Architectural Soup:
A proposed very general purpose computer

Ian Weaver

Ph.D. Thesis

University of Newcastle upon Tyne

September 1989

ABSTRACT

This thesis is concerned with architecture for long term general purpose computers. The work is based on current trends in machine architecture and technology. Projections from these generated "Architectural Soups". An Architectural Soup has the potential to emulate many different machine architectures. The characteristics of this class of machine are, three dimensional, simple cells and a simple communications topology, which can be reconfigured at a very low level. This thesis aims to show potential usefulness and viability of machines with such capability.

Methods of programming are considered, and important design issues are investigated. A specific implementation architecture is described and illustrated through simulation. An assessment is made of the architecture and of the simulator used. In addition, the implementation architecture is used as the basis for a VLSI design, which shows the simplicity of a Soup cell, and provides estimates of the possible number of cells in future machines.

ACKNOWLEDGEMENTS

Although this work is declared as mine, I am indebted to many people for their assistance and opinion. I particularly wish to acknowledge the considerable help and useful comments received from my supervisor, Richard Hopkins, and the members of my supervisory committee, Martin McLauchlan and Brian Randell.

I also wish to acknowledge support which has indirectly assisted, from Mike Brudenell, Shirley Craig, John Clowes, Pete Lee, Ian Scoins, all at BNFL, Andy, Bart, my family and Tree.

Funding was provided by the Science and Engineering Research Council.

INDEX

Chapter 1: Introduction

1.1 INTRODUCTION

Any computer is ultimately bound by its physical size, the speed of light, the properties of the technology and the construction methods used to build it. Any other restrictions that a computer may have are due to the imposition of man-made architectural decisions.

Broadly speaking there is a spectrum of computer architecture, ranging between general purpose and dedicated purpose. The former is intended to be applicable to many different types of problem, whereas the latter is optimised for a specific class or classes of tasks. Dedicated purpose computers are usually easier to design as the use for which they are intended can be rigorously defined. This is not the case with general purpose machines. A key concept in the generality of an architecture is the notion of its efficiency. The dedicated purpose computer is poor for general purpose use as its optimisation will normally prevent efficient solution of tasks other than those for which it was intended. Similarly, a general purpose computer is unlikely to be the optimum for any one particular task as it has not been optimised towards the task. The criterion for the 'ultimate' general purpose architecture, therefore, is the ability to emulate ALL possible dedicated purpose machines as EFFICIENTLY as possible.

There are often many approaches to solving the same problem. To give a specific example, in the area of image processing [ROS82] examines methods of utilising cellular computers for parallel processing of images at the region level. He expresses the need for a cellular computer with the ability to dynamically re-configure itself, in parallel. An alternative approach is taken by [FOU85], who illustrates the applicability of three-dimensional MIMD controlled processor structures such as pyramids and cubes. This is a different algorithm for the same problem. A general purpose machine should be good for both approaches, not biased in favour of either.

This chapter will examine current architecture and technology. Based on this, the suggestion will be made that a method by which a general purpose machine can perform such emulation, without favouritism, is to be reconfigurable at a very low level. The remainder of this thesis is concerned with detailed examination of low level reconfigurability.

1.2 VON NEUMANN AND THE NEED FOR PARALLELISM

A consequence of the Von Neumann architecture is that the small amount of silicon devoted to processing, usually only 2-3%, is kept very busy, whilst silicon devoted to memory is relatively idle [HIL85]. As machines get larger more opportunity will be lost due to the poor use made of the majority of silicon. Even in the higher utilised processing elements there is inefficient use of transistors. This wasted capacity can make machines with simple instruction sets faster than machines with comprehensive ones (see reduced instruction set computers [KAT84]).

The Von Neumann architecture is a poor candidate for the 'ultimate architecture' due to the Von Neumann bottleneck. Methods used to speed up Von Neumann machines include caching, pipelining, vector processing, and the interconnection of small numbers of machines. These computers ARE faster, but hard to program well. On supercomputer architecture [DEN80] writes "Current architectures require intricate programming to obtain a fraction of their potential." Usually, the effective performance of supercomputers ranges between only 5 and 25 percent of peak performance [HWA87].

Hardware is inherently parallel [GOL85] so the available resource will be under-utilised as a consequence of imposing a sequential architecture upon it. Also, to efficiently emulate any particular dedicated purpose circuit, the emulator must be parallel. Consider neural computers. Here there is a requirement for large numbers of switches where each switch only performs a simple operation in order to alternate between two states [ABU87]. In this case a high degree of parallelism and inter-connectivity seem of more importance than the speed of the individual switching actions. It is likely that future machines will be highly parallel [MEA80]. The present trend is pushing the granularity of parallelism down [HWA87].

To illustrate trade-offs between sequential and parallel architectures consider two computers, both efficiently implemented, using similar technology and the same amount of silicon real-estate. One machine is Von Neumann architecture and the other is highly parallel with simple processing elements. The Von Neumann machine will be appropriate for sequential algorithms. However, it is unlikely to be as appropriate as the parallel machine for highly parallel tasks requiring simple processing operations. The efficiency of a parallel machine for parallel tasks will depend on how well the tasks map on to the machines particular architecture. For sequential tasks the parallel machine is unlikely to be as efficient. The simplicity of each processing element will make execution of the task on an individual element complex. The sequentiality of the task will make efficient utilisation of the majority of the processing elements difficult, and if feasible at all, the speed of solution will be restricted by the communications overhead between the elements.

Arguably, any computer which has a fixed architecture must be biased in favour of the algorithms which map well on to it. (Thus no computer is truly general purpose.) The more dynamic the architecture the more general it will be, provided any overheads incurred in control and communication are not large. In order to have an architecture which is good for both parallel and serial problems, I propose examination of a machine with the simplest possible underlying architecture. This could be dynamically programmed to emulate other architectures. The thesis aims to show the viability of the principle of such machines. From the outset it is clear that control and communication overheads could be large, so such machines might never be of practical use. The thesis will examine these overheads and illustrate some trade-off benefits which can be incurred as a result of high flexibility. This thesis also aims to give insight into the capability and flexibility possible in future general purpose machines. It is hoped that such examination will give insight into future design considerations.

1.3 TRENDS IN ARCHITECTURE

Current trends in architecture were examined, in particular, machines which were not Von Neumann architecture. Distinction was made between technology improvement and architectural improvement. These factors are often related. On a basic level, improved technology can permit more complex architectures, an example being a decrease in feature size, permitting faster components and higher density. An example of architectural improvement is parallel as opposed to sequential. Technological improvements can be estimated long term by examining the theoretical capabilities of materials and assuming all production difficulties will be overcome. (For a discussion of the maximum theoretical capabilities of silicon see [MEA80].) Architectural innovation is harder to predict, but by combining the assumptions of capability of future technology and examining the trends in current architecture it should be possible to obtain indications of the architecture of future machines.

There are many alternatives to pure Von Neumann architecture, for example, Illiac IV [SLO71], Burroughs Scientific Processor [STO77], Cray-1 [BAS77] [KOZ80], Massively Parallel Processor [FUN77] [BAT80], Cyber 205 [KOZ80], Apiary Network Architecture [HEW80], Systolic Arrays [MEA80] [MOO86], Heterogeneous Element Processor [SMI81], NYU Ultracomputer [GOT82], Configurable Highly Parallel computer [SNY82], data-flow and reduction machines [DEN80] [TRE82], Cray X-MP-2 [CHE83], Erlangen Multiprocessor System [FRI83], Semantic Network Array Processor [MOL84], Reduced Instruction Set Computers [KAT84], Geometric Arithmetic Parallel Processor [DAV84], the Connection Machine [HIL85] [KUN86] [HIL87], INMOS Transputer [INM85], CLIP [FOU85], Hypercubes [KUN86], and the Encore Multimax [ENC86] [LEE87]. Much of the following work is based on an examination of these architectures and on a recent survey reviewing the 'state of the art' in supercomputer architecture ([HWA87]).

Generally speaking there are two approaches to parallelism. Firstly, to have a small number (O(10^1)) of very powerful processors, as in the Cray, and secondly, to have a large number (O(10^2) - O(10^5)) of simple processors, as in the Connection Machine or array processors. In both of these approaches there is a high degree of regularity in processing units, the individual processors within a machine usually have similar (if not identical) function and a similar communications topology. [HEW80] identifies this as a need for both homogeneity and isotropy in an architecture in order to simplify software so that complicated optimisations do not need to be performed. (Homogeneity being each processing unit appearing the same in order that there are no distinguished locations. For isotropy there are no distinguished directions.) Considering the first approach of small numbers of powerful processors, closer examination reveals more similarity with the second approach. The powerful processors are typically speeded up using pipelines and vector processing units. If each section in a pipeline and each element of a vector processor is counted as a separate processor then each of these large processing elements can be considered to be a number of smaller elements grouped together. A difference between these two approaches to parallelism is in the degree of regularity. The systems which comprise of large numbers of simple processors are more regular.

Five basic architectural trends can be identified amongst multiprocessor systems which have large numbers of simple processing elements. Firstly the number of processing elements is increasing. Secondly the processors have progressively simpler functions. Thirdly processors are tending towards a multiple instruction stream and multiple data stream (MIMD) architecture ([AND87]) with each processor capable of processing instructions and data independently of other processors in the machine. There are designs now for MIMD parallel machines comprising of thousands of autonomous processing elements such as the NYU Ultracomputer [GOT82]. Fourthly, having a larger number of processing elements is enforcing a simple regular communications topology between elements. The trend is towards a near neighbour mesh, although there are practical restrictions as the building of a large mesh will generate complex interconnection wiring if many circuit boards are required. Alternative interprocessor communication schemes in supercomputers can be seen in [HWA87], and a taxonomy of schemes is given in [LIP87]. The near-neighbour mesh is particularly popular in Wafer Scale Integration for the interconnection of physically adjacent processors, for example work see [JES86]. The mesh permits exploitation of locality, but data movement is often limited to adjacent processing elements [SNY82]. One consequence is that distribution of data to a large number of elements may require significant time. The last trend identified is that memory is becoming more incorporated into each individual processing element. This is because communications bottlenecks are common when using a large global memory. In a typical computer more than 90 percent of silicon is devoted to memory [HIL87]. Distribution of the addressing of this memory amongst a number of processors permits faster access and a higher degree of parallelism. Doubling the amount of silicon afforded to the processors does not significantly effect the total amount of silicon required for the machine since the percentage currently afforded to processors is small in comparison with the percentage afforded to memory. It should be possible for future machines to have an increased degree of processing power without significantly effecting memory capacity.

An additional issue highlighted through examination of architecture, concerns the difficulty of designing optimal circuits. Switching theory provides formal methods for minimising the number of gates required. Unfortunately this does not necessarily give best solutions as the topological properties of the circuit interconnections must also be considered. Designs requiring more gates but having simpler more regular interconnection topology are often faster [MEA80]. "As switching components become smaller and less expensive, we begin to notice that most of our costs are in wires, most of our space is filled by wires, and most of our time is spent transmitting from one end of the wire to the other." [HIL85]. Even in the simple communication topology of the NYU ultracomputer most of the machine volume is occupied by the communications network and its assembly is seen as being the dominant cost [GOT82]. One of the problems in the architecture of parallel computers is in the choice of processor communication scheme [RUZ86]. For example, the "workers" in an "Apiary" [HEW80] do not share any physical memory but communicate by packet-switching message passing. Passing messages of arbitrary sizes and performing complex operations on these messages demand powerful node processors [HWA87]. For some tasks the penalty paid for such large processors may be prohibitive. However, flexibility is needed in the communications network in order to enable formation of a large number of processing elements in a pattern best suited to the problem at hand [HIL87]. It will be difficult to obtain such flexibility with simpler communication processors.

