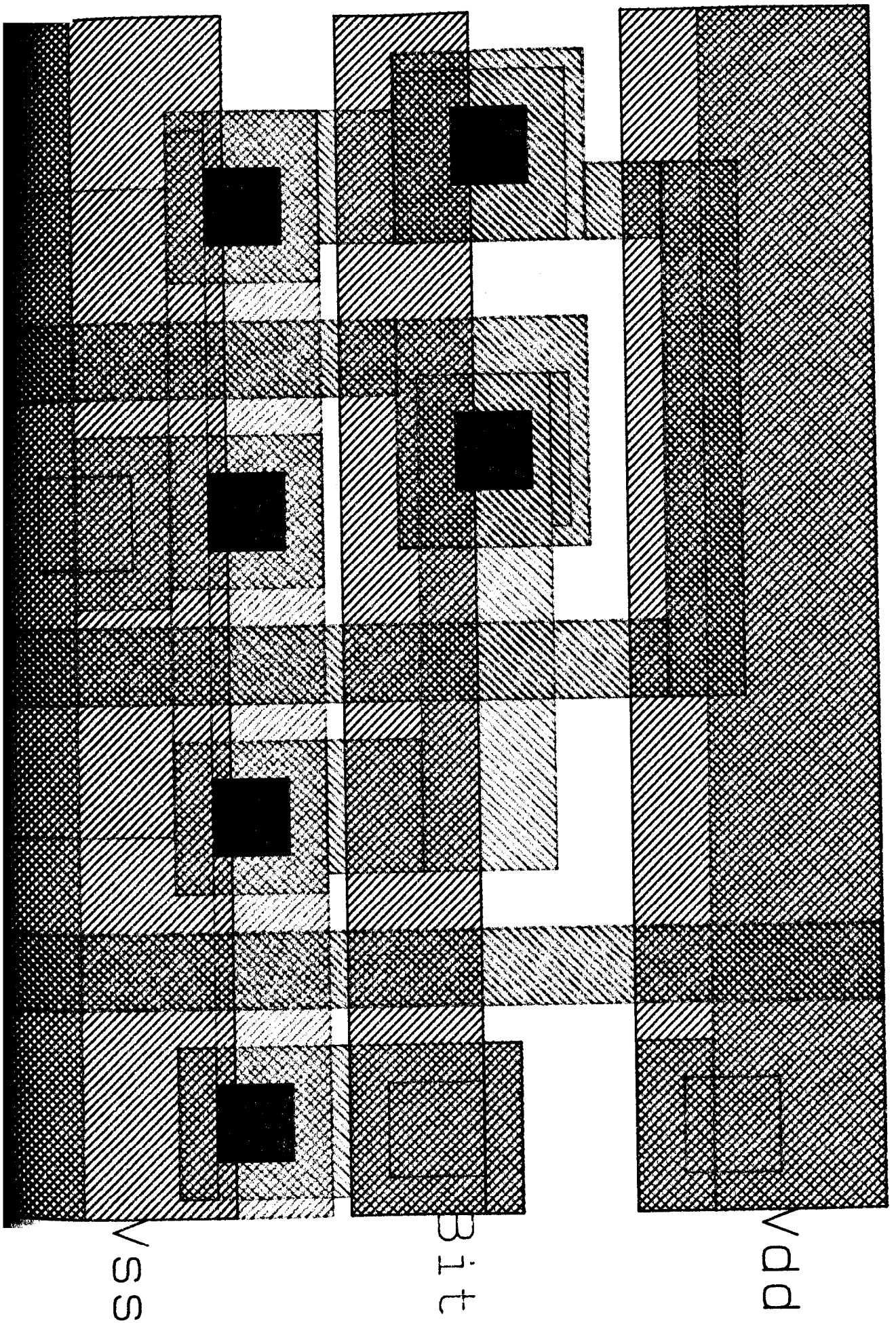


Figure 4–4: Three Transistor Dynamic RAM Layout (a).

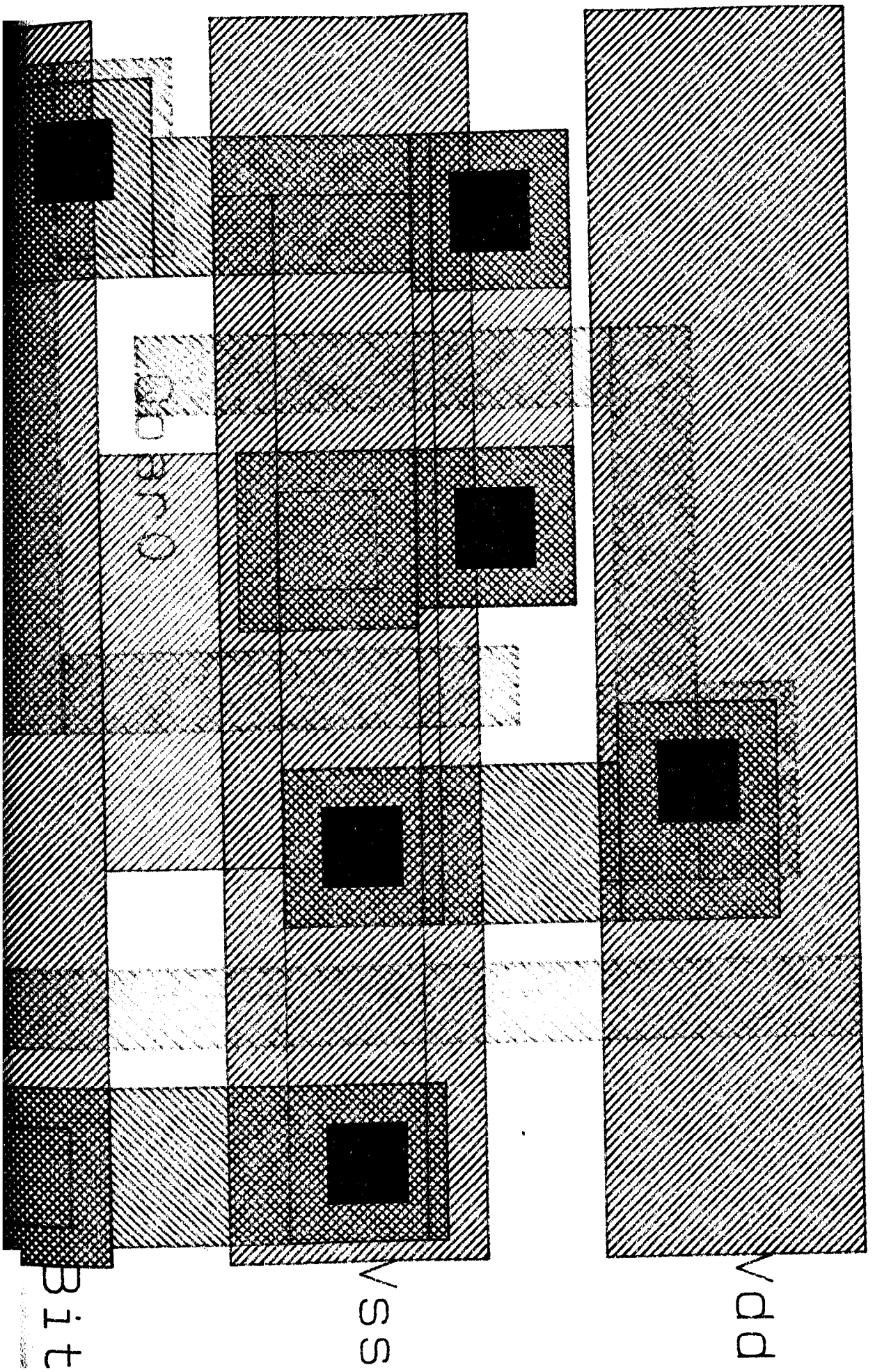


Figure 4–5: Three Transistor Dynamic RAM Layout (b).

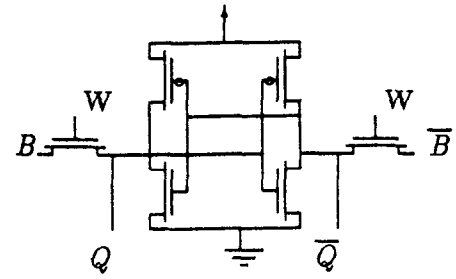


Figure 4–6: Six Transistor Static RAM.

Five Transistor Static. As previous design but with only one bit line and one word line (figure 4–9,4–10). This design is used by the XILINX programmable gate array [Xilinx86]. The use of a single bit line requires that word line be bootstrapped above Vdd during writes. The cell is now only being 'pushed' from one direction rather than 'pushed' and 'pulled' and bootstrapping is necessary for the problem case where $Q = 0$ and $B = 1$. Raising the word line voltage to $V_{dd} + V_t$ prevents degradation of voltage level by the pass transistor. This design reduces the RAM cell width by about a quarter over the six transistor case.

Three Transistor Static. This design replaces the p-channel load devices with undoped polysilicon resistors on a second polysilicon layer and uses buried contacts for poly-diffusion connections (figure 4–11). Both of these steps require non-standard processing. This is the design used by many commodity SRAM's and can be as little as 30% the size of the normal 6-transistor one (for example, Motorola's 64K SRAM uses this technique resulting in a storage cell size of $16.6\mu m$ by $11.4\mu m$ with $1.5\mu m$ rules). Connoisseurs of VLSI layout are referred to [Electronics86] for a plot of this cell.

Much of the improvement comes from the fact that with no p-channel devices no well and hence no well taps or substrate contacts are required. If we also use only one bit line then we have a three transistor static cell which is not much bigger than the dynamic version but needs no fancy peripheral circuitry. This is the optimal design for CMOS Static RAM's. The densest layouts cannot be used

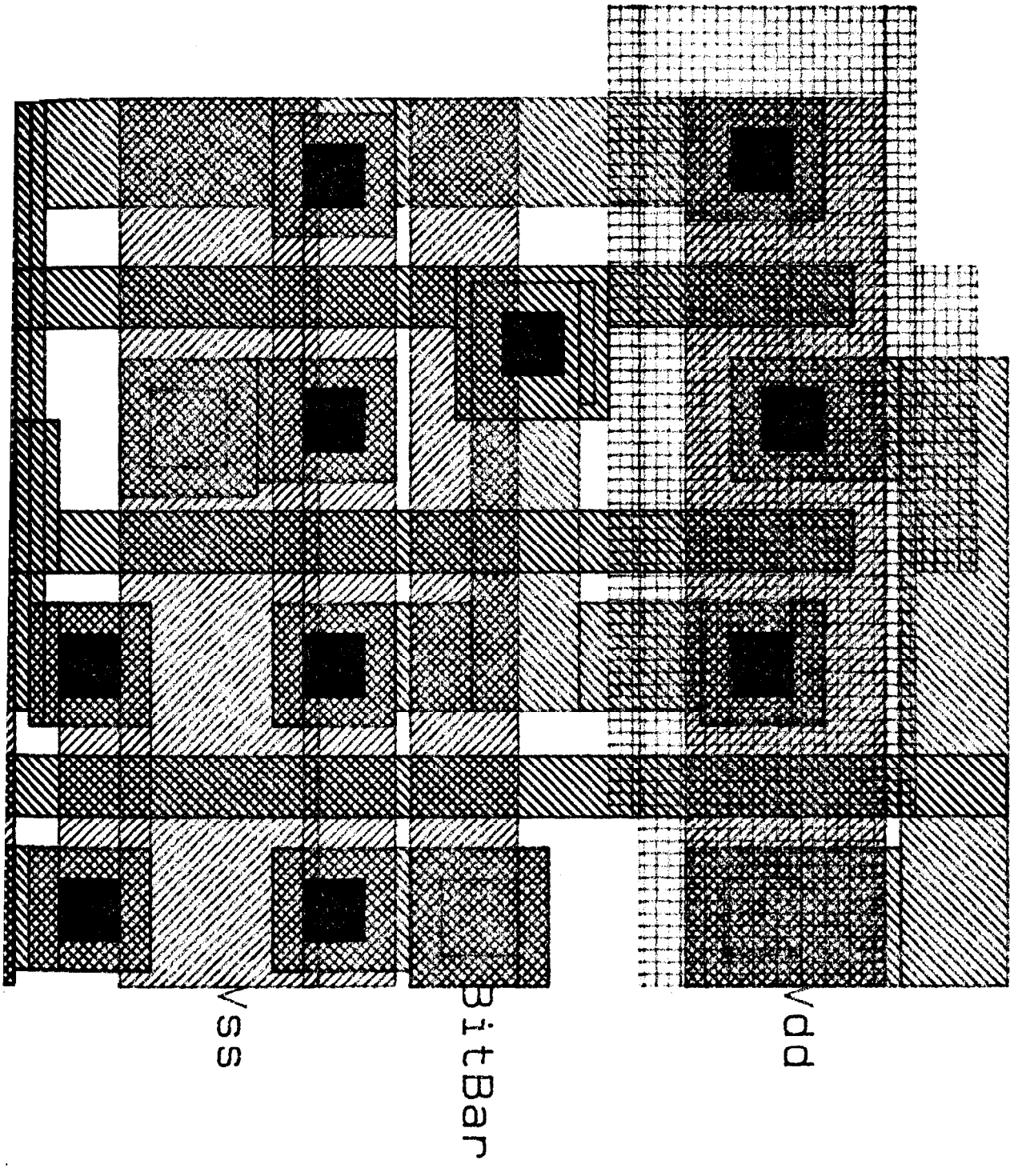


Figure 4–7: Six Transistor Static RAM Layout (a).

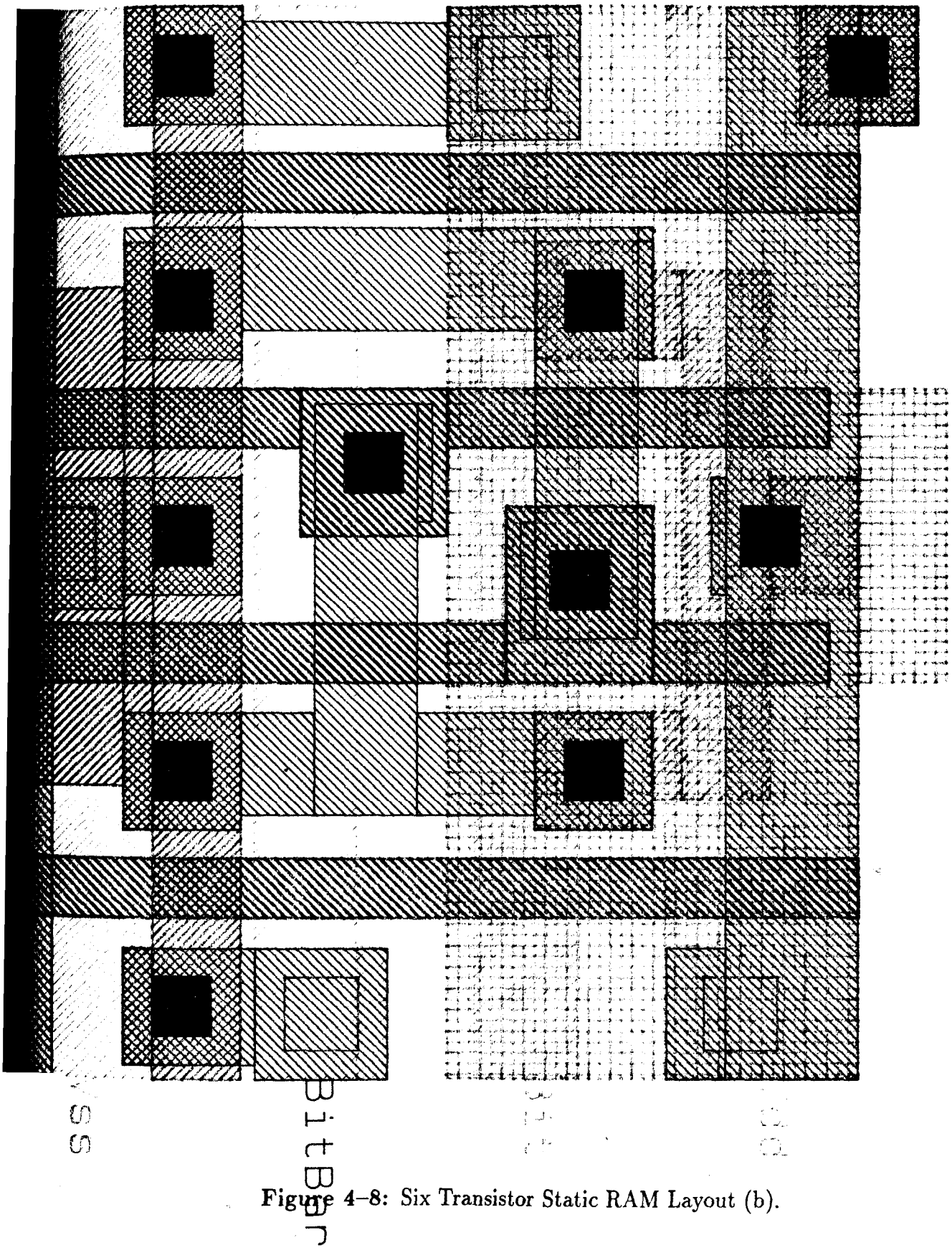


Figure 4–8: Six Transistor Static RAM Layout (b).

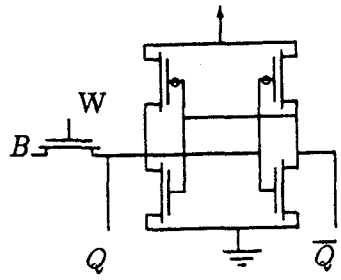


Figure 4–9: Five Transistor Static RAM.

Type	Trans.	Word Lines	Bit Lines	Outs	X (μm)	Y(μm)	Area (μm^2)
Static	6	2	2	Q,$\overline{Q}$	36.5	34	1241
Static	6	1	2	Q,$\overline{Q}$	29	37.5	1087.5
Static	5	1	1	Q,$\overline{Q}$	29	33	957
Dynamic	3	1	1	Q,$\overline{Q}$	31.5	25.5	803.25
Dynamic	3	1	1	Q	31.5	23.5	740.25

Table 4–1: RAM Area Comparison.

in configurable logic applications because they do not allow easy access to the Q and $\overline{Q}$ outputs.

There is some doubt as to whether this design will continue to be optimal as technology improves. It is possible that with submicron design rules and power supply voltages reduced to 3.3V (this is the JEDEC standard voltage for submicron VLSI) it may be necessary to go back to p-type devices as loads and the normal 6 transistor design to obtain sufficient noise immunity [Lineback86].

4.2.4 Design Choice.

There are several considerations which imply that RAM designs for control store applications can never be as dense as those in all memory arrays:

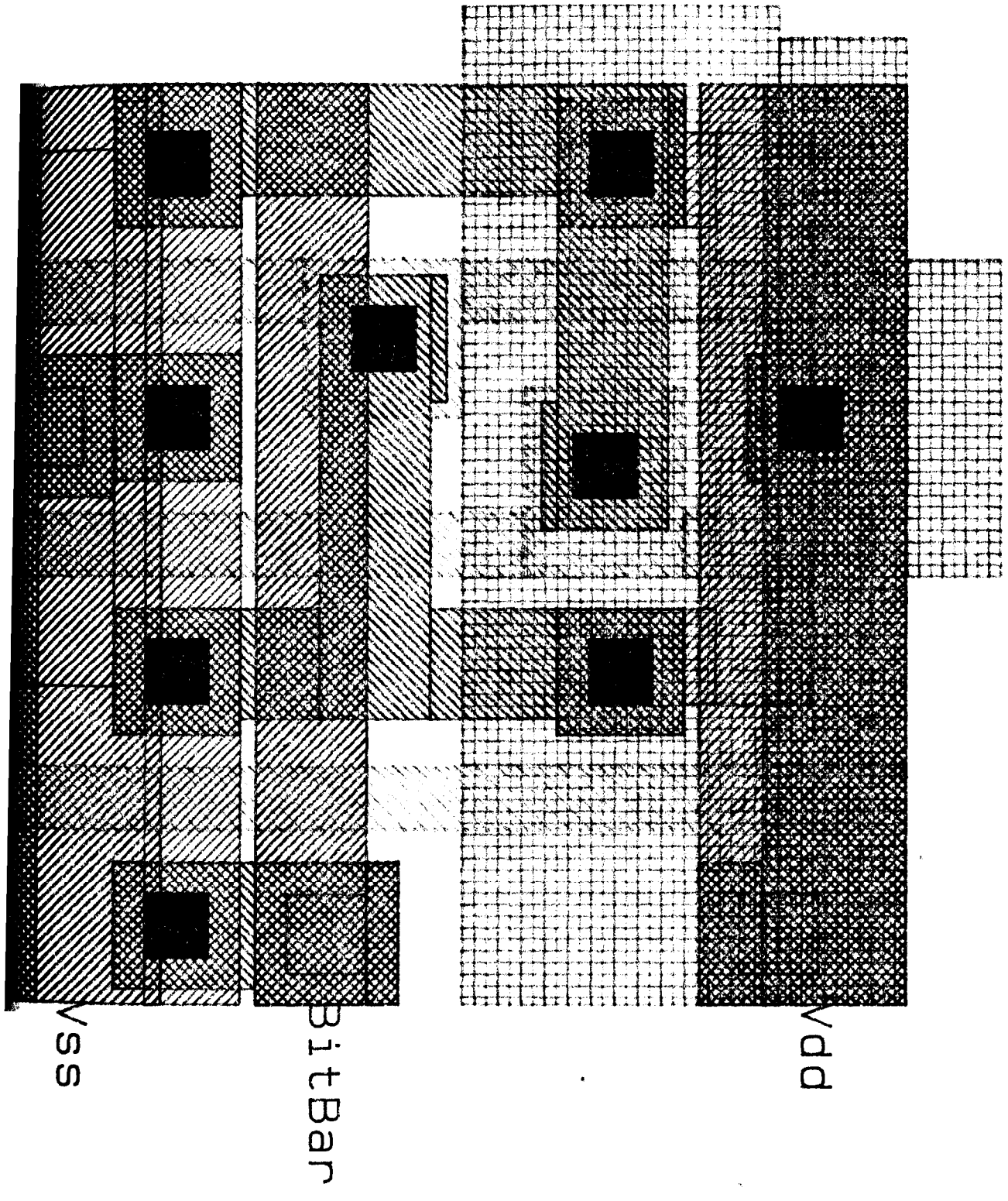


Figure 4–10: Five Transistor Static RAM Layout.

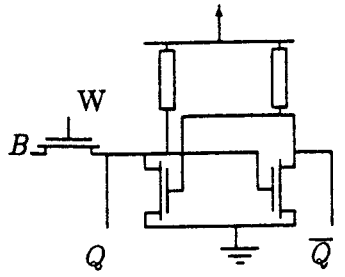


Figure 4–11: Three Transistor Static RAM.

1. Power Supplies. In very dense memory arrays it is common practice to distribute power (and sometimes ground) on polysilicon or diffusion layers. This can considerably increase density since the size of a RAM cell will otherwise often be forced by metal spacing. This technique is only practical because of the very low power consumption of memory arrays: metal power lines are needed in a programmable structure to supply the logic elements. The metal lines must be quite wide to cope with worst case dynamic power consumption.

2. Access to State. When the RAM is used to control a switch it is necessary that Q and possibly $\overline{Q}$ are brought out of the cell. This breaks up the array structure and reduces density. One reason for using a design in which one RAM cell controlled one switch would be that only Q would have to be brought out and the resulting increase in the number RAM cells would be compensated for by a smaller RAM cell design. In the case of the 6 transistor cell there is no size advantage to only bringing out one signal but in the three transistor design the size reduction is significant.

3. Substrate Contacts. In a dense RAM array one can get away without as many substrate and well contacts as one needs in a logic circuit provided that the periphery of the array is well covered. In the 3 transistor design where only n-type transistors are used they are unnecessary within the array. This is important because the spacing rules between diffusion and substrate contacts are quite large and the contacts themselves waste space.

Despite these disadvantages there is no doubt that the density of the control RAM could be significantly improved by using the sort of special processing used in commodity SRAM chips. There are three main areas where this could help.

1. Wells. The rules about containment of p-transistors within a well and separation of n-transistors from the well edge can add up to $10\mu m$ of wasted space, this is about two metal pitches. Processes which reduce this gap (e.g. by deliberately sacrificing yield to loosen design rules) or provide high resistance devices as pull-ups so only one kind of transistor is needed are very advantageous.

2. Buried Contacts. In standard CMOS it is not possible to connect directly between diffusion and polysilicon layers. Instead two contacts (metal-diffusion and poly-metal) each approximately the size of a buried poly-diffusion contact are required and there are spacing rules about how close together these may be. This also wastes space on the metal plane which could otherwise be used for through signals. The cross-coupled structure of the RAM cell can benefit greatly from direct connections.

3. Two Polysilicon Layers. This is used with the poly-resistor design to allow the resistors to be fabricated on top of the n-type devices further reducing area.

There is no doubt that the size of the basic RAM cell could be reduced by at least a factor of two using the techniques above. Use of forty-five degree lines is also important for several RAM cell designs.

For the prototype design only standard processing was available so the choice was between the five transistor single word line design and the six transistor double word line design. The double word line design while being slightly larger allows more sharing between adjacent cells, is electrically less complex and fits in better with the multiplexor layout discussed in the next section so it was selected.

4.3 Multiplexor Design.

As described in Chapter 3 the functionality of the basic cell can be implemented by a number of multiplexors controlled by RAM cells. Now that we have decided on the design of our RAM cell we must consider the layout of these multiplexors. Our cell design calls for 2:1, 4:1 and 8:1 multiplexors, we will deal with the design of the 4:1 multiplexor only. Designs for the other sizes of multiplexors used can be derived in a straightforward way although some of the designs illustrated below cannot be directly extended to 16:1 or larger multiplexors (because of limitations on the number of pass transistors between buffers).

One optimisation which can be used is to have inverting rather than non-inverting multiplexors. This means that signals which are routed through an odd number of cells will be inverted when they reach their destination. The logic units can cope with inverted input signals simply by changing the logic function being implemented and, except in pathological cases where an extra cell function unit could be required, functions can be chosen so that off chip signals have the correct polarity. These translations can be performed automatically by the program which produces configuration data for CAL's and hidden from users.

4.3.1 Candidate Designs.

Several multiplexor designs were considered:-

Full CMOS Complex Gate Multiplexor. This design (figure 4–12) was quickly ruled out because of its size. Note that because the pull-down and pull-up paths can pass through three transistors to get the same driving capability as an inverter built from minimum sized transistors all the transistors have to be three times as wide. It could make more sense to use minimum sized transistors within the gate and add a normal inverter to buffer the output.

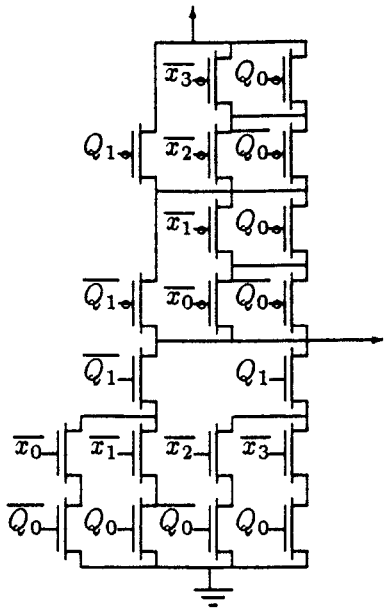


Figure 4–12: Full CMOS Multiplexor Circuit Diagram.

Mostly NMOS Complex Gate Multiplexor. This design (figure 4–13) replaces the PMOS transistors with a single p-channel pull-up which could be put in the same well as the RAM pull up transistors. Although it is less than half the size of the full CMOS multiplexor it is still unacceptably large.

Transmission Gate Multiplexor. This design (figure 4–14) implements the function as a switching structure: i.e. inputs are connected to sources and drains,

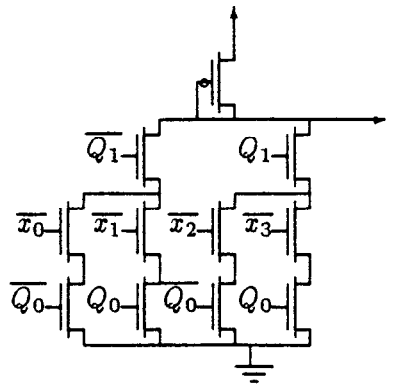


Figure 4–13: Mostly NMOS Multiplexor Circuit Diagram.

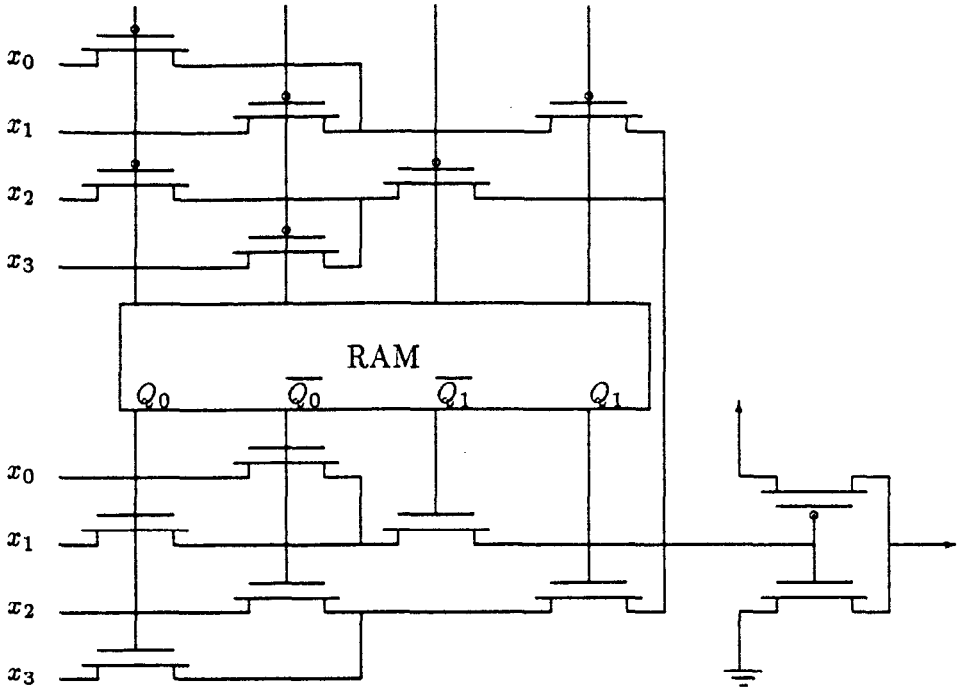


Figure 4–14: Transmission Gate Multiplexor Circuit Diagram.

not gates. It has an advantage over pass-transistor designs in that transmission gates do not compromise logic levels. The need for true and complement control signals is not a major disadvantage because the RAM can supply both. The problem is the number of transistors required and the fact that half must be in one well and half in another, the circuit diagram is intended to suggest an appropriate layout with n-type and p-type transistors grouped together and the RAM cells central.

Pass Transistor Multiplexor. This design (figure 4–15) is extremely compact while still providing high performance. Of course, because only n-type transistors are used there is a problem with reduced logic 1 voltages. Provided that the depth of each multiplexor before buffering is limited and the chip power supply system ensures that there are no large degradations in supply voltage in the middle of the array this is perfectly acceptable.

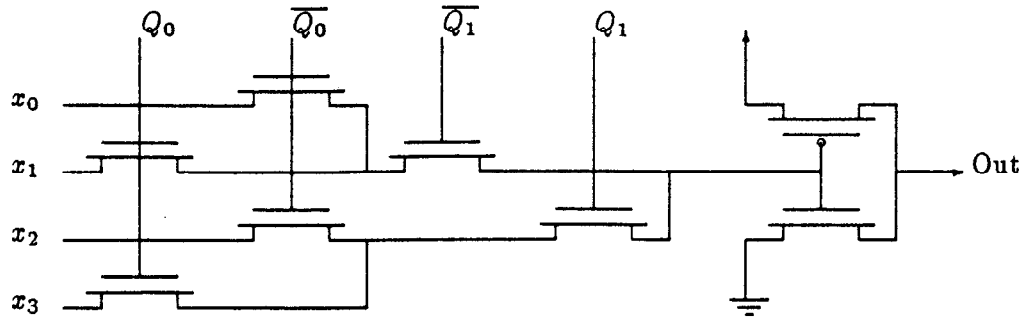


Figure 4–15: Pass Transistor Multiplexor Circuit Diagram.

A layout for this multiplexor is given in figure 4–16.

RAM Cell/ Switch Multiplexor. The 'tree' designs of pass gate multiplexor all have multiple switches along the path between inputs and buffers. This reduces performance because of the extra resistance. One could get away with a single transistor on each input if a $lg(n)$ to n decoder was placed between the RAM and the controlled transistors. Unfortunately, this would require too much area. A better idea for small multiplexors is to use n RAM cells: one per switch (figure 4–17). This implies that the efficiency is no longer increased by having a number of inputs which is a power of two. The overhead is not quite as bad as it seems because denser RAM cell layouts can be used when only one state signal needs to be brought out. This design would be very suitable for a configurable logic chip which used 3 transistor CMOS or 1 transistor dynamic RAM cells. There a considerable area penalty if standard 6 transistor RAM's are used.

This multiplexor has one important problem not shared by the other designs: not all possible RAM values correspond to legal routing permutations. In some cases contention between input sources will result causing static power dissipation. The problem is significant at power-on when there are random values in the control store and during refreshes if a dynamic RAM is used. The problem can either be accepted or global signal can be provided to switch the multiplexor outputs to

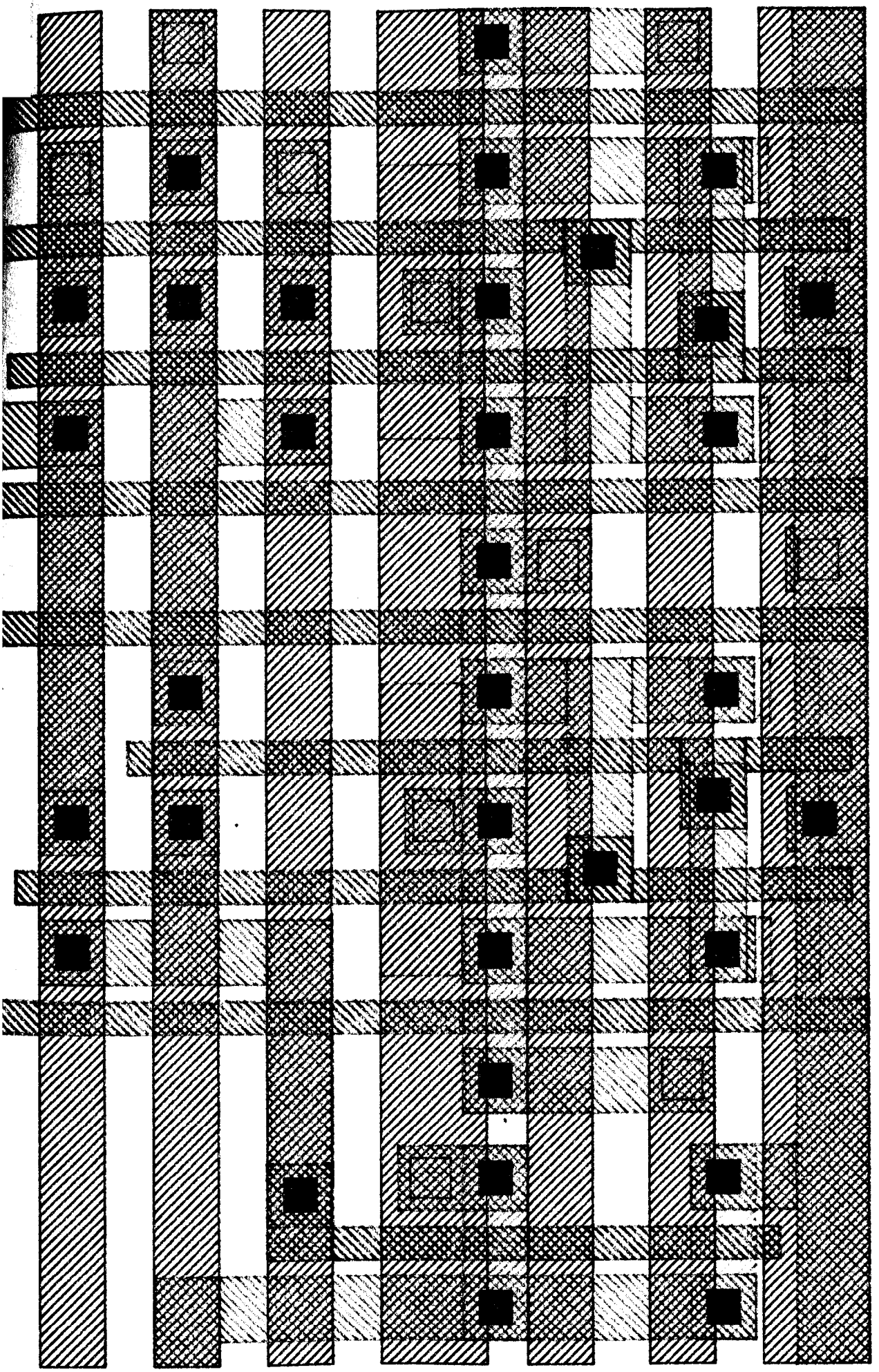


Figure 4–16: Pass Transistor Multiplexor Layout.

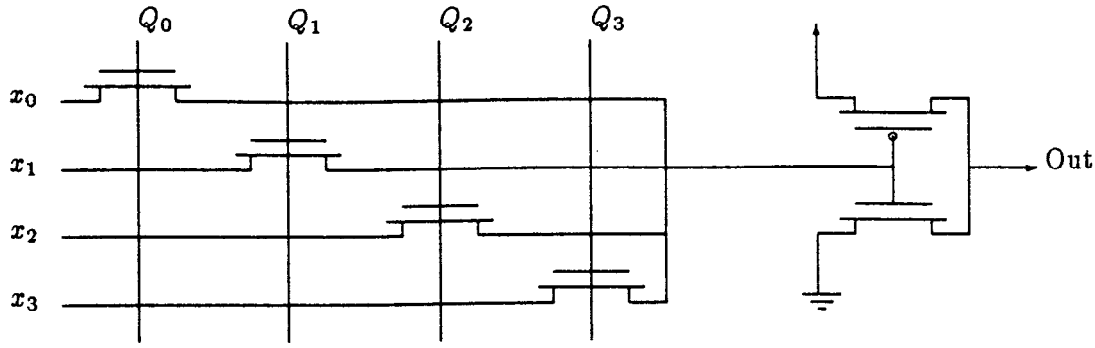


Figure 4–17: RAM Cell/ Switch Multiplexor Circuit Diagram.

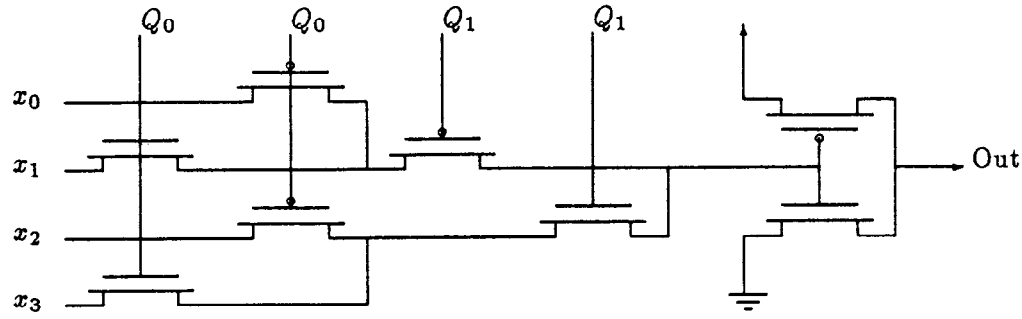


Figure 4–18: Mixed Pass Transistor Multiplexor Circuit Diagram.

a high impedance state: this is especially easy if clocked multiplexors are being used.

Mixed Pass Transistor Multiplexor. At first glance, it appears that one could get the best of both worlds by using both n-type and p-type transistors in the pass transistor tree (figure 4–18). In this case only Q would need to be brought out from the RAM. Unfortunately, this design has serious electrical problems: any path through the tree which involves both n-type and p-type transistors will have both the logic zero and the logic one voltages degraded. This compromises noise margin to a dangerous degree. The lower conductance of the p-type transistors requires wider transistors to produce equal performance and the need for an n-

well will cause a significant area penalty. Although the 4:1 mixed multiplexor is problematic, 2:1 mixed multiplexors can be extremely useful if care is taken. They are used, for example, in the input selection network of the Xilinx LCA combinational function.

4.3.2 Performance.

Perhaps the most important performance parameter for a CAL is the delay and bandwidth of the routing multiplexors. This was simulated using the worst case loading configuration where every multiplexor in the next cell selects the output of this cell (e.g. if we are talking about a NORTH output multiplexor then the North,West,East,X1 and X2 multiplexors in the cell to the north all select South). The results are given in table 4–2: the times are between the input step waveform going through 2.5V and the output of the heavily loaded multiplexor following it. Note that since the multiplexors are inverting the rise time figure corresponds to a falling input signal. It is important not to read too much into these figures because of problems with the simulation program used (Chapter 5): convergence could not be obtained under identical conditions for all the circuits.

The simulations were done on hand coded circuits (not circuits extracted from actual layouts) and used worst case process parameters: for these reasons direct comparison with performance figures for the Xilinx LCA architecture would be unsafe. Such comparison could only be done fairly using measurements on fabricated chips.

It is worth pointing out an additional advantage of using inverting multiplexors to route between cells. Normally the rise time of signals in CMOS will be longer than the fall time (because n-type transistors are more conductive). Using double width p-type transistors can help but it increases the load capacitance (since each multiplexor drives 4 similar multiplexors in an adjacent cell) thus marginally increasing the fall time and requires more area. When considering the worst case delay of signals through a long chain of non-inverting multiplexors one would have to consider a rising signal whereas with inverting multiplexors one must consider

Name.	Fall Time.	Rise Time.	Notes.
Full CMOS	2.2ns	2.5ns	Unit n, Double Width p
Mostly NMOS	1ns	5.6ns	Triple width n, Double Length p
TxGate	3ns	2.7ns	Double width p-type
Pass Transistor	2.4ns	3ns	Unit n and p type
RAM Cell/ Switch	2ns	2ns	Unit n and p type

Table 4–2: Multiplexor Delays.

the average of the rise and fall times. Thus the delay through a long chain of inverting multiplexors will be approximately the same for high going and low going signals. For this reason it was decided to make both n and p type transistors in the inverter minimum size.

Summary. Of all the multiplexor designs studied the one RAM cell per switch design has the best area and delay characteristics: however suitable RAM control stores require special processing. Of the other designs the pass transistor multiplexor is the clear favourite because of its efficient use of control store and small size. Its performance is also competitive with the less area efficient designs. Another advantage of this design is the natural way in which the logic circuits can be mixed with the control store in a dense array using the normal CMOS layout style.

4.4 Cell Design.

Given our design for the basic 4:1 multiplexor and the cell design outlined in the last chapter how do we build up a VLSI layout for a whole Configurable Logic cell?