Some issues in the architecture of the processors, memory and communications schemes have been illustrated. The section which follows will investigate future technology to identify forms of architecture which may be favoured.

1.4 TECHNOLOGICAL PROSPECTS

New strategies for interconnection and architecture will have to be devised to cope with the problems of size reduction [BAT88]. "In the foreseeable future, short-channel effects and the non-statistical behaviour of devices spanning only a few tens or hundreds of silicon atoms will require a drastic rethinking of the basic device technology underlying most of present-day circuit integration." [NUD85]. Trends in technology suggest future machines will be much larger in terms of numbers of active devices possible. The complexity of the DESIGN problem will necessitate machines being constructed in a highly repetitive manner from very simple atomic structures [BER85]. Uniformity is of major importance in VLSI [SNY82]. The difficulty is that "seen from the software user, these same machines must show a different face: they are dynamic information structures which can be arbitrarily modified in order to adapt them to the immense diversity of sequential and parallel software" [BER85]. The architecture therefore needs to be simple and repetitive but capable of performing complex tasks.

The component density permissible on a chip is increasing but fundamental limits are being approached [NUD85]. Perhaps within a decade reductions in feature size will run up against the limits of circuit technology [BAT88]. Once the limit has been reached then increasing the number of components on a chip will require that the chips are physically larger. Research in this area includes work on Wafer Scale Integration [JES86] and three-dimensional computers such as [PRE83] [GRI84] [NUD85]. An attempt at a three-dimensional computer can be seen in [NUD85] where silicon wafers are stacked on each other. Although connectivity between the wafers is poor this research suggests that forms of three dimensional technology will be available within the foreseeable future. [NUD85] predicted that it would be possible to produce a working version of his three-dimensional cellular machine by approximately 1990. Chip layering of 20-30 layers is apparently feasible with existing technology [GRI84] [NUD85] [POR87]. The anticipated benefits of three-dimensional silicon circuits are, easier more systematic wire routing, shorter wires, and as a consequence more compact circuits [ROS83]. ([ROS83] demonstrates dramatic efficiency improvements in circuit realisations over those attainable in two dimensions.)

An important manufacturing characteristic of cellular three-dimensional computers is the potential for low cost fabrication. The construction of conventional computers involves many processes that are time-serial. Assembly of cellular three-dimensional machines could be more parallel [GRI84]. There are still many problems to be surmounted, such as power distribution (heat) and the creation of an apparently fault-free medium, but the trends suggest that these problems can eventually be overcome for general purpose architecture. The design of large DEDICATED three-dimensional structures will be more complex than current VLSI design, in order to take advantage of three dimensional connectivity, because dedicated architectures are likely to be irregular. Even with current design tools, the production of a two-dimensional chip is an extremely time consuming process due to the complexity of the electrical factors. This design complexity again suggests the need for simple regular architecture.

1.5 SOME MOTIVATIONS FOR LONG TERM GENERAL PURPOSE ARCHITECTURE

It has been suggested that future technology is likely to be three-dimensional and that general purpose architectures imposed upon this technology will be parallel. Trends in architecture, compounded with the three-dimensional construction complications, suggest that the machine will have a large number of simple processing elements. Each element will contain its own small memory and will only be capable of direct communication with immediate neighbours. Software and hardware arguments have suggested the system needs to be simple and regular in order to facilitate design. Increased technological complications suggest that the design of any irregular architecture, such as a dedicated architecture for such technology, could be significantly more complex.

It was seen in section 1.2 that flexibility is required in the size of the architecture's processing elements in order to obtain high utilisation. Flexibility is required in memory organisation to permit parallel access, if required, and fast sequential access when parallel access is not. Flexibility is required in the communications network in order to permit reconfiguration of the memory and processing elements towards the task in hand, however, the elements of the communications system should be simple. The reconfiguration control of a large communications system consisting of many small communications elements will be a complex task. This will be augmented, because only simple processing and memory elements are available within the machine for such control. In the event of a machine being designed which has the flexibility suggested above, the ability to control the architecture will therefore be of crucial importance.

1.5.1 A programmable array of nand gates

It is theoretically possible to design a machine comprising a regular programmable array of NAND gates. Conceptually, ignoring temporal issues, the nand gates can be organised to make any functional circuit required. The processing element of a machine can be provided through arbitrary interconnection of two input nand gates which can generate all of the Boolean functions of two variables [GIB83]. A simple memory element can be created via the interconnection of two nand gates to form an SR flip-flop [GIB83]. Communication can be provided by chains of gates with the unconnected input held high to make gates act as inverters.

By programming an array of gates it should be possible to create processing elements, memory and communication paths where they are required, to create architectures which fit the application algorithms. The problem with this solution is that there are still many design decisions to be made. For example, the size (numbers of inputs and outputs) and interconnection topology of the individual nand gates will influence the interconnection possibilities, and hence favour different architectures. The control of the interconnection of the gates will be of paramount importance. There are many possibilities. For example, the architecture could be programmed via a mesh of busses which are externally programmed in a similar manner to the way memory is addressed, with connections being made or broken depending on the conditions at the bus intersections. Long busses will be slow. An alternative is that topologies could permit a cell to be programmed by examining the state of the cells surrounding it, raising the possibility of internal reprogramming. A penalty is that cell complexity would increase in order to perform this examination. The benefits of internal reprogramming and local communication must be considered against the simplicity of a global bus system.

The notion of a mass of programmable nand gates can be used to conceptualise the possibilities of a machine with limited architectural restriction. The implementation possibilities are large and there are still many architectural decisions which need to be made. This thesis will examine areas requiring such decisions.

1.6 THE AIMS AND STRUCTURE OF THIS THESIS

This chapter has examined trends in computer architecture. It has made the distinction between dedicated purpose and general purpose machines, problems with current architectures have been stated, and long term predictions for future computers made. This thesis will examine some possible architectures for general purpose computers in the very long term based on these predictions.

Chapter 2 proposes a type of parallel architecture based on the projections made in chapter 1. Components of the architecture are, simple, homogeneous and general purpose. The possible capabilities of such machines will be discussed and similarities with some existing ideas and architectures are identified.

Chapter 3 details implementation issues. A broad set of topics are examined, as factors which now require little consideration will become more important in future architecture. A consequence of this broadness is that individual topics are not considered in depth.

Chapter 4 discusses a specific design, meeting the considerations discussed in the previous chapters. The machine discussed is symmetrical in three-dimensions, hence, cannot be built with current technology.

Chapter 5 illustrates this design through simulation. Experience with programming and simulation of a three-dimensional computer and directions for further experimental work is given.

Chapter 6 presents a VLSI design for one processing element of the architecture illustrated, to show the design's simplicity, and the simplicity of the architecture. The design is then utilised to obtain a projection for the possible number of processing elements in a future realisation.

The final chapter presents a summary and makes concluding remarks on the validity of work done, with direction for further work.

Chapter 2: Architectural Soup

2.1 INTRODUCTION

Computational problems frequently have many methods of solution. For any rigorously specified problem that may be solved by computer there will be optimal electronic circuits. The closer the architecture of a computer to a 'minimum solution circuit' the faster the solution can be found. 'Architectural Soup' is a proposed type of computer with the capability to be reconfigured at a very low level to, it is hoped, efficiently emulate such circuits. (The term Architectural Soup is derived from the myriad of potential architectures available from within the machine.)

When considered at a low level, computers consist of processing elements, memory and wires. Consider a machine with simple cells, which may be dynamically reconfigured to provide these components as required. Such a machine should be able to imitate the function of any feasible computer, however, there will be differences in the speed of individual components and in the size limitation of the machine.

This chapter will define Architectural Soups. The architecture of such machines will be shown to be an extension of the trends identified in chapter 1. This is followed by an examination of the possible capabilities and limitations of such machines should fabrication become feasible.

2.2 ARCHITECTURAL SOUP

Architectural Soup is a class of architectures as opposed to any single architecture. There are many possibilities for machines with simple processing elements which have the capability to emulate other machines. In order to define the class, a basis architecture will be defined. Membership of the class is determined by assessment of a candidate machines ability to emulate this basis architecture.

2.2.1 A basis architecture

Consider a three dimensional machine consisting of a very large number of simple cells. For example 100x100x100 cells is small. (Estimates for the total number of cells in a future machine will be made in chapter 6.) The cells must exhibit the properties of homogeneity and isotropy in that they have identical function and that orientation is of no concern. (Discontinuity problems at the surface of the machine will be examined in Section 3.4 .) The technology of the machine must be sufficiently stable and the architecture be such that the actions of all cells are controllable and deterministic so that repetition of the same operation from the same starting state will yield the same result. The basis is not defined as having a global clock to aid in these control issues as Soup cells could also function asynchronously. (Synchronisation issues will be considered in Section 3.6.)

Cells in this basis architecture are cubic. Each has six immediate neighbours, left, right, back, front, up and down. Up and down are included in order to utilise three dimensional space, as opposed to two dimensional. Cells are abutted in a three dimensional complete near-neighbour mesh. Communication is with immediate neighbours only, via a one bit wide input and a one bit wide output to each abutting neighbour.

The functionality of cells is provided by programming the required primitives between the six inputs and six outputs of a cell. The method by which this programming could be performed will be discussed later in this chapter. The functional primitives required in each cell are a wire, a one bit memory and a simple processing element. The wire - It should be possible to program the interconnection between ANY input to and ANY output from, the cell. An additional desirable characteristic of the wire function is that there should be no noticeable degradation over a path through the machine which consists of many such wire functions. The memory - A memory primitive has two inputs and one output. When the control input is signaled, the value on the data input is stored in the memory. The processing element - This can be provided by a two input nand gate, two input nand gates can be interconnected in order to provide all the possible functions of two Boolean variables [PRO87] and hence all functions of N Boolean variables. In order for the machine to be isotropic, it must be possible to place nand gates, memory elements and wires between any inputs and outputs of a Soup cell. In addition to these functional primitives, the Boolean constants 1 and 0 are required. It must therefore be possible to program any output of a cell to generate these constants.

2.2.2 Simulation and Emulation

Many machines will be capable of simulating the basis Architectural Soup but in order to emulate it they will require simple cells and a simple communication topology. Based on the dictionary definition [OED33] the difference between a simulation and an emulation, in this case, is that an emulation is much closer to the true speed of the target architecture. An emulator would require a very direct correspondence to the elements and connections in the basis. It will require large numbers of simple processing elements, a highly distributed memory, and the capacity for fast local communication. Consider a simulation of the functionality of the basis machine on current Von Neumann machines. This would not be a Soup as the quality of emulation would be poor. Consider a machine identical to the basis, with the exception that each cell has only one nand gate which is positioned between left and right inputs and the front output. Such a machine would be able to emulate the basis architecture by programming a group of these cells to act similarly to one basis cell. This provides routing such that nand gates 'appear' to have the ability to be connected in every feasible orientation. As the number of cells in such a group will be small this architecture would consequently be an Architectural Soup.

2.2.3 Why this architecture is proposed

Taking into consideration the requirements and trends established in chapter 1, the first architectural requirement of a Soup is that the number of processing elements must be large. This should be fulfilled as a consequence of the simplicity of the Soup's cells and communication topology. It should be remembered that the simplest cells do not necessarily give the most space efficient architecture, due to the amount of control circuitry required. Cells which have more complex function but require less control may permit a denser packing. Cells with functions simpler than memory, nand gates and wires could be considered, for example, the interconnection of several cells to form a single nand gate. A possible benefit of this would be to achieve higher cell density. Control of such an architecture, however, is likely to require more control circuitry, and be a prohibitively complex task.

A second trend identified was that a machines individual cells have very simple functions. The cells in a Soup have simplicity which, in fact, renders them incapable of meaningful computation on their own. With such simplicity there would be little purpose in every cell of the machine performing the same function at the same time. It is suggested that such a machine must require both a multiple instruction stream and a multiple data stream, that is to say the third trend.

The fourth trend suggests that the communication system in the basis has a near-neighbour mesh topology. By Soup definition, any machine capable of emulating the basis will require a topology close to the near-neighbour mesh, otherwise the quality of emulation will be poor. (Due to the simplicity of the cells and topology, circuit switching techniques will likely be used as opposed to packet switching.)

The final trend identified was that of memory being incorporated with each processing element. In the above architecture there is a one bit memory for each one bit nand gate processing element. This represents maximum memory/processor interleaving.

This style of architecture appears to be an appropriate candidate for three-dimensional technology. Cells are simple and identical so permitting a simple design. For example, a small area consisting of a few cells of the machine can be designed in detail and used to make generalisations for a larger machine based on simple repetition. As the active devices are in a regular lattice, modelling of this architecture is simplified. One of the major problems with three dimensional machines is heat generation. The ability to model heat generated in a lattice of cells will simplify this complex aspect of design. Considering manufacturing characteristics, it has been mentioned in section 1.4 that manufacture of a three-dimensional machine could be simplified by using parallelism in the manufacturing processes. A regular array of simple processing elements will make the Soup a candidate for such parallelism.

The flexibility aspects, identified in chapter 1, must also be considered as motivations for general purpose architecture. The simplicity of the Soup's cells are such that the machine can be programmed so that processing elements, memory and wire, are configured as required. This provides the potential for flexibility in processing, in the memory and also in communication. An important consequence of this is that, theoretically, a machine with the above capabilities could be efficient at executing both parallel and serial algorithms. As the architecture is highly parallel, it should be possible to reconfigure the machine in order to emulate a variety of parallel architectures. This could include any future alternative machine architectures which follow the trends outlined. In addition, due to the simplicity of the cells, it should still be possible to reconfigure the machine to emulate serial architectures. Penalties for this flexibility will be, the amount of control hardware required, the overhead of any unused elements within each cell when it is programmed, and the delay incurred in programming the control circuitry.

2.3 PROGRAMMING CONCERNS

It has been shown that an Architectural Soup follows projections of machine trends. Consideration will now be given to the means by which such a machine could be programmed. It is proposed that it will be possible to program at this low level because machine trends imply this style of architecture. The programming of the architecture can be viewed from two standpoints, firstly as being at a level just above that of silicon chip design, and secondly at a level below machine code programming.

Consider the view of programming being similar to silicon design. It can be argued that we are not yet highly skilled at programming in this manner. However, programmers of the Soup will be less concerned with electrical factors, such as power distribution, than a silicon designer, as these will have been considered before the machine was manufactured. Programming a Soup can be viewed as nearer semi-custom VLSI design than full-custom. (For a description of semi-custom design see [HIC83].) A two dimensional realisation of a Soup could perhaps be conceptualised as a surface of gate-array chips which have a dynamically programmable metal layer.

[GOL85] states that integrated circuit design is currently in a phase similar to that of software design thirty years ago when the first compilers became available. The target domain for silicon compilation is complex, but as the possible number of features available on a chip increases, designers are more concerned with aspects of size than of obtaining high utilisation in the previously limited area of silicon real-estate. As a consequence the tools used by silicon designers are now tending towards software methods [DAL84]. A silicon compiler described by [KEL85] can automatically generate self-timed circuits from a behavioural description. (A behavioural description specifies the input/output mapping, as opposed to the explicit physical structure of the architecture.) Algorithms are viewed by the compiler as having two dimensions , which are mapped directly onto the two dimensions available on planar chips. One dimension is used for the flow of data through the algorithm, the other for parallelism. This suggests that it should be possible to produce a compiler for Soup architecture, especially given the increased flexibility of a third dimension.

Alternatively, the programming of a Soup could be viewed as programming at a level below machine code. Considering this approach, programmers are no longer concerned with writing programs at the machine code level. It is assumed, in the majority of cases, that compilers will generate efficient code. For a suitable machine code it should be possible to generate circuits which implement equivalent function. A compiler could then specify which circuit primitives were required as opposed to producing a machine code. A problem of such a system will be synchronisation. Timing difficulties exceed all other design problems by "an order of magnitude" [KOE86]. [KOE86] goes on to state that the self-timed approach is capable of tackling not only the problems of timing, but also design complexity and testing. A penalty of the approach is increased circuitry.

A Soup can be seen to be more software dependent when compared to existing machines since the underlying hardware has, effectively, had less structure imposed upon it. It follows that some of the properties sought in hardware design will also become sought in software design. For example, simple regular structures may be preferred in circuit generation as repetition is likely to be simple. A speculation is that compilers may become more concerned with environmental issues, such as heat generation. Heat in silicon is primarily generated from the active devices as opposed to wires [ROS83]. Synchronisation protocols may, therefore, have an additional requirement of slowing down circuits switching frequently, so as to prevent excessive heat production.

Considering both standpoints, programming a Soup in some form does appear feasible. This will be at a level in between that of machine code and silicon design.

Methods of programming will now be considered. The manner by which a machine is to be programmed will influence the manner in which it is controlled. The following discussion is simplified, further issues will be considered in chapters 3, 4 and 5.

2.3.1 Machines with no dynamic reprogramming capacity

Consider first a machine which can only be programmed when switched on. This type of machine has no capacity for dynamic reprogramming of its architecture. Assuming the machine is in a receptive starting state, it is possible to consider specifying an architecture by using a high level language similar in form to a VLSI description. This description can be algorithmically decomposed to generate the necessary programs for each individual cell of the Soup. Once the Soup has been programmed, the users perform applications programming on the emulated machine. This architecture has similarities with the properties of static microcode, that is, remaining the same for a period of time but having capacity to change if required. (Microprogramming is used to implement control of a processor in a "systematic and flexible" manner [HAY78]. It is intended to permit tailoring of a computer to a particular problem, or type of problem, by providing a closer interface to the underlying architecture [ECK79]. It is argued that the extra delay from the circuitry required to implement the microcode is offset by the improvements in speed gained from achieving a better utilisation of the underlying resources [ECK79].) A benefit of this capability in Soup architecture is that a general purpose off-the-shelf hardware unit can be purchased and tailored, using software methods, towards the particular environment in which it is to be used.

2.3.2 Machines with dynamic reprogramming capacity

If a Soup has the capability for its architecture to be changed dynamically, there is the possibility for individual applications or programs selecting a favoured architecture from a set of standard architectures, which can be loaded as required. The user programming language could permit specification of the required architecture or give hints as to which architecture might be preferred, leaving the final choice to the compiler. The compiler could then generate code optimised towards the chosen architecture. If a Soup permits more rapid dynamic change then changing architectures during execution could be considered. This would benefit tasks where the amount of parallelism varies depending on previous results. The complexity of performing such transformations is likely to be large, requiring an overhead of circuitry to detect when the change is required. A benefit would be that the Soup architecture need show little favouritism towards individual application languages as it could restructure itself in a short period of time to favour others. This is similar to the properties of a dynamically changeable microcode. For example, a microcode capable of changing its function during execution depending on how the algorithm is executing. The speed benefits need to be greater than the complexity of the programming task and the speed at which the microcode can be changed. Little attention has been paid to microcode with such dynamic properties to-date. Such properties will be more complex to control. Also, microcode has been predominately used on Von Neumann architecture machines. As the architecture of the underlying machine remains essentially Von Neumann there is little benefit to be gained from dynamic change. An important difference gained by use of a Soup is that a much higher percentage of the architecture could be changed so such dynamic flexibility may be beneficial.

For the programming methods discussed, there is a trade-off between time spent generating and loading the initial architecture and time spent running the application on it. For example, an algorithm can be implemented as a regular lattice of identical processing elements or alternatively as an irregular lattice of several different elements. The initialisation time saved by regular design could offset any benefit from the irregular case being closer to a minimum solution circuit.

A particular issue which arises from an ability to change architecture is that, historically, there has been a need for software developed for old computers to be supported by new models. Given that the architecture of a machine could be specified and compiled into programs for the Soup, the Soup would then be able to emulate any architecture (if it had appropriate peripherals) and hence execute old software, although there will likely be considerable speed reductions. This is called upward compatibility. As there is the physical three-dimensional restriction on the building of any future computer and a Soup has such low level emulation capacity, it would be able to emulate any FUTURE machine architecture (including other Soups). It will therefore have the ability to execute software for FUTURE machines for silicon style technology. I term this anomaly 'downward compatibility. Obvious problems would arise in the possible size and speed of the machine but previously it has been more important that software can be executed (in acceptable time) rather than that it executes well. Software would gain in portability since it could execute on more machines. For example, applications could incorporate a specification of the required architecture in some form of standardised specification language. It is important to remember that it is the specification, rather than the 'circuit programs', which can be transferred from one machine to another. It will not necessarily be possible to transfer circuits since any difference in feature size or material technology between machines will effect timing considerations. For example, in a silicon design a component may no longer function if it is scaled down [MEA80].

2.3.3 Highly dynamic machines

If a Soup has a highly dynamic architecture then reprogramming of applications architectures from within the machine can be considered. This would permit dynamic creation of dedicated architectures. An example of work done in dynamic switching can be seen in [POR87] where the switching of arrays, both in form and structure, during the course of computation, is considered. [POR87] states that "Such capabilities are desirable to cope with the computational demands of multi-stage algorithms.". (For a discussion of highly parallel dynamic systems see [HIL85].) A simple example relevant to a Soup is that an addition operation specified in a program can be compiled into an adder circuit no larger than required, with the appropriate synchronisation to neighbouring circuits. A more complex example would be a dynamic stack which claimed and released Soup cells as required. This introduces two requirements on the Soup architecture, the ability to identify free circuit space and the ability of a circuit to replicate itself into this free space. Many additional problems will result from having such flexibility in Soup architecture. For example, difficulty in deciding which circuits to use (a problem encountered in optimising compilers), although the cost of performing these decisions could be reduced by extraction of parallelism. The issues involved in this flexibility are comparable with those of tailoring microcode to provide the specifically required function, and also of the microcode modifying itself during execution depending on previous actions taken.