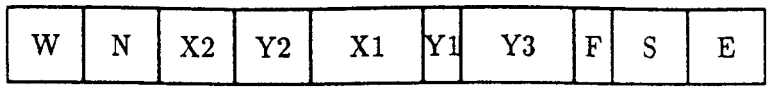
Several possible floorplans are given in figure 4–19: the labelled boxes implement the configurable logic function and routing multiplexors described in the last chapter (figures 3–9, 3–10). There were three main criteria used in selecting the floorplan.

1. Wiring Space. The RAM cell bit lines and power lines are running horizontally along rows from left to right on metal 2 and the word lines are running vertically on polysilicon. This means that it is much easier to have wiring areas running left to right than top to bottom since it is hard for metal wires to cross the rows of RAM cells. This suggests using the one row or two row floorplans.

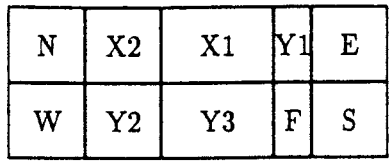
2. Aspect Ratio. It is important to maximise the ratio of the 'area' (number of cells) to 'perimeter' (number of I/O connections) to reduce pin count. This favours near square basic cells and militates against the choice of the one row design.

3. Wire Length. For performance reasons one wants to keep the longest wire within a cell to a reasonable size. It is also desirable that North to South and East to West signals traverse approximately equal distances to keep the delays approximately the same.

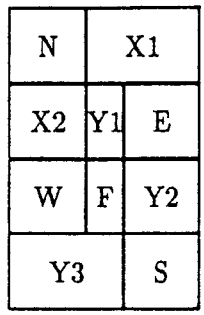
For these reasons the two row design was chosen. At the top and bottom of the cell are strips containing the controlling RAM and multiplexor output buffers. These face a central area containing the pass transistor logic and wires which implement the routing and logic function.



One Row Organisation.



Two Row Organisation.



Four Row Organisation.

Figure 4–19: Basic Cell Floorplan.

The two row organisation of the cell groups n-type transistors in the middle and p-type transistors along the top and bottom edges. This allows well sharing between adjacent cells. Designs which did not group transistors of the same kind together would pay a heavy penalty in area because the design rules force a large spacing between different transistor kinds.

In order to realise the 20 functions 6 bits of RAM are used to control the function block. The longest path from a cell input through the function block to a cell output is through 3 multiplexors. The multiplexors in the function block are much less heavily loaded and therefore faster than the routing multiplexors giving a worst case delay between a cell input change and a consequent change in cell output of around 10ns.

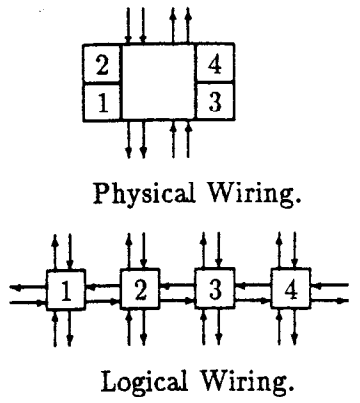


Figure 4–20: Logical/Physical Wiring Scheme.

4.4.1 Aspect Ratio.

One of the main challenges in the design was to get a logically square array of cells in a physically near square chip. The two row design has an aspect ratio of about 3.5 to 1. It was very difficult to see how to change the layout to improve the aspect ratio without sacrificing large amounts of area because the layout is constrained by the RAM design. The solution chosen was to build a rectangular array of cells with four times as many cells in the vertical as the horizontal direction: this results in a near square chip after extra peripheral circuits in the horizontal direction (for RAM programming and global signal buffering) are counted in. These cells are then wired up with groups of two cells vertically connected as if they were separated horizontally (figure 4–20): this results in a logically square array. This requires small wiring channels areas at the left and right of the basic cells to implement the more complex structure. These wiring channels were done by hand using all three wiring layers and result in a minimal increase in total area (initially automatically generated routing was tried but it proved to be extremely inefficient - the router attempted to place all vertical signals on one metal layer and all horizontal ones on the another and wasted a large amount of area doing unnecessary layer changes).

4.5 Input Output Blocks.

In this section we will consider the design of Input/Output (I/O) circuits for the configurable logic architecture. Firstly, we will consider the reasons for providing special I/O and secondly we will consider appropriate designs for CAL's which are to be used as stand-alone EPLD's and as components of larger arrays.

There are three main reasons for providing special I/O logic.

Pin-Count. There can be a large mismatch between the number of wires that can emerge from the edge of a VLSI array architecture and the number of wires that can be connected to pads. The total number of inputs and outputs in a CAL array grows as $8n$ where n is the dimension of one side of the array, the number of cells in the array grows as n^2 (we will consider only square arrays since they have the best ratio of perimeter to area - CAL designers can avoid building rectangular arrays). The largest package readily available when the prototype was designed had 144 pins, although packaging is improving rapidly and the latest LCA chips use 175 pin packs [AMD88]. This means that after taking into account necessary overheads like power and ground and control signals the biggest size of array which can be accommodated with all I/O signals going off chip has 16 cells on a side. This is a serious problem since one could have a 32x32 array if only silicon area needed to be considered.

Power Consumption. The driver transistors in pads have length to width ratios hundreds of times larger than the minimum size transistors in the centre of the array because pads have to drive highly reactive loads and therefore need to source and sink relatively high currents. CMOS pads are rated at between 1 and 20mA. These currents are not just arbitrary maxima but will actually flow every time an output changes as the external capacitance is charged or discharged. This means that even if a package with enough pins were available the potential

power consumption of a chip with 128 (=32x4) output pads would require careful consideration.

The number of pads required does not grow simply with the number of inputs and outputs since it is necessary to add additional power and ground pads for every $n, (n \simeq 8)$ output pads simply to cope with the demands of the pads themselves.

Interface. Another important reason for having separate I/O blocks is to interface with other systems. For example, it may be desirable to have some bidirectional pads at the edge of the array to allow the programmable device to be connected to a Tri-State bus. This consideration is more important in less general EPLD architectures where special latches for input and output signals are often provided.

4.5.1 The CAL EPLD Approach.

EPLD's are generally used as stand-alone devices so one wants to put as large an array as possible on one chip. It is also desirable to use low pin count Dual-In-Line (DIL) packages to reduce costs and board real estate. This implies a high degree of multiplexing at array edges. The CAL philosophy is that special purpose I/O blocks should be avoided if at all possible because they add a different kind of entity to the system. This complicates everything from the chip design itself, to the CAD system to the user's conceptual view of the architecture. I/O blocks tend to be irregular and ugly things - the kind of design where users must look at a manual to see if a particular situation can be coped with.

It was decided, therefore, to use the routing network within the array cells to provide the multiplexing required. At the moment it is necessary to share one bidirectional pad between two inputs and two outputs to match up the capabilities of the silicon and the packaging. Cells are grouped in pairs: the first cell's output drives the output data line to the pad and the second the enable line to the pad. The input from the pad is connected to the input lines of both cells. A diagram of this arrangement is given as figure 4–21. Normally, the function units of the

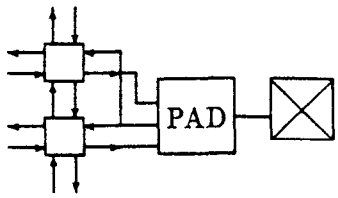


Figure 4–21: CAL Pad Multiplexing Scheme.

peripheral cells connected to the pads will be used for I/O functions (e.g. inverting or latch output signals or using the constant 1 and 0 functions to make the pad a dedicated input or output).

The CAD software will handle the I/O problem as follows: the user will design his system as an array of cells smaller than that available on the chip and the software will attempt to embed his design in the physical cell array so that all I/O signals can be routed to the pads. In some cases this will not be possible and the program will give an error message.

4.5.2 The CAL Array Approach.

This CAL design has a very different I/O requirement from the EPLD: in order to build up a large array from many smaller single chip arrays all inputs and outputs from each smaller array have to be available externally. This seems to limit the size of the arrays which can be fabricated as a single chip size unit to sixteen by sixteen cells. This is indeed the size of the first CAL chip designed for this application.

Multiplexing several inputs or outputs onto the same pad transparently to the user is quite attractive in the CAL architecture and need not lead to excessive performance losses. Even general purpose CMOS pads taken from ASIC libraries can switch fast enough so that the pad-propagation delay between two cells on neighbouring chips is about the same as that through 3 routing multiplexors. In a catalogue part CAL design carefully designed pads and appropriate packaging technology coupled with the fact that pads need only be capable of driving a

single input on an adjacent chip could further reduce the delay. Most paths in user designs will go through tens of routing multiplexors and several levels of gates so the pad delay is relatively minor. Transparent scaling of the CAL architecture across chip boundaries is possible using a scheme which takes advantage of two other factors:

1. Most Potential Connections will be Unused. Most designs will use only a small fraction of the available connections at the edge of the array but the connections used will vary from design to design. By including some extra routing area at the chip edge the degree of multiplexing required in most designs (and hence delay in crossing chip boundaries) can be reduced.

2. Used Connections will change state relatively Infrequently. We have seen that the pads can switch much faster than the average signal in a user design. Many user signals will switch very infrequently. For this reason a scheme which transmits *changes* in user signals rather than *values* of user signals is potentially much more efficient. An important side effect of this scheme is a reduction in power-consumption: in a traditional time-share multiplexing scheme if two output signals have different constant values the shared pad would have to change state every cycle using more power than an implementation with separate pads for both signals.

Naturally, such a scheme would cause an increase in the complexity of peripheral circuits where CAL must interface with other logic families: one way around this problem would be to allow the specialist I/O circuits on a given side the CAL chip to be programmed to select a traditional EPLD-like multiplexing scheme rather than the specialist scheme for CAL to CAL communication.

In the longer term special packaging for CAL array parts (such as that used to place many RAM chips together within a single unit) would be desirable since to decrease the load on 'internal' output signals and allow smaller pads dissipating less energy to be used. This is an important reason for considering wafer scale integrated versions of the architecture (Chapter 5).

4.6 Programming.

The design goals for the RAM programming logic are firstly that it should have a clean and easy to use external interface, secondly simplicity to ensure that it works on first silicon and uses little area and thirdly that it should allow reasonably fast read and (more importantly) write access. These goals are considerably different from those of normal RAM designs where speed of reading and writing is paramount.

4.6.1 External Interface.

The best external interface for CAL depends on its application: the EPLD design has different requirements from the array component design.

EPLD Design. In the EPLD design the most important goal is to reduce the complexity of the interface to the outside world. This design cannot expect to have a friendly computer to load its programming data. Pinout must also be minimised to reduce cost.

For these reasons it was decided to make use of static shift-register counters as address logic rather than conventional address decoders. This means that users of CAL's need only generate a serial data stream and a clock signal for programming thus reducing system overhead. The use of static shift registers means that the only timing constraint is on the maximum clock frequency. This decision has the side effects of reducing area, design complexity and pad requirements for the CAL.

Control. The programming circuitry requires five external signals:

1. *PROG.* This signal is analogous to the $\overline{CE}$ input on conventional RAM's: it tells the programming logic to pay attention to the read/write signal.

2. *Rd/*$\overline{Wr}$. This determines the direction of data flow on the *Din/Dout* pad.

3. *Clock.* This signal is used to clock the shift register counters to address successive RAM cells. Data input and output is synchronised to this clock.

4. *Reset.* This signal clears the counters to address the first RAM cell.

5. *Din/Dout.* This is a bidirectional pad for data input and output: the direction is controlled by the read/write input.

The circuits for controlling the RAM's are very much simpler than in standard RAM designs. There are no sense amplifiers for reading: instead a simple inverter is used. Bit Lines are pulled up statically using long p-type transistors rather than the conventional precharge circuitry to simplify timing. The transistors within the RAM cell itself are sized conservatively to ensure reliable read and write operation.

Array Design. CAL arrays will be most useful as a board within an existing computer system: it makes sense, therefore, to map the control store into the memory of the host and provide logic on chip to do address decoding. In order to reduce the number of pads required (this is important because the arrays are already pad limited) it makes sense to share pads between address and cell inputs. A control signal would determine whether addresses or data were being passed. Pad sharing would make it hard to change the contents of the control store during computations - this may be a serious drawback in some applications.

4.6.2 Internal Design.

This section will deal with the internal design of the EPLD interface described above. This was the interface used on the prototype CAL chip and the only one designed in detail. Many of the points would also be applicable to the address based array interface. An implementation of the array based interface could use one of the 'textbook' decoder designs [Weste85].

Word Lines. The word lines are run in polysilicon vertically and have a large RC delay. Simulation showed it was necessary to install repeaters on these lines every

10 cells to get reasonable access times for the RAM's. Even with these repeaters in a 1024 cell array there would still be a 25ns (worst case) delay between the input to the word line buffer and the word line transistor in the farthest RAM cell.

The most obvious method of decreasing this delay would be to run word lines on a metal layer: this would incur a very large area penalty with current technology although extra metal layers could change this situation. The other alternative would be to duplicate the address decoding logic, in this case only a serial data stream or a set of address lines would need to be distributed on metal. The first step would be to have two address decoders one at the top and one at the bottom of the array, halving the length of the polysilicon word lines. This technique would probably be necessary if the CAL were extended to 64x64 arrays in $1\mu m$ technology. Placing extra address decoders in the middle of the array to further reduce word line length is not particularly attractive since it would increase the length of inter-cell communication paths.

Bit Lines. The bit lines are run horizontally in metal 2. The RC factor is small enough to allow reading and writing logic to be placed at one side of the array. Conventionally, this logic would be placed centrally and the array would probably be partitioned further. In a CAL, however, the access time of the RAM is secondary to the delay time between cells and splitting the array would mean routing inter-cell signals across about $200\mu m$ of RAM control logic. This is made harder by the fact that this logic must be pitch matched to the CAL cell (or the vertical dimension of the chip would be increased) and the layout is quite tight. It would be possible to duplicate the bit line logic on the other side of the array to half bit line length if speed was felt to be critical (the advantage of this would be mainly on reads where the RAM transistors which must act through the relatively high resistance of the unit size word line pass transistor drive the bit line capacitance).

The write cycle time is more important than the read cycle since reading the RAM is only necessary for testing. The consequence of this fact and the long word line delays is that, in the current design, there is no reason to fine-tune the RAM

control logic for speed. It is very easy to ensure that the bit line voltages are valid when the word line goes high and this is all that is required. It is also possible to use static pull ups on the bit lines because the shift registers impose an order on the write requests. Although it takes a relatively long time for the pull ups to restore the voltage on the bit lines no cell in the row can be accessed again until the shift register has cycled through all the other rows.

Since the only time the RAM will be read is during testing (to ensure it is functional) it was not critical for performance reasons to have a sense-amplifier and a simple inverter was used instead. The reason for this decision was that if any errors were made in the sense amplifier design the chip would be untestable.

Shift Register Counters. The shift registers are built from simple ratioed CMOS master slave latches using a two-phase clocking scheme. The row and column shift registers have the same circuit but different layouts because they must pitch match to different cells. The column register layout is slightly convoluted because its width must be less than or equal to that of a RAM cell. As well as the clock signals a clear signal is necessary. The first registers in the row and column chains master latches clear to 1 to insert the 'tokens' which are shifted round. The row register clocks are generated by splitting the CLK signal from off chip: ϕ_2 corresponds to CLK=0, ϕ_1 to CLK=1. The column clock signals are generated by splitting the feedback line of the row registers: thus the column register is clocked every time the row register has gone thorough a complete cycle.

Cycle Times. The major constraint on timing is the word line delay, once the word lines are high the RAM cell will write very quickly. If the users data input clock frequency is set to 10MHz then there is a large safety margin, at 20MHz the clock signal cannot be symmetrical since ϕ_1 (corresponding to CLK=1) must be high for at least 30ns to allow for word line delays.

4.6.3 Possible Improvements.

The RAM programming logic is the area of the CAL chip most likely to be changed in later designs. This is because it was deliberately kept very simple: one major reason for this was that more complex designs could not be simulated adequately with the tools available. RAM control logic has complex timing and electrical characteristics and there is little chance of producing a working design on first silicon without good simulation. Despite this, high performance, particularly on write cycles will be required from RAM's which are to be used as array components to allow fast downloading and changing of programming information. Some possible improvements are listed below.

Cell Based Programming. Currently, the CAL programming is based on RAM addresses rather than cell addresses. In some applications it would be preferable if the chip could interpret a cell based protocol e.g. program cell at $< x, y >$ with $<$ 20 bits of programming information $>$. The mapping between RAM addresses and cell addresses is complicated by the logical/physical wiring scheme and the fact that the programming data for one cell is on two separate rows of the RAM. To implement such a protocol a small controller would be needed on chip: some speed reduction could also be expected. The Xilinx LCA product uses a fairly complex programming protocol.

Parallel Writes. In theory, it is possible to write all the cells in one column of the RAM in parallel. If this is combined with a very fast serial or parallel interface to the outside world much faster write times can be obtained (c.f. Video RAM's). The main cost would be in the complexity of the programming interface to the CAL (since there would be a very fast burst of input as data was entered followed by a pause while the programming occurred). An important reason for providing such an interface would be to reduce the testing times for CAL's where a large number of different configurations must be input. This technique would be a natural extension to the present design which already duplicates reading and writing logic for each RAM row.

Duplicated Bit and Word Line Logic. As has already been mentioned bit line and word line logic can be duplicated on each side of the array to reduce delays. This, coupled with careful design of the storage cell and peripheral circuits could reduce the cycle times of the CAL RAM to the same order as that of commercial static RAM's.

4.7 Global Signals.

4.7.1 G1 and G2.

The G1 and G2 signals are distributed horizontally along rows and vertically up the right side of the chip in metal 2. There are two levels of buffering, one per row connected to the common vertical bus and a single buffer to drive the vertical bus at the base of the array. Because of the low resistance of metal 2 two levels of buffering with large transistors suffice. The worst case delay on these lines is small enough to be ignored in user designs.

4.7.2 Ftest.

This is the global test output which can be taken from the function block of any cell. Each cell has a single bit of RAM which will be 1 if that cell is to drive FTEST. Because the state of the RAM's is unknown on power up and the chip must not fail if erroneous data is programmed into it this line is made open drain so that contention between cells cannot occur. At the right edge of the array is a pull up for each row and a simple charge sharing sense amplifier. To improve speed the line is not pulled up all the way to V_{dd} or discharged all the way to V_{ss}.

The global signal is derived by having another common drain line running vertically along the right of the chip with a similar pull up and sense amplifier at its base. The sense amplifiers are necessary to increase speed since the utility of this function depends on it being able to work at the normal operating speed of

the device. Simulations suggest that the delay between the FTEST output of the farthest cell and the output line to the pad will be about 6ns.

The sense-amplifiers for this function are the only 'tricky' circuits on the chip. It was thought reasonable to include them here because if they fail then only a small part of the chip's function is lost: if they are successful then they could also be used in the RAM read logic.

4.8 Power Supply Distribution.

Power supply distribution is a harder problem for configurable circuits than for most designs because power consumption can vary greatly according to the circuit being emulated. Two conditions must be satisfied: metal migration limits on conductors must not be exceeded and worst case voltage drop on the path to any active component must be small. Limiting the voltage drop is especially important in this design because of the use of n-type pass transistor logic which has less noise margin than fully complementary logic.

4.8.1 Array Requirements.

In order to determine power supply requirements a SPICE model of the basic 4:1 multiplexor was built. It quickly became apparent that the RAM power consumption was negligible compared to dynamic power dissipation in the multiplexors. It was decided that the power supply system should be designed to cope with every multiplexor in the array passing a 50MHz signal. This resulted in a current consumption for the 1000 cell array of 528mA.

Real designs where signals pass through several levels of logic will clock nearer 10MHz, most 'data' signals will not change value every clock cycle and a large proportion of the routing and functional hardware will not be in use. Setting the speed to 10MHz divides the power consumption by 5 and the other two factors

will probably amount to another factor of 4 giving a current consumption in the array of around 25mA.

Although the worst case consumption will never happen in normal user designs someone could conceivably configure the array with a path through all the multiplexors and arrange a fast input signal. Where possible computing structures should not be susceptible to physical damage from running legal programs. Another good reason for allowing for high currents is that ring oscillators can be formed on the array by random values in the control store on power up: it is extremely unlikely that such a configuration would approach the worst case but it is better to be safe than sorry. Systems which use CAL's (or other general configurable parts like the XILINX array) should program the array as soon as possible after power up to break ring oscillators and make sure pads are driven by proper signals. Future CAL chip designs may include circuitry to clear the control store on power up, this is already available in the latest Xilinx LCA.

Sizing the conductors appropriately to support the worst case current meant that the voltage drops in the centre of the array were insignificant. The strategy adopted is shown in figure 4–22. All these tracks run in metal 2 whose resistance and metal migration limits are three times better than those of metal one. The array is supplied by four power and four ground pads. Vertical buses $100\mu m$ wide in metal 2 run up the edges of the array. These are connected to $12.5\mu m$ wide horizontal buses which travel along the rows of RAM cells. Since the horizontal and vertical buses are refreshed at both ends their current capacity is effectively doubled. It turned out that the horizontal power buses did not increase the width of the RAM cell design which was forced by diffusion separation rules. It must be noted that if one of the very dense RAM cell designs discussed earlier had been used taking this conservative approach to power supply design would have resulted in a large area overhead.

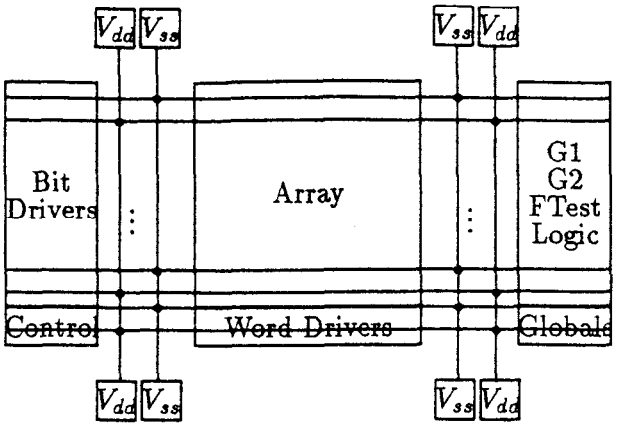


Figure 4–22: Power Supply Grid.

4.8.2 Pad Requirements.

The pad requirements obviously vary from array to array and with the pad library used. Only output pads are important for these calculations. The only design taken to this stage was for a 16x16 array which required 64 output pads. In this design one could just get by with the 4 power and 4 ground pads for the array providing power to the pad ring as well. It was necessary, however, to carefully arrange the pads so that no output pad had too many other output pads between it and the nearest power pad. The present design uses standard library pads, future designs will probably split the power sources for the pads and logic circuits to reduce power supply noise caused by heavy pad switching currents and use specially designed pads: this could potentially provide a significant area reduction and performance increase.

Unfortunately, worst case design of power supply for pads is not feasible for designs intended to be subcomponents of larger arrays. Even if wide enough metal wires and many power pads were provided the heat dissipated would require special packaging. Users must take responsibility for ensuring that excessive demands are not made: it is highly unlikely that a real design would use most of the output pads or if it did that they would be switching as fast as theoretically possible.

CAD programs could be developed to detect when such situations could occur and warn the user.

4.9 Summary.

This chapter has concentrated on the engineering of an implementation of the CAL architecture in $2\mu m$ CMOS. The major considerations in any such design have been discussed. Ways of improving the implementation given more design effort and better processing technology have been presented.

Chapter 5

Specific VLSI Implementations.

This chapter will consider specific mappings of the configurable logic architecture into silicon. Three specific designs have been attempted. The first of these is a chip to implement CAL, this chip contains a 16x16 array of the configurable cells designed according to the principles in the previous chapter. The second is the Configurable Logic Array (CLA) which uses the second metal mask instead of RAM to personalise an array of basic cells. This system was designed in conjunction with Genbao Feng who is responsible for the function block design used. The CLA was designed to allow direct mapping of systems prototyped using the dynamically programmable technology into silicon. Since only one mask is being changed techniques such as laser zapping can be used to configure standard chips quickly and cheaply. This architecture is interesting from another point of view as well: it occupies the middle ground between dynamically programmable systems where configurability is expensive and cellular designs are most efficient and full-custom silicon where configurability is cheap and cellular designs are inefficient.

The third topic covered in this chapter is a proposed wafer scale version of the dynamically programmable architecture. Wafer Scale Integration (WSI) appears to hold much promise for dynamically programmable hardware: the original research in this area (e.g. [Shoup70]) had wafer scale integration in mind - although he considered each cell as being one chip on a wafer. Technology has improved a little since then!

5.1 The CAL Chip.

5.1.1 History.

Throughout the course of this project it has been intended to design a chip to implement dynamically programmable CAL. This chip design went through three major iterations as the processing technology available changed: from $6\mu m$ NMOS to $4\mu m$ CMOS to $2\mu m$ CMOS. Only the final design was covered in Chapter 4 but the experience with the other two means that all the major design problems have been tackled in more than one way giving high confidence that the final design is near optimal.

During the course of this project two design systems were used: the department's internal ILAP system [Hughes82], for which a new leaf cell editor QV [Kean86] was written to support this design and MAGIC [Hamachi85] from the University of California at Berkeley. It is safe to say that without the Berkeley VLSI Design tools and the availability of a commercial $2\mu m$ process the chip could not have been designed efficiently enough to be viable.

The present design consists of a library of leaf cells, these cells were designed as components of a (32 by 32) 1024 cell array and all transistor and power line size computations were done assuming this size of array. Use of more advanced design automation systems would allow for autoscaling of power lines and key driver transistors with array size. These present cells can be used to build arbitrary sized logical arrays up to 32 cells wide and 32 cells high (a 16 by 64 physical array): note the limit is on maximum dimensions in a given direction to limit wire lengths rather than on the 'area' of the array.

5.1.2 Chip Design.

As can be seen from the plot of the CAL (figure 5–1) the design is fairly dense and consists of a large array of cells with peripheral circuits on the left and right

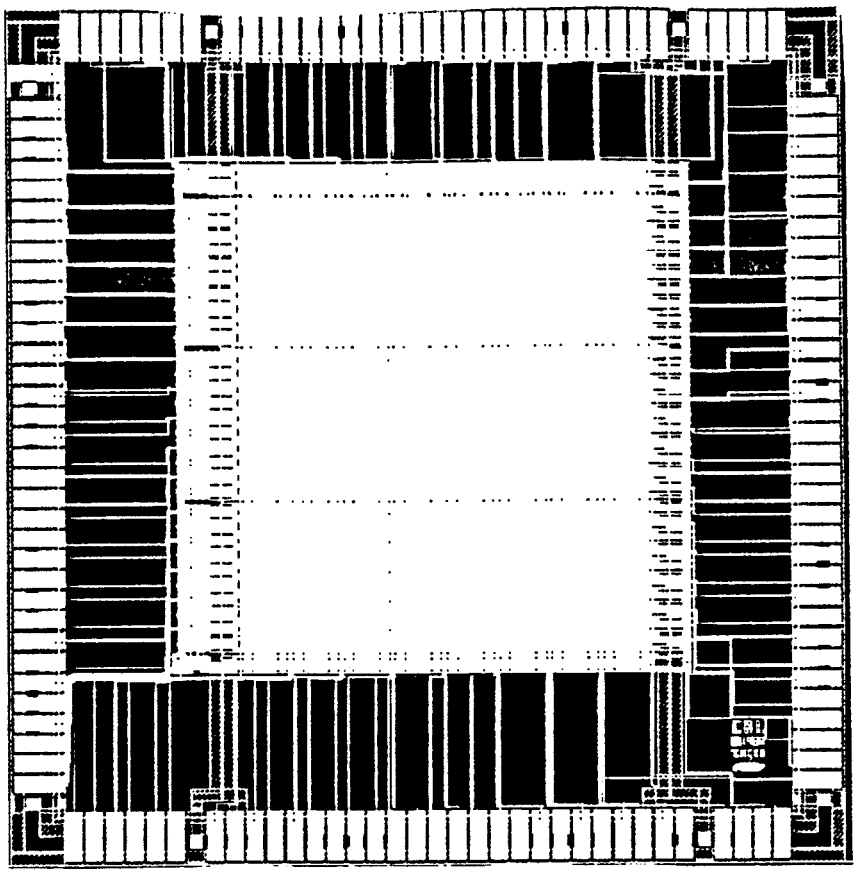


Figure 5–1: CAL Chip Plot.

and word line drivers and control logic at the bottom (figure 4–22). The logic on the left of each row is concerned with reading and writing the RAM and the logic on the right is concerned with the G1,G2 and FTEST signals for that row. There are also large power busses on either side of the array - these are sized for the 1024 cell design.

The logical 16x16 array on the prototype CAL is formed from a physical 8x32 array and, after counting in peripheral circuitry, has a core (i.e. before pads) symbol size of 4817 by 4596μm. All peripheral signals from the array are brought out then and after taking into account power, programming and the global G1,G2 and FTEST signals the chip uses all the pins in a 144 pin Pin Grid Array (PGA) package. This design of CAL could be used to produce an array big enough to emulate an ASIC by connecting multiple chips together at board level. This version of the design has been fabricated but because of very long delays at the processing plant it was not shipped until after the funding for this project ran

out. Testing carried out at the processing plant shows that the chip has no gross electrical problems (such as power-ground shorts or latchup) but it has not been possible to build a full test rig for the device (this would require a fairly complex interface to a host computer in order to download programming information and monitor chip inputs and outputs).

For PLD applications,on the other hand it is preferable to have as large an array as possible on one chip and use a multiplexing scheme at the edge of the array. For example, a logical 30x30 array would have a core symbol size of 8177 by 7450 μm and could share one bidirectional pad between two peripheral cells (the output of one cell driving the Three-State control and the output of the other driving the pad output, the input from the pad being connected to both cells inputs). This design would fit in a 68 pin DIL package.

Table 5–1 gives an estimate of how array size will grow with processing technology. Naturally, such figures should be taken with extreme caution since key design rules such as metal pitch do not scale in the same way as the transistor gate width used to typify the technology. It can also be expected that the cell design itself will be improved in the next design iteration and it is not unlikely that major process improvements such as provision of extra metal layers will occur. The entry for 'special' CMOS indicates a process with some support for RAM: about half the CAL area is taken up by RAM cells and processing support for them could have a major impact on array size. We assume that we want the chip to be around 1cm on a side to maximise the number of cells while obtaining reasonable yield.

5.1.3 Design Validation.

Two simulation programs were used to validate the CAL design: SPICE [Vladimirescu87] and RNL [Terman87].

SPICE. SPICE (version 2G6) was used for circuit level simulation of key components such as the RAM cell and the basic 4:1 multiplexor and to obtain timing information for the global signals. SPICE was perhaps the least satisfactory com-

Technology.	Array Size.	Notes.
Generic CMOS $2\mu m$	32x32	> 1cm on a side
Generic CMOS $1.5\mu m$	32x32	Reasonable Size
Generic CMOS $1\mu m$	48x48	Reasonable Size
Generic CMOS $0.8\mu m$	64x64	Reasonable Size
Special CMOS $1\mu m$	64x64	Reasonable Size

Table 5–1: Projected Improvement of Array Size with Processing Technology.

ponent of the Berkeley tools package: it has an extremely poor user interface and the numerical algorithms used are very temperamental. It is normally necessary to juggle with the simulation temperature and add nonexistent large resistors and small capacitors to critical nodes to force the numerical algorithm to converge. This made it impossible to set up simulations which described exactly the circuit and timing parameters that one wanted. SPICE simulations were done with hand-coded rather than extracted circuits to give enough control to make the simulations converge. The simulations which were done give a high confidence of design correctness but the timing figures which resulted (Chapter 4) are questionable and should be read as absolute worst-case figures (resistance and capacitance values were overestimated and slow process models were used to force convergence since SPICE seemed to have more problems with fast rising and falling signals).

RNL. RNL is a switch level simulator with a LISP based interface developed at MIT and improved at the University of Washington. It was used with circuits extracted from the chip layout by MAGIC. RNL has a very limited circuit model in which transistors are replaced by one of two specified resistance values according to whether the simulator thinks they are ON or OFF. Higher resistance values are used for n-type transistors passing logic 1 and p-type transistors passing logic 0. There are three possible logic values 0,1 and X. This model works reasonably well for fully complementary CMOS designs but has problems with ratioed circuits

such as RAM cells and n-type pass transistor logic. Circuits with feedback are prone to 'catching X's' where a real circuit would quickly leave the intermediate state. Means are provided for giving 'hints' to allow correct simulation of such circuits but to some extent this defeats the object of the simulation: this feature was only used for areas which had already been verified using circuit simulation.

For the purposes of simulation a 6 by 3 logical (3 by 6 physical) array of cells was built. This represents the smallest repeating unit in the design so that simulations done using it would check every component of the larger arrays (for this reason RAM column buffers were inserted in the array although they are not necessary for so small a design). Even with this reduced array size switch level simulation was very time consuming: much of the time went on circuit initialisation (where the simulator attempts to assign initial values to nodes in the circuit) because of the amount of state in the design. Every simulation on the array had to be left to run overnight on a SUN 3/260 and had normally completed by the morning. This severely restricted the number of simulations that could be done.

Once this model had been built several simulations were done to test the correctness of the design:

1. RAM Programming. The programming circuitry was verified by a simulation which programmed every RAM cell on the chip and then read back the results. Several patterns were used (e.g. 101010...., 11001100..., 000000..., 11111....). In the course of this simulation several errors were found in the programming circuitry and corrected. Some redesign of peripheral circuits was done (changing ratioed logic to fully complementary logic) to make simulation easier.

2. Cell Function. This simulation was run on extracted data representing a single cell function unit. The function unit (Y1,Y2,Y3 multiplexors) was presented with every possible input pattern (plus some repeated patterns to check the latch functions). The output was checked to ensure that all the cell functions were performed correctly. As a result of this simulation some minor changes to the function unit design were made to reduce the maximum

path between buffers from four n-type pass transistors to three n-type pass transistors.

3. Chip Function. This simulation was done in two parts: firstly the chip was loaded with programming information representing a particular design and secondly the configured chip was presented with data inputs to check that the design was implemented properly. The design chosen was two of the toggle flip-flops from the stopwatch example (Chapter 7). In the course of getting the flip-flop example to work many smaller designs were tried and it is unlikely that any major faults remain undetected. The only component which has not been fully verified at the chip level is the $FTEST$ signal whose sense amplifier circuitry was beyond the capabilities of RNL: it is, however, fairly simple and the amplifier design has been verified using SPICE.

Other Verification Tools. As well as simulation the netlist extraction tools available were used to check the continuity of important signals. This proved to be a very useful technique since a netlist extraction can be done much faster than a simulation. Using this technique a power ground short in the control section (which had been causing some very strange simulation results!) and several short breaks in RAM bit lines were detected. The bit line breaks are a good example of the limitations of graphical editors: it is often hard to see that a wire is broken for a short distance when it is passing over other circuitry e.g. a light purple metal 2 bit line will not show up well over a blue metal one wire and a red and brown shaded p-type transistor. Net list checks are the best way of detecting such errors.

The geometrical correctness of the design was verified using MAGIC's on line design rule checker. This was one of the most useful facilities in the MAGIC suite and saved a lot of the time spent staring at plots in the previous $4\mu m$ design.

5.1.4 Testing the CAL.

Failure Modes. There are three basic areas in which a CAL could fail: the cell routing area, the RAM cells and the peripheral circuitry. Well over half the

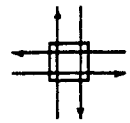


Figure 5–2: Testing: Initial Cell Configuration.

transistors in a CAL are in RAM cells so this will be the most common failure area. If a failure occurs in a RAM cell it is very likely to affect either a bit or a word line, failures which affect these signals will wipe out a whole row or column of RAM cells. Failures in the cell wiring area will usually have no effects outside of that cell, except that neighbouring cells may receive erroneous inputs. Failures in the peripheral circuitry could easily knock out either the whole chip or very large sections of it.

Test Algorithm. We will assume that we wish to detect if a chip contains any faults so that it can be discarded rather than locate a given faulty cell to allow row and column replacement. The following procedure could be adopted.

1. Test the RAM. There are many published algorithms for testing RAM's; one of the best known is the Algorithmic Test Sequence (ATS) algorithm [Knaizuk77]. This test can be expected to detect most faults.

2. Initial Configuration. Configure the array so that all signals pass straight through (figure 5–2). Start at the bottom left and test each cell in turn until you reach the top right.

3. Cell Test. The test for each cell seeks to exercise all configurations of each multiplexor. Since all the other cells in the array are routing 'straight-through' changes in the current cells outputs are visible at the edge of the array (faults in intermediate cells between the current cell and the edge of the chip could force the current cell test to fail - this is not important since we are only interested in the presence, not the location, of faults on the chip).

(a) North Test. S,W,E select N. Toggle North input to array and check that S,W,E respond if not discard chip.

(b) South Test. N,W,E select S. Repeat.

(c) East Test. N,S,W select E. Repeat.

(d) West Test. N,S,E, select W. Repeat.

(e) X1 Test. N,S,W,E=F. Function=X1. FTEST=TRUE, X1 selects N,S,E,W, G1,G2 in turn toggle inputs, check outputs respond, check FTEST responds.

(f) X2 Test. X2 selects N,S,E,W in turn.

(g) Function Test. X1 selects N,X2 selects S: check all functions in turn for all input permutations 00,01,10,11. Check latch - needs more vectors.

(h) Finish. N selects S, E selects W, W selects E, S selects N. Ftest=FALSE. Next cell.

The procedure outlined above is far from optimal in terms of number of test vectors which must be applied. More complex testing protocols which take advantage of the fact that many cell multiplexors can be tested at the same time could easily be developed.

5.1.5 Fault Tolerance.

In this section we will briefly discuss the CAL design with reference to fault tolerance. The main motivation for this section is that cellular array designs are often associated with fault tolerant computing.

The first two failure modes mentioned above can be treated to some extent by providing an extra row and column of cells on the chip. If a single cell failed then the row and column it was in could be configured out of the array and the spare row and column added. This would provide some protection from single defects and may be worth considering: the economics of this approach are very process specific. Note that if the fault was in a RAM cell and it caused failure of

all cells on a bit or word line then it would be impossible to program the other cells in the row (or column) to route 'straight across' the faulty row or column so this technique would not work. The present CAL design was designed for area efficiency not fault tolerance: if fault tolerance was seen as an important goal then shift register control store and extra connectivity to cells on a hexagonal or octagonal grid should be considered (see [Lee86] for a discussion of how to design fault tolerant arrays using these extra connections).

5.2 CLA Implementation.

The Configurable Logic Array [Kean87] is a mask programmable version of the cellular architecture: the original idea of a mask programmed variant of Configurable Logic and the original function block design are due to Genbao Feng. The implementation of this system predates many of the design decisions described in earlier chapters. In particular the implementation of the basic function block is not optimal (although the same functions are provided) and the global signals are not available. There are no fundamental reasons why the system could not be updated to include these features. The cell designs described here were done in $4\mu m$ p-well CMOS rather than the $2\mu m$ n-well CMOS of the CAL chip.

Experience with the CLA system has shown that while it can compete with single mask change gate arrays in terms of density it is not significantly better. Gate array systems are highly developed and have many man years of investment in software tools and, therefore, it does not seem worth developing the CLA further. This decision is reinforced by a change in philosophy: originally it was intended that designs would be done using cell based tools then prototyped using CAL's and implemented using CLA's. The present system envisages designs being done at a high level in terms of logic blocks and interconnections: tools would then be provided to produce cellular implementations which could be tested using CAL's. Existing silicon compilers could then be used to produce implementations of the

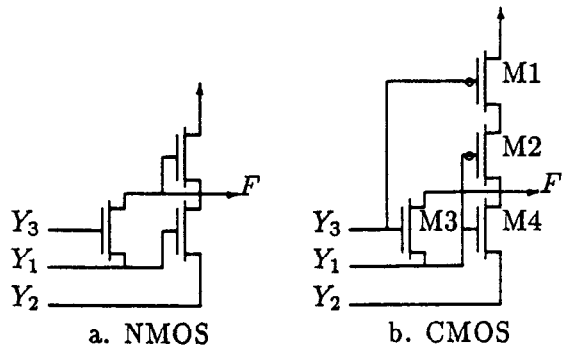


Figure 5–3: Basic Block Circuit Diagram.

higher level designs. The reasons for this change in philosophy will be discussed in more detail in Chapter 6.

5.2.1 Basic Design.

The basic function block is based on an NMOS design in [Chen82] (Figure 5–3 a). This was converted to CMOS (Figure 5–3 b). The original programming table is given as Table 5–2.

Input Inverter. Some of the functions in the programming table require the complementary form of one of the input signals (X_2). In a practical implementation these functions will either require two cells (one to act as an inverter) or an extra inverter must be added within the basic cell itself. The design goals of the CLA are best served by adding an extra inverter. It is likely that in most user systems using these cells this will also save area since a single inverter is much smaller than a four transistor function unit with its associated routing area. Note that if the function units were used in a gate array layout with separate wiring channels it might well be better to use two function units.

Output Inverter. Although this new design can perform all the functions of two boolean variables it has some electrical problems. Functions where $Y_1 = 1$,

Number.	*Function*	Y_1	Y_2	Y_3
0	$ZERO$	X_1	0	1
1	ONE	X_1	1	0
2	X_1	X_1	1	1
3	$\overline{X_1}$	X_1	0	0
4	X_2	X_1	X_2	$\overline{X_2}$
5	$\overline{X_2}$	X_1	$\overline{X_2}$	X_2
6	$X_1 \cdot X_2$	X_1	X_2	1
7	$X_1 \cdot \overline{X_2}$	X_1	$\overline{X_2}$	1
8	$\overline{X_1} \cdot X_2$	X_1	0	$\overline{X_2}$
9	$\overline{X_1} \cdot \overline{X_2}$	X_1	0	X_2
10	$X_1 + X_2$	X_1	1	$\overline{X_2}$
11	$X_1 + \overline{X_2}$	X_1	1	X_2
12	$\overline{X_1} + X_2$	X_1	X_2	0
13	$\overline{X_1} + \overline{X_2}$	X_1	$\overline{X_2}$	0
14	$X_1 \oplus X_2$	X_1	$\overline{X_2}$	$\overline{X_2}$
15	$\overline{X_1 \oplus X_2}$	X_1	X_2	X_2

Table 5–2: Original Programming Table.

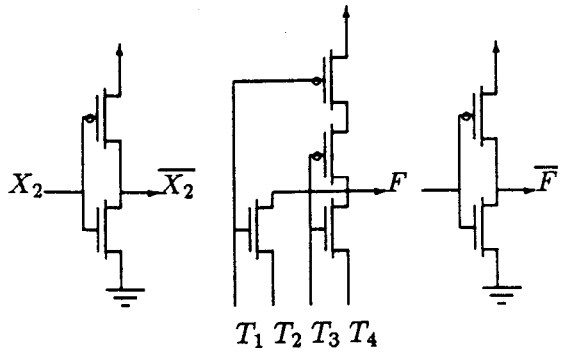


Figure 5–4: New Cell Circuit Diagram.