An application where such dynamism might prove useful is neural computing. Similar features which can be identified when considering a neural computer and a Soup architecture are the requirements for fine grain computation with massive parallelism, the potentially very high communication bandwidth, and a distributed and self-organising control mechanism [HWA87]. In particular it is possible for a neural computer to program itself [ABU87]. The ability for a machine to modify its architecture from within, it is proposed, will be of major importance.

Work on dynamic reconfiguration can be seen in Cellular Automata theory [COD68] [PRE84] [WOL86]. One of the original purposes for the study of Cellular Automata was to determine how computers could be made to reproduce themselves [PRE84]. An automaton is similar to a Soup in that it consists of a near neighbour mesh of identical processing elements. Such machines are identified as having the ability to, in theory, compute all computable functions, to reconfigure other parts of the cellular space, to reproduce itself in any quiescent, accessible and sufficiently large region of the space, and to emulate other Automaton. It is stated that "If an economical realisation can be found, Cellular Automata provide the capability of extending the computing power of a system in small or large increments and of reorganising these increments to suit various special needs". [COD68]. (The Connection Machine is described as a realisation of a Cellular Automaton [HIL85] [KUN86] [HIL87]. One important similarity with a Soup is that any single cell within the Connection Machine is incapable of meaningful computation on its own, groups of cells are connected together to form "active data-structures".) A difference between an Automaton and a Soup is that cells in Automaton are defined as synchronised to a global clock. An Automaton is one realisation of a Soup. A Soup could emulate an Automaton and hence possesses similar capabilities. However, there could be other realisations of a Soup where cells are not synchronised to a global clock.

2.3.4 Discussion of programming methods

Several programming methods with varying degrees of dynamism have now been considered. It is important that an Architectural Soup which can only be programmed once at switch-on, can be used in a similar fashion to a machine which can be dynamically programmed, by configuring the machine to emulate the dynamically reconfigurable Soup architecture. This ability to emulate other architectures is a crucial factor when defining a machine as a Soup. There will be emulation penalty, but this is offset by the frequency of use of dynamism. Dynamism must be frequently used, or there will be little benefit gained from an architecture which has this capacity. The additional control would make circuits which do not require such dynamic changes run slowly.

It has been shown that there is a range of possible dynamism within the program control circuitry. There are many possibilities for Architectural Soup architectures, the common factor being their ability to emulate each other. A major issue appears to be the binding time of the architecture. The method of programming which can be employed is affected by the quantity of architecture specified at time of manufacture, the quantity at switch on, and the quantity which can be dynamically changed. The speed of change will be important in deciding the relative merits of any dynamic reconfiguration. For a system which does not change dynamically, a global transmission system such as a mesh of global busses which are controlled externally, would suffice to force cells into the required configurations. However, as the trends in chapter 1 suggest that communication will only be local, a global bus is unlikely to be a good solution. Alternatively, it can be considered that a cell (or group of cells) has the capability to force a neighbouring cell into a required configuration, with new circuits only being loaded into the external cells of the machine. Such a system will be illustrated in chapter 4. A third possibility is that a cell switches itself, depending on past events and its environment, such as in Cellular Automata. However, this was thought unlikely to be of practical use for machines with such simple cells, as the circuitry required for this function would likely be complex.

2.4 SUMMARY

In Chapter 1 the projection of technology trends supported a prediction that future technology will permit large three-dimensional arrays of transistors. A proposed style of architecture for such technology is the Architectural Soup. The main properties of this architecture are that it is highly parallel, simple and regular. An Architectural Soup could be used for particular tasks by configuring itself at a very low level to emulate a good architecture on which to solve the problem. The assessment of the quality of this emulation is difficult as it is technology dependent. It can be argued that as machine trends are leading towards this style of architecture, it is likely that emulations will be reasonable. Provided that the Soup is sufficiently large it could, theoretically, be programmed to emulate directly (more so than current computers) any existing or future computer. The emulation would be slower than machines that have been emulated, due to the overheads of set up and control that will be required for a machine with simple cells. It is argued that this is offset by the flexibility benefits.

The technological design of three-dimensional systems will be more complex than current machines. With the increased importance of factors such as heat and non-statistical behaviour, it may be that in practice it will ONLY be possible to design a system which is simple and regular. The simplicity of a Soup's basic cells would simplify hardware design. For example, it is easier to model a regular system. Regularity simplifies production, for example, there is capacity for a high degree of parallelism to be employed in manufacture. As software becomes more responsible for concerns which were previously architectural considerations, the properties sought in hardware design will become more important in software design. Software designs possessing simple regular structure will be favoured, and it is possible that software may become concerned with factors such as heat.

In section 2.2 it was shown that it is possible to conceive an architecture which is efficient for both parallel and sequential code. The run-time efficiency is dependent on the control over-head required. The following chapter will illustrate implementation issues in more detail, to assess factors requiring control, and illustrate complexities of the future design problem. A specific example of an architecture with a capacity for internal reprogramming will be illustrated in chapter 4.

As the control overhead for Architectural Soup machines is likely to be high it is thought unreasonable to design a computer with cells at the gate level other than for the interest aspect of examining a machine with such flexibility. However as current trends project to this style of architecture, architectures with only slightly more complex cells than the machines described here are likely to be practical.

Chapter 3: Directions of investigation

3.1 INTRODUCTION

This chapter is concerned with implementation issues. Due to the diversity of topics which need to be considered when designing a machine it is not possible to perform a thorough investigation into all design issues. It is therefore intended to present an overview of possible directions and issues for investigation, in particular, to illustrate factors which are likely to be of increasing importance in future machines. The factors which will be examined are, theoretical considerations, material properties, surfaces and interface considerations, initialisation problems, synchronisation, control circuitry, fault tolerance and some alternative technology to silicon style machines.

3.2 THEORETICAL ASPECTS

Restrictions on the architectural form of a machine can be established from examination of relevant mathematical theory. Consider the manner in which the cells of the machine can be interconnected. This will effect the number of neighbours which can abut an individual cell. The physical interconnection architectures which are possible can be determined by examining the theories of compacting solids, for example work see [TOT64] [LOE76] [MAN82]. The number of regular polyhedra which compact together to fill an area of three dimensional space is finite. Some examples of such space-filling polyhedra are shown in Figure 3.1. Consider one cell of an Architectural Soup as being one such polyhedron. The manner by which these polyhedra abut will determine the number of immediate neighbours a cell has and hence limit the interconnection topologies that are possible between cells. (The theoretical topologies possible would, in practice, be restricted to those permitted by the properties of the material and limitations of construction technologies. In particular some materials may not favour a simple cubic design, such as in the basis.)

There is a trade-off of functionality between cells and functionality within them. The functions available within a cell will effect the number of neighbours that a cell should

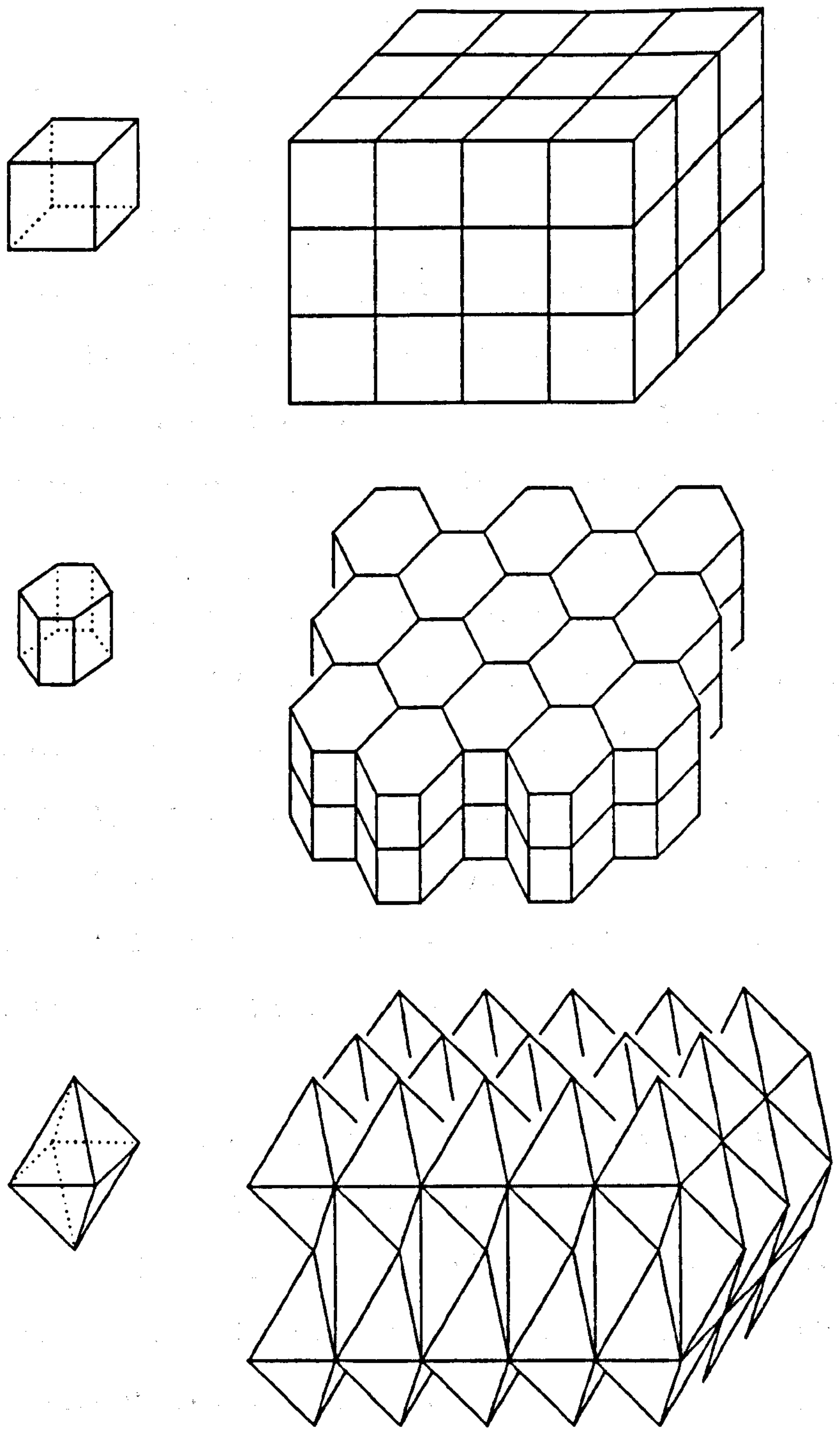
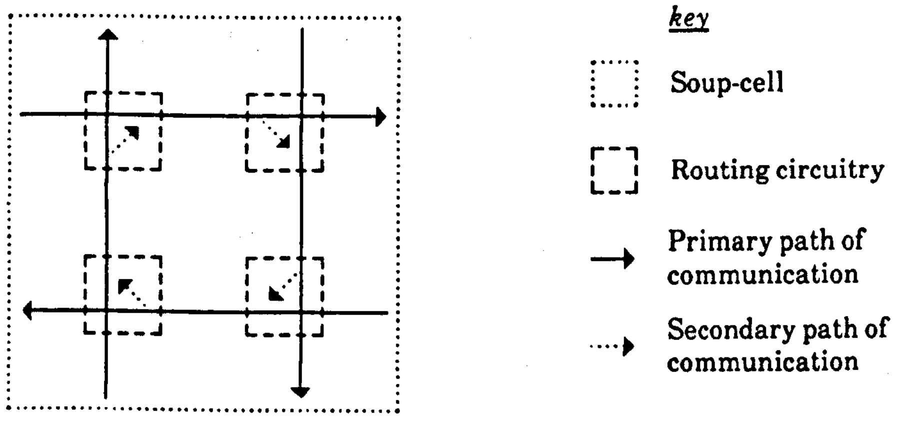


Figure 3.1: Examples of space-filling polyhedra

have. For example, if a cell has a complex processing element or a large memory it may justify a complex communications topology. When considering routing, there are many possibilities available from within a Soup cell. Each cell in the basis architecture, as described in chapter 2, has six inputs and six outputs. For isotropy, it was stipulated that it be possible to connect every input to every output. This is an exhaustive method of routing and so likely to be more complex than is required. Such a system will need extensive control to select the required combinations from the large number of wiring possibilities available. It should also be noted that with this system, the cost of increasing the number of neighbours will be relatively high due to a combinatorial explosion in the complexity of the cell. This suggests that a small number of neighbours will be a major design aim for an exhaustive approach to inter-cell routing.

The opposite of an exhaustive routing set is the minimum routing set. This utilises the smallest number of connections which permit communication from any direction to any other. One possibility for cells similar to the cubic basis architecture, would be to have each input primarily connected to the output on the opposite face, while incorporating the ability to redirect the output to one other face, as in Figure 3.2. For example the input to the left face would primarily be routed to the output on the right face, but it would also be possible to direct this output to the front face if required. If cells remain orthogonal then information can still be routed from any direction to any other, by connecting together several adjacent cells to route signals to the required direction. Such a routing system would increase linearly in complexity with the number of neighbours that a cell has, so permitting a higher number of neighbours before the complexity of routing control becomes prohibitive. A penalty with a minimum routing set, however, is that where several cells may be required, only one is required with the exhaustive approach. An architecture based on a minimum routing set is illustrated in chapter 4. This architecture was chosen for its simplicity.



key
Soup-cell
Routing circuitry
Primary path of communication
Secondary path of communication

Figure 3.2: Example of a minimum routing set soup-cell (simplified to two dimensions)

3.3 PROPERTIES OF SILICON-STYLE MATERIALS

Design of silicon at approaching minimum feature size is complex. For a detailed analysis of the properties of silicon and the complexity involved see [MEA80] [CAR80] [CHI82] [LIN82] [GIB83]. This design problem will be exacerbated by the increased complexity possible with three dimensional circuits. For example, higher connectivity would permit smaller more densely packed components [PRE83] [ROS83]. This shortening of wires in circuits will potentially make components faster. However, this will also increase other design problems, such as the amount of heat generated. The scale of features in cells will effect the manner in which they are implemented. For example, as VLSI feature size reduces, the relative time cost of wiring increases over the cost of the switching components [MEA80]. The distinction between switching states becomes blurred due to current leakage and imperfections [BAT88]. Smaller active circuits have a reduced drive capability [NUD85]. These factors suggest that the smallest possible cell size will not necessarily be the best. Considering manufacturing aspects, the smaller the feature the tighter the controls needed and the lower the yield. Not all problems will increase in building a large three-dimensional machine. For example, a simplification of the power distribution problems found in Wafer Scale Integration, unlike in current silicon chips, is that there is no requirement for high power driving circuitry needed for communication between several chips within a wafer [GRI84].

The complexity of manufacturing three dimensional devices will influence the resultant architecture. For example, if it is as easy to use three dimensions as it is to use two dimensions then cubic or spherical machines can be considered. However, if utilisation of the third dimension is difficult then machines will essentially be only a small number of surface layers, such as the computer described in [NUD85] which consists of the interconnection of a few two-dimensional wafers.

Section 2.2.2 illustrated the ability to model part of the machine and make generalisations, due to the regularity of the architecture. The design of a Soup should be simpler than designing three-dimensional machines which are based on existing architectures, due to the simplicity of the Soup cells and the regularity of the inter-cell architecture. A consequence of the simplicity is that it will be easier to design and accurately model the electrical considerations of an individual cell. Once this has been performed, interaction in a regular group of such cells could be examined and generalisations made for a larger machine based on repetition. This would be a difficult task for complex and irregular architectures as such generalisation can not be made.

3.4 SURFACES AND INTERFACE CONSIDERATIONS

The surface area of a Soup is a factor in interface considerations as it is the point from which communication will occur. The ratio of surface area to volume is important. A thin film of material provides a higher surface area to Soup cell ratio than a sphere which has a low ratio. The optimum ratio will be determined by the speed of the Soup cells and the input/output requirements. (For insight into possible capability of future input and output devices see work on highly parallel optical stores such as [ABU87] or [HUT87].) If the permissible input/output speed is a restricting factor this may limit the maximum useful size of the Soup. This problem is compounded in larger Soups as the average communication path from Soup-cells to peripherals interfaced to the surface will also increase. Input/output intensive tasks may, as a result, take longer to execute.

The surface of a material is particularly important for controlling the materials behaviour and properties [JAS77].

For example, the abrupt termination of a crystal lattice at the surface must result in a unique arrangement of the surface atoms. The surface of a machine is most prone to unpredictable external factors such as electrical interference. The surface will also be required to dissipate heat. These factors suggest that one of the most important areas of the machine (input/output) is also likely to be the most unstable if designed in the same manner as the cells in the remainder of the machine. It is therefore suggested that cells in the proximity of the surface will require a different design from those cells at greater depth. This will introduce a programming irregularity. Irregularity may be acceptable, given that a programming irregularity at the edge of the machine is inevitable due to the discontinuity of material. In systolic arrays this problem is solved by routing the outputs from one side of the machine back into the inputs on the opposite side. Examples of rings and torus architectures performing this function can be seen in [MEA80] [MOO86] [POR87]. However, this is unlikely to be a feasible solution for an Architectural Soup due to the large communication distance between cells on opposite sides of the machine relative to the distance between adjacent cells within the machine. An alternative is that the architecture can to an extent be folded to reduce this communication distance. For example consider a cubic Soup. The left and right surfaces of the machine could be folded together to form a ring, and the top and bottom surfaces could be folded to obtain a doughnut shape. (It is not possible to join the front of this Soup to the back as these are now the internal and external sides of the doughnut.) Such a machine is likely to be hard to manufacture due to its irregularity, and similarly hard to interface to and program. For these reasons the cubic machine is favoured. If it were required, a small 'doughnut architecture' machine could be emulated on a large cubic Soup.

Compatibility of a Soup with the machines it is required to communicate with will be an important factor. It is possible that communication with the outside world may suggest a communication mechanism for Soup cells in the proximity of the surface. This may influence the communication topology of all cells within the machine in order to maintain regularity. For example, a fundamental problem with input/output devices is metastability [MEA80] [KOE86]. This necessitates determination of whether an intermediate signal is of a high or low value. One method of tackling metastability problems is to use asynchronous communication protocols, the result being that occurrences of metastability only cause the system to slow down temporarily [KOE86]. If such an interface scheme were adopted it may be desirable to make the remaining cells within the Soup communicate using asynchronous protocols.

3.5 THE INITIALISATION PROBLEM

The initialisation problem concerns transformation of a machine from an initial random state into a known and useful state. For Soups the initialisation problem can be subdivided at a cell level. Firstly, a cell must acknowledge that it does not contain a valid program, and secondly, it must accept the required program. Considering the first problem, cells could be designed which assume an empty state when the machine is switched on by using the initial power characteristics. An alternative is that a cell could be forced into an empty state by the cell's neighbour(s), and once in the empty state the cell could force a neighbour (or any neighbours) which are not in an empty state similarly. Cells on the surface of the machine could be forced into a suitable state to start this process - it should not be complex to generate programs consisting of all ones or all zeros in the machines surface cells, by connecting every surface cell's program input to all high or all low input signals as appropriate. The architecture of the machine could be such that a cell receiving an all-ones or all-zeros program will send a similar message to its neighbours and then place itself in the empty state ready to receive the appropriate initial programming. After a period of time all of the cells in the machine should be in the empty state. The time taken for initialisation will depend upon either the speed at which cells can be reprogrammed or the speed of input to the Soup. A system based on this will be described in chapter 4.

There will be problems with such a start up system. For example, in a machine which can program itself internally there will be a risk of some initial random state, such that the machine will start reprogramming itself. This programming may be resistant to any start up circuitry, for example, if it can reprogram itself faster than the initialisation circuitry can clear it. It must be ensured, therefore, that the machine is unable to arrange itself into a random state such that it cannot be initialised. Similarly, there is the possibility of accidentally generating an all-ones signal in one cell during execution (for example as a result of external interference) which could result in a reprogramming of the entire machine. Another initial problem would be if a Soup cells initial random state is an illegal electrical state such that the cell does not function as expected. Avoidance of this unintended function (as with all circuits) will require special attention when cells are designed.

3.6 SYNCHRONISATION ISSUES

Both the designers and the programmers of a Soup will be concerned with synchronisation. The designer must provide sufficient synchronisation so that the system can be used, while programmers must generate the required synchronisation so that applications algorithms execute in the required manner. The boundary between the designers concerns and the programmers concerns is not distinct. Low levels of synchronisation are concerned with controlling the function of an individual Soup cell and the communication between the cell and its immediate neighbours. In current silicon design the preferred method for such synchronisation is to distribute a global clock signal and synchronise circuits to this signal. However even on a relatively small silicon chip transmission of such signals can require much complexity and thought (see [MEA80] [YAK85] [KOE86]). For example the clock signal needs to be clean and sharp at all locations. The additional complexity incurred with the size of a large and three dimensional machine will make clock distribution difficult, although it should be noted that the simplicity and regularity of the machines architecture should again serve to simplify this design problem over irregular alternatives for such technology. A synchronous system will likely return a low Soup cell performance since the clock must run at the speed of the slowest cell function. For simple cells the disparity between the individual elements such as wire and memory is likely to be large. Also, as feature sizes get smaller, the increased wire delay relative to device delay, due to the scaling, enforces a slowing down of a synchronous system that distributes a common global clock. The clock rate must be reduced to compensate for signal skew [YAK85]. Clock skew removes any assumption that all components of a design receive the clock signal simultaneously [KOE86]. The greater the distance between a cell and the clock source the more skewed and degraded the signal will become. Regions of such a Soup would vary in stability as cells closer to a clock will likely function more reliably. A slow clock speed is not always a major consideration. The extensive parallelism permitted in a three dimensional computer described by [GRI84] permits many operations utilising relatively slow clock speeds. ([GRI84] also notes that having a slow clock speed in his architecture reduces power dissipation problems.) Similar results are obtained in neural computers where a slow processing speed is offset by the massive parallelism possible [HWA87][ABU87].

It is claimed by [YAK85] that the self-timed design approach is capable of tackling the problem of timing. Self-timed circuits offer benefits in better design and possible increased speed but may require more circuitry. For a machine with simple cells, the overhead of implementing self-timed protocols in every cell is likely to be expensive. This suggests a need for a simple system for synchronisation within each cell and generating all other synchronisation through software. To illustrate the feasibility of this approach, where synchronisation is a major concern, see work on the automatic translation of algorithms from a high level language into self timed systems [KEL85] [YAK85] [KOE86].