$Y_2 = 1$, $Y_3 = X$ (e.g. ONE) involve transmitting high voltages through M4 (figure 5–3 b) which will degrade the output voltage by V_T. Another problem is that functions with the input condition $Y_1 = 1$, $Y_2 = 0$, $Y_3 = 1$ (for example X_1X_2 with $X_1 = 1$, $X_2 = 0$) rely on contention between M3 and M4. This is not a problem when the current comes from the extra (input) inverter but it is serious when it comes directly from a cell input. The combination of a cell with a degraded output voltage driving several cells which rely on contention could cause failure. Therefore, an extra inverter was added to buffer the cell's output. This inverter has wider transistors than those in the function block which increases the cell's output driving capability. At this time the basic four transistor block was changed to a four terminal version from the original three terminal version. This allowed some functions which previously relied on contention to be performed under normal CMOS working conditions. Previous work in this area has sought to find circuits which minimise the number of input terminals required and would consider an extra terminal to be a major deficiency, however it caused no penalty in our layout since the extra via (contact between first and second layers of metal) could be placed in the area between the edge of the p-well and the p-type transistors which would otherwise have had to be left empty.

From the new programming table (Table 5–3) it can be seen that two of the possible combinational logic functions in the present cell draw current from the X_1 input. The first $\overline{X_1} + X_2$ can always be avoided by swapping X_1 and X_2

Number.	*Function*	*T1*	*T2*	*T3*	*T4*	*Output*
0	*ZERO*	0	X	0	X	FBAR
1	*ONE*	1	0	0	0	FBAR
2	X_1	0	X	X_1	0	FBAR
3	$\overline{X_1}$	0	X	X_1	0	F
4	X_2	X_2	0	0	X	FBAR
5	$\overline{X_2}$	X_2	0	0	X	F
6	$X_1 \cdot X_2$	0	X	X_1	$\overline{X_2}$	FBAR
7	$X_1 \cdot \overline{X_2}$	0	X	X_1	X_2	FBAR
8	$\overline{X_1} \cdot X_2$	$\overline{X_2}$	0	X_1	0	F
9	$\overline{X_1} \cdot \overline{X_2}$	X_2	0	X_1	0	F
10	$X_1 + X_2$	X_2	0	X_1	0	$\overline{F}$
11	$X_1 + \overline{X_2}$	$\overline{X_2}$	0	X_1	0	$\overline{F}$
12	$\overline{X_1} + X_2$	1	X_1	X_1	$\overline{X_2}$	$\overline{F}$
13	$\overline{X_1} + \overline{X_2}$	1	X_1	X_1	X_2	$\overline{F}$
*13	$\overline{X_1} + \overline{X_2}$	X_2	$\overline{X_2}$	X_1	0	$\overline{F}$
14	$X_1 \oplus X_2$	X_2	X_1	X_1	X_2	$\overline{F}$
15	$\overline{X_1 \oplus X_2}$	$\overline{X_2}$	X_1	X_1	$\overline{X_2}$	$\overline{F}$
16	*D* Latch	Q	X_1=CLK	X_1=CLK	$\overline{X_2} = \overline{D}$	$\overline{F}$
17	*RS* Latch	R=X_2	Q	Q	S=X_1	$\overline{F}$

(X = Don't care.)

Table 5–3: New Programming Table.

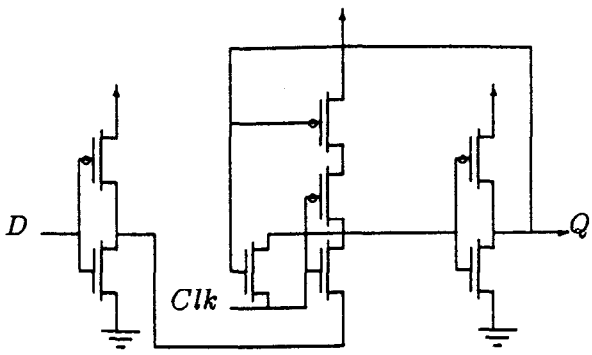


Figure 5–5: D Latch Circuit Diagram.

using the cell's internal routing and using $\overline{X_2} + X_1$ instead. The other function $\overline{X_1} + \overline{X_2}$ can usually be worked around in practice, for example, by using X_1X_2 and changing the logic functions of the cells connected to this cell's output to use the complemented signal. This final problem could be eliminated by using the connection marked * in the table - this was not used because it would have required additional routing area within the cell.

Latch Function. The additional output inverter has an important side effect - it allows static latches to be built within one cell. There are several ways of building latches but we will describe only D latches here since they can also be used with the scan-path circuitry described below. The circuit diagram is given as Figure 5–5. Note that the extra inverter is essential to allow positive feedback.

This latch design relies on contention and will draw current from its clock input in the state D=1 ($\overline{D} = 0$), $CLK = 1$, $Q = 1$. The contention imposes a design constraint on the user: no more than ten cell inputs which sink current may be connected to a cell output (the figure ten comes from SPICE simulation and allows a large safety margin - it is highly dependent on process parameters and could be improved by changing the length and width of the transistors involved in contention). This constraint is important since it is common practice to connect large numbers of clock inputs to a common clock line - extra cells may be required as clock buffers.

5.2.2 Final Design

The cell was designed in double metal p-well CMOS to lambda rules using λ=2μm ($4\mu m$ technology) it was 140 by 132 μm. These design rules are extremely conservative. The layout was done using the ILAP design system [Hughes82] and the QV layout editor [Kean86] and is shown in figure 5–6. In this design power must be distributed on metal 1 since metal 2 is reserved for personalisation: this is unfortunate since the resistance of metal 1 is about 3 times higher than that of metal 2 and the electronmigration limit is correspondingly lower. This is not a problem in $4\mu m$ technology (since the design rules force wide metal lines anyway) but would cause an area overhead in a more modern process.

5.2.3 Scan Path Cell

This version of the cell has the capability to link any latch in the user's design into a scan path to improve testability [McCluskey85b,McCluskey85a]. To make use of this functionality users must design their circuits as combinational logic blocks separated by latches. These latches then provide for controllability and observability of the combinational logic via the scan path. Test patterns can be generated automatically using well known algorithms such as [Roth67].

The extra functionality is provided by 5 additional transistors as shown in Figure 5–7. Although the number of transistors is increased from 8 to 13 the new cell is only 162 by 132 μm an area increase of about 15%. This low overhead is possible since the testability circuitry makes use of previously wasted space under the metal 2 routing wires and uses polysilicon wires for the associated global clock and control signals. The tradeoff is of performance against area - performance can be sacrificed because testability circuits will only be used once.

A version of the scan path cell designed to commercial 2μm rules on a $0.5\mu m$ grid was 96μm by 85μm or 38% of the lambda rule design area. The metal 2 rules which are the determining factor in this design did not scale as well as the polysilicon and diffusion rules resulting in a less than expected size reduction.

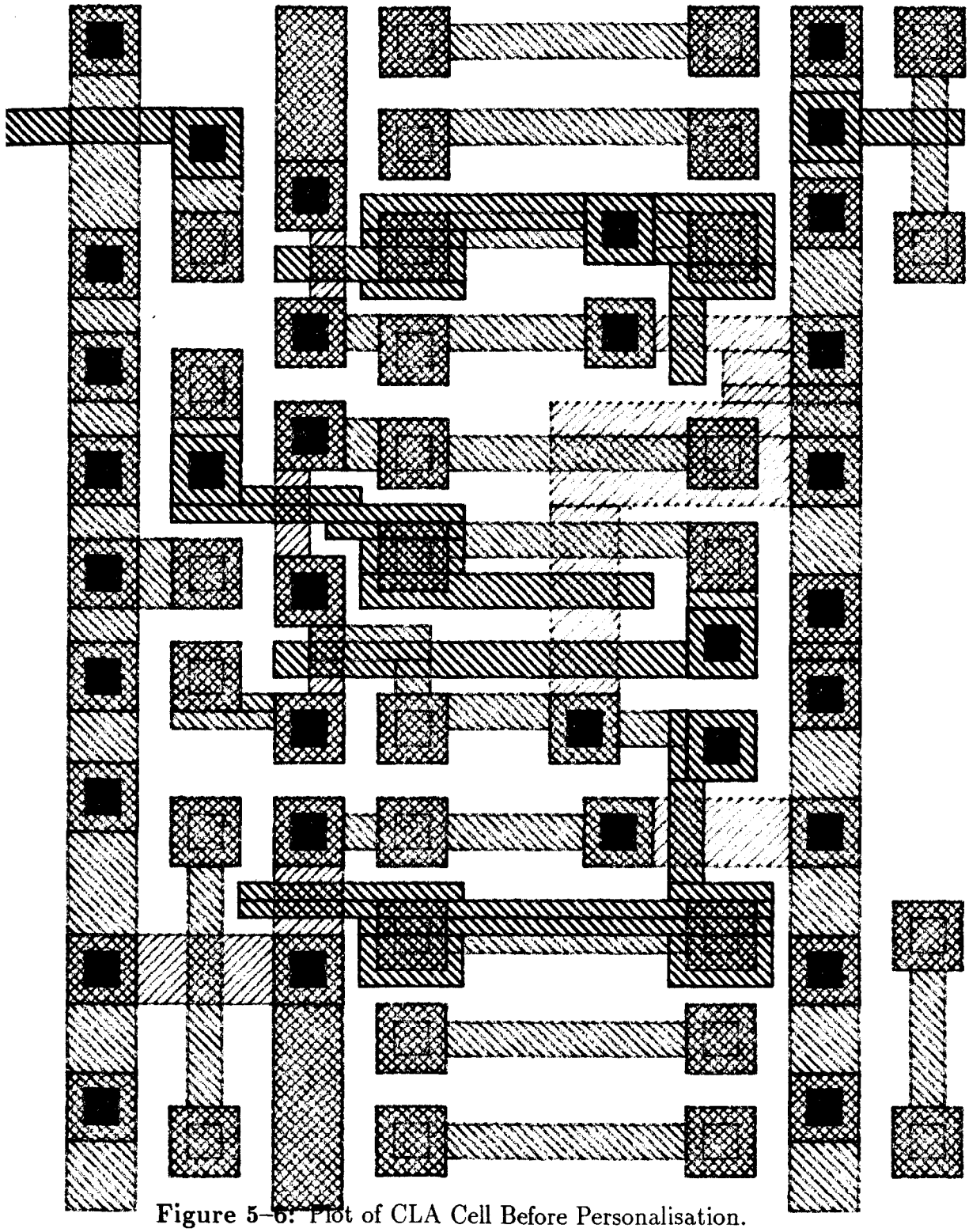


Figure 5–6: Plot of CLA Cell Before Personalisation.

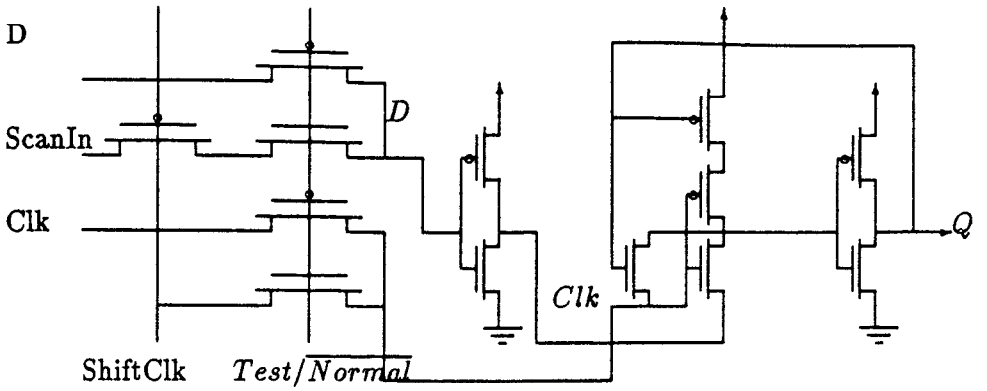


Figure 5–7: Scan Path Cell Circuit Diagram.

Another factor was the decision to use approximately the same width of metal 1 power and ground rails to allow for large arrays of cells.

There are 3 new signals associated with this function (in this discussion we assume that the cell is a latch cell on the scan path, other cells will connect scan-in directly to scan-out using metal 2 personalisation):

Test/Normal - When this signal is low the test circuitry is disabled and the cell functions as a DLATCH using the user clock and user data inputs. When this signal is high the cell implements a D flip-flop. The D input comes from scan-in and the Q output goes to scan-out. The flip-flop is clocked by ShiftClock.

ShiftClock - This clocks data through the shift register built from D flip-flops in the test mode. When ShiftClock is low the master latch is loaded and when shiftclock is high the slave latch is loaded. Note that the master latch is dynamic whereas the slave latch is static.

ScanIn - This signal comes from the Q output of the previous D flip-flop in the scan path chain. The *ScanIn* input of the first flip-flop and the output of the last are taken off chip.

The extra pass transistors (particularly the p-n combination on *ScanIn* considerably complicate the electrical design of the cell and must be sized to avoid

excessive degradation of signal voltages on clock input lines due to contention within the function block.

5.2.4 CLA Personalisation.

Given the basic function block described above we must personalise it using metal 2 connections overlayed on the basic design in figure 5–6 to obtain the desired function and routing permutation. All of the routing permutations provided by the multiplexors in figure 3–9 (except the global signals) must be available. This is done by separating the routing areas into non-overlapping 'tiles' of metal 2, the CAD software described in Chapter 6 selects which tiles are necessary to implement a given set of connections. The tilings were done as leaf cell layouts using QV (treating each tile as a symbol) and the editor output files were converted automatically into a form which could be included in the CAD programs. This approach makes it easy to change processes since there are no process dependent parameters within the human written part of the software.

Function Block Personalisation. The tiles shown in Figure 5–8 are responsible for implementing the logic function given the appropriate inputs on the X1 and X2 vias. The output is made available on the Y1 via if the output inverter is not used (F functions in table 5–3) or the Y2 via (for $\overline{F}$ functions). The X3 via can be used to input X1 allowing additional functions such as RS latch to be implemented. The T1,T2,T3 and T4 vias are the terminals of the basic block described above (Figure 5–4): the V_{dd} and V_{ss} vias provide logic ones and zeros as required (Table 5–3). Since there are only a few functions the CAD software uses a look up table to determine which tiles must be used. Tiles F50, F51,F52 and Y3 are always used, they are connected to Y1 or Y2 as appropriate to allow the routing software easy access to the function output without knowing which of the Y1 and Y2 outputs was used.

Routing Personalisation. The routing of the present cell is shown below (Figure 5–9). The tiles whose names begin with 'R' are the metal 2 routing tiles and

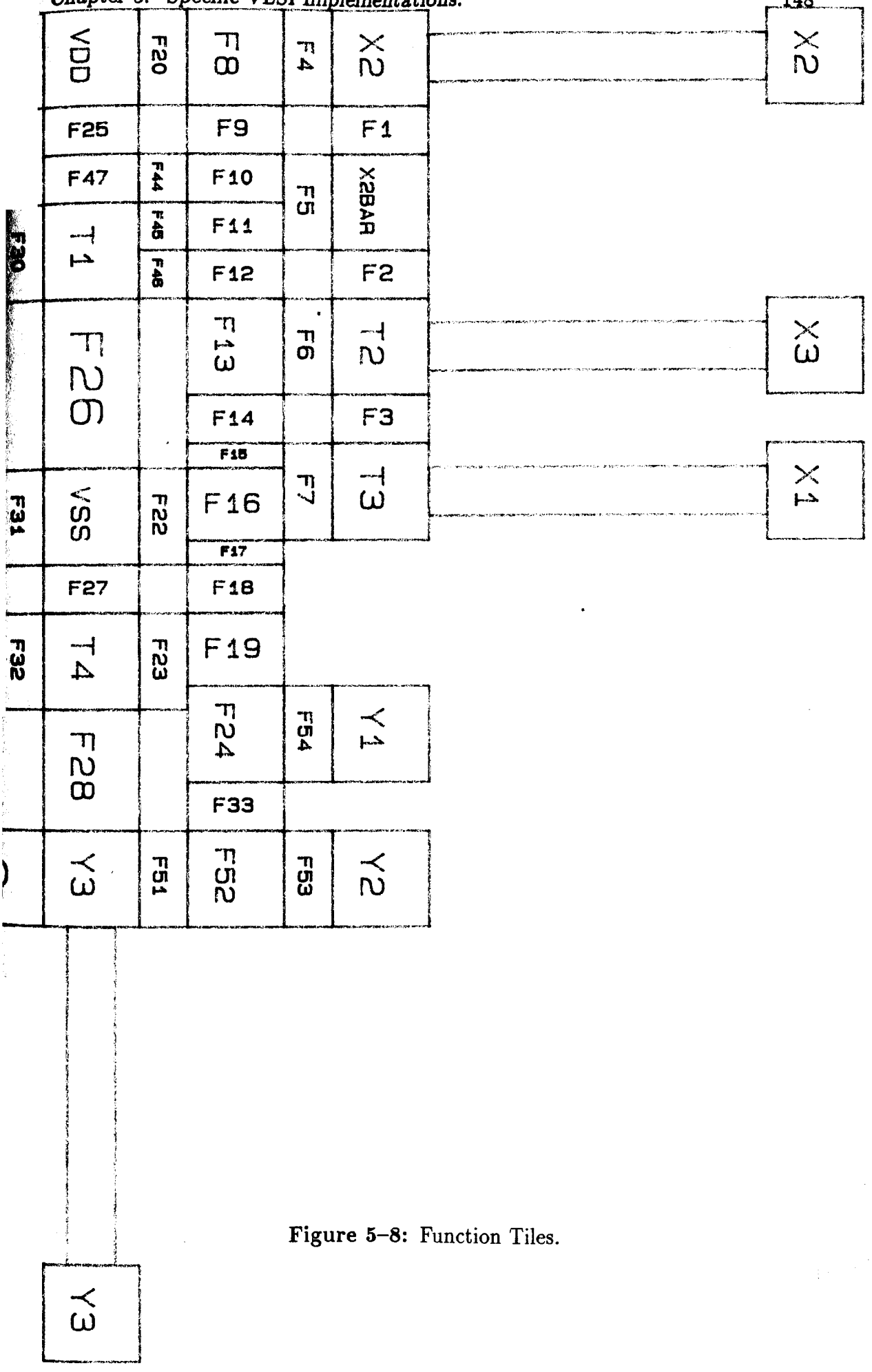


Figure 5–8: Function Tiles.

tiles starting with 'P' are dummy tiles to mark input and output ports: all the rest are via's. Tiles whose names start with 'V' connect to metal one 'bridges' under routing wires. The 'X' tiles are function block inputs and the 'Y' tiles function block outputs. Two tiles with the same name are electrically connected by a metal 1 bridge. The CAD software (Chapter 6) selects which of the routing tiles are necessary to provide a given configuration. The layout of this routing area to allow all legal input output permutations without having wasted tiles was the most time consuming part of the cell design.

At present routing area dominates the cell size. This is because of the rich routing possibilities provided and the relatively poor width and separation rules for metal 2.

5.2.5 Useful Techniques.

Although the CLA itself is not considered to be worth pursuing further some of the techniques used in it are of interest: in particular the use of pass transistor logic to obtain efficient implementations of fairly general functional units. The transistor count for a latch implemented using this technique is less than half that of one in a gate array library. If one is prepared to use p-type pass transistors (this is acceptable provided buffering is placed immediately after them) then a very area intensive layout is possible (figure 5–10). This design works because (as we saw in Chapter 3) all functions of two boolean variables can be computed using a three input multiplexor with appropriate values on its terminals (table 3–1). The mask level layout for a multiplexor using p-type pass transistors is almost identical to that of an inverter so an array can easily be customised using metal connections to make a given stage either a multiplexor (on the left of figure 5–10) or an inverter (on the right).

Extra inverters for producing $\overline{X_2}$ will only be provided when necessary rather than in every 'cell' unit as in the present case. Simple transformations on a user's design at the gate level could reduce the number of functions which required input inversion. CAD software could also share the inverted form of an input

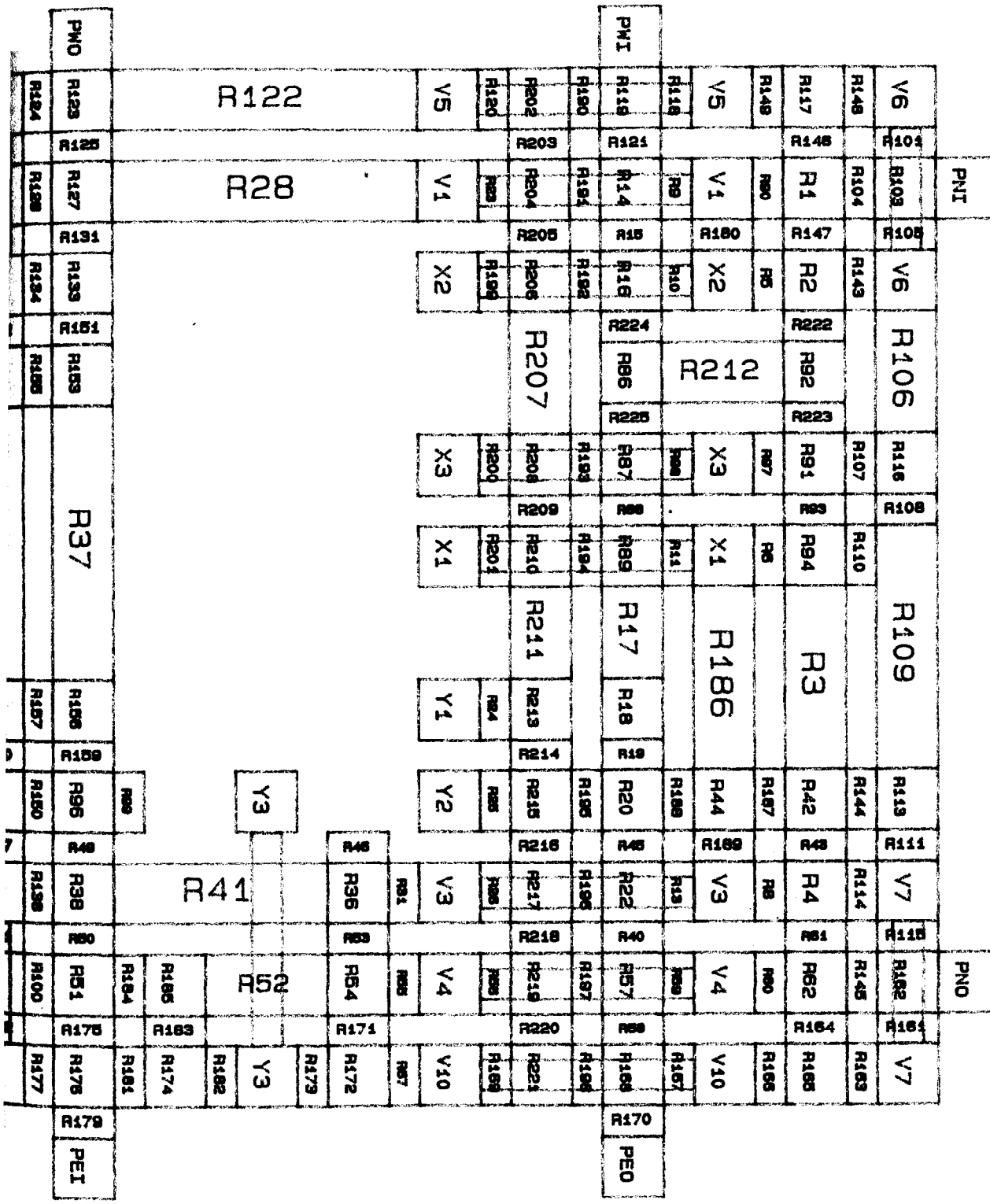


Figure 5–9: Routing Tiles.

amongst several cells which required it using wiring rather than duplicate inverters Similarly, CAD tools could be developed which, based on a set of connection rules, would only add output buffers for level compensation where absolutely necessary (many low fan-out local signals would not require it). Using this optimisation functions like AND, which would normally require 6 transistors (a four transistor NAND gate and an inverter) could often be implemented with 2, and even if an output inverter was used only 4.

There are some problems with this technique: in particular the noise immunity is lower than that of complementary logic. In many applications this may not be critical and the area advantages are considerable.

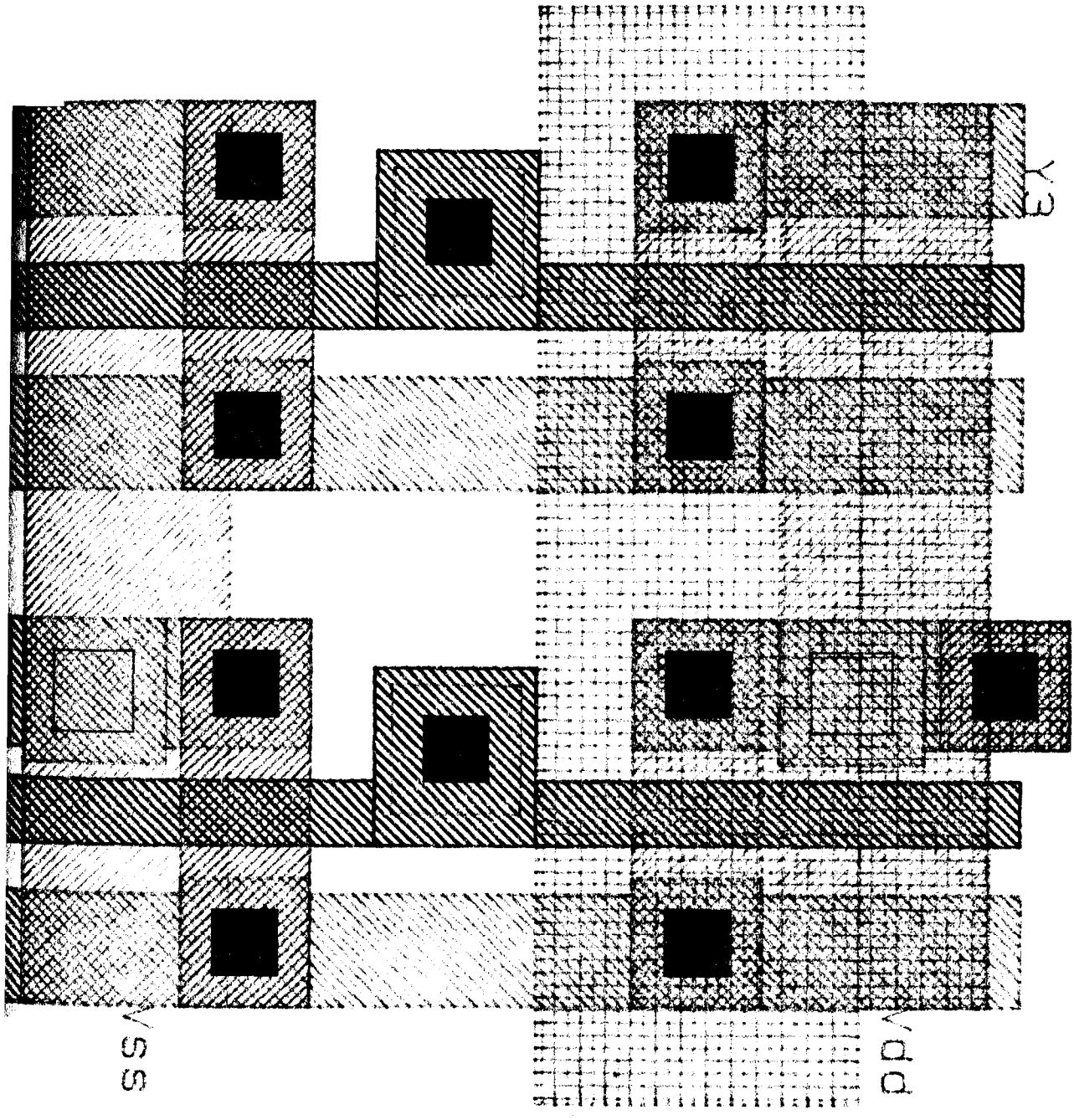


Figure 5–10: Function Block Stage Layout.

5.3 Wafer Scale Version.

This section will consider methods of mapping the dynamically programmable version of the architecture into Wafer Scale Integration (WSI). There are several reasons for looking at wafer scale versions of this architecture [Moore85].

1. I/O Bottleneck. As we have seen in previous sections the VLSI CAL design is pad limited: that is the size of array which can be used is determined not by area limitations but by the number of package pins available. With wafer scale integration between-chip wires are around the same width as on chip wires so the bottleneck disappears. There is also the speed limitation caused by the pad drivers: this is not significant with the current configurable logic architecture but could be important in a pipelined system (section 2.4).

2. Pad Cost. A design with a large number of I/O pads incurs two important penalties: firstly because of the size of the driver transistors, bonding pads and protection structures there is a large area overhead caused by the pad ring ($1.5mm$ in X and Y would be a good estimate). The pads also have a potential power consumption as great as that of the array.

3. System Costs. Wafer Scale Integrated systems have the potential to cut dramatically system costs and physical volume by eliminating the need to dice and package individual chips and solder them on a board.

4. Regularity. At first glance this architecture seems very suitable for WSI because there is only a single, easily tested component.

The hope is that a WSI version of the CAL architecture would gain enough advantage from these factors to offset the large area overhead over statically configured circuits.

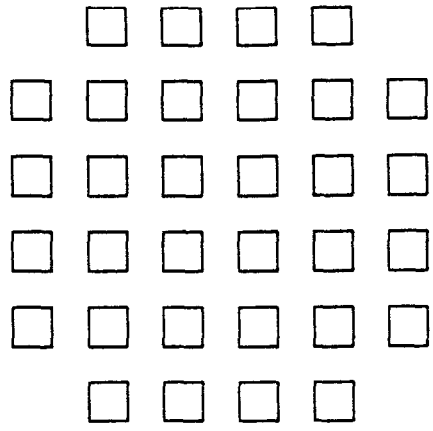


Figure 5–11: Silicon Wafer.

5.3.1 Design Overview.

The basic WSI configurable logic system would consist of a large number of chip level units on a wafer (figure 5–11). Many of these chips will be faulty. (It is not possible just to design a huge wafer sized chip because there are too many global row and column signals in a RAM. Faults affecting any of these signals would cause a whole row or column of cells to fail. Such faults are almost certain to occur in every row and column so the yield of the wafer would be almost zero). The problem is to build an array of chips containing as many of the good chips on the wafer as possible and none of the faulty chips. Ideally one would like all the good chips to be used: this is the situation in normal VLSI technology.

Given this scenario the first question is what size of chip unit to use. There are four important factors to be considered.

1. Unit Yield. The larger the unit chip the more likely it is to fail. Also, a

single fault causes a whole unit to be lost so the area lost per fault is higher with large units.

2. Reconfiguration. If large units are used then there will be fewer units per wafer. Since there are fewer units there will be fewer faulty chips: this means that less reconfiguration circuitry will be required. The number of wires in the reconfiguration circuitry remains constant (because the wiring paths for the larger units are wider) but the control overhead is dependent only on the number of logical switches - not on the number of wires being switched.

   Large amounts of reconfiguration circuitry themselves affect the wafer yield to some extent (the reconfiguration areas are nearly all wiring channels and the wires are often wider and separated more than wires within the chip units: so the yield of the reconfiguration circuits is substantially higher than that of the units themselves).

3. Chip Efficiency. The area of a CAL is $(o_x + xc_x)(o_y + yc_y)$ where o_x and o_y are the width and height of peripheral overhead circuitry in the X and Y directions x and y are the number of cells in X and Y and c_x and c_y are the cell dimensions. This implies that the ratio of chip area to cell area is maximised in a large chip. There is, of course, a maximum array size imposed by wire length constraints in the unit design; we will take this as 64×64 cells in $1\mu m$ technology.

4. Ease of Design. It would be useful if the basic unit was a 'nice' size e.g. 16×16 cells (because of simplifications in functions like address decoding and user interface) and even better if the same design used for board level chips could be used for wafer scale chips by simply removing the pads.

Before we can proceed any further we need to make some assumptions about how yields vary with unit size. Two excellent references on yield models are [Stapper83], [Stapper84]: here we will use a simpler model derived from this work based on [Turnbull85], [Fourman85]. The reason for using a simple model is that it

allows realistic computations to be done using pencil and paper with public domain information: the more complex models require special CAD programs and detailed information about a particular process. The general form of the yield equation is $Y = Y_0 e^{(-fAD)}$ where Y_0 is the yield of pinholes and vias which is taken as constant at 80%, A is the area of the device and D is the defect density (we will use two values of D in this analysis $0.02mm^{-2}$ and $0.01mm^{-2}$ the first corresponds to current practice and the second to the near future), f is a modifier to take account of the fact that not every processing defect causes functional failure (this is more important for areas like wiring channels where only some masks are significant and we will take $f = 1$). We must also consider the size of wafer to use: we choose $10cm$ wafers and assume $64cm^2$ of usable area after [Fourman85]. One should note that other authors have considered different sized wafers e.g. $15cm$ wafers with $132cm^2$ of usable area in [Turnbull85]: direct comparison of WSI system complexity can therefore be misleading. We have also assumed $1\mu m$ processing technology resulting in a factor of 1.5 improvement in the X and Y dimensions of the $2\mu m$ CAL design: $1.2\mu m$ processing for ASIC's is already commonplace. Table 5–4 shows projected yields assuming 100% harvest of working chips for different sizes of chip unit. The working sites per wafer figure is the number of sites working at 96% confidence, not the expected number of working sites.

From this table we can see that if good harvest can be obtained (i.e. most working chips can be used) it should be possible to get a $2^{17} = 131072$ cell wafer scale CAL at reasonable yields using 16×16 cell units.

5.3.2 Reconfiguration Methods.

In this section we will look at several possible ways of building good arrays from larger ones with faulty cells.

Use Cell Interconnect. The cells themselves provide a general switching system so it is worth looking at an architecture which connects all the chips on the

Array Size	*Yield*	*Sites per Wafer.*	*Working Sites*	*Working Cells*
D=0.1				
16x16	0.79	650	505	129280
32x32	0.77	162	120	122880
64x64	0.68	40	24	98304
D=0.2				
16x16	0.78	650	498	127488
32x32	0.73	162	102	104448
64x64	0.58	40	20	81920

Table 5–4: Projected WSI CAL yields.

wafer up into one huge array and uses the cell's switching capabilities to avoid defective areas.

All rows and columns in the array which contain faulty cells must be configured out (by arranging for all the working cells in them to route straight across the faulty row or column). This could be done either at the chip level, removing faulty rows and columns of chips or we could take advantage of partially functioning chips and do it at the cell level. Let us imagine the 650 chip array as having 26 columns of 25 chips (the actual situation is more complex because wafers are circular). The chance of a column containing all correct cells is $p^{26} = 0.79^{26} = 0.002$. This means that there would be no yield. Things are no better in the single cell case since although p is higher (it is hard to say how much higher since a fault in one cell can affect all the others on the same RAM bit or word line) we now have $p^{(16\times26)} = p^{416}$.

Use a special Structure. The embedding of grids of processing elements in switching structures to allow fault tolerance has been studied for many years [Manning77]. Although there are a number of algorithms for embedding strings in grids with good harvest [Manning77,Catt78,Lea85a], embedding grids in grids is

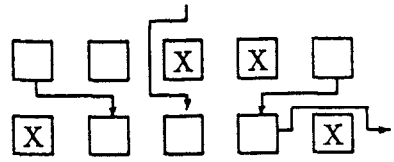


Figure 5–12: Building Good Arrays From Bad.

a much harder problem. Formal analysis suggests that hierarchical tree like structures are necessary as the number of devices gets large [Leiserson85,Jesshope85]. This analysis is predicated on delay scaling with wire length in the reconfiguration structure: this is not as significant if different fabrication techniques are used for wafer sized wires. Simulations [Negrini86], [Lee86] have shown that if additional (not nearest neighbour grid) connections to allow substitution are provided then good harvests can be achieved in reasonable size arrays. The problem is the amount of wiring between the chip sized units: to get good harvest it is necessary to be able to perform many different reconfigurations (figure 5–12). This results in large wiring channels: in [Negrini86] there are 6 logical connections between cells in the vertical direction and 7 in the horizontal direction. Each of these connections represents 32 wires (16 inputs and 16 outputs).

We take the wire pitch as $5\mu m$, this is larger than normal for $1\mu m$ processing since we wish to ensure very high yield. This implies that in a 25 by 26 cell array about $5 \times (16+16) \times 7 \times 25 = 28000\mu m = 2.8cm$ will be used up in the Y direction and $2.4cm$ in X. The area remaining for chips is $(8-2.8)(8-2.4) = 29.12cm^2$ or 45% of the usable area on the wafer. The wiring area could be reduced by using the same sort of multiplexing scheme recommended for pad sharing: perhaps by a factor of two in each direction: in this case 68% of the wafer area could be used for chips. It should be noted that this overhead is not a CAL specific problem: any wafer scale system with wide nearest neighbour grid connections - including many of the systolic algorithms proposed in the literature - would have the same trouble.

5.3.3 Implementation of Reconfiguration System.

At this point we must consider the implementation of the chosen reconfiguration system: there are two main approaches currently under investigation.

5.3.4 Normal Processing

In this approach no special processing capabilities are used for the reconfiguration circuitry. Reconfiguration is provided by transistor switches. Perhaps the best example of systems designed using this approach is the Brunel WSI Associative String Processor (WASP) [Lea85a]. This approach has two main advantages:

1. Availability. The technology for doing this kind of design is readily available.

2. Cost. Since no special processing is involved this is the poor man's way to Wafer Scale Integration. In terms of volume production of wafer designs it also has advantages: wafer specific processing could well be a bottleneck in a production environment.

There are, however, three very important problems with using standard processing technology:

1. Propagation Delays. The wiring necessary to avoid faulty subsystems on a wafer is necessarily long: possibly several centimetres. The characteristics of wires fabricated using normal VLSI technology are not ideal for wires of this length.

2. Power Routing. There are two problems here.

   (a) Switching. Active devices have too high an impedance in the ON state to be used to switch power lines. This means that faulty subunits must be left connected to the power grid. Power ground shorts and other failures which result in excessive power consumption in subunits cannot be ruled out. It must be remembered that the power and ground

nets are the longest in the design and that other mechanisms (e.g. latchup [Glasser85]) apart from metal shorts can cause high power-ground currents in CMOS. In an array structure it is not inconceivable that a 'data' wire might short to a power line in one place and a ground line in another. A power ground short in any subunit could render the whole wafer unusable. Supplying power to faulty and unused units as well as good units will significantly increase the wafer's power consumption even if there are no shorts present.

(b) Area. The metallization on current VLSI technologies was never designed to handle the sort of currents involved in supplying whole wafers of chips. Each chip may require as much as 50 to 100mA if the application is highly concurrent. Attempting to supply the current using standard metallization results in having a large grid of wires each of which is several hundred microns wide. The WASP chip [Lea85b] used a grid of wires $700\mu m$ wide in metal 1 in the X direction and $200\mu m$ wide in metal 2 in Y. If we assume the same size of tracks in our system we would get overheads of $25 \times 700 = 1.7cm$ and $26 \times 200 = 5.2mm$; this would use up 27% of the silicon area. Obviously the figures would be different for Configurable Logic but this approximation is enough to suggest that we look for a better approach.

5.3.5 Special Processing.

This strategy uses extra processing steps after testing of 'chip' size subunits on the wafer to allow a good system to be built. This could be done either by patterning additional layers using electron beam machines [Steinvorth85] or by using laser programmable links [Raffel85]. This approach can tackle both the signal propagation and the power distribution problems.

1. Signal Propagation. The signal propagation problems in normal VLSI devices are caused by the thin silicon dioxide dielectric. If a special polyimide dielectric is used with an intervening ground plane between silicon and the

	VLSI		WSI		
Wire Length (cm)	5	20	5	5	20
Wire Width (μm)	3	3	5	10	10
Wire Sep. (μm)	3	3	5	15	15
Wire Thickness (μm)	0.5	0.5	.75	5	5
Dielectric Thickness (μm)	0.5	0.5	1	5	5
Dielectric	SiO_2		Polyimide		
Propagation Delay (ns)	15	240	5.2	0.3	1.2

Table 5–5: WSI Interconnect Delays.

wiring planes then the interconnect delay can be dramatically reduced. Table 5–5 shows figures taken from [Steinvorth85]. This microtransmission line interconnect with delays even for long distances of the same order as that through a single CMOS inverter has major implications for the conventional wisdom on systems realisable using VLSI/WSI.

2. Power. Even if no other special processing is used it makes sense to use a third metal layer for power wires. This should not be technically difficult because these wires are necessarily wide and thickness is an advantage. Distribution of power on a third metal layer in large gate arrays is now commonplace. It is also useful to have the ability to make and break power connections at the wafer level after testing either by using connections to special power planes as in [Steinvorth85] or by laser cuts and welds on a predefined power grid as in [Raffel85].

These techniques have four major problems.

1. Individual Attention. Since the final processing steps depend on where faulty chips occurred they are wafer specific. Much of the efficiency of VLSI processing stems from the fact that all the chips are identical allowing batch

production techniques. The final customisation could easily become a bottleneck in situations where large numbers of wafers were being produced.

2. Yield of Reconfiguration Circuitry. It is very important that the extra reconfiguration circuitry is reliable. If this circuitry fails then the whole wafer is lost irretrievably after considerable investment in testing the individual chips. This implies that there must either be redundancy in the reconfiguration circuits or an extremely high yield.

3. Infant Mortality. This refers to wafers where components fail shortly after the initial test. When irreversible additional processing is used to configure the wafer there is no way of recovering from failures caused by infant mortality. The additional processing steps can themselves be expected to stress the tested components. It is not clear yet how serious this problem will be: it may be that an extended 'burn-in' will be necessary before final configuration. Note that this would only affect the latency, not the throughput, of a wafer production line.

4. Cost. Extra processing stages inevitably mean that the cost of producing a wafer is increased. However, the situation is slightly more complex since extra wiring layers allow more circuitry to be put on each wafer so in a given system fewer wafers may be required. Additional processing may also increase yield by allowing better reconfiguration structures to be used so fewer wafers have to be manufactured to get a given number of good ones.

5.3.6 Technology Choice.

Since there was never any intention of actually building a Configurable Logic wafer, only of investigating the feasibility of such a design it makes sense to assume the availability of special processing technology where this could significantly simplify the design process. Attempts to design wafer scale systems using completely standard CMOS processing technology are setting themselves unnecessarily hard problems.

Even if one does not want to use wafer specific processing it will be helpful to use a process with additional metal layers. This is not a particularly challenging processing problem because the pitch and thickness of the additional layers can be larger than those of the standard VLSI interconnects. Additional wiring layers have two important benefits: firstly they greatly simplify the power distribution problem and secondly they allow wafer level signals to be routed over the top rather than round the edge of chip level blocks. As we have seen extra wiring channels to support reconfigurability could cost half the silicon area of the wafer. Power supply routing could cost 25%. The analysis is not complete but it is clear that without additional wiring layers the number of chips per wafer will be reduced by about a factor of three: with additional wiring layers the power supply and reconfiguration wiring will cause very little overhead (since it can be done above the functional circuits). The need for additional interconnect layers in wafer scale systems is not surprising: after all conventional technology uses the tracks of the printed circuit board to provide the same sort of functions.

Given additional interconnect layers the architecture could be implemented using any of the reconfiguration strategies mentioned above, dynamic reconfiguration, laser cutting and welding or discretionary wiring. Of these the method which presents the best interface to the designer is discretionary micro-transmission lines.