The compromise to the clock distribution problem, to date, has been a hierarchy of communication. [KOE86] proposes a hierarchy of communication levels with increasingly sophisticated protocols. At the bottom level there are isochronic regions, at the next level simple protocols are used, while at higher levels features such as in local or wide area networks can be seen. The notion of self-timed systems, where systems are structured by the interconnection of self-timed modules communicating without the use of a common clock is described in [YAK85]. For example, a Soup might have a global clock to provide signals to ensure that small groups of cells function correctly, but as clock skew will occur, regions are assumed to have their own equipotential domain. Communication across two domains requires asynchronous protocols. One particular benefit of a self-timed approach is interfacing to the outside world, with a reduction of metastability problems ([KOE86]). A problem is likely to arise in uniformity between Soup machines, that is, in different machines differing degrees of synchronisation will be left to the software.

3.7 CONTROL CIRCUITRY REVISITED

This section aims to illustrate control possibilities. The discussion is general in that it does not aim to illustrate any particular possibility in detail. (A specific example of an architecture will be given in chapter 4.) In section 2.3 it was noted that there will be many possibilities for the program control circuitry. The form of circuitry will be technology dependent but will also be influenced by the manner in which the machine is to be used. To consider all possible control mechanisms for all possible technologies it is necessary to consider the manner by which things can change. It was proposed in chapter 2 that useful inference can be made through an examination of polymorphism. Polymorphism is seen as being fundamental [SNY82] since computers can radically change their function by changing their programs. As there are many forms of polymorphic transformation there will be many possibilities for the means by which a computer may change its function. (From the standpoint of flexibility, a general purpose machine should support many means of transformation.)

Decisions must be made concerning the functions that should be contained within an individual Soup cell and how the cells should be interconnected. To provide reprogramming capacity, it is necessary to determine, if the Soup needs to be in a particular state in order to perform a programming event, the number of cells required to perform programming of another cell, and maximum and minimum numbers of cells which can be programmed in a single programming action. On functionality, it is necessary to decide whether cells have continuous or discrete function. Also, communication distance and input/output bandwidths will effect architectural decisions and hence effect the control circuitry. In terms of general polymorphic transformations, this suggests consideration of the transformations which are possible from within the Soup, the starting conditions required for each transformation, and the grain size and controllability of the transformation (for example, is it possible to stop, change or reverse the course of the transformation). Other factors important to polymorphism are whether there are any chain reactions or side effects which must be considered and the importance of local environmental factors to a transformation.

Clearly there are many properties to be considered when deciding on control possibilities. A detailed examination of polymorphism should determine these possibilities. For example, a monotropic transformation is one which is essentially one way [SOX76]. This suggests a write-once read many times medium, with the possibility of particular architectures being permanently set into a machine if a design is satisfactory. This is similar to programming a read only memory. Some benefits of such a Soup would be, reducing initialisation delay over a general purpose machine as the architecture configuration is permanently programmed, and provision of dedicated machines could be simplified with an ability to program a dedicated architecture from a standard off-the-shelf general purpose component. Much optimisation of circuits could be justified for machines being designed on general purpose architecture where the intention is to create such dedicated machines.

3.8 FAULT TOLERANCE

The more faults in a machine the less efficient it will be. Effort will be required for fault tolerance. The yield of VLSI and Wafer Scale Integration devices decreases exponentially if the silicon area required for a device increases [SU86]. The yield also decreases if smaller feature sizes are used for the components of the device. The construction of a large three dimensional solid which is fault free is probably impossible. For example, dislocations and fractures of material often occur during manufacture and will stay with the machine throughout its working life. Circuits will fail over time. If these faults are transient then they need to be tolerated temporarily. If, however, the failures are permanent then the removal of faulty components will be difficult, if not impossible. A key concept in fault tolerance is that weak spots in a design need to be identified and improved [LIP87]. Faults can occur in any area of the machine. Consider global transmission systems such as a power network. A single fault can potentially corrupt the whole system unless cells of the machine have a degree of independence enabling tolerance against unexpected interactions from neighbouring cells. Faults could also occur in the control mechanism, for example, effecting the method by which a cell becomes initially loaded or reprogrammed. The architecture must always permit some form of input and output otherwise it will no longer be possible to use the machine.

One of the main fault tolerance techniques used is redundancy. This can either be physical (replicated hardware/software) or temporal (repeated execution) [LIP87].

On a Soup, fault tolerance could be performed by software, for example, the automatic generation of N-Modular Redundancy circuits. ([KOE86] describes the generation of such circuits and also illustrates applicability of self-timed systems to fault tolerance.) The degree of fault tolerance in an application could be varied by software methods. An alternative is that each Soup cell incorporates its own redundancy circuitry. A problem with this approach, however, is that each cell needs a minimum of three copies of its circuitry and additionally three adjudicators. As a result the cells of the machine are likely to be significantly larger. An alternative technique used in hardware fault tolerance is to maintain spare areas of circuits which may be switched in to replace an area known to contain a faulty component. For an example, see work on Wafer Scale Integration and array processors such as [LIP87]. Due to cell simplicity, the maintenance and control of areas of spares, and the extra programming complication and communication distances which will result to communicate across these areas, suggest that this also is not a good solution. It is suggested that fault tolerance could be more a software concern than a hardware one.

An interesting possibility is that an unreliable Architectural Soup could be used to emulate a more reliable one, although the cells of the emulated machine would be slower and fewer in number. If the emulated machine was still unsatisfactory then the emulation could be repeated so that the emulated machine emulated a machine identical to itself (except in size and speed). A recursive emulation could be performed until a machine of sufficient reliability is achieved. (This only applies to the functionality aspects of the cell, as it is still necessary to have inherently fault-tolerant power networks. A fault in this could prevent any utilisation of any of the cells of the machine.) Different applications will require differing degrees of fault tolerance. One solution is to implement a system which has little hardware fault tolerance in the cells but permits the user to use software methods to emulate a more fault-tolerant Soup if required.

3.9 ALTERNATIVE TECHNOLOGY

The discussion in this chapter has focused primarily on digital silicon systems. It is possible that the cells of a Soup could be analogue, with infinite function controlled by a continuous voltage. The complexity of deterministic control for such circuitry would be immense. If it is feasible then a possible application would be in the area of self-learning systems, such as neural networks, where systems adjust to a problem over a period of time. Some example work has been done in the area of analogue perceptrons in [LPM87]. Note that the emulation of an analogue system on a digital system and vice-versa has not been examined. These are likely to be non-trivial tasks. This suggests the possibility of there being two types of Architectural Soup, those primarily synchronous digital and those primarily asynchronous analogue.

Some alternative construction mediums to silicon style materials are optical systems and biological systems. Differences are likely in machine manufacture and flexibility of use. Little investigation was performed into these materials but for illustration of capabilities a brief synopsis is given.

3.9.1 Optical computers

An optical computer differs from silicon machines in that photons are used as opposed to electrons. Optical processors have the potential of computation far beyond the foreseen limits for electronic processing technologies [KUN86]. For example, it is possible to multiplex multiple information streams simultaneously through the same transistors, "multiple beams of light can pass through lenses or prisms and still remain separate." [ABU87]. This suggests a Soup which has multiple tasks executing in the same Soup cells but at different wavelengths. Heat is also less of a problem as there is no resistive heating when light moves through conductors or channels. (Heat is likely to be a major concern in three-dimensional construction technologies.) This would make optical technology a good candidate for large three-dimensional Soups. Considering the material properties of optical components, they have the benefit of being naturally very radiation hard but a drawback in a tendency shatter on physical impact [KUN86]. A good example of current work in optical computing and a possible application to the neural computing problem of pattern matching can be seen in [ABU87]. The possibility of a high density three dimensional information store using optical holography is mentioned in [KUN86].

3.9.2 Biological computers

A biological computer uses chemical interaction. One possible construction medium is organic semiconductors. Memory and processors based on these are discussed in [AND87] and [JAS77]. Individual cells of such computers are likely to be significantly slower than optical or electronic machines. [HWA87] gives example speed estimates for a biological neuron as being 10^-3 as opposed to 10^-13 for optical and 10^-16 for electronic equivalents. Some of the main benefits of biological systems lie in its flexibility and the generation of a three dimensional machine should not represent significant difficulty. Liquids also have the additional capability that component parts can be physically moved, unlike solids where the underlying hardware cannot move, such effects are achieved by movement of 'state'. Liquid computers are likely to be slower due to the extra control that will be required for the liquid properties. An interesting possibility with regards to a Soup is that it may be possible to increase the processing power of a liquid machine by simply adding more liquid. Similarly, a liquid computer would be more susceptible to the installation and removal of the systems component parts whilst in service, for example, for fault tolerance and maintenance purposes. (Such physical dynamic reconfiguration whilst the system is in service is an aim in a silicon architecture described in [HEW80].) A wilder speculation is that gaseous computers may be considered. This offers the possibility of an extremely flexible system, but the sparsity of the elements will likely make such computers prohibitively slow, if feasible at all.

Given that there are alternatives to silicon style materials, the combination of several construction materials should be considered for future machines. For example [HWA87] states that we may never build a pure optical computer. More likely is a system with electronic logic and optical interconnects. Such a system is described by [HUT87], consisting of both optical and electronic elements on a single semiconducting substrate. This system could be applied to a Soup to provide a fast optical global bus, such as a three dimensional machine which is primarily silicon but has a regular optical communication system. This could facilitate faster circuit loading and input/output, as the surface restriction of a purely silicon machine could be reduced. The benefits of this are offset by the technological and manufacturing complexity of interfacing between the two technologies within the machine.

3.9.3 Embedded computers

The environment in which a Soup is to be used may effect the choice of construction medium. For example, consider embedded computers which reside within the devices they control and where the compatibility of the machine to its environment may become of paramount importance. Materials have many properties, for example, hardness, toughness, elastic and plastic stress and strain, thermal expansion, thermal conductivity, and electrical resistivity. (See [VAN70] [VAN73] [JAS77] for a general examination of materials and their properties.) It is insufficient to consider electrical properties alone when designing a Soup as factors such as stability, durability and ability to manufacture the material are also major concerns. These factors are influenced by the environment in which the Soup is to be used.

3.9.4 Interfacing dedicated architectures

A dedicated computer should always be faster at solving the problems for which it is intended than a general purpose machine, due to the overhead of the extra flexibility required to be general purpose. If certain architectures are consistently used in particular environments it may be beneficial to build a dedicated machine and interface it to the general purpose Soup. When appropriate tasks are identified they can be transferred from the general purpose machine to the dedicated machine (and vice-versa in a primarily dedicated environment). The compiler and operating system have the additional complexity of deciding when it is beneficial to use the dedicated machine considering factors such as communication delay and competition for the resource by other processes.

3.10 SUMMARY

This chapter aimed to highlight the diversity of topics which need to be considered when designing a machine. A consequence of the necessary broad discussion is that individual topics could not be considered in depth. Many factors are related so making a decision with respect to one, influence the possibilities for another. Factors such as theoretical restriction, properties of implementation technology, surface and interface considerations, initialisation, synchronisation, control and fault tolerance aspects all require consideration. In deciding on a particular design factor, other related factors must be taken into account. As there are many architectural trade-offs that may be made there are likely to be many possibilities for future machines. The following chapters will investigate one possible architecture in more detail.

Chapter 4: On the implementation of an Architectural Soup

This chapter will describe a specific implementation of an Architectural Soup and highlights motivations for the choice of architecture and problems identified with this implementation. Chapter 5 will detail experience with a simulator for this implementation and chapter 6 will illustrate a VLSI design for one processing element.

4.1 MOTIVATIONS FOR A CHOICE OF ARCHITECTURE

It is not possible to manufacture a working Soup prototype due to two-dimensionality and the currently limited area with VLSI technology. A specific implementation architecture will be proposed in this chapter and a small three-dimensional machine will later be simulated. The primary aim is for illustration purposes. Once an architecture is designed simulation techniques can be used to show if it is possible to program a machine from an initial random state. If this is possible, then the complexity of designing programs/circuits for the machine will need examination due to the increased flexibility over current VLSI given by a third dimension. Simulation can also examine some of the dynamic programming issues in order to assess the difficulty involved. (Dynamic programming considerations were discussed in chapter 2 and chapter 3.) Considering the complexity of such a simulation task, insight may also be obtained into future simulation difficulties in general from problems with the simulator itself. Two-dimensional VLSI simulators require large computational resource and it is difficult to represent the simulation results in a clear manner. It is possible that a low-level simulator of a three-dimensional machine will be prohibitively slow and the results impossible to understand. A purpose of simulation work is to assess what is feasible to simulate.

Designing an architecture requires consideration of the work detailed in the previous chapters and also for the aims outlined above, as this will facilitate implementation of illustration. An architecture which is simple and regular will make illustration of the machines internal state less complex.

4.2 THE ARCHITECTURE

This section gives a general description of a specific architectural Soup architecture, followed by an examination of each cell primitive component required.

4.2.1 General Description

The functional requirements of chapter 2 were that each processing element in the basis Soup contained the primitives of a wire, a memory element, a nand gate, and the constants 1 and 0. In order to design a Soup-cell using these elements, some form of selection must be provided to determine which outputs from the Soup-cell logic should be routed to the inputs of the neighbouring Soup-cells. This selection is performed using a multiplexor which is controlled using a register called the multiplexor program register. Applications circuits specify the required multiplexor programs for every Soup-cell multiplexor within the Soup. (An example of a multiplexor program will be given later in this chapter.) A program is a series of multiplexor routing definitions. For a multiplexor to become programmed without the use of a global bus it must receive its program from the surrounding Soup-cells. In order to do this there is a need for some form of control unit in each Soup-cell to supervise the loading of multiplexor program registers. A cell in this machine will, therefore, comprise of logic, multiplexor, multiplexor register, and control circuitry.

To examine the functional aspects of Soup-cells without the complication of three-dimensionality, consider a cell which has a single 'previous' cell sending information to its logic and only one 'following' cell to which information is sent, see Figure 4.1. Such a cell will be called a 'face-cell' as opposed to a Soup-cell for reasons that will be explained later in this section. The face-cell requires inputs from the previous face-cell to its one bit memory and nand-gate. In order to decide which information should be input to the following face-cell, the face-cell's multiplexor must select between output from, memory, the nand gate and also the constants 1, 0 and undefined.

Many such face-cells could be joined together to form a string with uni-directional communication. Each face-cell in the string would take input from its previous face-cell and

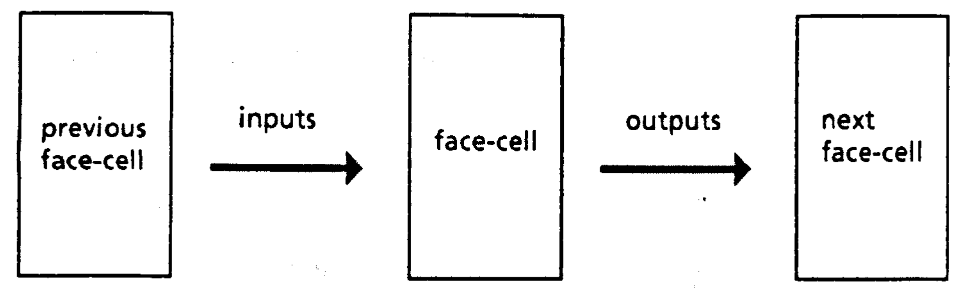


Figure 4.1: Simple face-cell string architecture

give output to the following face-cell. Consider the initial loading of the multiplexor programs of such a string. It has been suggested that cells in a Soup should receive their multiplexor programs from the neighbouring cells. In this architecture there is a choice of two cells (the two immediate neighbours) from which a face-cell can receive its multiplexor program. Loading multiplexor programs in the same manner as the processing elements communicate will make for a design which is easy to understand, see Figure 4.2. This is facilitated by multiplexor programs being loaded at the start of the face-cell string and permitted to travel the length of the string until their progress is blocked. Blocking may result from either the following face-cell multiplexor containing a program or because the far end of the string has been reached. This movement of face-cell multiplexor programs along a string is controlled by the control unit in each face-cell.

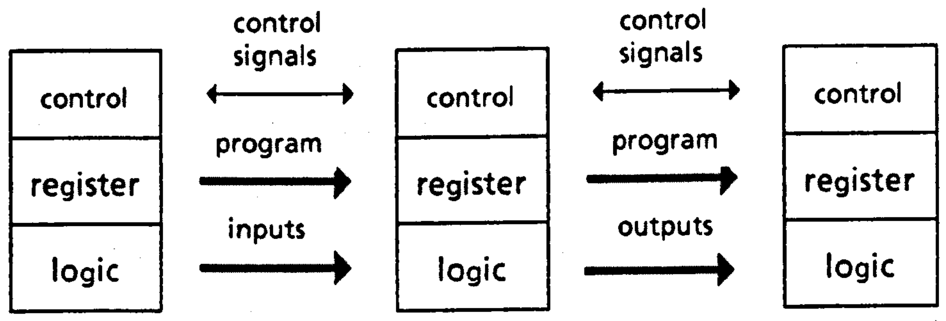


Figure 4.2: Face-cell program loading architecture

A uni-directional string of such face-cells would of course serve little purpose. Also, the architecture needs to be extended to utilise three dimensions. In the basis architecture of chapter 2 a machine consists of a regular three-dimensional lattice of cubes abutting in a near-neighbour mesh, each cube having six neighbours, each abutting one face of the cube. A cube is termed a Soup-cell. Consider now placing a face-cell on each face (hence the name face-cell) of each Soup-cell, with each face-cell communicating in a different direction. When Soup-cells are abutted in three-dimensional space, face-cells can be connected to corresponding face-cells in the previous and following Soup-cells in order to form the string architecture described above. Each Soup-cell comprises six face-cells, which are themselves components of six independent face-cell strings, which communicate in six orthogonally different directions. (This Soup-cell contains six times the functionality required for the basis, that is six nand gates and six memory elements.) Face-cells on the same faces of Soup-cells are connected in strings which receive input from the face-cell communicating in the same direction in the previous Soup-cell, and send output to the following face-cell communicating in the same direction in the following Soup-cell, Figure 4.3. For example, consider a Soup-cell at coordinate [height, width, depth] in the Soup. It has six face-cells communicating in six different directions. Consider the face-cell which communicates upwards. It will take its input from the upward communicating face-cell in the Soup-cell below , [height-1, width, depth], and passes its output to the upwards communicating face-cell at [height+1, width, depth].

Since no communication has yet been defined between the six face-cells in a single Soup-cell, such an architecture is little improvement on the single face-cell string architecture above. Three-dimensional space is now being utilised but it is not possible to communicate from a face-cell string to any other face-cell string, or even backwards along the same string. Consider adding a limited communication between the six face-cells within the same

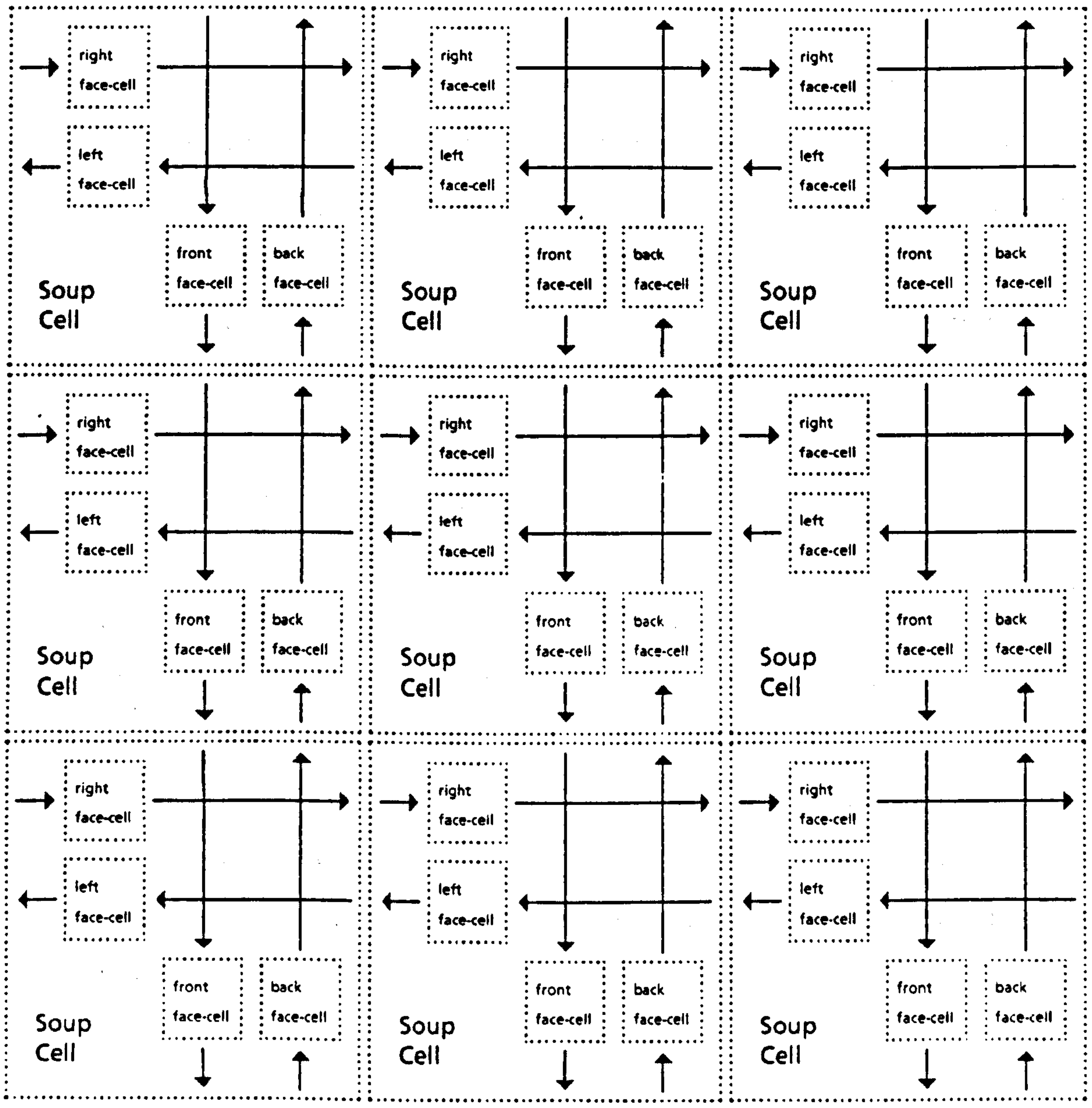


Figure 4.3: Face-cell chains (simplified to two dimensions)

Soup-cell by extending the multiplexor of each face-cell, so that it included one more input and one more output. The six face-cells then become connected by a ring when the inputs and outputs are linked together. Since it is possible to communicate between two face-cells within a Soup-cell and along the face-cell strings, it is possible from any face-cell in a Soup-cell to communicate along a face-cell string in any direction. By communicating through a path consisting of multiple face-cell strings in different directions it is possible to communicate from any Soup-cell in the machine to any other Soup-cell, and also from any face-cell in any face-cell string to any other face-cell anywhere in the Soup.