It is important to note that although extra metal layers will be required wafer-specific processing is not. The only area where wafer specific processing would be useful is breaking power supply lines to faulty chips. This can be done very quickly using a laser: we only require a few hundred cuts per wafer (rather than the hundreds of thousands or millions to pattern a whole mask) so this need not be a bottleneck in a production environment.

5.4 Summary.

The 16x16 CAL array prototype chip covered in this chapter illustrates the viability of the configurable logic architecture. Much larger arrays are possible in a commercial device by increasing the silicon area and taking advantage of better processing technology: 64x64 cell arrays should be possible with leading-edge technology.

The CLA device where the cellular structure is mapped into an array programmed by a single mask being changed was covered. This implementation is competitive with traditional gate arrays in which only a single mask is changed. Much of the advantage comes from the much more general function block. It is not considered that this architecture is viable commercially given the huge investment in tools for traditional gate arrays, however it is possible that large density improvements could be made in such gate arrays by changing to a more powerful basic function block.

The wafer scale integrated version of the CAL technology promises a huge number of configurable cells. Like previous attempts at wafer scale integrated designs what appeared at first to be straightforward turned out to have many hidden costs. Perhaps the most surprising result is that extra interconnect layers are much more useful than wafer specific processing for a WSI CAL implementation. Such units could be used as coprocessors within workstations for specific problems. It is unclear whether Wafer Scale Integrated versions of the architecture can offer sufficient improvement over VLSI versions with state of the art packaging to make the extra processing required cost effective.

Chapter 6

CAD Tools for Configurable Logic.

This chapter will discuss Computer Aided Design tools for Configurable Logic. There are three main topics: a discussion of the applications of CAL's and the CAD tools required for each of them, a discussion of datastructures for representing CAL arrays and discussion of important silicon CAD tools and how to convert them for cellular systems. The tools developed during the course of this project are also covered.

6.1 Applications of Configurable Logic.

Three main applications of Configurable Logic were identified in Chapter 1. In this section we will discuss their distinctive CAD requirements.

6.1.1 EPLD Replacement.

In this application designs will normally have to fit on a single CAL chip. This means that the design size will be small enough to allow manual design optimisation and that efficiency is of paramount importance. This application requires very low level tools analogous to VLSI mask level editors, or assemblers on a conventional computer allowing complete control of the cellular structure.

Subsidiary tools such as simulators and schematic editors are also important. A good example of this kind of environment is that provided by XILINX for their LCA product [Xilinx86].

6.1.2 ASIC Prototyping.

One of the major goals of the configurable logic system is to provide a prototyping capability for ASIC's. This has major implications for the design of CAD software since it is desirable that a system be implemented which can convert a single source format efficiently into either a silicon or a cellular implementation. The tools required for this are dependent to a large extent on the design style of the silicon implementation. There are two main approaches to this problem: either one designs a cell based system with a back end which can generate silicon or one attempts to build support for CAL into an existing silicon system.

Cell Based System. The main advantage of a cell based system such as the Configurable Logic Array (CLA) discussed in Chapter 5 is the direct one to one correspondence between the design emulated by the CAL and the final silicon implementation. This gives a very high probability of success on first silicon and provides a smooth migration for CAL EPLD designs into ASIC's. Since the CLA system need only involve a single mask change it can provide low cost and fast turnaround for relatively low density designs. Gate net lists can also be extracted from the CAL design and used as input to silicon compilers for higher density ASIC's where all the masks are changed.

This approach has several drawbacks:

1. Efficiency in Silicon Layout. Efficiency in the utilisation of silicon area in full custom designs is dependent on making use of special array structures such as PLA's, RAM's and ROM's. By the time a design has been reduced to the cellular format the information necessary to make use of these structures is no longer present. To make use of such structures the cellular and silicon design processes would have to be separated at a functional rather than a

structural specification level. A direct correspondence between cellular and silicon implementations also prevents the use of many other important circuit technique within the silicon design.

2. CAD System Complexity. Many important and difficult CAD problems (notably behavioural compilation, floorplanning and global routing) are specified almost identically for silicon and cellular designs. Implementations of algorithms for these problems are available within existing silicon design automation system and have required many man years of development. Many engineers are already familiar with silicon design automation systems. For these reasons it makes more sense in terms both of functionality provided to the user and ease of implementation to build support for configurable logic into an existing silicon design automation system rather than produce a completely new CAD system.

Silicon Based Systems. Here we are attempting to add CAL support to an existing silicon design automation system. It is desirable that any legal design in the silicon system can be emulated using the CAL, since any mismatch will cause considerable inconvenience to users of the system. However, there are several important areas where this cannot be achieved with the current CAL design:

1. Wired Logic. The CAL is a completely gate based system and the routing network prevents gate outputs being connected together. Thus CAL cannot emulate structures such as three-state, open drain and precharged buses directly. Gate based silicon systems have trouble here as well: for example MODEL ([Lattice86]) needs a special wired-or library part to get round its normal checks on output connections when transmission gates are used. Precharged and static open drain wires usually only occur in manual silicon designs. Gate level simulation programs do not handle such structures well either. It may appear that the design of the CAL should be changed to allow three state lines: however (as was explained in Chapter 3) provision of bidirectional lines would cause a large increase in the control store requirements.

Note that the dynamic (precharged) CMOS logic families which are gaining popularity can be emulated since the problem is with connecting multiple gate outputs together rather than precharging *per-se*.

2. RAM's and Register Files. Most silicon compilers have macros for these functions in their library and many user designs take advantage of them. Emulation of large stores cannot be done efficiently by the current CAL architecture. A partial solution might be to allow the CAL control store to be used as a memory within user designs on a per chip basis (i.e. a given chip looks like a memory rather than a block of logic to other chips in the system). Another solution would be to provide RAM chips connected into a board level switching system on an emulation product.

3. Analogue Parts. It is becoming fairly common for ASIC's to include some simple analogue functions such as operational amplifiers or analogue to digital converters: obviously a completely digital system such as CAL cannot emulate such devices.

We will now consider CAD support for the more common silicon design styles.

- Gate Arrays. The first kind of design is the 'gate-array' or 'cell-array'. The floorplan of these devices is very simple and they are composed of macrocells (normally all the same height) from a library. These macrocells are implemented either as leaf cell designs or by custom metallization on an array of two transistor stages. This design style is very suitable for 'TTL-like' blocks of random logic.

  A very simple approach to emulating this style of design would be to manually implement all the library functions as cell designs. Each unit would be the same height and they would be composed in rows using exactly the same placement as the silicon design. Row feed throughs can be handled by adding columns of cells between library parts or using spare vertical connections within the library parts. The rows of library cells could then be connected using the channel router (described in section 6.8.1). This method

would be extremely easy to implement and give the close correspondence between the silicon and cellular designs which is desirable in this application. One drawback of this approach is that it would not be particularly efficient in terms of cell utilisation. This could be improved by a final optimisation phase but it may well turn out to be more convenient just to provide more cells.

- Sea-Of-Gates. This design style differs from the gate array in that there is no separation between wiring and logic areas: instead the whole chip area is covered by a 'sea' of transistors which can be built up into gates or ignored based on metal personalisation. The fact that wiring and functional areas are not separated potentially allows better use of CAL resources (normally cells in routing areas will not have their function units used). More complex, placement and global routing algorithms are required for this structure than the normal gate array.

- Full Custom 'Mega-Cell'. In this design style chips are built up from large library elements which could be RAM's or standard microprocessors. The ability to use familiar 'catalogue' components within silicon designs is a major advantage to novice IC designers. Emulation of such systems using CAL is problematic: a much better approach is to use board level prototypes with the corresponding catalogue parts. CAL could be used to emulate the extra blocks of random logic 'glue' which are normally required in these designs.

Emulation Speed. Configurable structures are inherently slower than fixed structures implemented in the same technology because of the additional switching systems. Thus, in the general case one cannot expect real-time emulation of target systems. Many ASIC designs, however, run much slower than the technology would allow. One important reason for this is that CMOS power consumption is proportional to clock speed and many systems value low power consumption more than speed. Another point is that CAL chips can be designed manually (because

of their regular array structure) as catalogue devices with architectural support from the processing technology (e.g. buried contacts for the RAM cells and special high-conductance switching transistors). ASIC's, on the other hand, normally use 'lowest-common-denominator' technology to allow a degree of process portability.

We would expect many ASIC designs to be emulable in real time and certainly emulated much faster than simulation using traditional hardware accelerators. Of course, CAL emulation is mainly a functional test and cannot give the detailed timing information about individual nets which is available from simulation.

6.1.3 Algorithm Implementation.

The difficulty of algorithm implementation using CAL depends to a large extent on the complexity of the control flow within the algorithm. A large number of 'cellular automata' and 'systolic' algorithms are known for important problems (see, for example, the bibliography in [Wolfram86]). These algorithms provide high performance based on an array of relatively simple processing elements. Direct implementation of these elements using CAL is often attractive. Normally these cellular implementations would be carefully hand optimised because the basic unit is simple and is repeated tens or hundreds of times.

Implementation of algorithms with more complex control flow or the need for significant storage is more difficult. An attractive architecture for this application is a normal microprocessor with a block of Configurable Logic acting as a coprocessor. Only speed critical 'inner-loop' code would be executed using configurable logic. Clever 'active' compilers would detect suitable loops based on the simplicity of the operations within them and high repetition counts. The structure of such a compiler is discussed in more detail in the next section and an example of the coprocessor architecture applied to an image processing problem is presented in Chapter 7.

6.2 Active Compilation.

In this section we will discuss the components of the software system required to convert critical components of an algorithm written in a high level programming language into a cellular implementation. Some of the ideas in this section are taken from a paper written in conjunction with Viitanen [Viitanen88b]. The term 'active-compiler' was coined by Gray [Gray88] to describe the complete system - it is 'active' because the result is the connection of active logic elements rather than a passive byte stream to be interpreted by another unit.

Historically, many research projects have been carried out into behavioural compilation for catalogue part and silicon designs, notably the work at Carnegie Mellon University on the CMU-DA system and related projects [Thomas83]. Recently, a highly developed system has been produced at IBM Yorktown Heights [Brayton86b]: this system features novel multi-level logic synthesis techniques and is probably the most fully engineered behavioural system available. We will model our discussion after the IBM compiler and another interesting system from Linkoping University [Peng88].

Figure 6 illustrates the proposed CAL program development process from high level language source to configuration information. This diagram is typical of most silicon behavioural compilation systems. The process is normally split into two parts: in the first part the behavioural representation is converted into a structural one (e.g. a hierarchical netlist of gates) and in the second (termed cell assembly) the structural representation is converted into physical layout. Tools for the second phase are fairly well understood so we will concentrate on the behavioural compilation step and take each component of the diagram in turn.

6.2.1 Source Language.

The choice of source language is central to the active compilation system: there are several schools of thought about the most desirable source language.

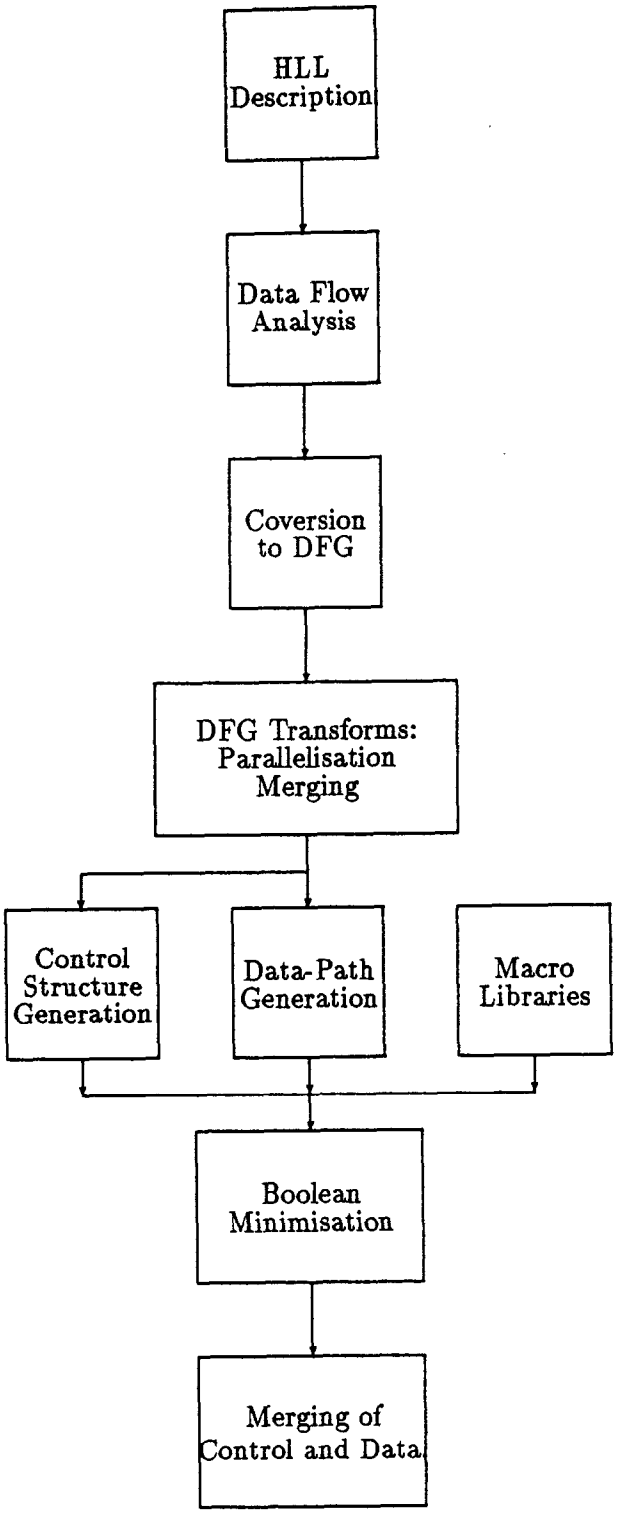


Figure 6–1: Behavioural Compilation Process.

Conventional Imperative Programming Languages. Use of a conventional high-level language such as C is the most attractive choice from the user's point of view since it could potentially allow him to get the benefit of the CL accelerator without making any changes to existing programs. This is the approach suggested in [Viitanen88b], however it is impossible to make full use of the CL accelerator using this method. There are several areas in which problems occur.

1. Word Length. One of the major efficiencies of programmable hardware implementations arises from the fact that operational units need only be as wide as required by the problem in hand. For example, in the image processing problem of [Viitanen88b] (Chapter 7) five bit comparators are required, in a C program to implement this algorithm the operands would probably be specified as 'int' (16 bits) or 'long' (32 bits). The extra precision has no cost in run time and negligible cost in space on a conventional processor but in a CL implementation area and time would both be about 6 times greater for a 32 bit comparator rather than a 5 bit comparator. The active compiler when presented with C source would have no way of telling that the extra 27 bits were redundant. To get maximum efficiency from a CL implementation the source language must provide as much information as possible about operand sizes as well as such factors as whether signed or unsigned arithmetic is required and handling of overflow conditions. An example of a high level language allowing this sort of information to be specified would be ISPS (Instruction Set Processor) which was developed to describe the instruction sets of conventional processors [Barbacci78].

2. Parallelism. Extraction of potential parallelism is central to use of configurable logic as a hardware accelerator. This can be done by simple data-flow analysis of the code but there are problems: normally programs written in imperative languages have little scope for parallelisation. One reason for this is that algorithms which run efficiently on conventional computers get no benefit from potential parallelism but can be speeded up by techniques which reduce it - e.g. complex branching sequences to 'special-case' the

computation performed. Other techniques which produce speedups on conventional computers such as the use of large lookup-tables to trade-off space against time are also inappropriate on configurable logic. To get the maximum benefit from parallelism it must be considered at a high level when the algorithm is being chosen: for example a C programmer might specify a quicksort algorithm where a hardware implementation would be faster using a distributed bubblesort. It is often impossible for a programmer to write efficient code without knowing which processor it will run on.

Language with Explicit Concurrency. In this approach the language allows the programmer to specify exactly which operations proceed in parallel. An example of such a language would be OCCAM [Inmos84]. This technique makes programming harder but will probably result in much more efficient compilation since the programmer can use extra knowledge about the problem to determine which parts run on the coprocessor. Use of OCCAM with transputer hosts and a self-timed design style within the programmable blocks could result in a very clean interface between the parts of the program running on the transputer and those running on the configurable structure. If OCCAM was extended to include extra information about operand sizing it would provide a very attractive source language from the point of view of efficiency of CAL utilisation and ease of active compiler implementation.

Functional Language. Use of functional languages for hardware description has been suggested because of the ease with which maximum parallelism can be extracted from them and the perceived advantages of these languages. Functional programs are normally written at a very high level with little or no concern for efficiency and it would require very advanced software to translate them into reasonable hardware implementations.

6.2.2 Choice of Code Segments.

There are several important criteria on which the choice about which sections of the code to run on the coprocessor will depend:

1. Simplicity. The section of code must be simple enough to fit on the configurable logic array. Sections of code with complex decision structures are less suitable than in-line code since, for efficiency, we must keep the control section of the cellular implementation much smaller than the data-path.

2. Heavy Computation. There is no point in mapping a section of code onto the configurable coprocessor unless it uses a great deal of computational effort. An important reason for this is that loading the configuration information into the coprocessor will be quite time consuming - we need enough speed improvement to compensate for this overhead. Detection of code 'hot-spots' is normally impossible from static analysis of source code, a better approach would be to make use of 'profiling' information taken from the algorithm running on the conventional processor with normal input data.

3. I/O Overhead. In an architecture in which all I/O from the coprocessor is routed via the host processor there are many simple operations which would run faster on the host than the I/O instructions to read and write data to the coprocessor. This consideration puts a lower bound on the complexity of coprocessor operations as well as the upper bound which comes from the limited size of the CL array. If a separate path is provided between the coprocessor and memory then the bottleneck is removed and we also have the potential for parallel operation of the normal processor and the coprocessor.

4. Possible Speedup. There must be a reason to suppose that the CAL coprocessor will be able to offer a speedup for the given loop. The easiest way to determine if a speedup is possible would be to implement all sections of code which meet the first two criteria and calculate the expected performance. Only those sections which showed a clear speedup would be implemented in

the final code. This approach would involve a lot of redundant computation so it may be necessary to use some additional criteria to cut down the number of sections considered.

With present technology, only loops or parts of loops with 20 to 30 instructions and iteration counts over a few hundred would be considered for running on the CAL coprocessor.

6.2.3 Synthesis Operations.

At this point we will assume that the program sections to be implemented on CAL have been selected and that the source language has been compiled into an intermediate data flow graph format representing the necessary precedence relations between the computations to be performed. Examples of such intermediate forms are the Value-Trace (VT) in the CMU-DA system, Yorktown Intermediate Format (YIF) in the Yorktown Silicon Compiler and Extended Timed Petri Nets used in [Peng88]; the information encoded in these formats is essentially identical so we will consider a generic Data Flow Graph (DFG) format in this discussion.

In order to obtain reasonably efficient hardware realisations it is necessary that many operations in the data-flow graph be performed by a single physical unit. Maximally parallel and minimally parallel implementations are normally equally unacceptable: the key task is to trade off area against parallelism (and hence speed) in a way which is appropriate for the current system. There are two steps in figure 6-1, where this merging of operation can be done. The first comes, when the sequential program is transformed to the DFG description. This utilises the control structure of the DFG and compresses suitable 'inline' sections of the program with no external data dependencies to single operational units. The second merging step comes with the Boolean minimisation. Here, additions by a constant and similar ALU-only operations can be merged with the following operation. Programmable logic circuitry handles several variables at a time providing another useful speedup over traditional processors. A third merging phase may also be desirable where multiple units with data-dependancies are merged. Naturally,

this reduces the amount of parallelism present and potentially reduces speed but it can also drastically cut down the amount of hardware in the implementation and the more compact unit could well be faster because of reduced routing delay. This merging phase can be done by manual intervention (perhaps using graphical tools to manipulate the DFG) or automatically using 'expert' systems or other heuristic techniques.

The example in Chapter 7 of hardware to search for the minimum of five words after adding a constant to each one is a good illustration of these optimisations. Several comparison operations can be merged into a single Boolean expression over all the five input words. The expression can be optimised at compilation time and the resulting logic function can be assigned to CAL cells. The efficient automatic realisation of such boolean expressions has only recently become feasible with advances in multi-level logic synthesis techniques [Brayton86b,Gregory86]. A sequential program for this operation would take at most two words at a time for processing, and consume several cycles for each suboperation. A parallel processor implementation would distribute partial comparisons to different processors and thus introduce a major communication overhead. The CAL implementation is clearly better than both the traditional sequential and the parallel approach.

6.2.4 Control Structure Realisation.

The control structure seeks to implement the DFG graph on the physical hardware mapping instances of computations on the graph into computation slots on physical units and routing the data to and from these units. Two implementation styles are common:

1. Data-Path / Control Path. In this approach the classical processor architecture of a data path controlled by a wide instruction word from a controller is used. This model is relatively easy to understand and implement but has two considerable disadvantages. Firstly, it does not cope well with multiple state machine systems. Secondly, in a cellular implementation delay within

the controller and on the lines between the controller and the data path would be almost certain to be greater than data-path delays.

2. Data Flow (Self Timed). In this methodology 'go' and 'done' control signals are routed side by side with data signals. Note that one control signal is usually associated with a whole word of data signals. Special self-timed operators are used to implement the basic control operations of join, split, conditional, loop [Seitz80]. This distributed control structure allows maximal parallelism in a pipelined implementation and does not suffer from excessively long control signal routing. Self-timed structures are especially suitable for our system because we are implementing only small blocks of code with very simple control structures. Traditional state-machine controllers are much more area efficient when complex decision structures are necessary: extra decisions require extra logic elements and associated routing in a distributed self-timed structure but only extra memory locations in the control store of a state-machine.

6.2.5 Cell Assembly.

After the optimisation of the Boolean expression, we have a complete structural description of the design in terms of a netlist of logical units capable of being implemented by the primitive cells in the target array. The next steps in the process could be termed cell-assembly and consist of floorplanning, global routing and local placement and routing of functional cells. Floorplanning and global routing are high level processes applied to large hierachical structures (e.g. our five way comparator) within the structural description. Given this high level plan detailed placement and routing within the large substructures and channel routing to connect them up into the final design is also required. Usually, heavy computation is needed in automatic placement and routing - techniques such as simulated annealing [Kirkpatrick83,Brayton86b] are often used in the floorplanning step to ensure good results. Good placement of the computational units is very important since excessive delays will result from long wires. Minimising the computation involved

is also necessary in a system like ours where frequent recompilations will occur as the program is developed. It is at this point that the advantages of the CAL architecture as a target for 'active' compilers become apparent:

1. The architecture scales transparently over chip boundaries. Realistic size systems will never fit on a single programmable chip given the overhead of the configuration memory thus it is essential that multi-chip systems be supported. Architectures which use 'special-purpose' input-output blocks are unsuitable for large systems since single units (for example large logic blocks) in user designs will be hard to split over multiple chips.

2. The architecture is completely symmetrical: this is important when floorplanning large systems since it allows large subunits to be rotated and reflected to obtain a dense packing. Algorithms for floorplanning silicon designs take advantage of this flexibility.

3. There is a single resource in the system. Large units are built up by composing small resources rather than breaking up large ones. One area where this is particularly important is channel routing. In a large design channels with several tens of tracks are likely to occur: in the CAL architecture there is potentially no limit to the number of tracks in a channel, although each additional track may require an additional line of cells (often two tracks can be fitted in a single line of cells). In an architecture such as the LCA with special fixed width wiring channel resources problems occur when that width is exhausted possibly resulting in routing failure or grossly inefficient use of resources.

4. The routing model is simple and safe. Routing in a CAL design is simple compared with other EPLD architectures such as the LCA [Xilinx86]: there is only one class of routing resource so there is no question about which is the most suitable for a given signal. All paths are fully buffered so there is no need to worry about logic levels. These factors are important because they allow the use of standard 'channel routing' algorithms which can produce high quality routing relatively quickly. More complex architectures can still be routed automatically using 'maze' routers but the results are likely to be worse and computation time significantly longer.

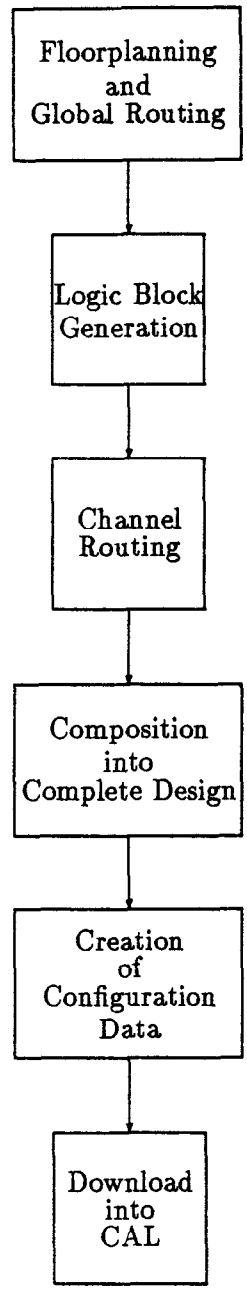


Figure 6–2: Cell Assembly Process.

6.3 Data-Structures for Describing CAL.

This section discusses the choice of a data structure and file format to describe the configurable logic arrays. This data-structure will be used by all the major design tools and is a critical component of the CAD system. There are several important design objectives:

1. Ease of Insertion. This is important for tools like graphical 'leaf-cell' editors where it will often be necessary to insert cells in the middle of an existing structure 'pushing' the existing cells out of the way rather than overwriting them.

2. Ease of Locating Neighbouring Cells. In many algorithms (for example, maze routing) it is essential to be able to locate the neighbours of the current cell quickly.

3. Overlapping. To take full advantage of the available cell resources it is useful to allow blocks of cells to overlap. This can often simplify design by allowing a library of common partial designs to be kept which can be composed by overlapping. The application of this technique is illustrated by the DES encryptor design in Chapter 7 where global routing wires are overlayed on large logic blocks.

4. Hierarchy. Hierarchy is essential to reduce the size of the design representation and minimise design effort.

5. Space Efficiency. Naturally, it is important to minimise the space requirements of the data structure to allow large designs to be done. Even on systems with virtual memory large IC designs can approach memory limits and some of the applications of CAL will require designs of similar or greater complexity.

Several data structures were considered:

1. Static Array. Normal (compiler allocated) array structures do not handle insertion of new rows or columns well and require a lot of copying to make space in the middle of an exisiting design. Another major problem in a hierarchical design containing many blocks of widely varying sizes is that the size of the array must be set large enough to hold the largest conceivable block. The store required is therefore determined by the maximum block size and the number of blocks so a hierarchical design with a large number of small blocks is not represented efficiently. This data structure also has problems with cut and paste editing operations which can gradually move a design across the plane. A design may start at (0,0) with length and width of 10 cells and finish up at (100,100) with the same length and width - the size of the array must be at least (110,110) to cope with this movement.

2. Sparse Array. Sparse arrays are more suitable for interactive design systems since the size of the array is effectively unlimited. Coupled with 'normalisation' to (0,0) offset for block bottom left co-ordinates when designs are stored this can solve the array size and 'creeping' problems. In some implementations insertion of extra rows and columns can be done without copying. Implementations can allow several array entries to have the same offset to support overlapping subunits. Indexing into a sparse array is significantly slower than indexing into a static one.

3. Corner Stitching. Corner stitching is a very popular data structure for VLSI leaf cell editors [Hamachi85]. Its realisation is more complex than the other structures but it can support all the requirements except overlapping very well.

4. Slicing Tree. Representation of the cellular structure as a hierarchical slicing tree was the method chosen in the initial implementation of the Configurable Logic tools. This decision was motivated by the chip construction tools described in [Wardle84]. A divided hierarchy with 'leaf' blocks containing only cell instances represented using sparse arrays and hierarchical blocks containing only instances of other blocks is also attractive since many

placement and floorplanning algorithms work with slicing tree data structures whereas channel routing and logic synthesis programs would prefer the array representation. The major disadvantages of this approach are its inability to handle 'cyclic' structures or overlapping blocks.

The sparse array data structure was considered to be the most suitable for the final implementation. It is implemented simply using a list of lists in SML [Harper86] - much more efficient pointer based implementations are possible in imperative languages. The file format is a fairly direct textual mapping of this structure. The file format is flexible in that it can describe the system at several different levels. This allows the the front end tools to use the same file format to describe the system before it has been totally converted to a cellular implementation.

The system is described as a collection of blocks. These blocks may contain instances of other blocks and base 'leaf' cells. An example CFG file is given below (this is the description of the toggle register shown in figure 7–1).

```
BLOCK toggle
BPORT Q OUT 2
RPORT Q OUT 1
RPORT CLR IN 1
LPORT CLK IN 1
ENDPORTS
CELL (0,0)
ESOURCE self
X1SOURCE north
X2SOURCE east
FUNCTION dlatch
ENDCELL
CELL (1,0)
ESOURCE west
WSOURCE self
X1SOURCE north
X2SOURCE east
FUNCTION nor
ENDCELL
CELL (2,0)
WSOURCE self
NSOURCE self
SSOURCE self
X1SOURCE north
X2SOURCE west
FUNCTION dlatch
ENDCELL
CELL (0,1)
WSOURCE east
ESOURCE west
SSOURCE self
X1SOURCE east
X2SOURCE west
FUNCTION or
ENDCELL
CELL (1,1)
WSOURCE east
ESOURCE west
SSOURCE east
ENDCELL
CELL (2,1)
WSOURCE east
ESOURCE south
SSOURCE self
X1SOURCE east
X2SOURCE west
FUNCTION x1orx2bar
ENDCELL
ENDBLOCK

BLOCK test
ENDPORTS
INSTANCE (0,0) NULL toggle
INSTANCE (3,0) NULL toggle
ENDBLOCK

ENDOFFILE
```

There are several elementary components in the data structure.

Cells. Each basic cell is described by the source selected by each of its multiplexors (including the *G1* and *G2* global signals in the case of the X1 multiplexor) and the function it is performing. The keyword FTEST is used to indicate that the function block output of this cell is to drive the global *FTEST* output signal.

Instances. Block instances have a transformation (mirror in X or Y, rotate by 90,180 or 270 degrees) and the name of the instanced block.

Blocks. The cells within a block are described as a sparse array of 'entries': an entry can either be an instance of another block (possibly rotated or reflected) or a 'leaf' cell. Each entry has an offset from a notional origin associated with it. Normally the bottom left cell will be at $(0,0)$ but this is not forced. Within graphical editors a common technique is to normalise offsets so that the lower left is at (0,0) when reading in blocks from a file then add a very large number (e.g. 1/2 maxint) to all coordinates. This allows structures to be added below and to the left of existing blocks without having to support negative offsets (which can complicate arithmetic in display operations which have to be fast). Overlapping implies that there are potentially multiple entries in the data structure with the same offset from the origin.

This data structure allows for fairly arbitrary overlap between components of a block: this can sometimes be very useful but their are obvious problems with clashes between overlapping cells. The behaviour of the system when clashes occur is not specified but in the absence of clashes overlapping blocks will 'add' together specifying the use of more of the cell resources. Hierarchical blocks can be flattened into fully instantiated blocks containing only leaf cells.

Ports. As well as information about the structures contained within them blocks have port information. Each side of the block has an associated port list. Ports are represented as an offset, a name and a type according to whether they are inputs to

or outputs from the block. One major disadvantage of this implementation is that cut-and-paste operations which move cells within the block must also manipulate a separate port data structure whereas if ports were associated with cells they would be updated automatically when the cell moved. The advantage of the present approach is that a block with ports and no cells can be used as a specification of a wiring problem for the channel router: which will then return a block with the same name and cells which implement the required connections.

6.4 Floorplanning and Global Routing.

Floorplanning is a high level placement process to decide the relative positioning of large subunits within a system. Floorplanning is distinct from other placement problems because the exact size and shape of the subunits is usually unknown: decisions are made based on size estimates and connectivity between subunits. There are two main approaches to the problem. Real systems will often have to work with a combination of fixed blocks and flexible blocks but one style is usually predominant.

Top Down Systems. In these systems the floorplanner is in control of the rest of the cell assembly system. Based on netlist information about the interconnectivity of blocks and number of gates within them the floorplanner formulates a proposed placement. It also determines target areas, aspect ratios and port placement information (things like which side a port should appear on and port ordering but not actual offsets) for the blocks which are passed to lower level 'cell-assemblers' which attempt to produce layout for individual subunits within the constraints imposed by the floorplanner. This method of floorplanning is typical for the 'sea-of-gates' design style.

Bottom Up Systems. In this case the floorplanner is faced with macrocells for subunits of fixed size and shape and the interconnectivity between them and

must attempt to find a good placement. This method is required for library based systems.

The floorplanning problem is specified identically for cellular and silicon designs and the same program can easily be used although some fine-tuning of the cost functions may be necessary to take account of the greater speed penalty on long interconnections. For this reason a special floorplanning program for CAL is not required or desirable.

6.5 Logic Synthesis Methods.

In this section we will consider the mapping of truth-tables or logic equations into array structures of basic cells. These techniques are of interest because the use of a fixed array structure reduces the complexity of the computation required to perform the logic synthesis by cutting down the search space and also does away with the need for a separate place and route stage to layout a netlist of gates. Array based logic synthesis methods (like PLA's in VLSI layouts) are normally used for large unstructured functions such as those found in sequencers.

This section will summarise various logic synthesis techniques which have been suggested in the literature suitable for use with the 2 input 1 output flexible gates provided by our cells, the next section will cover the logic synthesis tools provided for the CAL system. A very large number of papers have been published on this topic and here we will only attempt to provide an overview: the early papers cited here use widely varying notations and are of mainly historical interest.

Most early papers concentrated on the problem of synthesising a given function $f : \{0,1\}^n \rightarrow \{0,1\}$ using a particular architecture and failing if no synthesis was possible. This is not what is required in a practical system: we need an algorithm which can split up unsolvable synthesis problems into several solvable ones and combine the results. It is also necessary to be able to synthesise a solution for $f : \{0,1\}^n \rightarrow \{0,1\}^m$ with sharing of components between the individual one-output problems.

6.5.1 Cascades.

This is a very simple array structure capable of synthesising arbitrary functions. A cascade is a line of two input one output gates each of which takes one input variable x and a sub-result y. Most functions are not directly cascade realisable but they can be partitioned into several cascade-realisable functions which are then combined using an OR (or AND) collector row. Only 6 of the possible 16 two input one output functions are necessary to synthesise all cascade realisable functions [Minnick64]. There are two classes of cascades.

Redundant Cascades. In this class of cascade input variables are allowed to drive more than one column of cells. This allows permutation of inputs in different cascades which significantly increases the number of realisable functions. Obviously, this comes at considerable cost in array area and so these arrays are of mainly theoretical interest.

Irredundant Cascades. In this class of cascade input variables drive a single column. This is consistent with a very simple array layout. There are $\frac{(2\times6^n+8)}{5}$ functions which can be synthesised using n gate irredundant cascades [Maitra62].

6.5.2 Trees.

Trees are the next step up from cascades. Many different kinds of trees have been examined based on various decompositions of boolean functions (e.g. the Shannon Decomposition which gives rise to 2:1 multiplexor based trees). Tree designs require more complex wiring and are not as suited to array structures. Trees offer more directly realisable functions than cascades. Proposed synthesis algorithms fail to suggest decompositions of functions not directly tree-realisable. Like cascades both redundant and irredundant trees are possible

6.5.3 Tandem Nets.

Tandem nets are a very interesting generalisation of the cascade due to Butler [Butler78]. The problem with cascades of gates is that only a small fraction of switching functions are directly cascade realisable. Butler examined possible ways of adding extra Universal Logic Modules to maximise the number of functions realised. His technique requires at worst three modules for every one module in a cascade - cascades are special cases of his nets and can be implemented as efficiently as before. Transforming into an array layout we need, in general, two rows and two columns for every one row and column in a cascade array. It would be necessary to allow two columns right through the array and one or two rows according to the function being realised. The advantage is that many more functions can be realised: in fact Butler shows that as the number of input variables n increases $\lim_{n\to\infty} \frac{N_{CAS}}{N_{TAN}} = 0$. Unfortunately, Butler does not give a synthesis technique capable of breaking up general functions into a composition of tandem realisable ones. There is no reason to suppose that such a technique could not be found, however, and it could be worth looking for one.

6.5.4 Irregular Structures.

In this class of system there are no limitations on gate connectivity. It should be noted that it can be proved that minimal implementations (in terms of gate count) of even monotone boolean functions can require feedback [Rivest77]. Synthesis algorithms impose restrictions on gate interconnect, however, and no known algorithm can take advantage of feedback. This class of designs has become popular for irregular large functions only recently with the development of algebraic techniques for logic synthesis [Brayton86a], [Brayton86b]. It is questionable whether the overhead of the irregular wiring needed to support these structures outweighs the advantage of the increased number of functions which can be realised with the same number of gates.

n	*Cascade*	*Tree*	*Tandem Net.*	*Possible.*
2	16	16	16	16
3	88	152	240	256
4	520	2,680	6,448	65536
5	3,112	65,208	187,184	4.29×10^9
6	18,664	-	5,474,096	1.84×10^{19}

Table 6–1: Number of n Variable Functions Realisable.

6.5.5 Summary.

Table 6–1 shows the number of n-variable functions realisable with the main techniques described above. These figures are taken from several papers whose authors use slightly different definitions of the various categories: some authors, for example, allow variables to be input to the y input of the first gate in a cascade where others insist that it is held at a constant value. These differences can account for small differences in the number of functions claimed but the rates of growth are not in question. The reason for using the results verbatim and not converting to a common set of definitions is that the numbers are calculated from complex recurrence relations which are hard to derive and solve.

6.5.6 Binary Decision Trees.

Binary decision trees or diagrams provide a general method of describing logic functions with significant advantages over standard methods such as truth-tables and logic equations [Akers78]. The function is viewed as a tree of decisions to be made based on input variables. Sequential functions can also be described. There are very simple algorithms for generating multiplexor based implementations from these diagrams and for generating these diagrams from truth tables or logic equations. Some work has been done on generating array layouts for these trees [Oldfield83]. Many algorithms have been suggested for optimisation of de-

cision diagrams [Payne77,Cerny79,Thayse81]. None of these were developed far enough to be used in practical systems.

6.6 The Logic Synthesis Program.

In this section we will deal with the simple logic synthesis tools developed for CAL. The logic synthesis system generates a rectangular array of cells which implements a function specified as a minimised sum-of-products in the format output by ESPRESSO [Hamachi85]. The use of this standard format will allow CAL's to prototype systems intended to be implemented using PLA's. It also allows the use of tools such as state-machine compilers (e.g. PEG [Hamachi85]) which generate truth table output.

As in PLA designs the width of the cellular array is determined by the number of inputs and the number of outputs and the height by the number of 'product' terms. Since the width is fixed the goal of the algorithm is to reduce the height by minimising the number of product terms.

Single Output Function Algorithm. The program uses a generalised version of the cutpoint array algorithm in [Papakonstantinou72] this produces a cellular implementation of a single switching function from a minimised sum of products representation. The multiple input gates in the sum of products representation are first converted to cascades of two input gates giving an initial cutpoint array implementation. The implementation of the function

$$f = \overline{x_3}.\overline{x_4} + \overline{x_1}.x_2.\overline{x_4} + x_1.\overline{x_2}.\overline{x_4} + \overline{x_1}.\overline{x_2}.x_3.x_4 + x_1.x_2.x_3.x_4$$

is given in figure 6–3. This example is taken from Papakonstantinou's paper: note that in his model of cutpoint arrays constant 0 values are input on the left side of the cascades in the 'AND' plane and the top of the 'OR' plane and only the functions given in table 3–15 are available. The logic synthesis program avoids the need to route 0's to the edges of the array by using some additional cell functions but we will assume Papakonstantinou's method in this example.

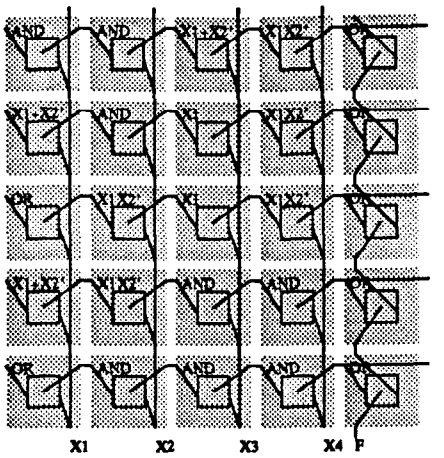


Figure 6–3: Stage 1.

The number of 'product' terms is reduced by 'merging' two cascades of gates into one using one of three rules given in the paper. Such merges are possible because of the much larger number of functions which can be realised using a cascade of general gates rather than just X_1X_2, $X_1\overline{X_2}$ and X_1 gates. The rules and the supporting mathematics are too long to quote here but we will present their application to the function above which can be verified by hand. In this example we can merge the first two 'product' terms into one giving figure 6–4. We can then merge the term resulting from the first merge with what was originally the first term giving figure 6–5. Two more merges are possible (figures 6–6 and 6–7) resulting in a single row implementation.

Papakonstantinou gives some statistical results for his algorithm which indicate that normally the merging rules will give an optimal or near optimal cascade implementation of single output functions. In use he obtained a 27% reduction in 'product' cascades over a minimal sum of products implementation. Cascade implementations of logic functions are sensitive to the order of input variables so to get optimal results it is necessary to run the algorithm for all possible permutations of input variables: this is only realistic when the number of inputs is small. Note that it is not necessary to reminimise the function to get another sum of products representation for each case: only the merging rules need be re-applied for each variable permutation.

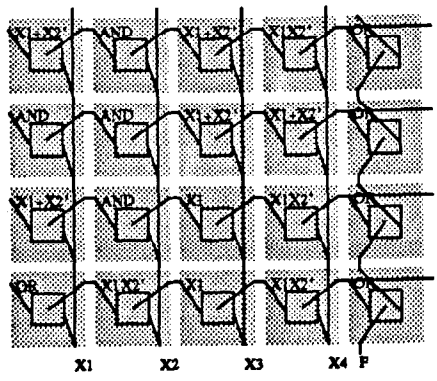


Figure 6–4: Stage 2.

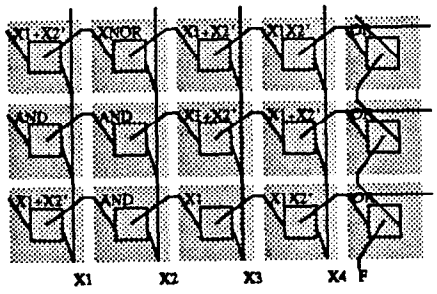


Figure 6–5: Stage 3.

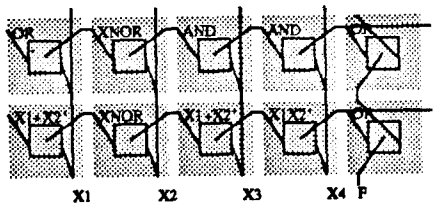


Figure 6–6: Stage 4.

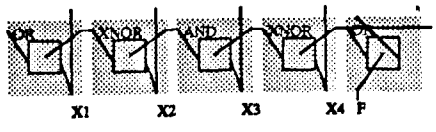


Figure 6–7: Stage 5.

Extension to Multiple Outputs. The key extension which must be applied to the algorithm is to make it compute several functions over the same set of input variables making the best use of product terms shared by several output functions. There are two obvious ways of approaching this problem:

1. Minimise each switching function separately and produce a cellular implementation for each one. Eliminate duplicate product cascades and build an array.

2. Minimise the switching functions together. Find all sets of product terms which always appear together and attempt to find merges within these sets. Use these merged sets to build an array.

Both these methods of extension have problems: the first method does not specifically choose good representations of the functions to allow a high degree of sharing of product terms; the second suffers from the fact that the sets of terms which always appear together are much smaller than the set of minterms which realise a single switching function and therefore merges are much less likely to be found. The present system uses the second method.

Other Extensions. The rules in Papakonstantinou's paper do not take advantage of some very significant possible savings from the use of general gates: general gates are only used in the AND plane but all the gates in the OR plane are identical.

If we allow $X_1 + \overline{X_2}$ gates to be used as well as $X_1 + X_2$ gates in the OR plane an important saving arises. If two 'product' terms are complements then one can be eliminated and an $X_1 + \overline{X_2}$ gate used instead of an $X_1 + X_2$ gate. If a function can be realised by a single cascade of general gates then so can its complement, the argument for this assertion is as follows.

1. If general 2 input 1 output gates are available then you just need to change the last gate in the cascade to invert the function (e.g. AND becomes NAND, OR becomes NOR etc.).

2. Maitra [Maitra62] showed that if a function was cascade realisable using all 16 possible gates then it was realisable using the six functions above.

This situation is different from the standard 'AND' type product terms where the complement of a function realisable with a single product term often requires several product terms to implement. Unfortunately, the number of cascade realisable functions grows as 6^n whereas the maximum number of product terms grows as 2^n so unless n is small we can expect very few merges. For this reason this technique was not implemented in the configurable logic system.

Another possible extension, not implemented in the current system, is to generate the cellular implementations of the product terms for AND collector rows as well as OR collector rows. The important point is that these will be different from the product terms in the OR collector implementation. With two implementations to choose from the program has more chance of finding shared product terms between switching functions. Of course, $X_1\overline{X_2}$ gates can be used as well as X_1X_2 gates to allow complementary functions to be eliminated as in the OR collector case.

6.6.1 The Configurable Logic ROM Generator.

This program was written to produce efficient implementations of multiple-output 'ROM-like' functions in which most of the 2^n possible product terms were required. The ROM generator is built up of two sections: an $n-1$ bit decoder built using 2^{n-1} product terms and a special OR plane using several gate functions. The last input variable is introduced at the bottom of each OR-plane column and each gate in the OR-plane computes a function of this variable and one of the minterms. Since the AND plane is a simple decoder we know that *exactly* one of the product terms will be high for any input vector. The gate functions in the OR plane are selected according to the desired truth table (table 6–2), where f indicates the ROM function. The idea behind this table is that gates in the OR plane pass either the output of the previous gate in the cascade (if their product term is zero) or generate one of two possible ROM entries depending on the last input variable

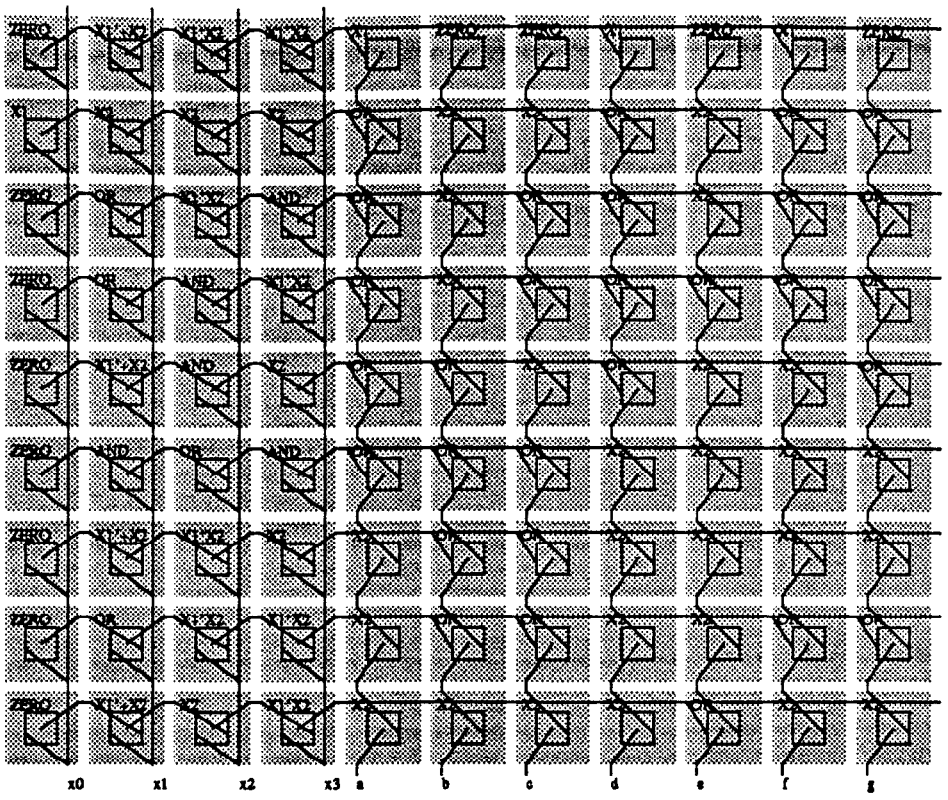


Figure 6–8: Seven Segment Decoder using Papakonstantinou's Algorithm.

(if their product term is one). The gate whose product term is high is guaranteed to have the input variable available since all the gates below it in the cascade will have low product terms and hence have passed it unchanged: similarly all the gates above it in the cascade will pass its output unchanged. Four functions are required ($X_1 + X_2$, $X_1 \oplus X_2$, $\overline{X_1}X_2$, X_2).

This method of implementing ROM's cuts the height (number of product terms) by 50% and the width by one column of cells. A further optimisation

Product (X_1)	Input (X_2)	Output (F)
0	0	0
0	1	1
1	0	$f(x_0, \ldots, x_{n-1}, 0)$
1	1	$f(x_0, \ldots, x_{n-1}, 1)$

Table 6–2: ROM Truth Table.

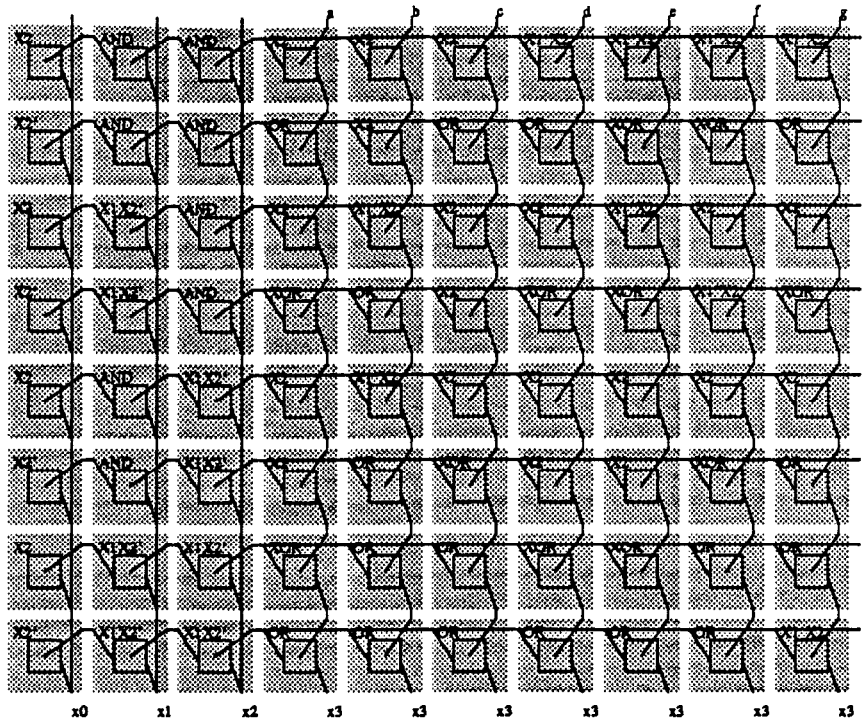


Figure 6–9: Seven Segment Decoder using Configurable Logic ROM.

would be to remove product terms from the $n-1$ bit decoder in the situation where the gates in the OR plane are identical for two product term rows p_1 and p_2 and the function $p_1 + p_2$ can be computed by a single cascade. This optimisation is not implemented by the present system since the expected number of such merges is small.

An implementation of the seven segment decoder example of the last section using this technique is shown in figure 6–9.

Pipelining the CL-ROM. Let us consider the effect of placing a row of pipeline registers within the ROM array across both AND and OR planes. Let n, m, p be the number of inputs,outputs and 'product' terms respectively, and r, c be the routing and computation delay times for one cell ($c \simeq 3r$). The maximum delay through the CL-ROM is then $(n-1)c + (m-1)r + pc$, the terms correspond to AND plane calculation time, routing to the last column of the OR plane and OR plane calculation. With pipeline registers in place the delay through each stage is

$(n-1)c+(m-1)r+p_sc+2c$, so only the last term is reduced. The delay through the ROM with s pipeline stages is now $s((n-1)c+(m-1)r)+pc+2sc$, the last term is caused by the delay through the pipleline registers. Normally there are many more product terms than inputs or outputs so pipelining can produce a very useful increase in throughput. In designs with a relatively large number of inputs, a column of pipeline registers between AND and OR planes can further increase throughput by allowing AND plane computation to proceed in parallel with OR plane computation.

6.7 Structural Layout.

There have been many placement and routing algorithms suggested for turning netlists of gates into silicon implementations. Provision of efficient structural layout algorithms is important in a developed cellular system to allow the use of multi-level logic synthesis programs which produce gate netlists - these are a vital component of the proposed 'active' compilation system.

There are two main steps in structural layout: placement and routing. In the placement step gates are clustered so that highly interconnected gates are close together and in the routing phase these gates are wired up. The simplest algorithm is called 'linear-clustering' and produces a long line of gates which is then sliced up into pieces of approximately the same length separated by wiring channels to produce a roughly square chip. This technique is not suitable for cellular designs since the individual 'gates' are much too small so nearly all the area would be taken up by wiring. It might be possible to use this technique in conjunction with larger 'macros' built up from several cells.

To make efficient use of the cell resources it is essential that the logical and wiring areas are mixed together. This is very similar to the situation in the 'sea-of-gates' silicon design style and it is very likely that placement and routing algorithms developed for these silicon systems could be adapted for configurable

logic. In fact, configurable logic's limited routing facilities simplify the algorithms even further and very good results are likely.

6.8 Routing.

6.8.1 Channel Routing.

The system provides a channel router to allow composition of cells according to a floorplan and global wiring plan. This code is implemented as an SML function which takes a block containing only port information (possibly read from a CFG file) and two integers representing the length and width of the desired channel and returns a block with the required routing and an indication of success or failure (the channel may not be routable in the specified number of tracks or cyclic constraints may force additional columns). Two algorithms have been considered for use in this system.

Recursive Decomposition. The router is based on a recursive decomposition of the subproblem along tracks. Each track generated is guaranteed to improve the situation resulting in an easier subproblem being handed to the next recursive call. The perceived advantage of this approach was that it was not necessary to specify the width of the channel in advance - the router automatically produced as many tracks as necessary. More conventional channel routing algorithms are given the number of tracks to use and fail if the routing cannot be completed.

Experience with this algorithm was not favourable since (because it worked on one track at a time and did not know how many tracks would eventually be used) it often put nets on the first tracks generated to make use of free horizontal routing resulting in much longer than necessary vertical routing. This was seen as an unavoidable side effect of the approach and so a more conventional routing algorithm which requires the width of the channel to be specified in advance was chosen for the final implementation.

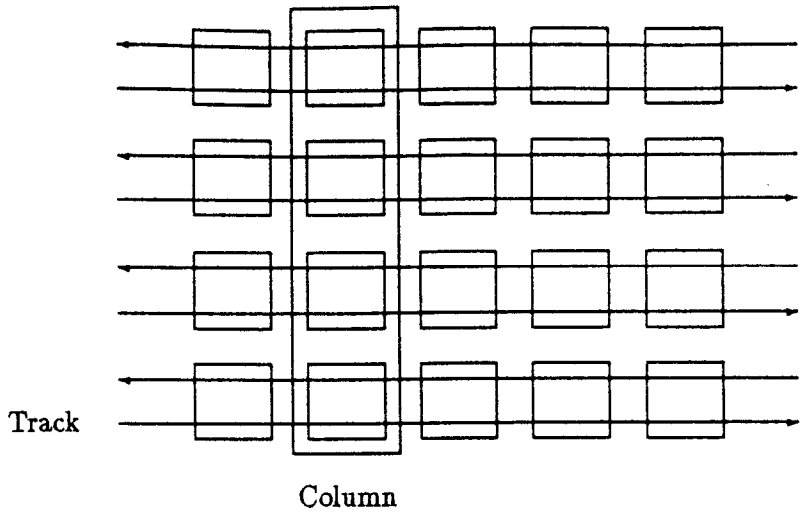


Figure 6–10: Wiring Channel Model.

Greedy Router. This algorithm is based on the Detour router from the Magic design system [Hamachi84], which in turn is a development of the system in [Rivest82]. It operates on a column by column rather than a track by track basis. At each column as much vertical wiring as possible is added to simplify the problem in succeeding columns. This is the algorithm used in the final implementation.

A diagram of the basic wiring channel is given as figure 6–10, note that the algorithm can be generalised to route 'switchboxes' with connections on all four sides. The channel is visualised as a rectangular grid of rows (called 'tracks') and columns. The 'greedy' algorithm operates by sweeping along the channel from left to right wiring up one column at a time: within each column it attempts to place as much vertical wiring as possible (hence 'greedy') to leave a simpler problem in later columns. Since there is no backtracking there is no guarantee that the best solution will be found or that a potentially wire-able channel will be solved: on the other hand channels are always wired within a reasonable time.

Figure 6–11 shows the basic structure we are operating on at each column. Given the nets which enter from the left, the top and bottom ports and a list of nets which must be extended to the right we must generate wiring in this column such that

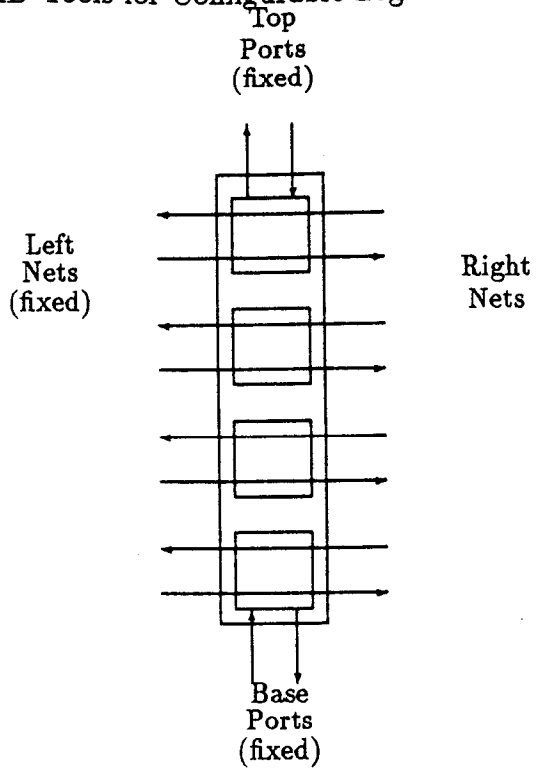


Figure 6–11: Wiring at Channel Column.

- The problem can still be solved.

- The remaining problem is reduced as much as possible.

The algorithm for creating the wiring in the current column can best be described as a series of rules. Note that these rules are identical to those in the original paper but their implementation is slightly complicated by the more complex routing available within each cell.

1. Bring New Nets into the Channel. First of all the router must bring any new nets into the channel. Nets are brought into the first available track, or when the net has ports to the left of the current column and there is no intervening vertical wiring to the track currently holding the net. If a net cannot be brought into the channel then the routing will fail so this step must be done first (since other steps could add vertical wiring which blocked these nets).

2. Merge Nets. Sometimes the same net will occupy more than one track in the channel. This can happen when the path from a track to a port in a previous column was blocked by essential wiring for other nets. The router attempts to 'merge' such 'split' nets by vertical wiring as soon as possible since they require additional area. When split nets cannot be merged they should be jogged closer together to minimise the amount of wiring needed to merge them in a succeeding column. Note that this step generates only vertical wiring and cannot cause routing failure.

3. Extend Nets to Right. In this step we extend all nets currently in the channel with ports to the right of the current column to the right (this gives the left port definition for the next call of the column router). Where possible 'rising' nets are jogged upwards and 'falling' nets downwards to be nearer to their next port. Heuristics can be used to prioritise the order in which nets are jogged (e.g. a net with ports on both sides of the channel may be better on a central track but a net with only one port to the right at the base of the next column should be jogged down as far as possible). It is important that these attempts to improve the problem for subsequent calls of the router do not stop any net from being extended across this column (since this would cause the routing to fail).

Figure 6–12 shows a small example channel generated by the channel router. We can see that nets B and E are both split at some columns within the channel but are quickly merged. Much larger wiring channel examples are given in the DES example of Chapter 7. Note that the current channel router is fairly primitive in terms of the rules applied within each column and produces significantly sub-optimal wiring channels, development of the router was stopped after the channels required for the DES example in Chapter 7 had been succesfully routed. Implementation of a high-quality channel router would be a major project (and should be tackled in an imperative language to achieve reasonable run times on large channels). The present system demonstrates that conventional channel routing algorithms can readily be adapted for cellular structures.

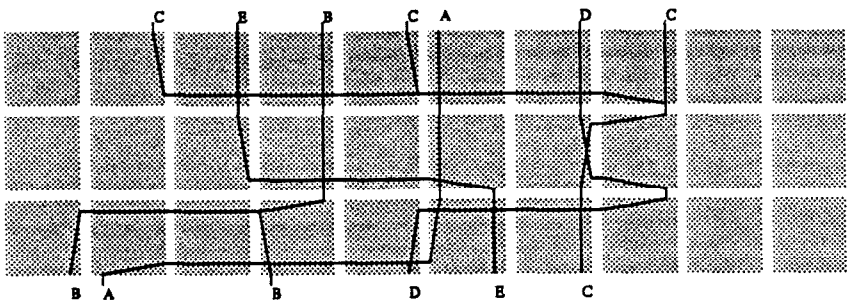


Figure 6–12: Example of Wiring Generated.

Pipelining Large Wiring Areas. In large wiring areas the delay through routing cells can be quite significant. One way to reduce the impact of this problem is to use pipelining within wiring channels: this is relatively easy to do since the function units of the cells within the channel are unused and the pipeline clock can come from low delay global signals. CAD software could be developed to implement the necessary pipelining automatically based on required throughput or return a failure message if this was impossible. Implementation of such software should not be difficult since it would simply involve counting cell delays for each path in a previously wired channel.

6.8.2 Maze Routing.

Whereas channel routing is used for separate wiring areas free from obstructions (although limited obstacle avoidance can be added) maze routing is required to implement wiring within blocks with existing connections. The basic component of this kind of router is an algorithm for finding the shortest path between two points in the presence of obstructions (e.g. Dijkstra's Algorithm or the Lee-Moore

algorithm), this computation is then performed for each net in turn. The problem is that wired nets will potentially obstruct nets which have not yet been wired, the complexity of maze-routing algorithms lies in strategies to avoid this contention by choosing a good order to wire the nets and recover from it (by ripping up nets wired previously) when it occurs.

Complex maze routers can be used for global wiring of large systems, such routers generally have 'backtracking' capability to allow rerouting of nets blocked by earlier connections, the concept of 'wavefront' routing based on the propagation and diffraction laws for physical waves when they hit obstacles is becoming popular [Xiong86]. The present system envisages a router working on single nets under direct user control: the user would determine the order in which nets from a net list were wired and could direct the system to rip up any given net. This router would be used as a component of the graphical editor described below.

6.9 Graphical Tools.

This section describes some graphical tools for manual layout of Configurable Logic systems. It should be read more as a specification of what is possible rather than a description of an actual implementation; although partial implementations of the functions described have been coded in SML [Harper86]. It was originally intended to produce fully functional versions of these tools to support the designs in Chapter 7 but it quickly became clear that the limitations of SML in terms of speed and low level control of the screen would limit their utility so much that it would be more convenient to do the small 'leaf-cell' designs by generating CFG files with a text editor - the large wiring areas and logic blocks were generated using the automatic tools.

6.9.1 The Leaf Cell Editor.

This is a simple graphical editor to allow the user to do hand crafted designs using the cells. This would be appropriate for commonly used functions such as

counters and adders where engineers may wish to use 'textbook' designs rather than less efficient automatic implementations. It also provides a tool for viewing larger designs to check their correctness.

Wiring would be done by clicking on the start and end points of the proposed wire, a maze router then makes the connection if one was possible. User's can control the wiring path by building long wires from several shorter ones thus constraining the maze router to follow the desired path. Cell functions are selected by clicking on the gate within the cell at which point a menu appears with the available functions on it. Special entries on this menu allow for the use of the FTEST and G1 and G2 global signals. The user can pan around the design using cursor keys.

Operations on hierachical blocks would also be supported. Editing operations would be provided to manipulate whole blocks (represented as rectangles) into the desired structure. This graphical representation could allow zooming unlike the fixed scale cellular representation (the representation of leaf cells must have a fixed scale because of the difficulty of scaling textual annotations like cell functions and port names).

6.10 Optimisation.

After a design has been produced using the tools outlined above there will often be considerable scope for an optimisation phase. Some of the most important areas where optimisation would be possible are outline below.

1. Gate Collapsing. Graph traversal techniques can be used to determine cases where gates are unused or several gates are being used to implement a function which could be implemented using a single gate. Such cases are very likely to occur in circuits composed of standard library parts and the availability of completely general two input gates makes collapses more probable. A gate collapsing phase which removed surplus gates would be easy to

implement and, although it would not save space in itself (since the placement of cells would not be changed) would reduce delays and provide more scope for following optimisation phases. Algorithms for this problem are well known and have been used for many years to improve gate level designs produced by simple logic synthesis programs which converted logic equations directly into gates [Shinsha84]. Similar techniques are used in gate array systems to remove redundant stages which arise when library parts are connected [Lattice86].

2. Use of Channel Area. In the naïve layout the function units of the cells in the wiring channel are unused. This phase would attempt to move cell functions into the wiring channel.

3. Local Perturbations. Given a cellular design one could attempt to move gate functions nearer to the 'bottom-left' of the array. This could be done by ripping up the wiring to a gate, moving it to a cell with an unused function unit and attempting to rewire the gate.

4. Compaction. In its simplest form this would consist of eliminating 'straight-through' rows and columns which may occur in wiring channels. More developed systems could use the constraint graph or 'zone-refining' approaches developed for silicon designs. Possibly at some time in the future a 'sticks' representation for cellular designs conveying only topological information and its associated fleshing program could be developed - this would be a significantly easier task than the corresponding silicon program where complex geometrical design rules must be followed.

6.11 Back End Tools.

Concurrently with the design of the prototype CLA stopwatch chip described in Chapter 7 some simple software tools for drawing cell connection diagrams, simulating designs and generating CIF were developed.

These tools were written in IMP [Robertson77] about a year before the front end code described above and run on locally designed APM workstations. They use a previous (slicing tree) version of the CFG file described above and have not been converted to read the new file format. This is because they were unnecessary for the remainder of the project and if they were to be ported to the SUN workstations it would make sense to rewrite them in C or SML: this effort was not considered worthwhile.

6.11.1 MODEL Interface.

At present simulation is provided by extracting a description of the circuit in Lattice Logic's MODEL language [Lattice86] which can then be simulated using the EXERT simulator. This approach was chosen because it was easy to implement and provided a low level simulation using well proven software thus giving high confidence that the stopwatch circuit was correct. This interface can also be used to extract designs verified using the CAL technology for implementation in silicon via the SOLO-1000 software suite.

6.11.2 CIF generation.

The tools to generate CIF from a CLA description are by far the most complex of the back end tools and will be discussed in detail. The CLA tool works within the ILAP system [Hughes82] in the same way as the PLA generator by producing a symbol with ports given a description in a file. It is accessed via the external routine call given below.

```
%external %routine %spec cla %alias "CELL_CLA"
                          (%string(31) symbol name,file name)
```

This allows CLA's to be integrated with other ILAP constructs such as PLA's, counters, custom leaf cells and, most importantly, pads. The present tools must generate geometry on all layers, not just metal2. It would be easy to modify them to produce only the metal 2 geometry to personalise standard sizes of configurable logic arrays.

The problem of deciding which routing wires to use within a cell given a description of the connections the cell is required to implement is not trivial and is approached as follows. A file is prepared manually giving a list of possible ways of doing each connection e.g. North Input to South Output may have five possible paths through the cell (the most obvious of which uses tile numbers R103,R104,R1,R90,V1,R28,R127,R128,R129,R130 in figure 5–9). A program then uses an exhaustive search over these paths to try to route each possible permutation. The present CLA cell has 4096 legal routing permutations (note that as discussed in Chapter 5 the CLA cells are slightly different from the current CAL cells in that there are no G1, G2 or FTEST signals).

An additional requirement is that the number of CIF statements output should be minimised and the number of symbols at each level of hierarchy should be kept relatively small. To do this symbols are generated for each of the possible intra-cell paths (e.g. North Input to South Output) used in an actual chip. There are usually very many fewer used paths than possible paths. These symbols instance the basic routing tiles shown in Figure 5–9. The program then generates a symbol for each distinct cell routing permutation within the design which instances the symbols for single intra-cell routes, again it would be surprising if more than a fraction of the possible 4096 permutations were used in a given design. Symbols are also generated for all legal cell functions which instance the basic function tiles shown in Figure 5–8.

These routing symbols are then superimposed as appropriate with one of the cell function symbols and a symbol with the unpersonalised basic cell to produce the required cell design. These additional levels of hierarchy are important in practice since they greatly reduce the size of the CIF files produced - by a factor of three in the case of the stopwatch example in the Chapter 7.

6.11.3 CAL Programming.

This program reads in a CFG file and generates programming information for CAL's. Unlike the other back end tools it is written in C and runs on a SUN workstation using the new CFG format. The program is complicated by the need to take account of inversions caused by routing multiplexors and modify cell functions accordingly. It must also have detailed knowledge of the layout of the CAL in order to find the particular RAM cell controlling a multiplexor in the cellular design. This involves compensating for the different logical and physical connection schemes used to produce a square array from rectangular cells (Chapter 4). The program outputs its results in the order required for the serial programming interface. As well as producing a 'binary' file for downloading into CAL's (at present this is a script for the RNL switch level simulator since no CAL's have been built) it produces a map showing the binary values within each logical cell in the CAL array. This is useful for debugging purposes.

6.12 Summary.

This chapter has discussed the design automation aspects of the cellular logic architecture and presented some simple design automation tools. These tools have been used successfully in the design of several major example applications (Chapter 7). They illustrate that the techniques used in VLSI CAD can successfully be applied to cellular systems. They do not constitute a fully engineered CAD system and were never intended to do so.

Given the large effort involved both in CAD system design and in learning a new CAD system as a user, it makes far more sense to integrate support for configurable logic into an existing design system than to create a totally new system to support it. This is especially true where configurable logic is to be used to prototype ASIC's.

Chapter 7

Design Examples.

This chapter presents four examples of designs done using the configurable logic system: the first (a digital stopwatch) is typical of the sort of system that would fit on a single CAL chip in an EPLD application and allows an area comparison of configurable logic with other design systems. The second example, a Data Encryption Standard (DES) encryptor/decryptor is much larger and serves as an example of the kind of application which could be tackled by a board full of configurable logic chips. The third example illustrates the use of configurable logic to accelerate 'inner-loop' calculations in an image processing problem. The final example shows how configurable logic can be used as a component of a very high performance system for fluid-flow simulation using a cellular-automata algorithm.

7.1 Stopwatch Example.

This section covers the design of a simple digital stopwatch designed to count up in tenths of a second to one minute and display the current time on three seven segment displays (tenths of seconds, seconds and tens of seconds). The watch is controlled by three signals:

INIT Clears the stopwatch to zero and puts it in the 'stop' state.

SS Start/Stop A high going edge in the 'stop' state starts the watch counting. A high going edge in the 'start' counting state puts the watch in the 'stop' state.

CLOCK 10Hz clock signal.

This design example was originally set as a practical exercise to MSc students to be implemented using PLA's. The seven segment decoders provide a good example of 'random' combinational logic while the counters provide a good example of classical sequential logic. This design uses three separate units for the three displays although the design complexity and area could be reduced by implementing it as a single state machine with multiplexed displays (since the large seven segment decoder logic would not need to be duplicated). This is necessary to comply with the design specification given to the students in order to make a fair comparison between design styles.

7.1.1 The Counter.

The counter is built from 4 toggle flip-flops (Figure 7–1). The basic 4 bit ripple counter is converted to a decade counter by a gate which sets the clear line when the counter gets to ten. The output of this gate is also used as the clock for the succeeding counter. The counter can also be cleared by the user's *INIT* signal.

Providing the clear capability was the only real problem in the design of the toggle flip-flop (the basic D latches have only clock and D inputs). The clear is provided by extra logic gates which force 0's onto the D inputs and 1's onto the clock inputs. A basic master-slave flip-flop can be built with only two D latch cells whereas this clearable implementation requires six cells.

The toggle flip-flop is a good example of the sort of component which must be designed manually rather than automatically. It was necessary to spend quite a lot of time to find a good layout for this function. As the system develops it would make sense to build up a library of efficient designs for such commonly used functions.

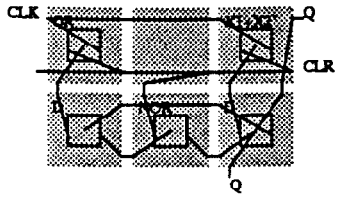


Figure 7–1: Counter Design.

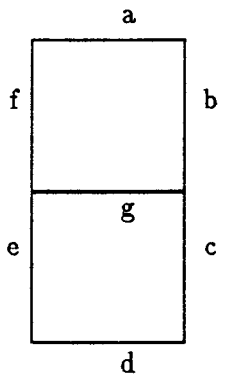


Figure 7–2: Seven Segment Display.

7.1.2 The Decoder.

The decoder (figure 7–3) takes advantage of the ability to produce any function of two boolean variables within one cell and uses several levels of logic rather than the two level logic normally used to implement such functions.

The decoder logic was designed by hand by Genbao Feng and took the greater part of the one-man week spent on the whole design. It is interesting to compare the quality of the logic synthesis with that of an automatic implementation generated by the newer design tools. The truth table for the function is given as table 7–1 (using the segment labelling of figure 7–2) and the minimised version output by ESPRESSO as table 7–2. This design can be compared with the automatically generated versions in Chapter 6. The Configurable Logic ROM implementation requires an 11 (4 inputs plus 7 outputs) by 8 (product terms) array

Input	a	b	c	d	e	f	g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	1	0	1	1
1010	X	X	X	X	X	X	X
1011	X	X	X	X	X	X	X
1100	X	X	X	X	X	X	X
1101	X	X	X	X	X	X	X
1110	X	X	X	X	X	X	X
1111	X	X	X	X	X	X	X

Table 7–1: Segment Truth Table.

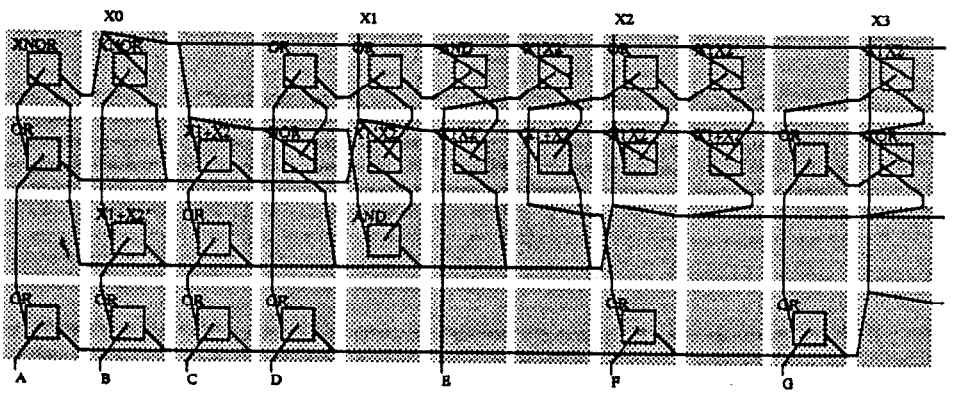


Figure 7–3: Seven Segment Decoder Design.

Input	a	b	c	d	e	f	g
X0X0	0	0	0	0	1	0	0
X00X	0	1	1	0	0	0	0
X000	1	0	0	1	0	1	0
XX11	1	1	1	0	0	0	0
X100	0	1	1	0	0	1	1
X101	1	0	1	1	0	1	1
X01X	1	1	0	1	0	0	1
X110	1	0	1	1	1	1	1
1XXX	1	0	0	1	0	1	1

Table 7–2: Minimised Segment Truth Table.

of cells compared with an 11 by 4 array when implemented by hand. The following points are worth noting:

1. Flexibility. The greatest advantage of the human generated implementation is the way the layout of the logic unit has been optimised to fit in with the chip pinout requirements and the counter design. Although there is a structure to the layout (note that input variables are routed horizontally across the array and outputs are collected using vertical wires) it is specific to this particular example.

2. Variable Ordering. In the automatically generated array the order of the input variables in the cascade of gates is fixed. Only the function performed by gates in the cascade can be changed. The human implementation gets a much better factorization of the logic equations by changing the order of variables in the gate cascades and using trees rather than simple cascades of gates. This can only be achieved by having a special routing plan for each function implemented.

3. Design Time. Although the automatic implementation is much less efficient it was done in about 10 minutes. The human generated implementation took several days.

4. Design Size. This example is at the limit of what can be efficiently hand designed by a person: larger functions would probably be designed just as inefficiently since the human would need to take steps to reduce the design complexity to a manageable level by reducing the design space.

7.1.3 The Control Logic.

The controller function (at the bottom left of figure 7–5) is implemented using a toggle flip-flop. The design is the same as those within the counters. The toggle flip-flop is clocked by the start-stop input and its output determines whether the counter should be stopped or run freely. The initialise signal clears the control flip-flop and stops the counter. The counters are stopped by AND'ing the 10Hz clock with the output of the control flip-flop: thus when the output of the control flip-flop is 0 the counters' clock input is held low and when it is 1 the counters receive the 10Hz clock input.

This is one area in which the configurable logic implementation is much more efficient than the PLA designs. These require a three phase 1MHz clock and use a whole PLA just to implement the start/stop function.

7.1.4 Floorplan.

The floorplan of the full watch circuit is given below as figure 7–4 and the layout in figure 7–5. It requires a 20 by 13 array of cells.

7.1.5 CLA Implementation.

Only that part of the design enclosed by the box was fabricated. The fabricated design with one decade counter, the control circuitry and the seven segment de-

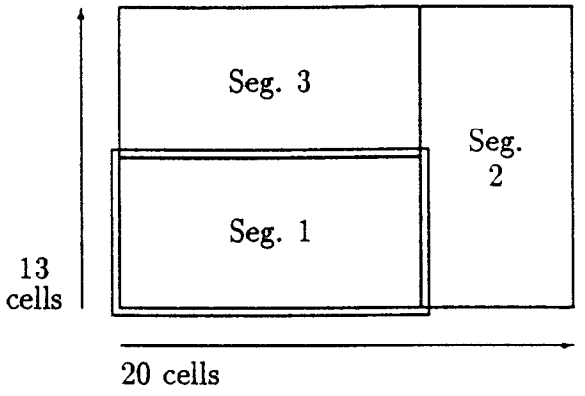


Figure 7–4: Stopwatch Floorplan.

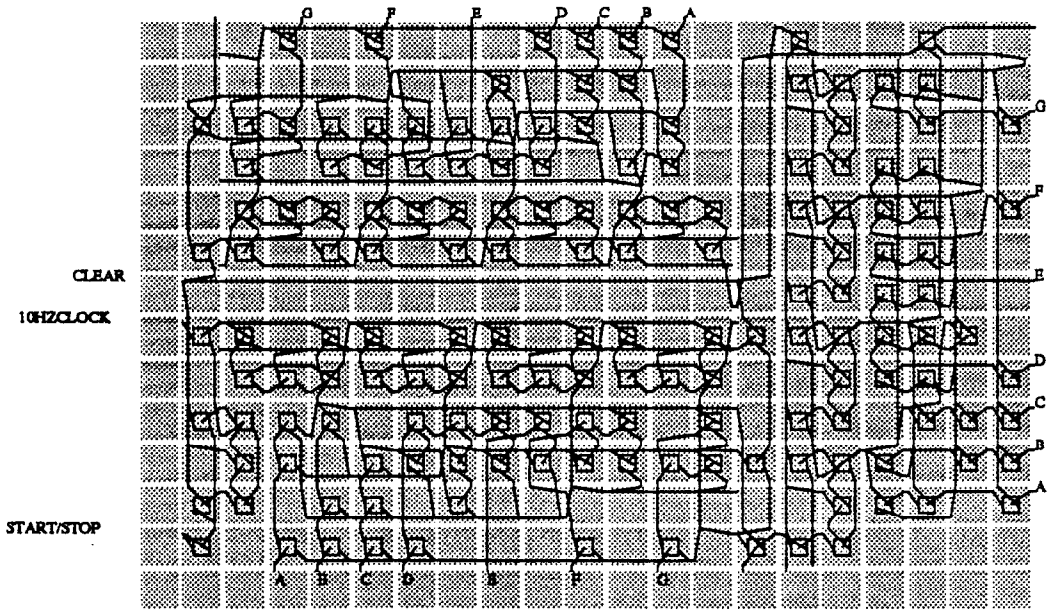


Figure 7–5: Full Stopwatch Design.

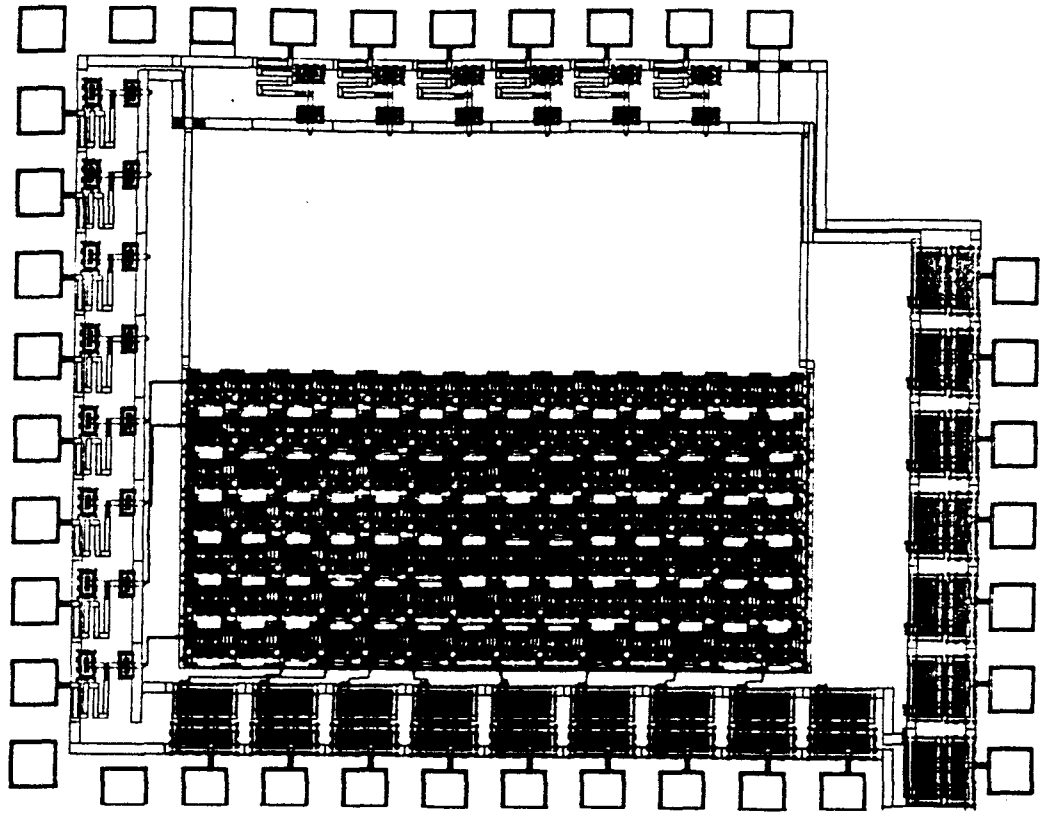


Figure 7–6: Plot of Stopwatch Chip.

coder required a 14 by 7 array of cells giving a core (i.e. before pads and pad routing) symbol size of 2016 by 932 μm.

Note that the design software could make a squarer chip (which would normally allow larger designs) by orientating the cells with their shorter ($132\mu m$) side along the longer (14 cell) edge of the target design. In this case the die size was fixed so there was no practical advantage in doing this.

The CIF plot of the final chip (figure 7–6) shows clearly the reason for only implementing one counter rather than the full watch: the large notch cut out of the top right corner of the standard die for test structures and the fixed die size meant that our design could not take full advantage of the silicon area. The pads are placed for use by the PLA based watch chip so many of them are not required by this design with its reduced functionality.

Test Results. The fabricated chip was tested and was found to be completely functional. A complete stopwatch was built using three of the prototype chips

and left running for several days, no problems were discovered. No attempt was made to measure the maximum speed of the devices: it is likely that the pads would provide the speed limitation rather than the logical elements. Of the 20 chips fabricated 18 functioned perfectly, one had a segment on continuously but was otherwise functional and one failed completely.

7.1.6 Size Comparisons.

The digital stopwatch example has been designed using several different styles: CAL, CLA (single mask change), PLA and multiple mask change 'optimised' (because wiring channels and logic areas are only as large as they need to be - the detailed chip floorplan is specific to a particular design) gate array. The optimised array designs were done by European Silicon Structure's SOLO-1000 system (previously Lattice Logic's CHIPSMITH silicon compiler) [Lattice86] using an input specification (in the MODEL language) generated automatically from the cell based design. This compiler has had many years of development and produces very high quality layouts for this kind of design: in this case it can also take advantage of the efficient gate level design done for the configurable logic implementation. The figures for this design are, therefore, better than average for this implementation style. The PLA designs were done by students as a practical exercise and have been fabricated using lambda rules with $\lambda = 2\mu m$ (i.e. $4\mu m$ technology). They are composed of four PLA's one for each of the segment units and one for the controller. The CLA example was also done to these rules and a section of it containing one segment unit and the control logic was built and tested. The CAL and gate array designs were done using the same commercial $2\mu m$ rules. The areas given are core symbol sizes (no pads), the CAL figures are for a 20 by 13 (logical) chip. The results are given in Table 7–3, in this case there is a factor of about 15 in area between the CAL implementation and the optimised array: special processing in the CAL design could significantly reduce this gap. Although a factor of 15 seems high it suggests that any gate-level design which will fit on a single ASIC could be prototyped using a board full of configurable

Design.	*Technology*	*Dimensions*
CAL	$2\mu m$	$5777 \times 3786\mu m$
CLA	$4\mu m$	$2858 \times 1724\mu m$
PLA	$4\mu m$	$1432 \times 1625\mu m$
Gate Array	$2\mu m$	$1544 \times 952\mu m$

Table 7–3: Design Size Comparison.

chips. Or, to put it another way, CAL has to be reprogrammed about 15 times to regain parity with normal logic.

7.2 Data Encryption Standard Example.

In 1977 the United States National Bureau of Standards (NBS) promulgated a new standard for encryption of unclassified data [NBS77] based on a proposal by IBM. Since then this algorithm has become the de-facto standard for data encryption worldwide. The algorithm is very suitable for implementation in hardware and several commercial products are available. In this section we will first outline the DES algorithm and secondly develop an implementation. DES provides an excellent example of an algorithm which can be speeded up greatly by bit-level hardware implementation and is also a good example of the sort of ASIC design which could be prototyped using CAL's.

7.2.1 Introduction to DES.

DES is a substitution cipher on 64 bit binary vectors based on a 56 bit key. The strength of DES lies in the complexity of the substitution. Two good introductions to DES are [Tanenbaum81] and for a more in-depth analysis by an IBM 'insider' [Konheim81]. Since it was first promulgated the DES standard has been the

subject of a great deal of controversy. Two main attacks have been made: firstly that the 56 bit key is not long enough (the original IBM proposal suggested 128 bit keys) and secondly that some sort of trap-door exists in the algorithm allowing rapid decipherment without knowledge of the key by those who know of it.

The first claim has some validity if the DES algorithm is used naïvely as a 'black-box': a hardware implementation of the algorithm using several hundred custom chips could do a key search in a reasonable time (this was first shown by Hellman [Hellman80]). It is certainly true if the keys are concatenations of 8, 7 bit, ASCII characters representing an English word rather than random 56 bit integers: in this case a key search taking advantage of letter frequency information is almost trivial with appropriate hardware. For this reason, in 1980 a follow up document was issued by the NBS detailing additional modes of operation of the cipher: these break the one to one relation between plaintext and ciphertext by making each ciphertext block depend on previous blocks as well as the current one [NBS80]. The cipher can also be strengthened by using multiple encipherments with different keys and generating good 56 bit keys from a longer string of english text using DES as a hash function [Konheim81]. These modifications are easy to implement and make key search totally infeasible. In terms of hardware implementations limiting the key length to 56 bits makes it much easier to place a whole encryptor on one chip.

The second claim has never been established despite a lot of work on analysing the cipher: the main reason for believing it is the refusal of the U.S. government to let IBM disclose all the criteria for the design of the cipher. No attempt will be made to analyse the cryptographic properties of DES here, a detailed analysis is presented in [Konheim81].

7.2.2 The DES process.

The DES algorithm is illustrated in figure 7–7. All the f-boxes are identical and it is obvious that a major design decision which must be taken is whether to have one reusable f-box or 16 separate ones to allow pipelining. We will consider

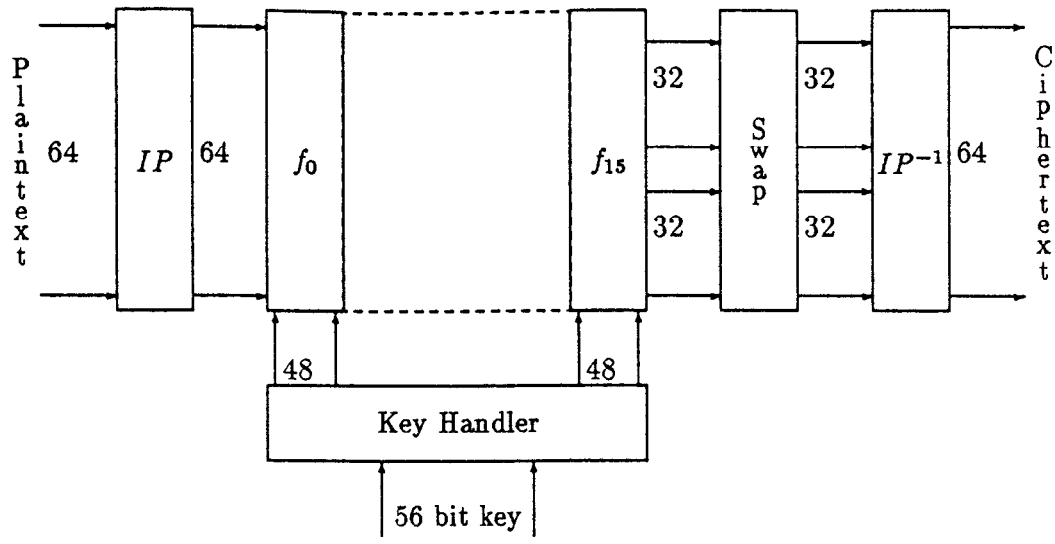


Figure 7–7: DES Block Diagram.

the major components in the block diagram individually. We will try to give an idea of the size of the wiring channels (unfortunately channel density cannot be quoted because it depends on the ordering and spacing of the ports which is implementation dependent) and we will give the number of product terms in a minimised PLA implementation (determined using ESPRESSO) for the logic blocks.

Permutations IP and IP^{-1}. The first step in the DES algorithm is to apply an initial permutation IP to the plaintext, the final step is to apply the reverse permutation IP^{-1}. These permutations involve simple wire swapping and could be implemented using the channel router however the resulting channel would be wide. The permutations are highly structured and more cost effective implementations than straightforward wire swapping can be found.

Design of the f-box. The heart of the DES process is the f-box which performs the function $\pi : (L, R) \rightarrow (R, L \oplus T(R))$, where L and R represent the left and right 32 bits of the 64 bit input. The transformation T is composed of an expansion (bit copying) to 48 bits modulo 2 addition with a key, substitution (in which each group of 6 bits is taken as representing a binary number which is then substituted

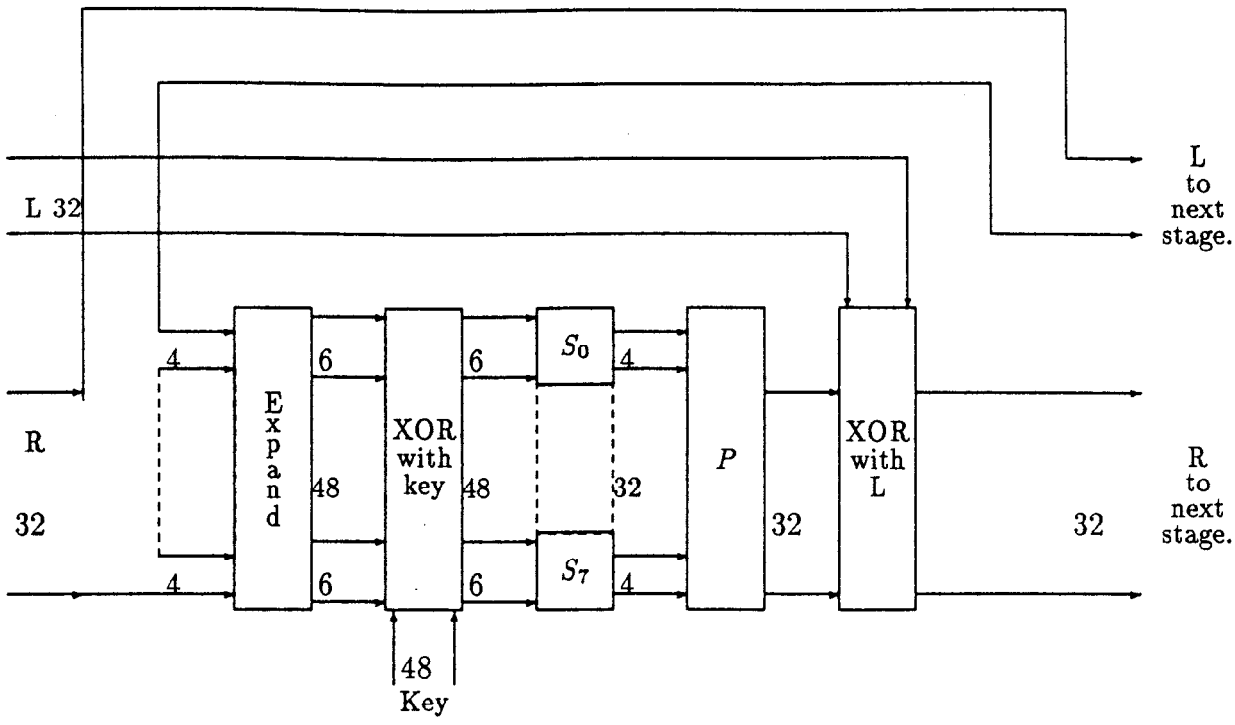


Figure 7–8: DES f-box Block Diagram.

for another 4 bit binary number using a lookup table, bringing the size back to 32 bits) and a 32 bit permutation (wire swapping).

The major components in the DES f-box are illustrated in figure 7–8. The important point to note is that the width of any hardware implementation is fixed by the number of bits to be operated on. We must concentrate on reducing the length of the unit. We will now consider the components of the f-box in order left to right.

1. Expansion. This step involves expanding groups of four bits in the data to six bits by copying the outer two bits as shown in Table 7–4. This step fixes the height of the implementation as at least 48 cells.

2. XOR with key. This is the critical operation in the whole cipher since this is the only point where the output ciphertext is modified by the key. A different 48 bit key derived from the user supplied 56 bit key is used in each f-box.

Before.	*After*
x_0, x_1, x_2, x_3	$x_{31}, x_0, x_1, x_2, x_3, x_4$
x_4, x_5, x_6, x_7	$x_3, x_4, x_5, x_6, x_7, x_8$
x_8, x_9, x_{10}, x_{11}	$x_7, x_8, x_9, x_{10}, x_{11}, x_{12}$
$x_{12}, x_{13}, x_{14}, x_{15}$	$x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}$
$x_{16}, x_{17}, x_{18}, x_{19}$	$x_{15}, x_{16}, x_{17}, x_{18}, x_{19}, x_{20}$
$x_{20}, x_{21}, x_{22}, x_{23}$	$x_{19}, x_{20}, x_{21}, x_{22}, x_{23}, x_{24}$
$x_{24}, x_{25}, x_{26}, x_{27}$	$x_{23}, x_{24}, x_{25}, x_{26}, x_{27}, x_{28}$
$x_{28}, x_{29}, x_{30}, x_{31}$	$x_{27}, x_{28}, x_{29}, x_{30}, x_{31}, x_0$

Table 7–4: DES Expansion.

3. S-Box. The S-box or substitution box has 4 outputs and takes 6 inputs, 4 'data' and 2 'control'. The control inputs select one of four 1 to 1 substitution functions over 4 bit binary integers. There are 8 separate S-boxes each of which uses different functions. These units are most naturally implemented as blocks of combinational logic. Running the functions through ESPRESSO produced very little minimisation - all 8 S-boxes required more than 50 product terms. This is presumably because the cryptographic properties of DES require an irregular function whereas logic minimisation relies on finding regularities.

4. P-Box. The P-Box or permutation box does a simple wire swapping operation on its inputs. This can be implemented using a channel router in a reasonable area.

5. XOR and Swap. The swap is necessary because without it only the right half of the plaintext would interact with the key. This would imply that 32 bits of the ciphertext would be identical to 32 bits in the plaintext (which 32 bits in ciphertext and plaintext would depend on the initial and final permutations). The XOR is also crucial to the complexity of the cipher

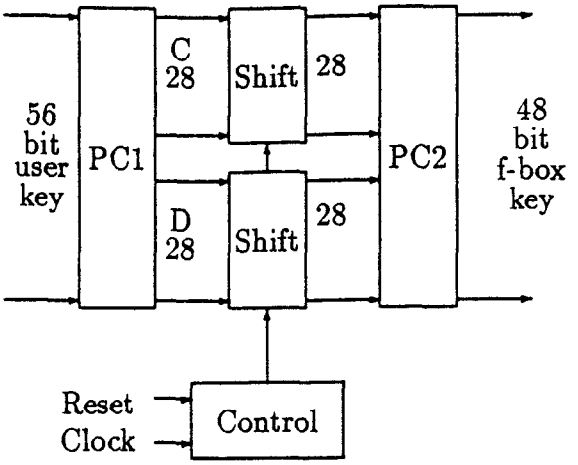


Figure 7–9: DES Key Handler.

because without it there would be two groups of 32 bits in the ciphertext each of which depended only on a group of 32 bits in the plaintext. Instead of being a 64 bit, 16 stage cipher it would be two separate 32 bit, 8 stage ciphers and hence much easier to crack.

Final Swap. This has the effect of cancelling out the swap in the final f-box. This is done to keep the cipher symmetrical so that decryption can be done by re-encrypting with the same key - thus only one DES unit is required (not two, one for encryption and one for decryption).

Design of the Key Handler. This logic controls the key which is applied to the f-box at each stage of the DES operation. This is determined by the key-schedule. The design of the key-handling unit is shown in figure 7–9. We will describe the major components in turn.

1. PC1. The PC1 unit applies an initial permutation to the bits of the key. This unit is 56 bits wide. It can be implemented using a channel router but the channel is fairly wide, better implementations based on the structure of the permutation are possible.

2. Shifters. After the initial permutation the key is broken up into to two 28 bit segments C and D which do not come into contact with each other again. At each stage in the cipher the C and D keys are shifted circularly a number of times determined by the controller according to the key schedule.

3. PC2. The PC2 unit takes the C and D input keys and does a selection and permutation operation on them selecting 48 bits from the 56 bit key. The top 24 bits of the result all come from C and the bottom 24 bits all come from D so it is possible to view PC2 as being two separate 28 to 24 bit units. The resulting 48 bit number is used as the key in the f-box. Again this unit can be implemented using a channel router, in this case the channel is reasonably small.

4. Controller. The controller generates either one or two clock pulses for the shift registers at each stage of the cipher according to whether one or two key shifts are specified in the key schedule.

7.2.3 Implementation of DES.

Floorplan. When we consider the amount of logic required to implement an f-box it becomes obvious that we cannot hope to have 16 of them in a reasonable sized array, instead we must use the same box for every stage of the cipher. We wish to avoid bringing the key in from the bottom because at least 48 cells will be required to input a 48 bit vector: if it is brought in from the side then no extra cost is incurred because the unit must be at least 48 cells high anyway. We also note that IP and IP^{-1} can be implemented in the same wiring area. This suggests the floorplan shown in figure 7–10.

Layout of f-box. When we examine figure 7–8 it appears that large wiring areas are going to be required for the L and R 32 bit words. It is also apparent that all the wires are going left to right: this would imply a large waste of cell resources since the basic cell can route right to left as well. After a little thought it becomes

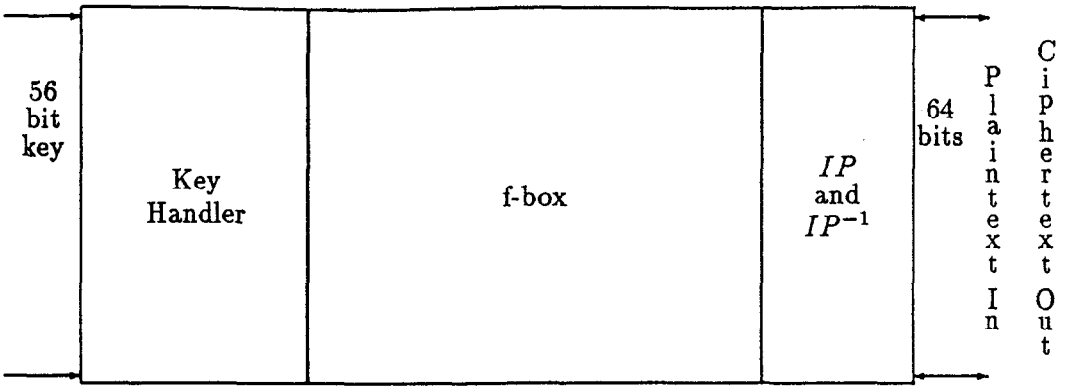


Figure 7–10: DES Floorplan.

clear that a much better layout of the f-box is possible (figure 7–11). By using registers at the right hand side we have avoided wasting large areas transporting L and R through the unit. By interleaving L and R we have made it extremely easy to XOR adjacent bits and swap over prior to the next stage. We are making use of the cell's ability to route R right to left to bring it to the left hand side for the initial expansion and XOR with the key. We will now consider the design of the major f-box components in turn.

1. S-Box. We start with the S-box because it is the most complex component and will constrain the rest of the design. In order to make use of the logic synthesiser described in the last chapter we have to adopt an AND plane/OR plane type layout for this unit with 6 input signals and 4 output signals. This means that the design will be 9 or 10 cells wide depending on which logic synthesis method is used. We can see from the floorplan that as well as performing the logic function the S box unit must route four bits of the R vector right to left: this is easily done using the right to left connections in the OR-plane. All the S-boxes will be expanded to be the same width as the one with the largest number of product terms. A diagram of one S-box unit is given as figure 7–12.

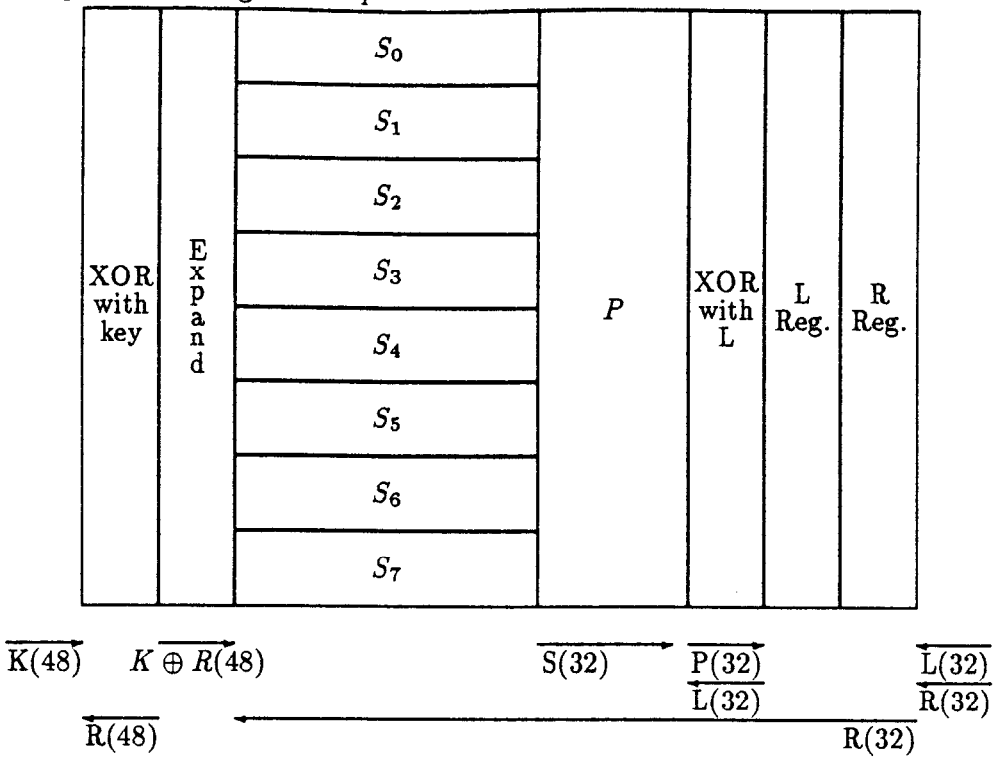


Figure 7–11: f-box Floorplan.

Papakonstantinou's algorithm could not find any merges in the S-box designs: this is not surprising because there were very few cases in which several product terms always appeared together. The S-boxes were, therefore, implemented using the CL-ROM generator and required 32 product terms. Although the logic block is only 9 cells wide we choose to use an extra line of cells between each S-box unit making its effective width 10 cells and fixing the height of the f-box at 80 cells. There are two advantages to this: firstly by increasing the height of the wiring channels we simplify the routing problem allowing a solution using fewer tracks, secondly the extra width simplifies the design of the other f-box components.

The S box outputs are generated in the order in which they appear on the opposite side of the P-box wiring channel rather than the 'numerical' order of the standard. This reduces the number of tracks required to implement the P box. Note that the right to left connection of R shown on the diagram is accomplished by overlaying a routing symbol over the automatically

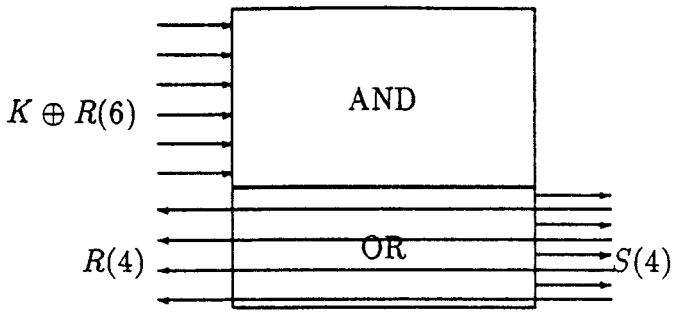


Figure 7–12: S-box Design.

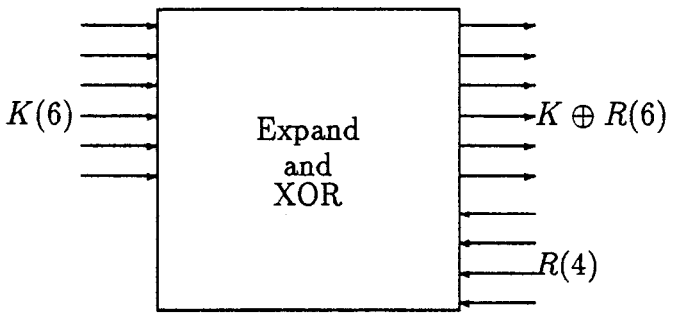


Figure 7–13: Expander Design.

generated S-box design: this is possible because the right-to left connections are not used by the CL-ROM.

2. Expansion and Key XOR. It turns out to be easier at the cellular level to implement both these functions within the same subunit. A block diagram of the design is given in 7–13. Note that there are in fact three slightly different designs for the first, last and intermediate expansion units because of the (x_0,x_{31}) wraparound connections (table 7–13). The 80 cell path for this wraparound connection is the limiting factor for pipelining this design. All but the first two columns of cells in the expansion unit are overlayed on the S-box so its effective width is only two cells: note the extra routing to bring the last bit of $K \oplus R$ to all the OR plane columns of the CL-ROM.

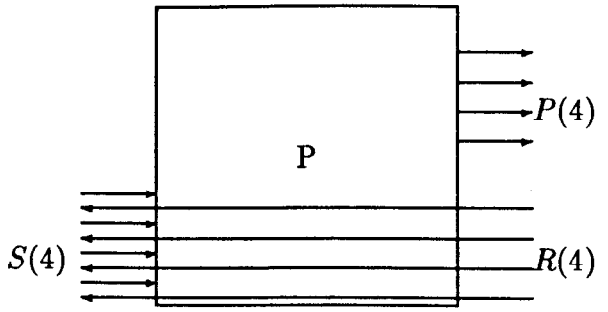


Figure 7–14: P-box Design.

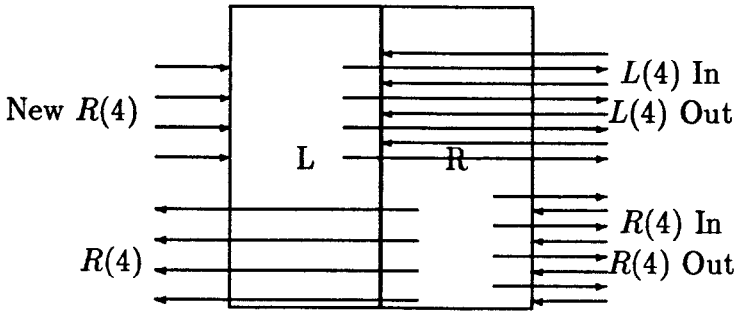


Figure 7–15: L and R Register Design.

3. P-Box. The P-Box is a large wiring channel which runs the whole length of the f-box. As well as implementing the wire swapping it also realigns the ports ready for the next stage (figure 7–14). The R vector can travel across the top of the wiring channel very easily since no right to left connections will be used: there is no need to inform the channel router about these signals.

4. L/R Registers XOR and Swap. The L and R registers are implemented as master/slave flip-flops to allow to new value to be loaded while keeping the previous value stable. The registers are broken down into 4 bit units which occupy 10 cells (pitch matched to an S-box). Within these units the bits are interleaved: swapping is implemented simply by a transfer between adjacent registers. A block diagram is given in figure 7–15

The layout of the whole unit is given in figure 7–16, note how the individual components fit together and the signal flow allows a very high utilisation of cell resources.

Layout of Key Handler. The layout of the key handler can follow the block diagram exactly. We will consider the implementation of the major subcomponents in turn.

1. PC-1. This routing converts 8, 8 bit bytes into a 56 bit number by throwing away the most significant bit of each byte and doing some simple scrambling of the remaining bits. It is questionable whether it is worth the overhead of hardware implementation since it will only be used when the key is changed, this is a relatively rare event. Fortunately, the permutation is structured in such a way that it can be implemented using shift registers rather than a large wiring channel [Hoornaert84]. The shift register technique is explained in more detail for the IP channel below: its application to this channel is slightly less straightforward.

2. Shifters. These are implemented as clocked master slave registers with a feedback path. The shifters are pitch matched to the controller in width and to the wiring channels in height. The number of clock pulses determines the number of shifts. The layout is given in figure 7–17.

3. Controller. This unit can be implemented simply as a finite state machine using the logic synthesiser. It must generate either one or two clock pulses for the shifters on each f-box cycle: to do this it is clocked twice at the start of each f-box cycle and cycles round a 32 state sequence. The reset signal clears the state machine ready for the first cycle and is routed to the shift registers to load the next key through PC-1. When reset is low the registers are configured as two 28 bit circular shifters. In this case Papakonstantinou's algorithm provides a slightly better solution that the ROM generator. A small amount of manual routing is necessary for the feedback terms: the generator could easily be extended to do this automatically. Note that the

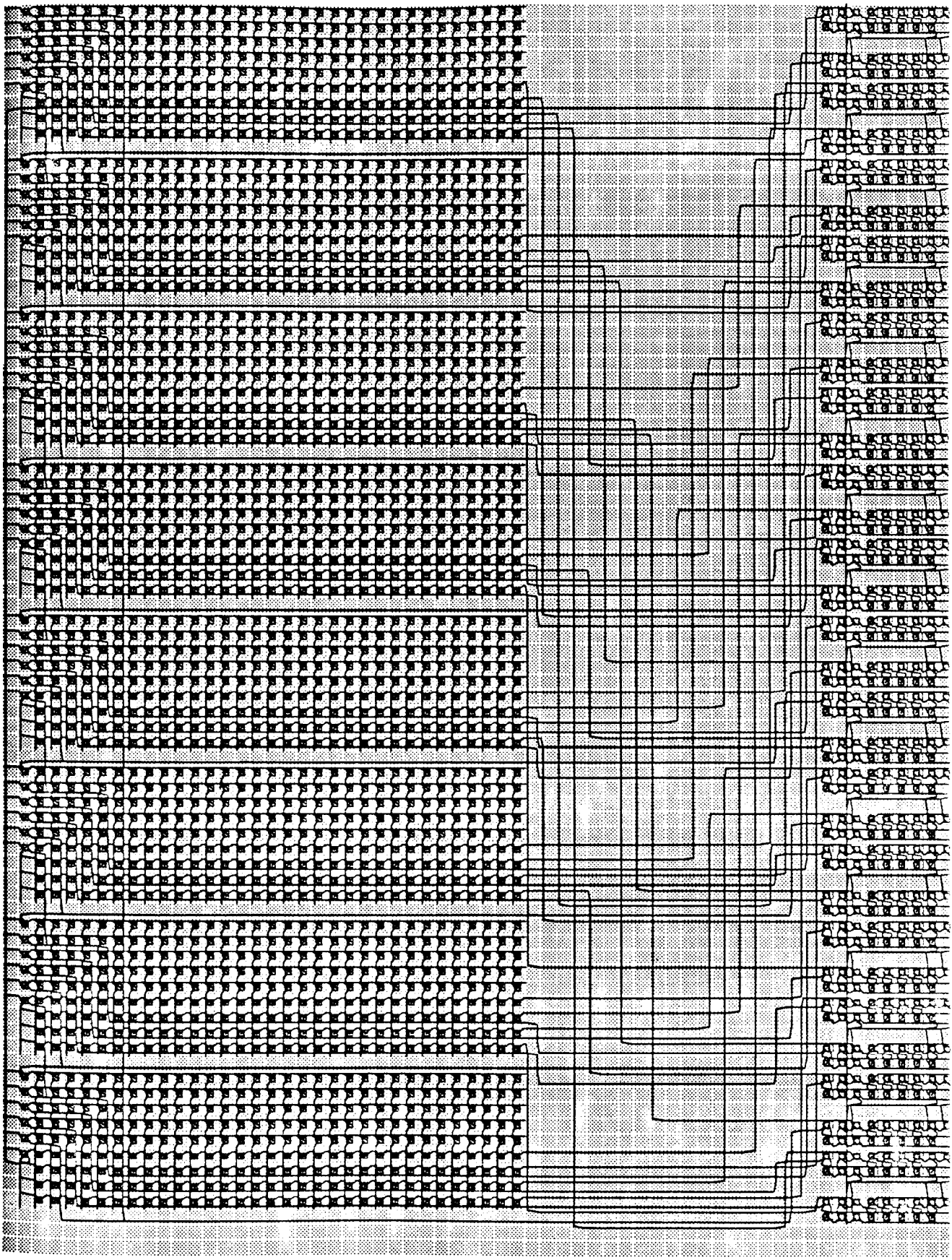


Figure 7–16: f-box layout.

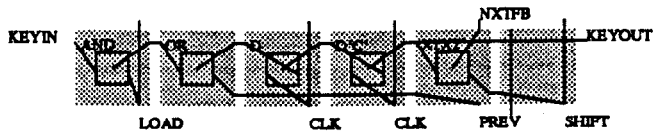


Figure 7–17: Shifter Layout.

layout (figure 7–18) could be optimised by using unused multiplexors within the logic array to provide the feedback connections.

4. PC-2. This unit is implemented using the channel router: it performs a permutation and selection operation on the shifted 56 bit key to generate a 48 bit key for the f-box. Since only 48 bits are used some of the ports on the left of this wiring channel are unconnected.

The layout of the whole unit is given in figure 7–19.

IP and IP^{-1}. A straightforward implementation of this large wiring area would require at least 37 cells. This large area requirement has to do with both the permutation itself and the fact that we are implementing both IP and IP^{-1} in the same channel. This means that there are 64 input and 64 output ports on both sides of a 80 cell channel heavily constraining the channel routing (there are very few 'free' columns which the router could use to reorder nets). This channel is a problem for silicon implementations of DES as well and we can borrow a technique from [Hoornaert84],[Hoornaert88] to implement it in a reasonable width. This technique relies on regularities in the IP permutation and replaces the large channel with a small one and some shift registers: it has the side effect of reducing the data rate but this is not a problem since the speed of the implementation is constrained by the f-box circuits which are used 16 times for every input through IP. The design is shown in figure 7–20 and the layout in figure 7–21.

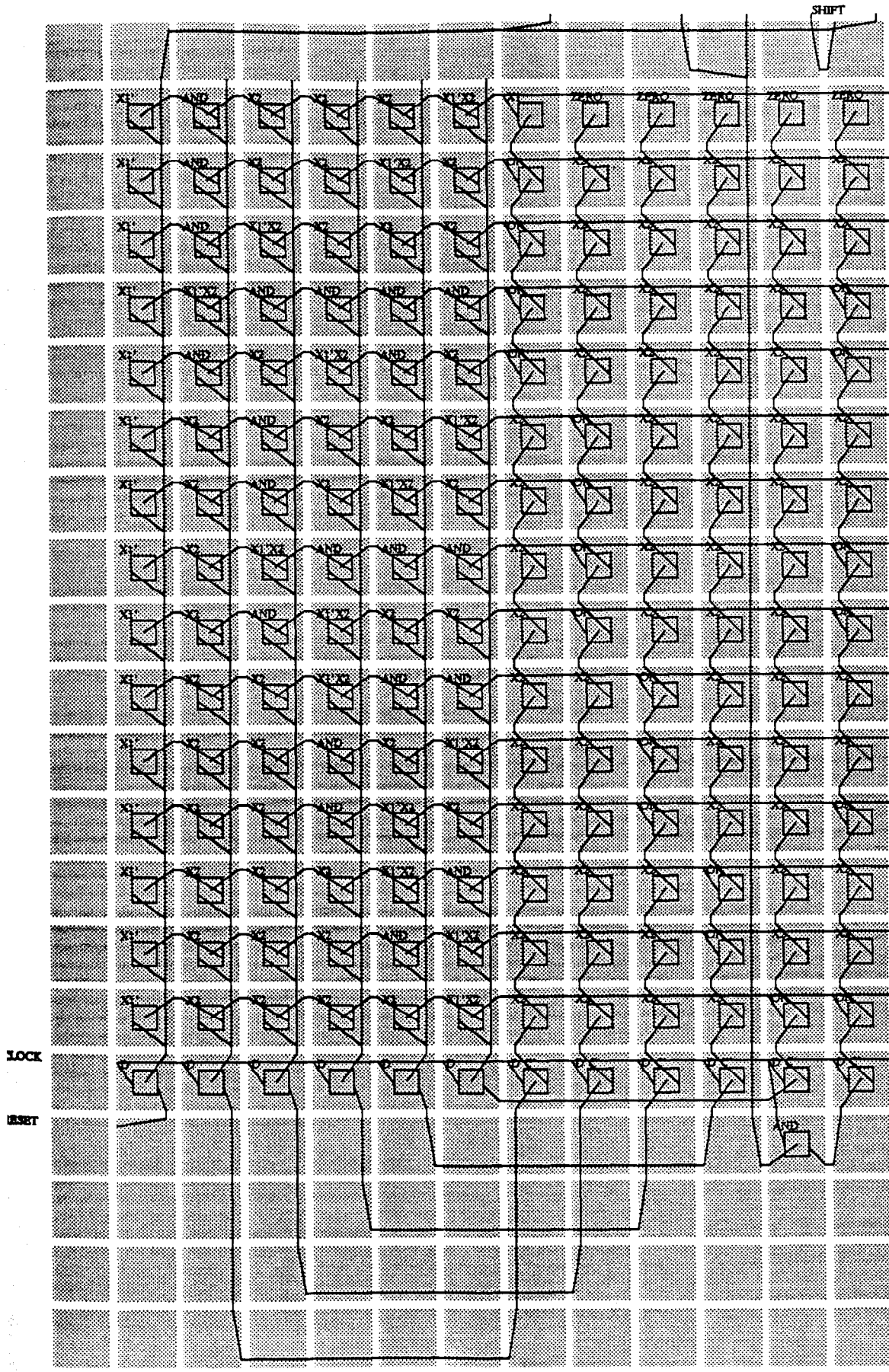


Figure 7–18: Key Control State-Machine Layout.

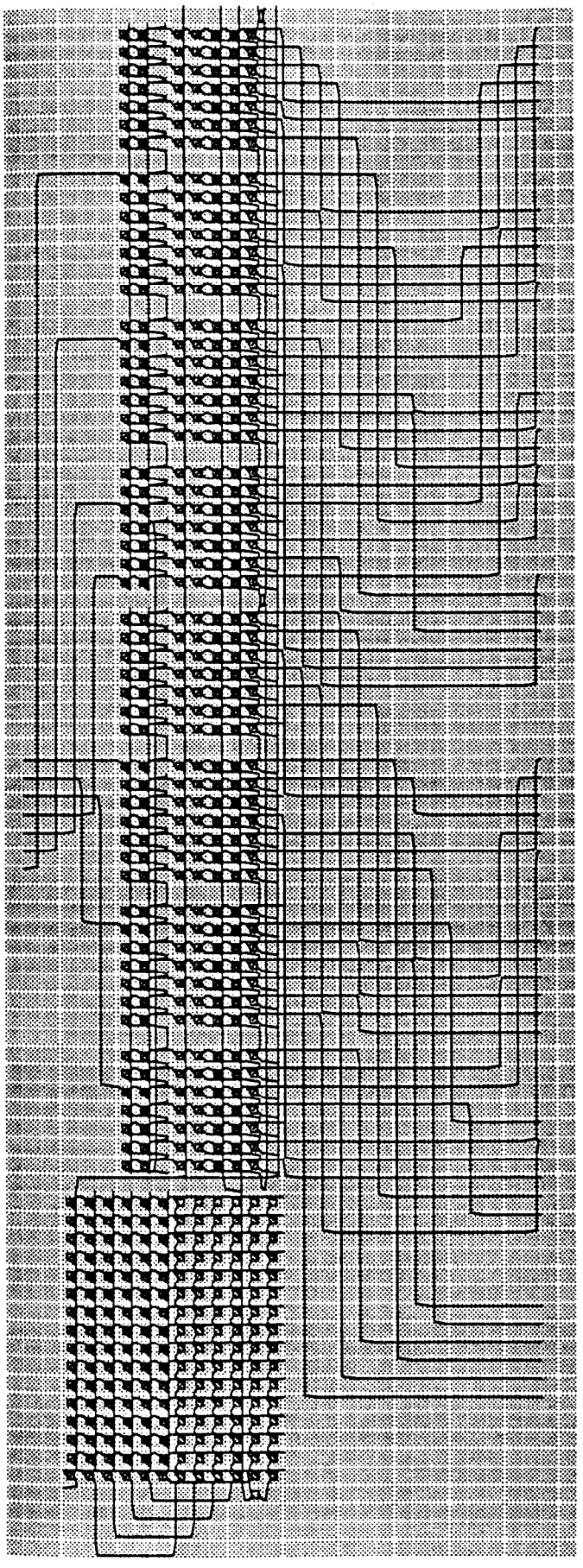


Figure 7–19: Key Control Unit Layout.

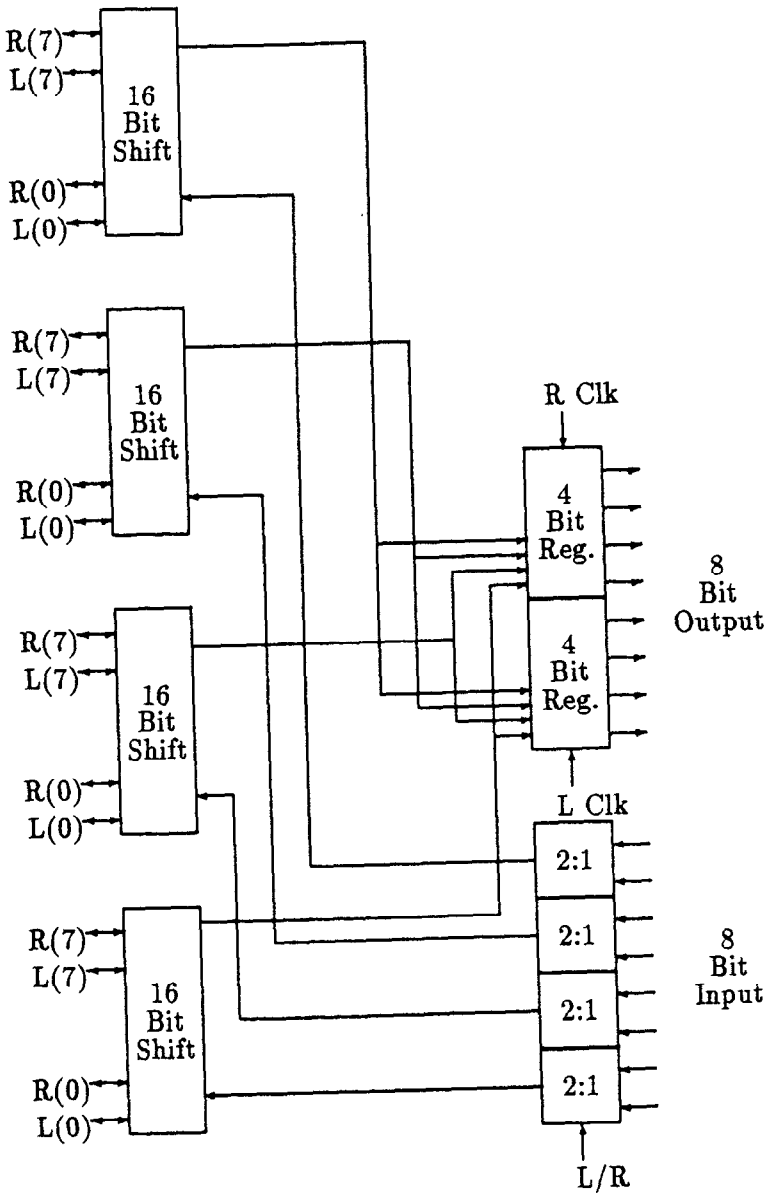


Figure 7–20: *IP* Design.

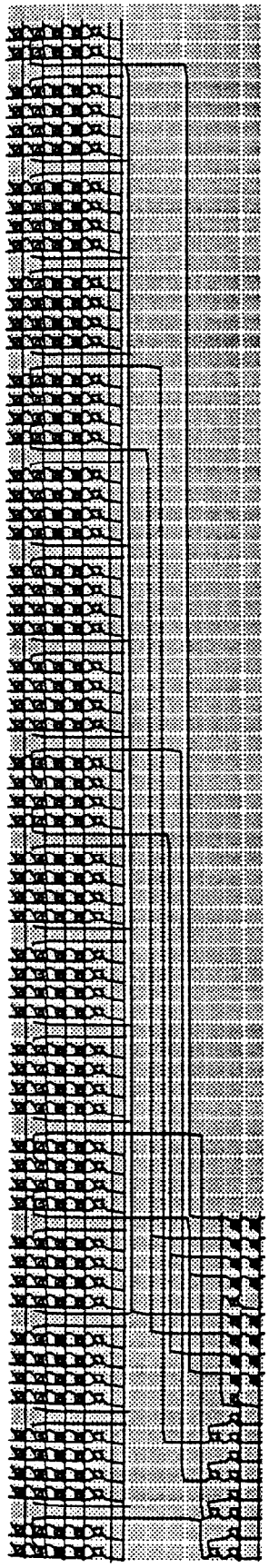


Figure 7–21: IP Unit Layout.

L							
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7
R							
56	48	40	32	24	16	8	0
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6

Table 7–5: DES Initial Permutation IP.

7.2.4 Pipelining.

The design of DES encryptor described above would be unnecessarily slow because of the accumulated delays through the f-box. It was decided, therefore, to separate the f-box into four pipelined stages to increase throughput. Note that the delays within the key controller and the IP unit are relatively unimportant because the computation within the f-box is much more complex and is repeated 16 times for every data input and keys are changed relatively infrequently. Any reasonable design of these units will be able to keep the f-box supplied with data and keys.

It was decided to aim for a throughput of 500,000 encryptions per second with a constant key. To achieve this one 64 bit ciphertext word must be calculated every $2\mu s$, since the f-box unit is used 16 times for each ciphering operation partially ciphered data must leave the f-box every 125ns. Table 7–6 shows the approximate delays through the major f-box components (we take routing delay through one cell as 2.7ns (the average of 3ns and 2.4ns) and calculation and output routing delay as 10ns. We can see that some of these components must be broken up into several pipeline stages to meet the performance objective. The last column of the

table shows the number of stages required. Within the CL-ROM we choose to place pipeline registers between the AND and OR planes as well as horizontally across the array, this decouples the AND plane calculation time which is quite high since there are 6 inputs. With pipelining between the planes as well as 3 stages across the array (two with 11 product terms and one with 10) we have a stage delay of $(m-l)r+11c = 3\times 2.7+11\times 10 = 118.1ns$. The ability to pipeline these large combinational logic units is a partial compensation for the loss in speed caused by using large numbers of two input gates rather than a single high fan in gate. Pipelining within wiring channels is easy to provide using the function units of the cells within the channel.

With the suggested pipelining the only stage which is close to the 125ns limit is in the KEYXOR unit - if this proved to be a problem a slightly more complex routing arrangement could split the 80 cell top-to bottom wraparound connection among several of the right to left routing stages - extra pipeline stages would not be required. The total pipeline length is 10 stages. Extra registers must be provided at the right side of the f-box unit to store the 'L' vectors corresponding to the ten 'R' vectors at the various pipeline stages within the f-box. The pipeline registers themselves are fairly small requiring only two cells for each master/slave register. They are clocked using the G1 and G2 global signals since propagation delay on the pipeline clock is an important consideration.

The time-line for the pipelining scheme is given as figure 7–22. The delay through the f-box is $16*(10*125ns) = 20\mu s$. It seems natural to pipeline the input and output phases through IP with this giving a total delay through the encryptor of $60\mu s$. In applications where large amounts of data have to be encrypted the throughput is of paramount importance so this is a reasonable design: we could reduce the delay to around $25\mu s$ by removing the extra IP pipelining at a small cost in throughput.

Unit	Direction	Route	Compute	Time	Stages
PBOX	R to L	20	0	54ns	1
SBOX	R to L	32	0	86.4ns	1
KEYXOR		85	1	239.5ns	2
SBOX	L to R	4	37	380.8ns	4
PBOX	L to R	65	0	156ns	2

Table 7–6: Delay Through Major f-box Stages.

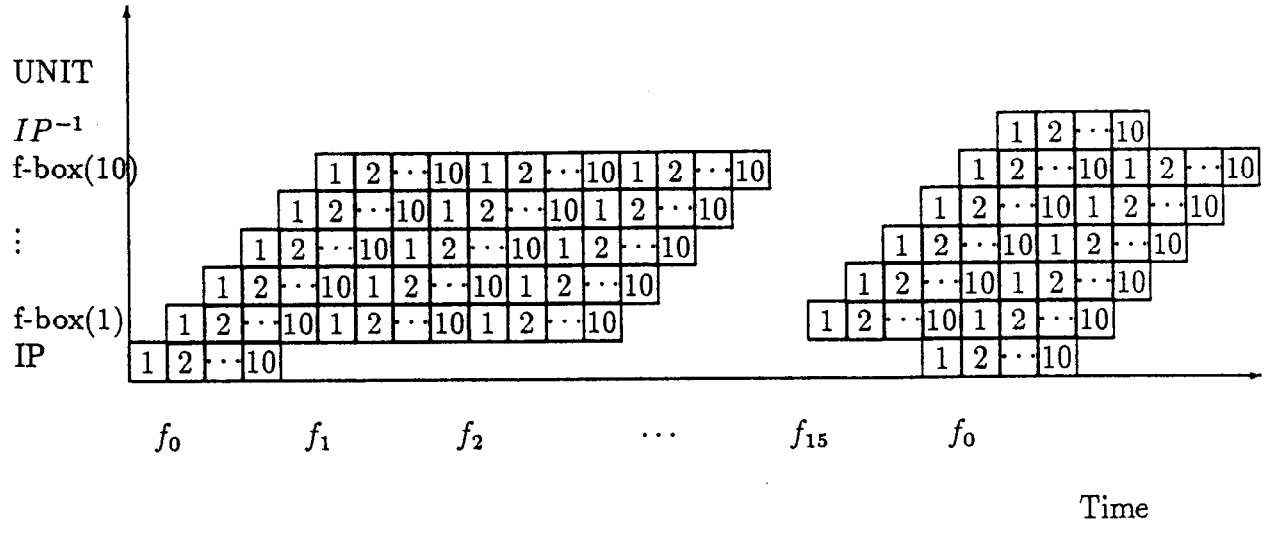


Figure 7–22: Pipelining Scheme.

7.2.5 Performance.

The performance of DES implementations can be measured in two ways: the number of encryptions per second with the same key and different data and the number of encryptions per second with a different key and the same data. The second metric corresponds to 'key-trial' in an attempt to break the cipher, the first to normal use of the cipher. The present design is optimised for normal use so we will concentrate on this problem. Table 7–7 shows figures for some DES implementations: the figures for SUN workstations are using the UNIX *crypt* function in the C library. This implementation is not particularly efficient and it is likely that the times could be significantly improved by recoding in assembly language and taking advantage of space time tradeoffs by using large data arrays to compute S-box functions. The time for a single encryption is given as well as the number of encryptions per second to show the effect of pipelining. The transputer array figure is a notional one assuming 1000 transputers and that each transputer has a good DES program allowing encryption ten times faster than a SUN 3/260: it is intended to illustrate the limitations of conventional parallel computers in this application. The time for the custom chip is taken from [Hoornaert88], this claims to be the fastest DES chip available with a throughput of 32M bits/sec (or 500,000 64 bit words/sec). Note that this chip has extra control logic to handle the more complex modes of DES in [NBS80] and also provides several registers for secure storage of keys on the chip. The CAL design makes use of several optimisations developed by the group which designed this chip. Although the CAL appears to be as fast it should be pointed out that $3\mu m$ processing technology was used in the custom chip which would be expected to be slower than the $2\mu m$ technology on the CAL and the CAL figures are based on circuit simulations rather than measurements on fabricated chips.

The CAL implementation of DES (including pipeline registers) requires an array of cells 88 cells high by 90 cells wide (27 for the key controller, 60 for the f-box and 13 for IP). This could be provided using an 6 by 6 array of 16x16 CAL chips or a 2 by 2 array of 64x64 CAL chips. The entire DES encryptor would fit on a single printed circuit board even with low density 16x16 chips.

Implementation.	*Encryption Time*	*Encryptions/Second.*
CAL	$60\mu s$	500,000
Custom Chip	$2\mu s$	500,000
SUN 3/50	0.5s	2
SUN 3/260	0.22s	5
Transputer Array	0.02s	50,000

Table 7–7: DES Encryptor Comparison.

7.3 Image Processing.

In this section we will illustrate the use of configurable logic to accelerate the 3-4 distance transform a calculation which has application in image pattern matching. This discussion is based on a paper written in conjunction with Jouko Viitanen [Viitanen88b] who originally suggested the problem and was involved with the development of the fast pattern matching algorithm used. The configurable logic implementation is compared with an implementation using a Xilinx logic cell array and an optimised assembly language program produced by Viitanen.

In this section we shall consider one possible system architecture using configurable logic consisting of a master CPU with a connected slave CL chip (or several of them). This architecture is used with Xilinx LCA's in the TAGIPS image processing system [Hanninen88]. The master processor controls the configuration of the CAL, sends data to it, and reads back the results. A potential advantage of this architecture is that the system could be programmed using a conventional High Level Language with a clever 'active' compiler [Gray88] selecting 'inner-loop' code sections for acceleration using the CAL coprocessor, this topic is covered in [Viitanen88b]. At present the division between normal and CAL processing is done manually. This architecture presents an attractive development environment but the need to channel data going to and from the CL array through

the microprocessor can be a bottleneck in many applications. Architectures with high-bandwidth connections and fast memories are necessary to take advantage of the raw power of large CAL arrays in highly parallel applications such as the cellular automata algorithm in the next section.

7.3.1 The Sample Problem.

Our example application, the Hierarchical Chamfer Matching Algorithm (HCMA), was first described in [Borgefors86] and fast algorithms for its calculation were presented in [Viitanen87] and [Viitanen88a] for the three-parameter translation-rotation problem. The advantage of HCMA is its robustness but in previous implementations it had the disadvantage of relatively long execution times.

HCMA is a model based, template matching operation. It uses simple operations, like addition, sorting and table lookup. The main phases in the recognition are image capture, feature detection (typically simple edge detection and thresholding to a binary image), calculation of a distance image (an image where the pixel values are proportional to the distance, or approximated distance, of the pixel from the closest detected feature), hierarchical pyramid creation from the distance image (quadtree, octree, or even pentatree rules can be used) and, finally, the actual matching. Coarse matching is done at the lowest resolution pyramid, trying every third or fourth translational position in both directions; rotation is estimated at the same time. Finer matching is done at higher resolution levels for a few best candidate positions.

The matching process involves transforming the model coordinates to different geometrically distorted positions with respect to the distance image. We are concerned in the practical case with three parameters: rotation and X-Y translation. Those values of the distance image that are addressed by the distorted model coordinates are picked up and accumulated. The accumulated sum is proportional to the 'average' distance of the model from the 'correct' position under the metrics used in the distance transform. This is a classical multidimensional optimisation problem where the distance image values form the cost function. The reduction

in computational load over the traditional template matching method (where the cross-correlation function is calculated between the model and the scene) is obvious since only additions are needed. References [Viitanen87], [Viitanen88a] show how the explicit polar coordinate transforms can be avoided in the geometric transforms. The local convergence properties of the distance image are good allowing several levels of the hierarchical representation of the image data to be used. This reduces the amount of data and the processing time considerably.

In this example we will improve the speed of the time consuming distance image calculation part of the algorithm using CL acceleration. In [Viitanen88a] the measured execution times on a TMS32010 image processing chip were 1.7 seconds for calculating the distance image and 10 to 20 seconds/model for matching against a 256 by 256 image with a maximum of 48 coordinate points in the model. For practical use in robotics, the processing times should be a few hundred milliseconds and we will show how to achieve this performance using Configurable Logic. The next section describes the calculations which must be performed in detail.

7.3.2 The Distance Transform Calculation

The 3-4 integer distance approximation is applied in the calculation of the distance transform (DT) used for creating the distance image. The approximation has small errors compared to the true Euclidean distance but is sufficiently accurate for practical use. The calculation of the distance transform involves two passes over the binary edge image, using the method of [Borgefors86]. The calculation in the first iteration is done as follows. Let F(x,y) be the two-dimensional discrete image array with row index x and column index y, where F(x,y) = 0 at valid feature points and maximum otherwise, then the corresponding distance image value for each array position in the first iteration is:

$$G(x,y) = min[F(x,y), G(x-1,y)+3, G(x,y-1)+3, G(x-1,y-1)+4, G(x+1,y-1)+4]$$

where processing is done row-by-row, in increasing x and y values. The second iteration is similar, expect that now the input image F(x,y) is the result of the first iteration, decreasing index values are used, and the signs of the index offsets above

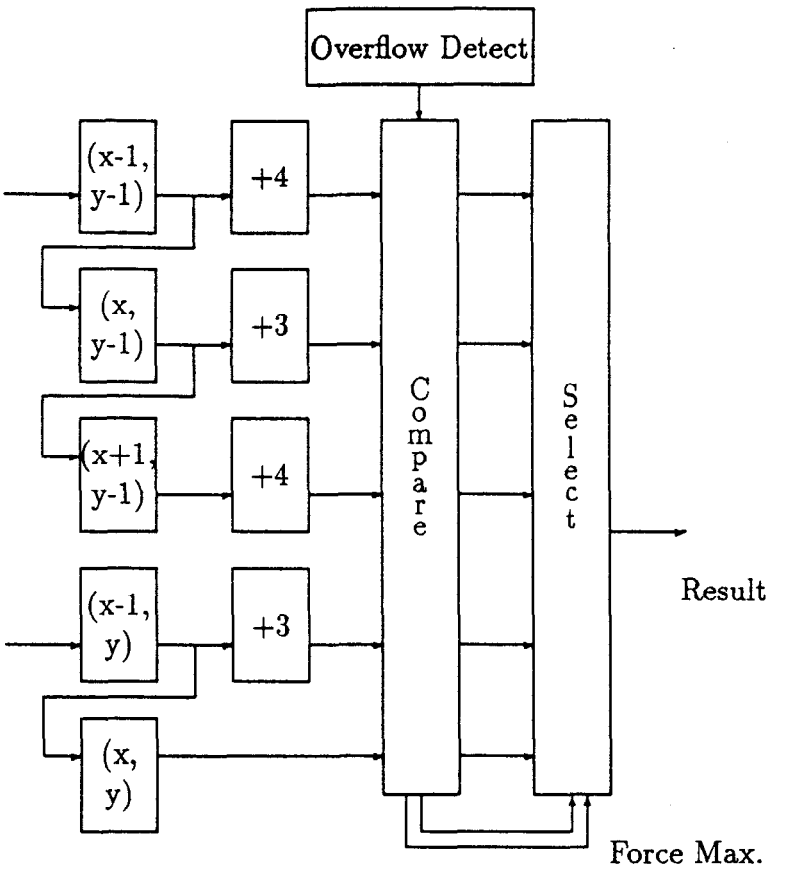


Figure 7–23: Distance Image Hardware Block Diagram.

are negated. A new search for the minimum has to be done among all the five pixels every time the mask is moved in the image (because of the additions). This makes the distance transform calculation fairly slow on a sequential computer.

A block diagram of the hardware to compute the 3-4 DT is shown in figure 7-23. It contains five registers in two groups corresponding to the two mask rows. The outputs of the registers are fed to adders which add the correct offsets. The most important part of the circuit is the parallel comparator section which selects the minimum of the five elements in one asynchronous, ripple-through process.

The structure of the comparator part of the circuit in figure 7-23 is shown in figure 7–24. This unit compares in parallel the corresponding bits of the five pixels, marked from A to E with five bit accuracy. The most significant bit (MSB) is marked with the highest number and the least significant bit (LSB) with zero.

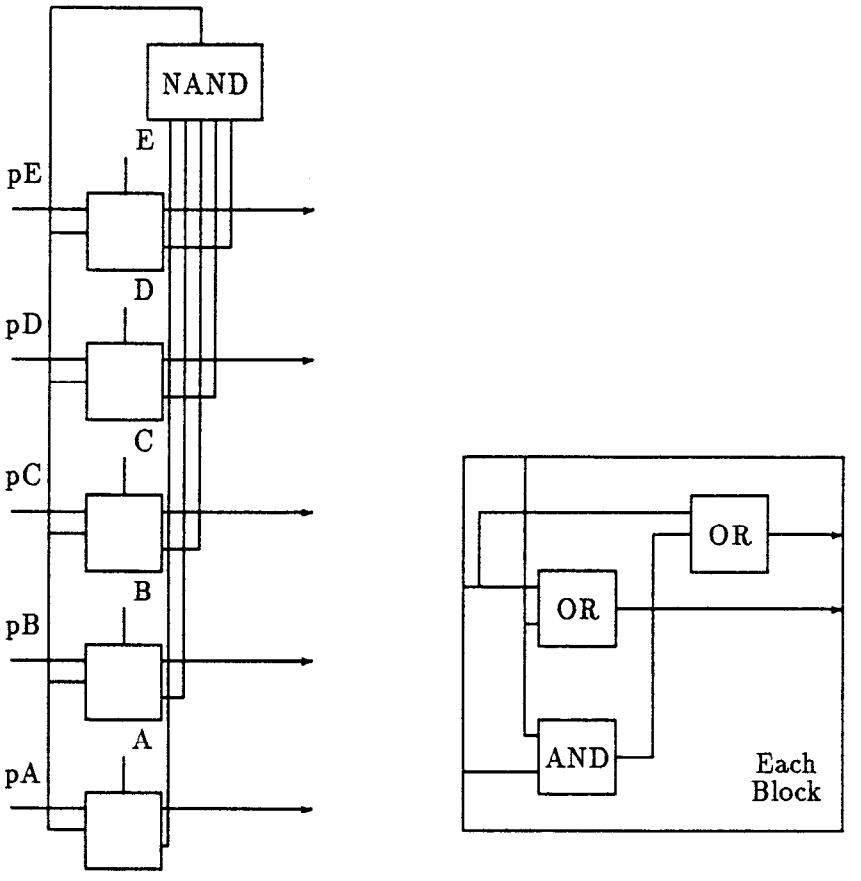


Figure 7–24: Parallel Comparator Logic Design.

The result of the comparison ripples down from the MSB to the LSB and is used to select the first pixel with the smallest value (several pixels may have the this value). The circuit gives approximately constant time for the comparison from a certain bit significance level. We could also have used faster lookahead to get a lower time for those comparisons where the minimum value could be determined at a high bit position but, since the host processor reads the results synchronously, the constant time approach was chosen.

7.3.3 LCA Implementation.

Figure 7–25 shows the Xilinx LCA design. The smallest available LCA with 64 configurable logic blocks was used: as we can see, only three cells were left unused and the routing capacity of the device was also fully used at many positions. It was impossible to implement the selection operation completely on the 8x8 CLB LCA and in the present design three bits of the selection are performed by connecting three output pins together on the circuit board and using tri-state control signals to do the selection. While this technique produces the correct answer the whole calculation has not been performed on the LCA and this should be noted when comparing areas. Use of this technique also makes it less feasible to reconfigure the LCA for another algorithm. A 10x10 CLB LCA would be required to perform the operation properly.

The total delay of the comparator is not a linear function of the equivalent gate delays along the signal path since the LCA evaluates all Boolean functions of up to four variables at the same speed. The performance of the circuit was estimated using the simulator supplied by the LCA manufacturer giving a maximum value of 250 ns for the selection of the minimum from the worst case MSB transition. The maximum delay for one CLB was 10ns with the LCA model used.

7.3.4 CAL Implementation.

Figure 7–26 shows a 'typical' bit slice through the CAL implementation of the distance transform unit. This design is different from the LCA one in that all the

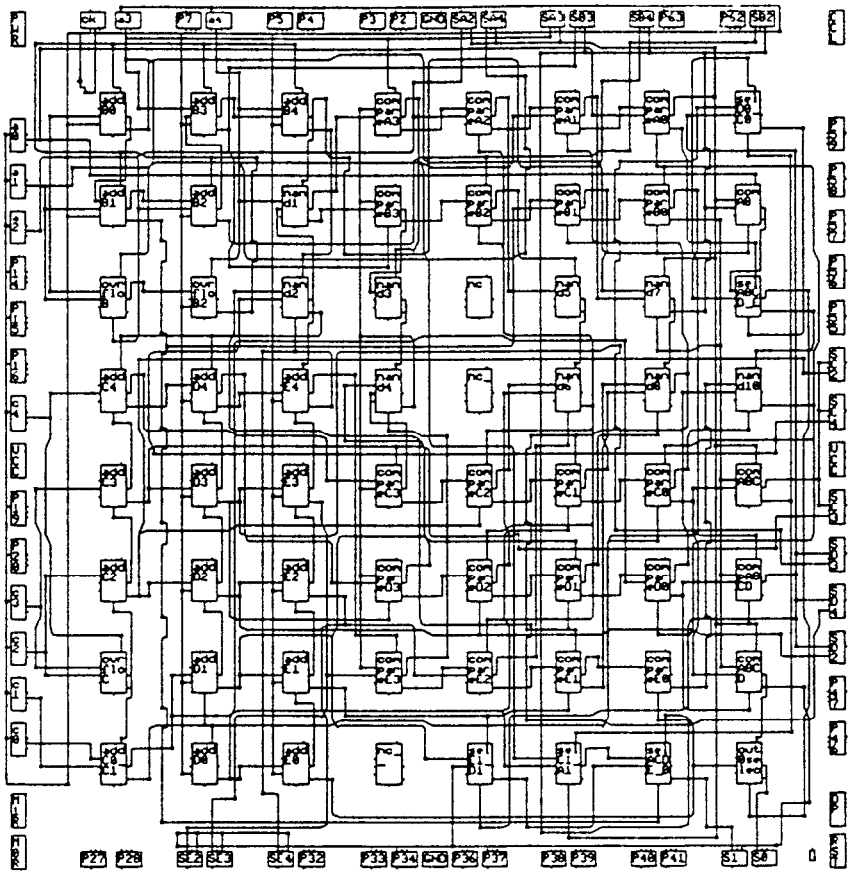


Figure 7–25: LCA Layout for Distance Transform Unit.

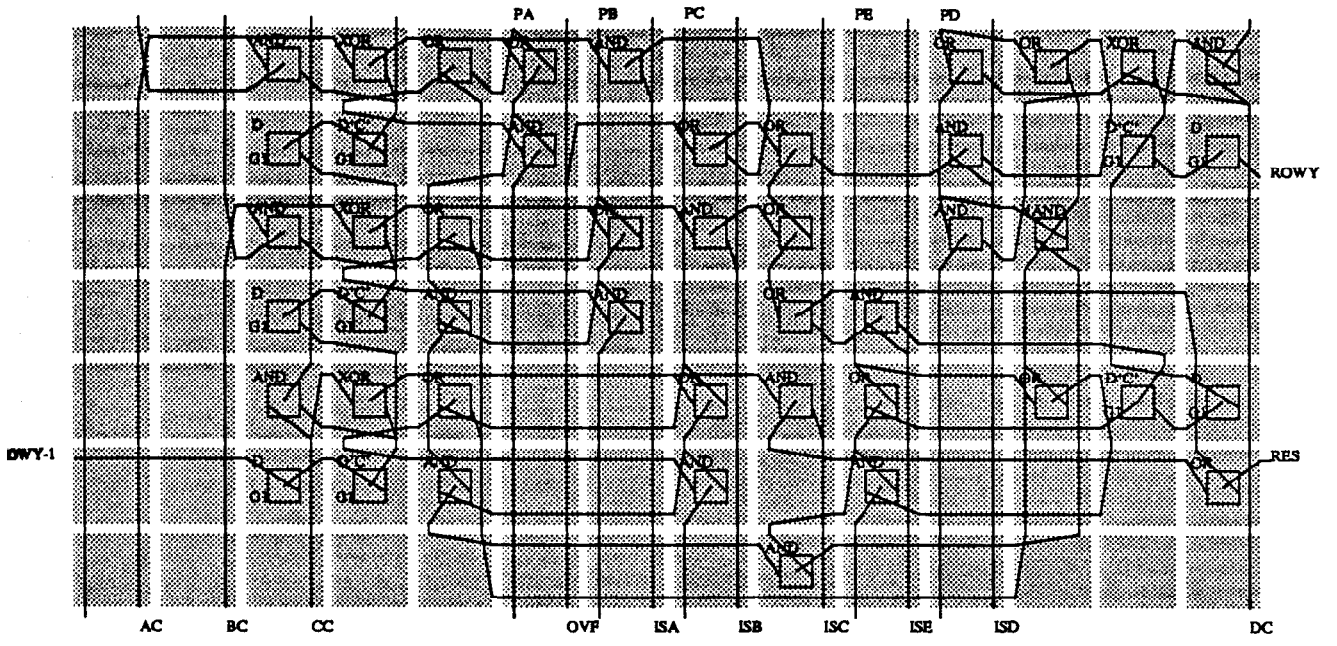


Figure 7–26: CAL Layout for Distance Transform Unit Bit-Slice.

major units are integrated into a single regular structure. The adders are at the far left and right of the central logic. Half adders are used since we only need to add constants - when we want to add 1 at the current bit position we would use XNOR and OR instead of XOR and AND to generate SUM and CARRY to the next stage. Each adder looks like a 2x2 square with a master/slave register on the bottom and the SUM (XOR) and CARRY (AND) gates on the top. The registers are clocked by one of the global signals (G1). Carry routing goes from bottom to top on the left and right.

The selectors and minimum detectors are implemented in the central area, 5 signals $pA \ldots pE$ go down the unit, and another 5 $isA \ldots isE$ up, $pA \ldots pE$ correspond to 'possibly' A through E and $isA \ldots isE$ to definitely A through E (after all bit positions have been compared). The '$isA \ldots isE$ signals are AND'ed with $A \ldots E$ and OR'ed (in the centre and far right of the design) to form a 5:1 selector and produce the result at this bit position (marked (RES)). The ovf signal is used to force RES high at all bit positions (corresponding to the maximum pixel value) if all the additions overflow. The right hand side of the central area is less

regular since it takes advantage of extra vertical space arising from only having one adder to save two columns of cells.

The complete Distance Transform unit is composed of 5 of these slices, all slices are identical except for the half-adders where the logic functions for sum and carry depend on whether 1 or 0 is being added at the present bit position. At the top of the unit three extra rows of cells are used to route carry out signals from the adders to the corresponding *p* lines thus inhibiting pixels whose values have wrapped around after an overflow from being selected. An AND gate over all the carry lines detects when all pixels overflow and this is used to set the *ovf* signal which runs up the array with the *is* signals and force all result outputs to ones (the maximum legal value). At the base of the design a row of cells is used to invert the *p* signals and feed them back to the corresponding *is* signals. Several *is* signals can be high if more than one pixel has the minimum value: the selector will still output the correct value in this case.

This layout trades some area for increased regularity: these four extra rows of cells could be eliminated if special MSB and LSB slices were designed. The complete design requires $5 \times 7 + 4 = 39$ rows of 14 cells or 546 cells in total.

7.3.5 Implementation Comparisons.

Comparison with Xilinx LCA. The best way to compare areas with the Xilinx LCA is by counting the control store required since this will cancel out differences in processing technology and die size. The LCA array has 12038 bits of configuration RAM: at 20 bits of RAM per CAL cell this represents 601 CAL cells. Thus the LCA and CAL implementations are almost equivalent in terms of area despite the large numbers of relatively high fan-in gates in the design. The Xilinx design has been fitted into a single chip where the CAL design represents an arbitrary shaped rectangle of cells. This is representative of the different ways the two systems are intended to be used CAL designs will normally be done in a large array built up from several chips and the comparator would represent only a small part of the total system whereas Xilinx arrays are intended for relatively

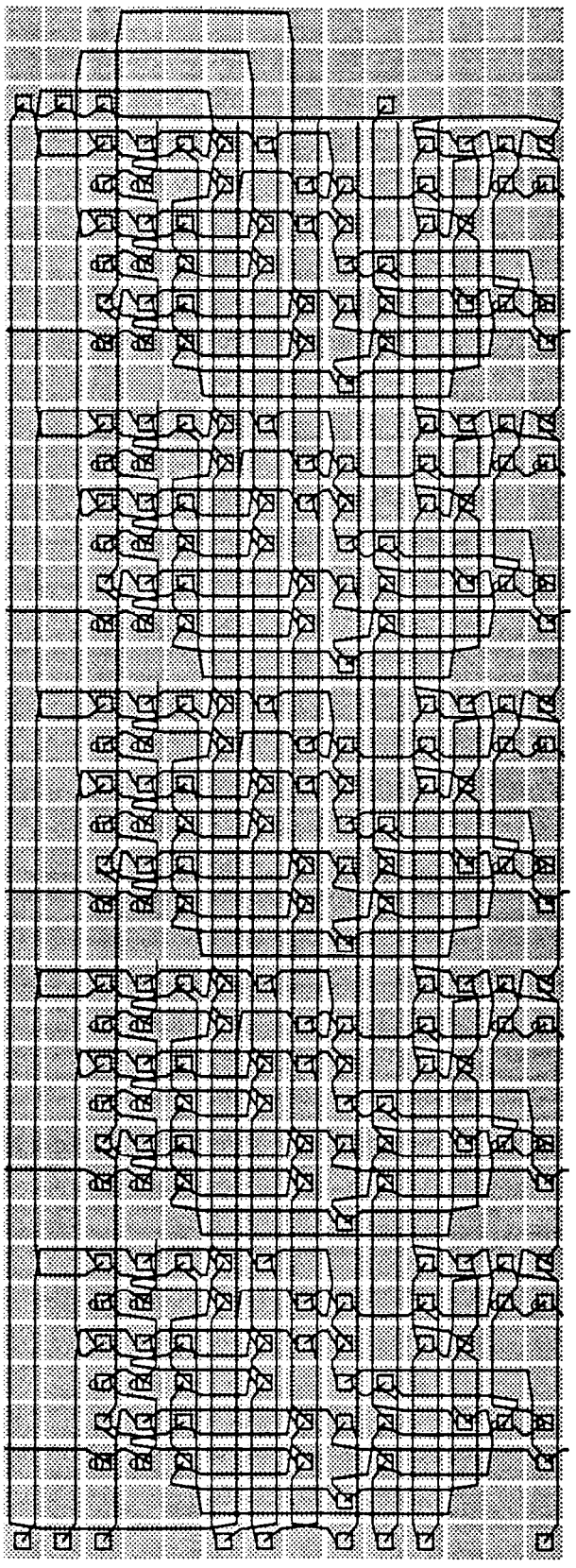


Figure 7–27: CAL Layout for whole Distance Transform Unit.

small designs which fit on a single chip. The CAL design could be folded into a squarer shape at the expense of some extra routing area.

There are three main sources of delay in the CAL design:

1. Carry Propagation. At each slice the carry signal incurs a delay of $6r + 1c$, where c is the compute delay (including routing to adjacent cell and r is the pass through routing delay.

2. p Signal Propagation. At each slice there is a delay of roughly $7c + 30r$ (in the worst case when the input data is high and the previous p line is low).

3. *is* Signal Propagation. The *is* signals incur a delay of $7r$ at each slice plus a delay of $5c + 10r$ for the selection operation after the data arrives at the MSB.

These three delay sources add up to $5(8c+43r)+5c+10r = 45c+225r = 1.06\mu s$: this is a lot larger than the 250ns reported for the Xilinx LCA. Much of this discrepancy has to do with the way the CAL delay figures were arrived at.

The method we have used to calculate CAL delays results in a large overestimate of actual circuit delays: this was intentional since it was considered preferable that any uncertainty went against the CAL. There are several places in which overestimates occur:

1. Circuit Simulation. The routing and computation delay figures for CAL come from slow SPICE models applied to a hand extracted circuit. When the circuit was being entered any capacitance and resistance estimates were rounded up. These factors could account for a factor of two.

2. Computation Speed. We have taken the computation speed as constant at 10ns for all operations: in fact this is a worst case. The actual computation speed depends on which multiplexors are used within the function unit and is often much smaller.

3. Routing Speed. We have assumed that the delay along a chain of inverting multiplexors is the delay through a single multiplexor times the number of multiplexors in the chain. The real situation is more complex since the invertors will start to switch long before the input voltage reaches the half way point. We would expect routing delays in a long chain of multiplexors to be much less than our simple model predicts (the Xilinx simulator uses an RC model of the interconnect to obtain more accurate results).

As well as the overestimates introduced by our simplistic method of delay calculation the effects of the improved processing technology (including the use of 'special' transistors, presumably with increased conductance in the switching units as well as better design rules) and the speed 'grading' of LCA parts must be considered. The Xilinx design has been in production for 2 years now and will have been finely tuned for speed; there is still a lot of scope for speed improvement in the CAL layout.

The CAL design is much more regular than the LCA design and was produced faster (about 2 days versus 1 week) despite the fact that it was done as a hand layout whereas the Xilinx design used automatic placement and routing tools. Overall, the speed of design and its regularity coupled with the slight area advantage tend to vindicate the 'keep-it-simple' philosophy of the CAL system.

Comparison with Conventional Processor. The corresponding operations on a TMS 32010 signal processor take from 29 to 37 instruction cycles of 200ns, so the speedup factor is from 23 to 30 (for Xilinx parts). The I/O overhead has to be added to both cases, depending on the actual implementation, so the total time for building the complete 256 by 256 distance image can be estimated to be about 64 ms using a fast host CPU with a 120 ns I/O cycle time. The TMS 32010 sequential implementation took more than a second.

7.3.6 Conclusions.

This design example seems to show that the proposed configuration of a conventional processor with a CAL coprocessor can offer significant speed-ups in some important problem domains. Automatic design of this kind of multi-level logic unit has become quite feasible with recent improvements in logic synthesis techniques [Brayton86b]. While it is not expected that automatic designs will approach the manual ones in area efficiency or speed they could still offer attractive performance improvements over normal processors.

7.4 Fluid Flow Simulation.

Recently, much attention has been focussed on cellular automata simulations of physical problems: [Wolfram86] contains a collection of important cellular automata papers and [Toffoli87] provides a much more approachable introduction to the topic based on one particular cellular computer the CAM-6.

The goal of these models is to set up a 'universe' based on a particular cellular automaton rule which mimics physical reality in some way and observe its evolution - this is to be contrasted with the more traditional approach in [Preston84] where the automata are used as data-processing devices to produce desired transformations in image processing. Of particular interest are the cellular automata models for fluid flow simulation: currently a large fraction of the world's supercomputer time is burned on this problem and cellular automata models promise cheaper and more powerful computers for this application. Although there was initially a degree of scepticism about cellular automata models in the physics community a lot of work has been done on validating the cellular models in the last few years and they have gained wide acceptance. Validation has been done both against experimental results and by showing mathematically that the cellular models approach known differential equations (e.g. the Navier Stokes equation) for particular 'regions' of interest [Frisch86],[Salem86],[Frisch87],[Shimomura87].

In this section we will outline the design of a configurable logic based machine for solving such problems.

7.4.1 The Model.

The most commonly used cellular automaton rule for fluid flow simulation is called 'hexagonal lattice gas' and simply specifies what happens when molecules collide on a hexagonal grid. All molecules are assumed to have the same mass and velocity: the density of the grid (i.e. the proportion of lattice sites which are populated at the start of the simulation) is proportional to the fluid's Reynold's number. An example rule is shown in figure 7–28. In the 'before' step all the arrows face towards the current site and in the 'after' step all the arrows face away: thus the situations can be coded up as six bit numbers representing the presence or absence of a particle travelling in a particular direction. Note that to model real fluids properly the rules must obey certain physical properties (e.g. conservation of momentum) and are symmetrical with respect to rotation and reflection.

To obtain interesting results we usually wish to place some object within the fluid. At those sites on the edge of the object a different set of rules will be used - e.g. particles reflect back with angle of incidence equal to angle of reflection. It is inefficient to reconfigure the update unit when these sites are being computed so the normal technique is to add an extra bit plane which contains ones along the outline of the obstacles. We then use an automaton rule with 7 inputs and 6 outputs (since the obstacles do not move) specified as the normal 6 bit rule when the 7th plane is 0 and the reflection rule when it is 1.

The rule in figure 7–28 does not use a true hexagonal grid but has approximated the hexagon on a rectangular grid using 45° diagonals: this is an acceptable transformation in most cases and is necessary to match the structure of the RAM memory which will hold the model points. In some cases a more accurate approximation will be required, this will involve a slight increase in complexity and some unused store in the RAM.

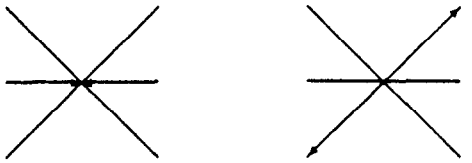


Figure 7–28: Example Lattice Gas Collision Rule.

Another point of interest is that the 'after' stage in our example is not the only one which conserves momentum - particles could also leave on the other two diagonals. Rules which make random choices between different possibilities are also of interest and can offer increased accuracy. While it is inefficient to add randomisation at each individual node update other techniques such as using one possibility on 'odd' lattice sites and the other on 'even' or using one possibility on 'odd' update cycles can be used and are reasonably effective.

Although the basic computation is very simple to get good results it must be repeated on a huge grid of points (2048x2048 is not uncommon) and perhaps 10,000 iterations must be run between samples. One such cycle would involve about 42×10^9 node updates. The need for parallel processing is obvious. After enough cycles have been run the model is divided up into larger sectors with about 64 sites per sector, the average direction of the molecules within each sector is converted into an arrow whose length represents the number of particles heading in the most common direction and a diagram such as figure 7–29 (taken from [Wolfram86]) is produced clearly showing the structure of the flow. The site data can also be used to calculate numbers of interest such as forces on obstacles and pressures with appropriate 'calibration' of the model.

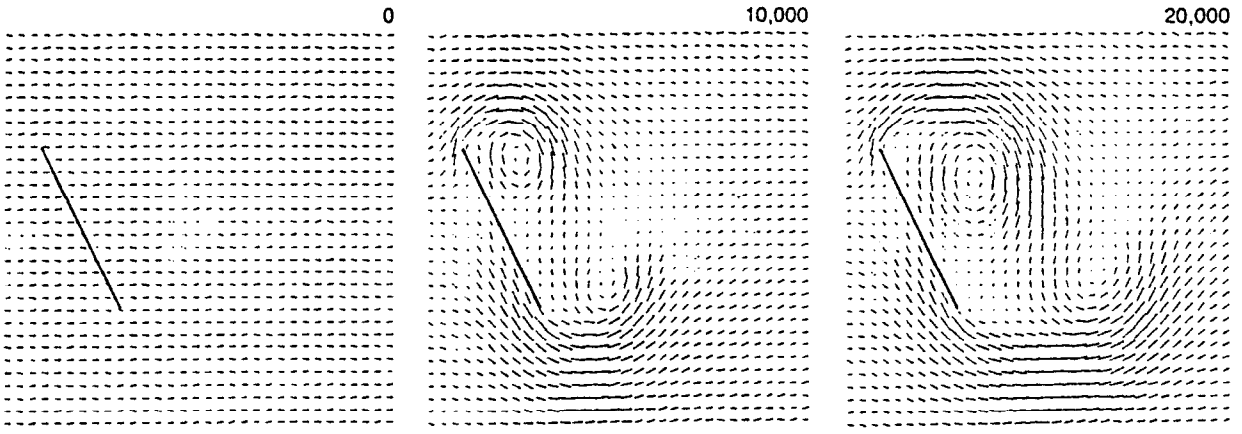


Figure 7–29: Example of Output from Lattice Gas Model.

7.4.2 Architecture.

There are several methods of organising the update computation.

1. One Processor per Site. This is the obvious method, using a hexagonal grid of processors and communication between them to deal with particle movement. This is not practical for real models with around 4 million sites.

2. Single Phase. In this implementation the calculation consists of both particle movement and site computation. Enough memory is provided within the computation unit to store the values at all sites adjacent to those whose new values are now being computed. Each site takes the inputs for its computation from the memory corresponding to adjacent sites (converting arrows leaving the adjacent site in the 'after' phase of the rule to arrows entering this site for the 'before' phase). This method is very easy to use when only a single update processor is available but the routing becomes complicated when several updates take place in parallel (because of the need

to store 'before' values for a given site after its new value is computed to allow adjacent nodes to perform their update).

3. Two Phase. In this approach [Cloqueur87] the memory subsystem is treated as several bit planes, the values in the planes at a particular address correspond to the 'before' data for the corresponding site. The computation unit performs the update and writes back the new 'after' value. After all sites have been updated the memory subsystem manipulates bitplane address offsets to perform a separate 'movement' phase in which the bit planes are 'realigned' so that in the next computation sweep cells again have 'before' values. For example, to move a bit plane 'up' one would add an offset of the number of pixels in a row. This assumes that memory addresses 'wrap around' at the edge of the bit plane. Wrap-around is actually very desirable in a cellular automaton system since otherwise the fact that edge sites have no neighbours in one direction can distort the results. This approach is very easy to implement, involves almost no performance penalty and simplifies the design of the update unit. We will assume the two-phase approach in this design, although, because the circuitry is reconfigurable it could also implement the single-phase method.

Performance Goals. Before starting out on a design like this it is important to consider the performance that one requires. Supercomputer implementations of automata problems currently achieve 10^9 site updates per second so we will take this as our goal and assume that we are working with 2048x2048 site lattices. Since 4M 7 bit words of memory is required for this size of model and 4096x4096 site models are already being suggested we also require that the memory subsystem be implemented using cheap dynamic RAM.

Update Unit Implementation. The first thing to consider is the best way to implement the site update computation on a CAL. The first thing worth noting is that the case where the 'object' bit plane is 1 is very simple, reflection simply involves a swapping operation on inputs. Use of a CL-ROM seems appropriate for

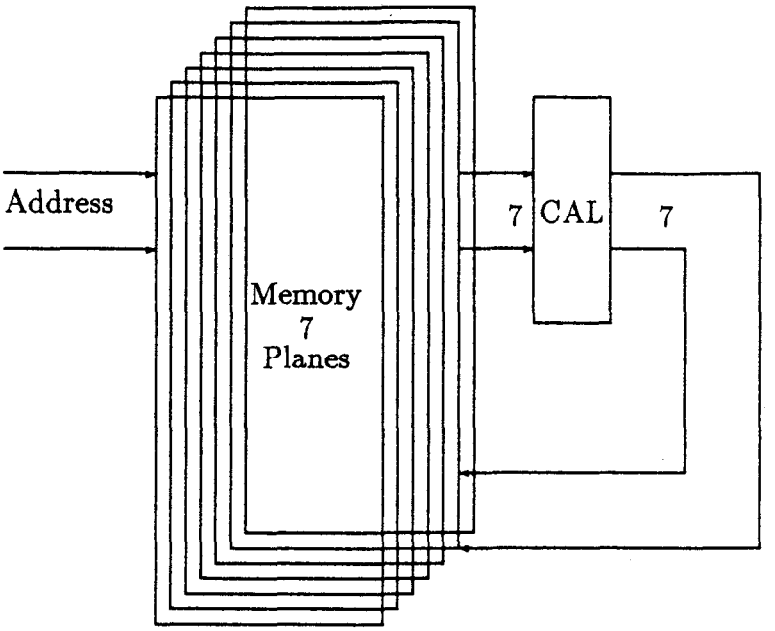


Figure 7–30: Basic Lattice Gas Simulator Architecture.

the hexagonal-lattice-gas rules when no object is present. We can integrate the reflection unit with the CL-ROM as shown in figure 7–31, note that for clarity the pipeline registers are not shown in this plot.

We have $n = 6$, $m = 6$, $p = 32$ so using the equations for pipeline stage delay (Chapter 6) and assuming 4 product terms per pipeline stage the delay is $5 \times 10 + 5 \times 2.7 + 4 \times 10 = 103.5ns$. There are 8 pipeline stages in the ROM plus one for the extra multiplexing. Use of dynamic memory suggests a pipeline tick of 120ns to allow a memory access to occur and data be routed to the CAL in a single cycle. We will assume that 64x64 cell chips are being used: each CL-ROM is 11 cells high and an extra line of cells is needed to route the 7th bit plane giving a total height of 12 cells. This allows 5 units per chip. Assuming that the multiplexors use a fairly sparse layout 6 cells wide we have a total width of 32 (product terms)+6 (multiplexor) +18 (pipeline registers=56 cells. We still have 4 spare rows and 10 spare columns of cells on each chip for any other circuitry we may want, e.g. wiring channels to simplify off chip routing. With 128 computation units (using 26 CAL chips) we can do 1.07×10^9 updates per second.

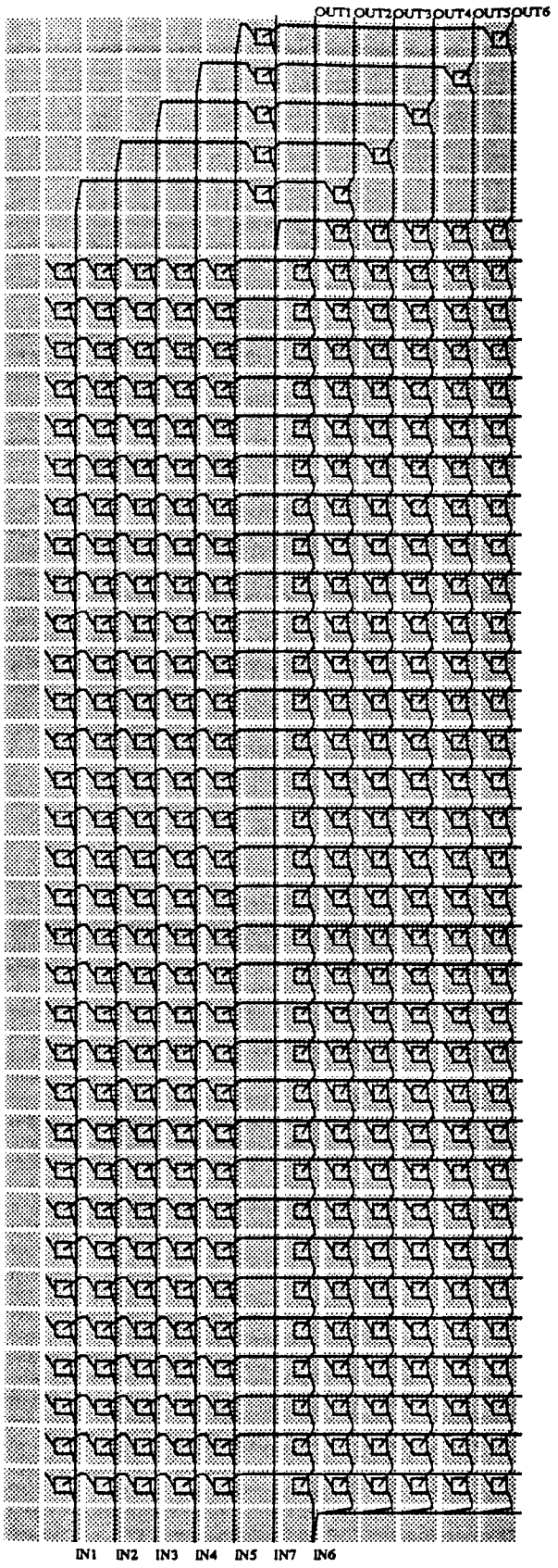


Figure 7–31: Layout of Single Update Unit.

Memory Subsystem. In order to achieve the very high data rate (6x128=768 bits every 120ns) required a special purpose memory subsystem is necessary. We will consider the design of one bit 'plane', all the bit planes can be identical. Each bit plane must provide 4M bits of RAM and deliver 128 bits every 120ns. If we choose memory chips which output 4 bit words then we need to access 32 chips. We also have to simultaneously write back information from the CAL into the bit plane memory. Given these considerations a single 4M bit plane memory could be built from two banks of 32 cheap 16Kx4 (64K) dynamic RAM's (figure 7–32). Larger 4096x4096 site models are already being suggested so a version of the system with 64Kx4 memories is also worth considering.

The addressing circuitry required is shown in figure 7–33, note that we need to decrement the address from which data are being read by the number of stages in the pipeline to get the write address. Each bit plane is assumed to receive the current 'site' address from a global controller. Naturally, all this control circuitry would be implemented using CAL chips to provide maximum flexibility.

7.4.3 Comparison with Previous Systems.

The configurable logic architecture differs from previous special purpose cellular automaton machines by having many processors and using configurable ROM's rather than lookup tables for calculations. The main disadvantage of the lookup technique is that the size of the RAM required grows as 2^n with the number of inputs n. RAM size and number of processors must also be fixed in advance - you do not get more processors when the rule is simple. Cellular automaton models with as many as 24 inputs have been suggested to increase the accuracy of the simulations. Naturally, if the rules were completely random functions of n variables configurable logic could do no better than lookup tables: however cellular automata rules for lattice gas equations are very regular (this follows from basic properties of the model such as conservation of momentum and symmetry which are part of the physical model). We can see the advantages of the configurable logic approach from the example above: if a RAM lookup table had been used

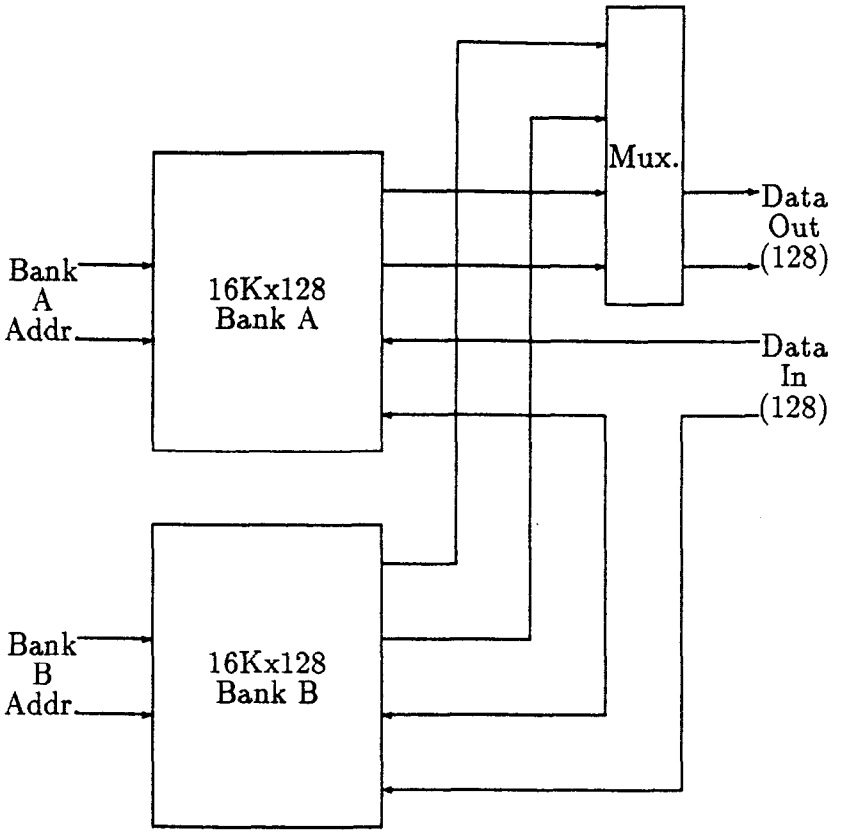


Figure 7–32: Architecture of Single Bit Plane.

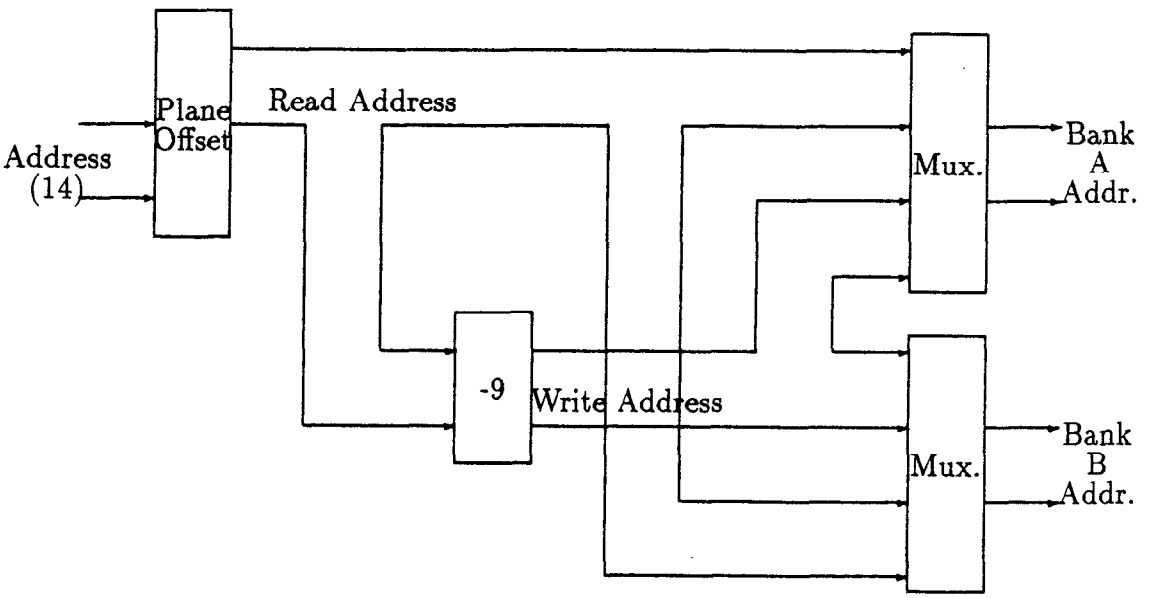


Figure 7–33: Memory Addressing for one Plane.

then the size of the RAM would be doubled by the seventh input bit, however with the CL implementation only a very small additional piece of logic is required. It is likely, therefore, that despite the number of inputs in the newer rules an implementation using relatively few logic gates will be possible. The structure of this implementation will vary from rule to rule. In the lattice gas model of this example rotation and reflection symmetry could be use to reduce the number of rules from 64 to 14 [Wayner88].

With a reasonable amount of hardware the Configurable Logic system can easily outperform any conventional computer on this problem: table 7–8 compares the expected performance of the system with supercomputer implementations and custom chips. The first Princeton figure is a maximum per chip. There are 1500 chips in the present configuration however I/O considerations on the SUN-3 host limit the performance per chip to 2/3 million sites/second giving an overall performance of 1000 million sites/second. This figure is disappointing for a custom chip and is a result of their method of organising the computation to reduce I/O bandwidth and avoid the need for a special memory subsystem - 75% of the Princeton chip area is taken up with a large shift register and there are only two update units per chip. The Princeton chip uses $3\mu m$ technology rather than the $1\mu m$ technology needed for a 64x64 cell CAL. The Connection Machine and Princeton Chip figures come from [Wayner88], the figures for the CRAY are from [Shimomura87]. The CAM-6 and RAP systems mentioned above do not support enough lattice sites and have much lower performances because they have only one lookup table for site update calculation.

7.4.4 Conclusions.

We can see that the CL design is extremely cost effective providing supercomputer performance from a system with fewer than 30 custom chips (even counting in control circuitry) and less memory than a low end workstation. Even when the cost of a host computer to control and display the results of the simulations is included the system is very attractive. The use of reconfigurable chips in the

System.	*Site Updates/Second* $x10^6$.
CRAY X/MP	500
Connection Machine	1000
Single Princeton Chip	20
Princeton System	1000
Single 64x64 CAL	40
CAL System	1070

Table 7–8: Performance on Fluid Flow Simulations.

update and control units will allow this system to evolve as the models become more complex, extra memory planes and larger update units are both easy to accommodate.

7.5 Summary.

The examples in this chapter have shown the suitability of configurable logic as an implementation medium for a range of designs. The stopwatch example is typical of small designs which could be implemented within a single configurable chip and the DES example is typical of a design which would require a larger array built at board level. The DES example is quite large and demonstrates the use of a wide variety of design techniques and particularly the importance of pipelining to reduce the effects of communication delays. The image processing example illustrates the use of configurable logic as an accelerating co-processor as well as allowing comparison with the Xilinx system. Finally the fluid-flow cellular automata application shows that extremely high performance can be obtained from configurable logic arrays in an important application.

Chapter 8

Conclusions and Future Work.

8.1 Overview of Thesis.

In Chapter 1 we covered the basic framework within which gate level configurable systems are designed and suggested target applications for the configurable architecture to be developed within the rest of the thesis.

Chapter 2 then took a 'high level' look at the resources available to implement configurable logic and the different ways in which they can be utilised. Metrics were developed for measuring the efficiency of architectures and some desirable properties of general purpose implementations were discovered. Timing disciplines for configurable systems were also considered.

In Chapter 3 we developed the ideas of Chapter 2 and introduced the idea of cellular arrays. We looked at what resources should be placed within each cell. Two main categories of resource were identified: functional and routing. Building on previous research in the area a new cellular architecture, Configurable Logic, was defined. Comparisons were made between the new design and previous configurable systems.

In Chapter 4 we developed the VLSI implementation of the Configurable Logic architecture specified in Chapter 3. The design parameters of the basic control store/ multiplexor combination were explored in detail with several possible implementations being considered. The design of input/output, memory control and power supply systems for a silicon implementation of CAL were discussed.

Chapter 5 discussed particular implementations of the configurable logic architecture in VLSI. Firstly, the prototype CAL chip design which implements a 16x16 array of dynamically programmed cells was covered. Area predictions for how the CAL architecture would scale with improved processing technology were given. Secondly, a statically programmed version, the CLA or Configurable Logic Array, in which the cellular array is configured by the second metal mask was dealt with. Thirdly, the possible extension of the VLSI implementation of CAL described in Chapter 4 to Wafer Scale Integration was discussed.

Chapter 6 dealt with CAD tools for the configurable logic system. The purpose of this chapter was to show that algorithms developed for silicon compilers can readily be adapted for cellular systems and that it should be practical to integrate support for configurable logic into an existing silicon compiler environment.

Finally, in Chapter 7 we covered examples of the application of the configurable logic technology. Four example designs were covered. The first example, a digital stopwatch chip is typical of the sort of application in which CAL could be used as an EPLD replacement and has been implemented using several different implementation technologies. Area comparisons were given to show where configurable logic fits into the design space. The second example, a DES encryptor, was a much larger system which requires many CAL chips built into a larger array at board level. This example illustrated the applicability of CAL to the prototyping of ASIC's or as an accelerator for conventional computers. The third design, a unit for computing the 3-4 distance transform illustrated the use of CAL as an accelerating coprocessor within a microprocessor system. The fourth design, a sketch of an computer capable of running cellular automaton models for fluid-flow simulation at supercomputer speeds illustrated the power of configurable logic in an important application domain.

8.2 Development.

At the end of this project the configurable logic architecture has been brought to a point at which it is approaching commercial viability. It can compete with anything currently available as an EPLD and system for prototyping ASIC's. The ASIC prototyping application is probably the most immediately commercial since there is no comparable system available and a large potential market for one. The more radical uses of configurable logic as an algorithm implementation technique also have great potential but would require more development, particularly of CAD tools.

There are many ways in which the CAL chip itself and the design automation tools could be improved. Some have been suggested in this thesis. The research part of the project has been completed: what is needed now is straightforward engineering development towards a marketable product.

8.2.1 Idealised Silicon.

As more and more complex systems are implemented using a single silicon chip or wafer it will become harder and harder to design systems correctly first time. Diagnosing faults in highly concurrent large scale systems is difficult because it is often impossible to look at internal values within the system. Testing such systems also becomes harder as they get larger. As the density of these systems increases the mismatch between the abilities of simulation on conventional computers and the complexity of the design gets worse. Formal methods of validating VLSI designs have been suggested as a means of solving the correctness problem but their practicality is questionable because of the computational complexity of automatic theorem proving and the difficulty of writing precise specifications of large systems. At best they only provide a partial solution since they do not attack the testability problem. An alternative approach is to reduce the cost of being wrong by using dynamically programmable 'idealised silicon': this would allow the traditional

iterative approach to the development of large software systems to be applied cost-effectively to hardware.

8.2.2 Other Architectures.

This thesis has deliberately concentrated on the development of one particular cellular architecture to the point at which it could be demonstrated to be useful. However, there is a huge range of cellular architectures many of which could be extremely important in particular applications. Exciting possibilities such as the self-configuring and repairing computers first suggested by Von Neumann [VonNeumann66] are approaching feasibility. Previous research in this area has been hampered by the inability of conventional machines to simulate large cellular arrays: Configurable Logic could be an 'enabling' technology allowing easy exploration of the cellular design space.

8.3 Virtual Cells.

It is worth asking whether the allocation of cells to functions can be done dynamically in the same way as memory is assigned in many conventional computers. Ideally designers could work with an infinitely large plane of cells but only 'active' cells would be mapped onto physical units at any given time. There are two ways in which this could be done.

8.3.1 Overlays.

By analogy with the programming technique in which large programs are broken up into smaller sections which can fit in memory large cellular designs could be broken up into smaller ones. It is easy to see ways to break up many algorithms: for example an image processing algorithm could be broken up into a series of transformations applied one after the other. Special cellular designs for individual transformation processors would correspond to 'overlays' and sub-results could be

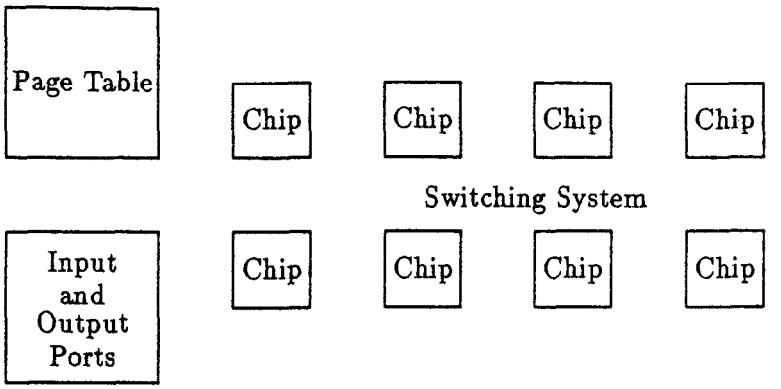


Figure 8–1: Virtual Array Design.

stored on disk in between steps. In some cases this would involve replacing of a pipelined computation with a sequential series of computations. A system which allowed overlays would be a straightforward extension of the present design.

8.3.2 Paging.

In this case the replacement of 'dead' computation units would be done automatically. We will imagine that we have the structure shown in figure 8–1. Each 'page' corresponds to one large chip full of cells, say a 64x64 array in $1\mu m$ technology. We will assume that there are several hundred such chips giving more than a million basic cells. Instead of being connected directly to their neighbours each block edge is joined to a switching system capable of arbitrarily connecting their edges. We assume that computations within the cells are synchronised to a system clock. Let us examine the additional hardware required to 'virtualise' this array allowing a number of subtasks to proceed as if they had an extremely large physical array of arbitrary shape to themselves.

8.3.3 Switching System.

The switching system has two main tasks: to connect non-adjacent chips together to form logical arrays and to connect internal chip edges to external signals. This second function is critical because large arrays will be very poorly utilised if I/O wires have to be routed through intermediate chips till they reach the system edge.

It may appear that the bandwidth required in the switching system is far too great to be feasible because each chip edge has 64 input and output signals switching at unpredictable times. There are three compensating factors, however.

1. Unused Signals. Many of the inputs and outputs will be unused and these can be found and ignored by looking at the configuration information.

2. Clock Frequencies. Because of the delay through the cell's internal switching system the clock frequency of any system implemented using the cells will be lower than that of the external switching system: therefore several cell signals can be sent along one external wire in one cell clock cycle. The clock to the cells can also be stretched to allow all inter-cell communication to complete: clock stretching means that the cell clock does not have to be set according to the worst case delay through the external switching system but only increased by any required additional delay.

3. Change Transmission. In digital circuits it is unlikely that a given output will change every clock cycle. This means that latches can be placed on all inputs and outputs to the chips, using these latches changes in output signals can be detected and only when they occur will communication through the network be necessary.

Nothing has been said so far about the topology of the communication network: because of the number of connection points (4 times the number of cells plus external ports) it will be impossible to use a crosspoint switch or similar structure. Topologies which grow as $O(n \lg n)$ where n is the number of connections such as hypercubes, omega networks etc. would probably be most suitable (see, for example, the discussion of network topologies in [Hillis85]). It would probably be advantageous to treat connections to neighbouring chips as a special case which did not need to use the general switching system.

8.3.4 Extra Cell Hardware.

We will now turn our attention to the additional hardware required within each cell to support virtualisation. Three conditions must be satisfied.

1. The computation must be restartable after swapping. In order to restart the computation we require two conditions: firstly the computation must be synchronised to a system clock - this implies that there will be a safe time to sample internal signals when one can be confident they will not change, secondly one must be able to save the value of all state within the array and all signals on the periphery of the array.

2. Swapping out a section should not affect the rest of the system. This function is provided by the latches mentioned above which reduce the bandwidth requirements of the routing system. This latching also means that other chips receiving inputs from the 'swapped' chip will not be affected since the signals will be latched. The first condition is most easily met by saving the output of all function blocks within the array (just saving function blocks which are emulating latches is not enough because gates can be interconnected to form state): this requires one extra bit of RAM per cell. In this application the global FTEST signal is not useful so we can easily use its space to provide this function. The third condition can also be met: we can monitor each gate output for changes easily since we have its previous value saved to allow computations to be restarted. A single global signal is required to check that no gates within the block have changed: again we can use the FTEST wiring so this will not increase cell size.

3. 'Dead' blocks must be identifiable. If no internal gate outputs change and no external signals change during one clock cycle then the block is 'dead' and can be swapped out. This is implied by determinacy: if nothing has changed in a clock cycle and no inputs change then nothing will change until an input changes - at this point the cell would have to be swapped back in. We can see, therefore, that the switching system must keep a table of

swapped cells and check for their inputs changing: when this happens they must be swapped back in.

8.3.5 Conclusions.

We have seen that it is possible to build a virtualisable array of cells. The major overhead would be the switching system. Such hardware would make the 'algorithm machine' application of cellular logic much more attractive by allowing multiple applications to share the same physical array of cells, supplying arbitrary shapes of cell arrays and allowing applications to be designed for cell arrays much larger than those actually present. These advantages would substantially narrow the usability gap between cellular arrays and Von Neumann machines and separate cellular technology from traditional one algorithm, one job hardware accelerators. However, it is important not to get carried away by the potential of such systems: it is far from clear that they can be implemented efficiently. Bottlenecks in the switching system or paging mechanism could easily remove all the performance advantage. Determination of the potential performance of such systems and trial implementations are promising areas for further research.

8.4 Conclusions.

At this point it seems appropriate to quote the last sentence of Shoup's thesis written 18 years ago [Shoup70]:

> It remains to be seen whether the computer science community and/or the commercial manufacturers will take up the banner of Programmable Logic. Hopefully, this dissertation has provided sufficient justification for further interest in this technique.

The advance of technology over the intervening years has made the arguments for configurable logic much stronger but at the present time there is less interest in

programmable cellular systems than there was twenty years ago. Twenty years ago the basic store and 2:1 multiplexor combination built from logic gates would require about 40 transistors (about 8 gates in a master/slave shift register stage plus another 3 in a 2:1 multiplexor and 3 or 4 transistors per gate): the same structure built from CMOS RAM and pass transistors requires 7 (3 transistors in the RAM, 2 pass transistors and an inverter). The transistors themselves are more than a hundred times smaller. Configurable logic was an idea which arrived before its time: now that its time has come it would be a pity to go on ignoring it.

8.5 Acknowledgments.

John Gray, Genbao Feng and Jouko Viitanen have contributed important ideas to this research, David Rees carefully proofread the first draft of the thesis and made many useful comments. Brian Wylie of the physics department provided some of the reference materials used in the fluid-flow example of chapter 7 and took time to explain the mysteries of fluid-flow modelling to a naïeve computer scientist.

I would also like to acknowledge the support of the Science and Engineering Research Council who gave me three years to follow my own ideas and European Silicon Structures Ltd. who fabricated the prototype CAL chips.

This thesis is dedicated to *BARRA*, a SUN 3/50 workstation, without whose selfless devotion to duty it would never have been completed.

Bibliography

[Akers78] Sheldon B. Akers. *Binary Decision Diagrams.* IEEE Transactions on Computers, C27(6):509–516, June 1978.

[Altera87] Altera Corporation. *ALTERA Data Book 1987.* Altera Corporation, Santa Clara, California, 1987.

[AMD88] Advanced Micro Devices. *PGA 3000 Series Data Sheet.* Advanced Micro Devices, Sunnyvale, California, 1988.

[Brayton86a] R. K. Brayton. *Algorithms for Multi-level Logic Synthesis and Optimization.* NATO Conference on Logic Synthesis and Silicon Compilation for VLSI design, 1986.

[Barbacci78] M.R. Barbacci, G Barnes, R. Cattell, D.P. Siewiorek. *The Symbolic Manipulation of Computer Descriptions: The ISPS Computer Description Language.* Technical Report, Carnegie Mellon University, 1978.

[Borgefors86] G. Borgefors. *On Hierarchical Edge Matching in Digital Images Using Distance Transformations.* Dissertation TRITA-NA-8602, The Royal Institute of Technology, Stockholm, Sweden 1986. (To be published in IEEE PAMI.)

[Brayton86b] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten and J. van Eijndhoven. *The Yorktown Silicon Compiler System.* Technical Report RC 12500, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Feb. 1986.

[Butler78] Jon T. Butler. *Tandem Networks of Universal Cells.* IEEE Transactions on Computers, C27(9):785–799, September 1978.

[Byte87] Theme Issue On Programmable Hardware. BYTE, January 1987.

[Catt78] R. C. Aubusson and I. Catt. *Wafer Scale Integration - A Fault-Tolerant Procedure.* IEEE J. Solid State Circuits, 13:339–344, 1978.

[Cerny79] Eduard Cerny, Daniel Mange and Eduardo Sanchez. *Synthesis of Minimal Binary Decision Trees.* IEEE Transactions on Computers, C28(7):472–482, July 1979.

[Chen82] X. Chen and S.L. Hurst. *A comparison of universal-logic-module realizations and their application in the synthesis of combinatorial and sequential logic networks.* IEEE Transactions on Computers, 31(2):140–147, February 1982.

[Cloqueur87] A. Clouqueur and D d'Humieres *RAP1, a Cellular Automata Machine for Fluid Dynamics.* Complex Systems 1, 1987 pp 584-596.

[Electronics86] *Building Fast SRAMs with no Process 'Tricks'.* Electronics, 81–83, August 7, 1986.

[Fairfield84] R. C. Fairfield, A. Matusevich and J. Plany. *An LSI Digital Encryption Processor (DEP).* Advances in Cryptology, Proc. Crypto 84, Springer Verlag, 1984.

[Finnegan81] J. Finnegan *The VLSI Approach to Computational Complexity.* In H.T Kung, Bob Sproull and Guy Steele, editors, *VLSI Systems and Computations* Proceedings of the Conference on VLSI Systems and Computations held at Carnegie Mellon University, October 1981.

[Fleisher75] H. Fleisher and L. Maissel. *An Introduction to Array Logic.* IBM J. Res. Dev., Vol 19,pp 98–109,March 1975.

[Fourman85] M. P. Fourman. *Redundancy Strategies for WSI.* In C. R. Jesshope and W. Moore, editors, *Wafer Scale Integration*, pages 72–82, September 1985. Proceedings of Workshop on Wafer Scale Integration held at Southampton University.

[Frisch86] U. Frisch, B. Hasslacher, and Y. Pomeau. *Lattice Gas Automata for the Navier-Stokes Equation.* Physical Review Letters, Vol 56 no 14, April 1986 pp 1505–1508.

[Frisch87] Uriel Frisch, Dominique d'Humieres, Brosl Hasslacher, Pierre Lallemand, Yves Pomeau and Jean-Pierre Rivet. *Lattice Gas Hydrodynamics in Two and Three Dimensions.* Complex Systems 1, 1987 pp 646-703.

[Glasser85] Lance A. Glasser and Daniel W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Addison Wesley, 1985.

[Gray88] John Gray. Private Communication.

[Gregory86] David Gregory, Karen Bartlett, Aart de Geus, and Gary Hachtel. *Socrates: A System for Automatically Synthesising and Optimising Combinational Logic.* Proc. 23rd Design Automation Conference, pages 79–85, 1986.

[Hamachi84] Gordon T. Hamachi and John K. Ousterhout. *A Switchbox Router with Obstacle Avoidance.* Proc. 21st Design Automation Conference, pages 173-179, 1984.

[Hamachi85] Gordon Hamachi Walter S. Scott, Robert N. Mayo and John K. Ousterhout (eds.). *1986 VLSI Tools: Still More Works by the Original Artists.* Technical Report UCB/CSD 86/272, University of California at Berkeley, Dept. of Comp. Sci. December 1985.

[Hanninen88] Pekka Hanninen, Jouko Viitanen, Juha Salo and Jarmo Takala. *TAGIPS-multiprocessor for Image and Signal Processing.* Proc.

STeP-88 Finnish Artificial Intelligence Symposium. pp. 37-42. University of Helsinki, August 1988.

[Harper86] Robert Harper, David MacQueen and Robin Milner. *Standard ML.* Technical Report CSR-209-86, University of Edinburgh, Dept. of Computer Science, 1986.

[Hellman79] M. E. Hellman. *DES Will Be Totally Insecure within Ten Years.* IEEE Spectrum, Vol 16:32–39, July 1979.

[Hellman80] M. E. Hellman. *A Cryptanalytic Time-Memory Tradeoff.* IEEE Trans. Inf. Theory, IT26:401–406, July 1980.

[Hikita79] T Hikita and H. Enomoto. *On the number of multivalued switching functions realizable by cascades.* IEEE Transactions on Computers, C28(5):371–374, May 1979.

[Hillis85] W. Daniel Hillis. *The Connection Machine.* PhD Thesis, Massachusetts Institute of Technology, Dept. Artificial Intelligence, 1985.

[Hoornaert84] Frank Hoornaert, Jo Goubert and Yvo Desmedt *Efficient Hardware Implementation of the DES.* Advances in Cryptology, Proc. Crypto 84, Springer Verlag, 1984.

[Hoornaert88] Ingrid Verbauwhede, Frank Hoornaert and Joos Vandewalle *Security and Performance Optimisation of a new DES Data Encryption Chip.* IEEE Jornal of Solid State Circuits, Vol 23:647–656, June 1988.

[Hu72] Sung C. Hu. *Cellular synthesis of synchronous sequential machines.* IEEE Transactions on Computers, C21(12):1399–1405, December 1972.

[Hughes82] J. Gordon Hughes et al. *VLSI Design Tools.* Technical Report, Edinburgh University, 1982.

[Inmos84] INMOS Ltd.. *OCCAM Programming Manual.* Prentice Hall International, 1984.

[Jesshope85] C. R. Jesshope. *Communications in Wafer Scale Systems.* In C. R. Jesshope and W. Moore, editors, Wafer Scale Integration, pages 65–72, September 1985. Proceedings of Workshop on Wafer Scale Integration held at Southampton University.

[Kamabayashi79] Yahiko Kamabayashi. *Logic Design of Programmable Logic Arrays.* IEEE Transactions on Computers, C28(9):609–617, September 1979.

[Kean86] Tom Kean. *User Manual for QV Version 1.0.* Internal Technical Note. Edinburgh University, Computer Science Department, September 1986.

[Kean87] Tom Kean and Genbao Feng. *Configurable Logic: An Approach to the Rapid Implementation of ASIC's.* Technical Report CSR-234-87, University of Edinburgh,Dept. of Computer Science, 1987.

[Kirkpatrick83] S. Kirkpatrick, C.D. Gelatt Jr. and M. P. Vecchi. *Optimization by Simulated Annealing.* Science Vol 220: Number 4598 pp 671–680, 13th May 1983.

[Knaizuk77] J. Knaizuk Jr. and C. Hartmann *An Optimal Algorithm for Testing Stuck-At Faults in Random Access Memories.* IEEE Transactions on Computers, Vol C26(11) pp 1141-1144, November 1977.

[Knuth73] Donald E. Knuth. *The Art of Computer Programming Vol 1: Fundamental Algorithms.* 2nd Edition, Addison Wesley, 1973.

[Konheim81] Alan G. Konheim. *Cryptography, a Primer.* John Wiley and Sons, 1985.

[Lattice86] Lattice Logic Ltd. *Chipsmith: A Random Logic Compiler for Gate Arrays, Optimised Arrays and Standard Cells.* Edinburgh, UK., 1986.

[Lea85a] S. R. Jones and R. M. Lea. *Interconnection Strategies for the WASP Device.* In C. R. Jesshope and W. Moore, editors, *Wafer Scale Integration*, pages 82–91, September 1985. Proceedings of Workshop on Wafer Scale Integration held at Southampton University.

[Lea85b] K. D. Warren, M. B. E. Abdelrazik, R. D. McKirdy and R. M. Lea. *A Power Distribution Strategy for WSI.* In C. R. Jesshope and W. Moore, editors, *Wafer Scale Integration*, pages 54–61, September 1985. Proceedings of Workshop on Wafer Scale Integration held at Southampton University.

[Lee86] Myoung Sung Lee and Gideon Frieder. *Massively Fault-tolerant Cellular Array.* Proc. International Conference on Parallel Processing 1986, pp 343–350.

[Leiserson85] Tom Leighton and Charles E. Leiserson. *Wafer Scale Integration of Systolic Arrays.* IEEE Transactions on Computers, 34(5):448–460, May 1985.

[Lin87] Lin Huimin. *X Windows in SML.* Internal Technical Note. Dept. Comp. Science, Edinburgh University, September 8,1987.

[Lineback86] J. Robert Lineback. *The Fast Static RAM Moves Into the Mainstream.* Electronics, 121–123, August 7, 1986.

[Maitra62] K.K. Maitra. *Cascaded Switching Networks of Two-Input Flexible Cells.* IRE Transactions on Electronic Computers, EC-11:136–143, April 1962.

[Manning77] Frank B. Manning. *An approach to highly integrated computer-maintained cellular arrays.* IEEE Transactions on Computers, C-26(6):536–552, June 1977.

[Mead80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison Wesley, 1980.

[Minnick64] R.C. Minnick. *Cutpoint Celllular Logic.* IEEE Trans. Electron. Comput., vol EC13, pp. 685–698, Dec. 1964.

[Minnick67] R.C. Minnick. *A Survey of Microcellular Research.* J. ACM, 14(2):203–241, April 1967.

[MMI84] Monolithic Memories Inc. *Programmable Logic Handbook.* Monolithic Memories Inc., Santa Clara, California. 1984.

[Mohsen88] Amr Mohsen. *Desktop-Configurable Channeled gate Arrays* VLSI Systems Design, August 1988, pp24–33.

[Moore85] W. R. Moore. *A Critical Review of Fault-tolerant Chips and WSI.* In C. R. Jesshope and W. Moore, editors, *Wafer Scale Integration,* pages 1–8, September 1985. Proceedings of Workshop on Wafer Scale Integration held at Southampton University.

[Morris79] R. Morris and K. Thompson. *Password Security: A Case History.* Comm. ACM, vol. 22, pp. 594-597, November 1979.

[McCluskey85a] Edward J. McCluskey. *Built in Self Test Structures.* IEEE Design and Test of Computers, 29–36, April 1985.

[McCluskey85b] Edward J. McCluskey. *Built in Self Test Techniques.* IEEE Design and Test of Computers, 21–28, April 1985.

[Negrini86] Roberto Negrini, Mariagiovanna Sami and Renato Stefanelli. *Fault Tolerance Techniques for Array Structures Used in Supercomputing.* IEEE Computer, February 1986.

[NBS77] National Bureau of Standards. *Data Encryption Standard.* Fed. Inf. Process. Stand. Publ. 46, January 1977.

[NBS80] National Bureau of Standards. *DES Modes of Operation.* Fed. Inf. Process. Stand. Publ. 81, December 1980.

[Oldfield83] Jose S. Metos and John V. Oldfield. *Binary Decision Diagrams: From Abstract Representations to Physical Implementations.* Proceedings of 20th Design Automation Conference, 567–570,1983.

[Osman72] Mohamed Y. Osman and C. Dennis Weiss. *Universal base functions and modules for realising arbitrary switching functions.* IEEE Transactions on Computers, C21(9):985–995, September 1972.

[Papakonstantinou72] George K. Papakonstantinou. *A Synthesis Method for Cutpoint Cellular Arrays.* IEEE Transactions on Computers, C21(12):1286–1292, December 1972.

[Patil79] Suhas S. Patil and Terry A. Welch. *A Programmable Logic Approach for VLSI.* IEEE Transactions on Computers, C28(9):594–601, September 1979.

[Payne77] Harold J. Payne and William S. Meisel *An Algorithm for Constructing Optimal Binary Decision Trees.* IEEE Transactions on Computers, C26(9):905–916, September 1977.

[Peltzer83] Douglas L. Peltzer. *Wafer Scale Integration: The Limits of VLSI?* VLSI Design,pp 43–47, September 1983.

[Peng88] Z. Peng. *A Formal Methodology for Automated Synthesis of VLSI Systems.* Linkoping Studies in Science and Technology, Dissertation No 170, Dept. of Computer and Information Science, Linkoping University, S-58183 Linkoping, Sweden.

[Preparata71] F.P. Preparata. *On the Design of Universal Boolean Functions.* IEEE Transactions on Computers, 20(4):418–423, April 1971.

[Preston84] K. Preston and M. Duff. *Modern Cellular Automata.* Plenum, New York, 1984.

[Raffel85] J. I. Raffel. *The RVLSI Approach to Wafer Scale Integration.* In C. R. Jesshope and W. Moore, editors, *Wafer Scale Integration*, pages 199–204, September 1985. Proceedings of Workshop on Wafer Scale Integration held at Southampton University.

[Rivest77] Ronald L. Rivest. *The Necessity of Feedback in Minimal Monotone Combinational Circuits.* IEEE Transactions on Computers, C26(6):606–607,June 1977.

[Rivest82] R. L. Rivest and C. M. Fiduccia. *A Greedy Channel Router.* Proc. 19th Design Automation Conference, Las Vegas, 1982.

[Robertson77] Peter S. Robertson. *The IMP-77 Language.* Internal Report CSR-19-77. Dept. of Computer Science, University of Edinburgh 1977 (Revised May 1983).

[Roth67] J. Paul Roth, Willard G. Bouricius, and Peter R. Schneider. *Programmed Algorithms to Compute Tests to Detect and Distinguish between Failures in Logic Circuits.* IEEE Transactions on Electronic Computers, 567–580, October 1967.

[Salem86] James B. Salem and Stephen Wolfram. *Thermodynamics and Hydrodynamics with Cellular Automata.* Theory and Applications of Cellular Automata, Paper 3.10, pp 362–365. World Scientific Publishing Co., Singapore. 1986.

[Savage76] John E. Savage. *The Complexity of Computing.* John Wiley and Sons, 1976.

[Seitz80] Charles L. Seitz. *System Timing.* Chapter 7 of *Introduction to VLSI Systems.* Addison Wesley,1980.

[Shinsha84] T. Shinsha, T. Kubo, M. Hikosaka, K. Akiyama and K. Ishihara. *POLARIS: Polarity Propagation Algorithm for Combinational Logic*

Synthesis Proc. 21st Design Automation Conference, 1984, pp 322–327.

[Shannon49] C. E. Shannon. *The Synthesis of Two-Terminal Switching Circuits.* Bell System Technical Journal, Vol 28,59–98,1949.

[Shimomura87] Tsutomu Shimomura, Gary D. Doolen, Brosl Hasslacher and Castor Fu. *Calculations Using Lattice Gas Techniques.* Los Alamos Science, Special Issue 1987.

[Shoup70] Richard G. Shoup. *Programmable Cellular Logic Arrays.* PhD thesis, Computer Science Dept., Carnegie-Mellon University, March 1970.

[Siewiorek82] Daniel P. Siewiorek Leonard S. Haynes, Richard L. Law and David W. Mizell. *A Survey of Highly Parallel Computing.* Computer, 15(1):47–56, January 1982.

[Sobol83] Robert Sobol. *The Universal Synchronous Machine.* VLSI Design, pp 60–66, November 1983.

[Stapper83] C. H. Stapper. *Modeling of Integrated Circuit Defect Sensitivities.* IBM Journal of Research and Development, Vol. 27:549–557, 1983.

[Stapper84] C. H. Stapper. *Modeling of Defects in Integrated Circuit Photographic Patterns.* IBM Journal of Research and Development, Vol. 28:461–465, 1984.

[Steinvorth85] R. H. Steinvorth, A. S. Bergendahl, B. J. Donlan, G. F. Taylor and J. F. McDonald. *Wafer scale integration using discretionary microtransmission line interconnections.* In C. R. Jesshope and W. Moore, editors, *Wafer Scale Integration*, pages 31–45, September 1985. Proceedings of Workshop on Wafer Scale Integration held at Southampton University.

[Tanenbaum81] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall International, 1981.

[Terman87] C. Terman (modified by University of Washington). *RNL 4.2 User's Guide.* In *VLSI Design Tools Reference Manual*, Technical Report 87-02-01, Northwest Laboratory for Integrated Systems, Univ. Washington. Feb. 1987.

[Thayse81] Andre Thayse. *P-Functions: A New Tool for the Analysis and Synthesis of Binary Programs.* IEEE Transactions on Computers, C30(2):126–134, February 1981.

[Thomas83] D. Thomas, C. Hitchcock III, T. Kowalski, J. Rajan and R. Walker. *Automatic Data Path Synthesis.* IEEE Computer, December 1983, pp 59–79.

[Toffoli87] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines.* MIT Press 1985.

[Turnbull85] J. R. Turnbull. *An Analysis of the Interconnection Problem for WSI.* In C. R. Jesshope and W. Moore, editors, *Wafer Scale Integration*, pages 24–31, September 1985. Proceedings of Workshop on Wafer Scale Integration held at Southampton University.

[Valiant81] Leslie G. Valiant. *Universality Considerations in VLSI Circuits.* IEEE Transactions on Computers, 30(2):135–140, February 1981.

[Viitanen87] J. Viitanen, P. Hanninen, R. Saarela, and J. Saarinen. *Hierarchical pattern matching with an efficient method for estimating rotations.* Proc. of the IEEE Industrial Electronics Society conference IECON'87. Cambridge, Massachusetts, November 1987.

[Viitanen88a] Jouko Viitanen, Pekka Hanninen, Reima Saarela and Jukka Saarinen *An Efficient Method for Image Pattern Matching* Proc. International Conference on Parallel Processing for Computer Vision and Display, Leeds UK, January 1988.

[Viitanen88b] Jouko Viitanen and Tom Kean. *Image Pattern Matching Using Run-Time Reconfigurable Processor Architecture.* Prepared for submission to Proc. International Conference on Parallel Processing for Computer Vision and Display, 1989.

[Vladimirescu87] A. Vladimirescu, Kaihe Zhang, A.R. Newton, D.O. Pederson and A.Sangiovanni-Vincentelli. *SPICE User's Guide.* In *VLSI Design Tools Reference Manual*, Technical Report 87-02-01, Northwest Laboratory for Integrated Systems, Univ. Washington. Feb. 1987.

[Voith77] Raymond P. Voith. *ULM Implicants for Minimization of Universal Logic Module Circuits.* IEEE Transactions on Computers, C26(5):417–424, May 1977.

[VonNeumann66] John Von Neumann (completed by A.W. Burks) *Theory of Self-Reproducing Cellular Automata.* University of Illinois Press 1966.

[Wahlstrom67] S. E. Wahlstrom. *Programmable Logic Arrays.* Electronics, Vol. 40(25):90–95, Dec. 11, 1967.

[Wardle84] C.L. Wardle, C.R. Watson, C.A. Wilson, J.C. Mudge, and B.J. Nelson. *A Declarative Design Approach for Combining Macrocells by Directed Placement and Constructive Routing.* Proceedings 21st Design Automation Conference, pages 594–601, ACM-IEEE, 1984.

[Wayner88] Peter Wayner. *Modelling Chaos.* Byte, May 1988, pp 253–258.

[Weste85] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison Wesley, 1985.

[Wolfram86] Stephen Wolfram. *Theory and Applications of Cellular Automata.* World Scientific Publishing Co, Singapore. 1986.

[Wood79] Roy A. Wood. *A High Density Programmable Logic Array Chip.* IEEE Transactions on Computers, C28(9):602–608, September 1979.

[Xilinx86] Xilinx Inc.. *The Programmable Gate Array Design Handbook.* San Jose, CA., 1986.

[Xiong86] J.G. Xiong. *Algorithms for Global Routing.* Proceedings of the 23rd Design Automation Conference, 1986.