The architecture of a face-cell can be seen in Figure 4.4 and in detail in Figure 4.5. This architecture will be described further in the following sections. Figure 4.6 shows the interconnectivity between six face-cells in the same Soup-cell, using the 'from previous dimension' input and 'to next dimension' output illustrated in Figure 4.4 to form the ring.

The remainder of this section is concerned with more detailed explanation of the function of the component units of a face-cell of this architecture, namely the face-cells control unit, the logic component, the multiplexor, the multiplexor's program register, and a review of the control unit to explain how dynamic programming ability was incorporated into the face-cells architecture.

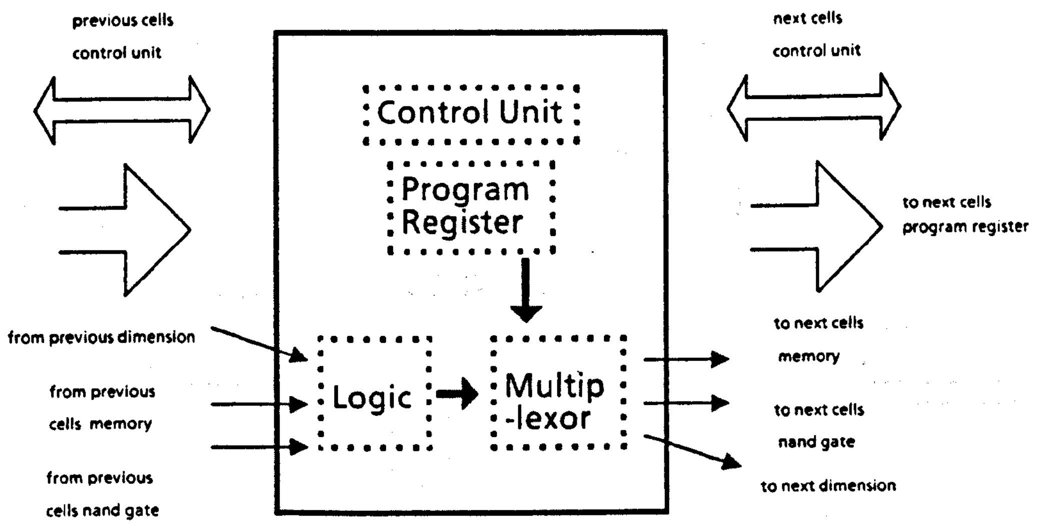


Figure 4.4: Simplified schematic for one face-cell

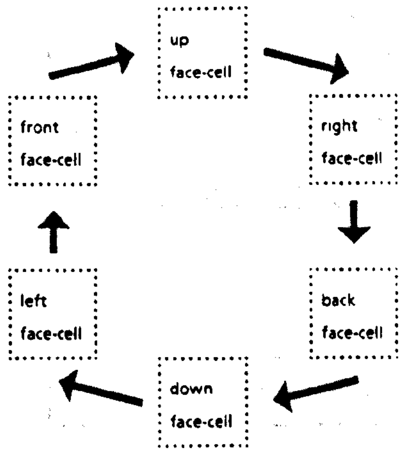


Figure 4.6: Interconnectivity between face-cells in the same soup-cell using 'from previous dimension' and 'to next dimension'

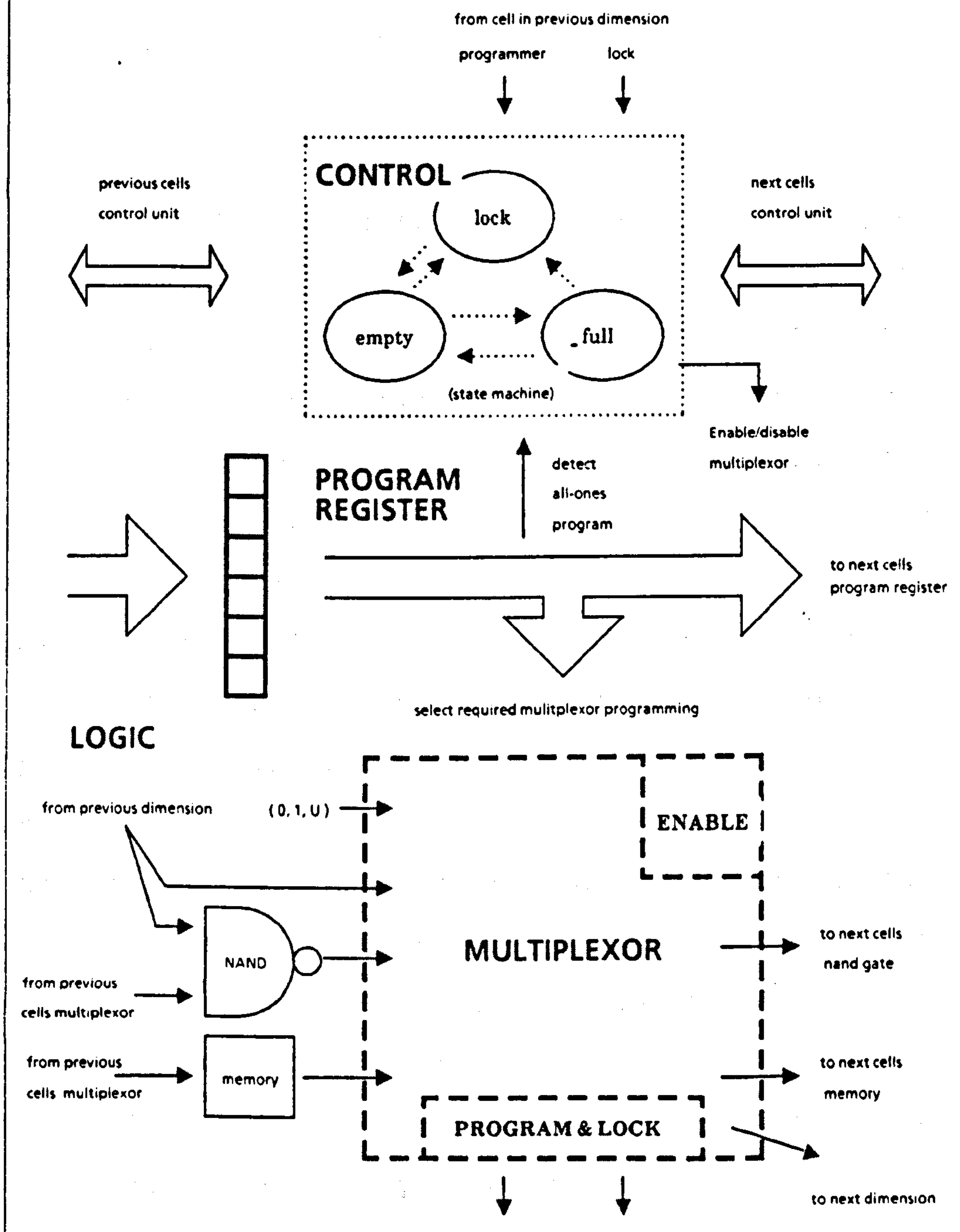


Figure 4.5: Face-cell architecture in more detail

4.2.2 Function of the control unit

The control unit is essentially a simple state machine. It can be seen at the top of Figure 4.4 and Figure 4.5. The states permitted are, empty, full and locked. Consider first the states empty and full. If a face-cell is empty, this is signaled to the control unit of the previous face-cell in the string. If the previous face-cell's state is full then it contains a valid multiplexor program in its program register. The program is copied (in parallel) into the empty face-cell. The state of the empty face-cell is changed to full and the previous face-cells state is changed to empty. If all face-cells in a string are empty then a program placed at the start of the string (an outside edge of the Soup) will traverse the whole string length, stopping only when the far end has been reached, the opposite outside edge of the Soup. If a second multiplexor program then placed at the start of the string it will only run as far as the face-cell which is one previous in the string to the face-cell containing the first program. By repeating this for all strings in the machine, on all six faces of the Soup, every face-cell may be programmed.

The control unit is also delegated responsibility for enabling and disabling the multiplexor. If a face-cell is empty, or if the following face-cell is empty, this implies the program in this face-cell is about to move to the following face-cell, then the multiplexor of the face-cell is disabled. (In Figure 4.5 the control unit can be seen to be connected to the multiplexor enable.) This prevents any spurious output results from the face-cells during loading of the multiplexor programs.

4.2.3 Logic

The logic component of the face-cell can be seen in the bottom left of Figure 4.4 and Figure 4.5. There are three inputs from associated face-cells, namely the memory input, the nand input and the 'from-face-cell in other-dimension' input. The memory element takes its input from the previous face-cell, as does one input to the nand gate in accordance with Figure 4.1 . The other-dimension input forms the inter-face-cell communication ring within a Soup-cell, in accordance with Figure 4.6 . The other-dimension input also serves as the second input to the nand gate.

Due to the need to disable face-cell multiplexors in the architecture, it was decided to use three-state (ternary) logic ([INT87] [MUK86]) in the logic component of face-cells. The three states used are 1, 0, and undefined. The implementation used for this requires two physical wires for every signal, namely high and low. If high=0 and low=1 then the signal was a zero, if high=1 and low=0 then the signal was a one, if high=0 and low=0 then the signal was undefined. The signal high=1 and low=1 was unused. (A similar two-rail ternary signal is described in [YAK85].) For an example of a ternary memory element consisting of two flip-flops see Figure 4.7. A ternary one or zero input signal to this circuit results in the same respective output signal. An undefined input signal would result in the output remaining as before. Note that a signal with both high=1 and low=1 would render this memory element into an unstable state. There is a risk of glitching during transit from a high to a low output, or vice versa, as the output signal could temporarily be such that both high and low are 1. The risk of such glitches occurring can be reduced, for example, by ensuring that the flip-flops clear faster than they set, by introducing a clock to synchronise local communication (see section 3.6 for the difficulty involved) or by designing the circuit using true three-state systems (for example experimental work [INT87] examines three-state transistors).

Figure 4.8 shows an implementation of a ternary nand gate which gives an undefined output (high=0 and low=0) unless both of the inputs are either one or zero signals, when it gives the nand of the inputs as the result. Similarly to the memory element, this circuit is susceptible to glitches.

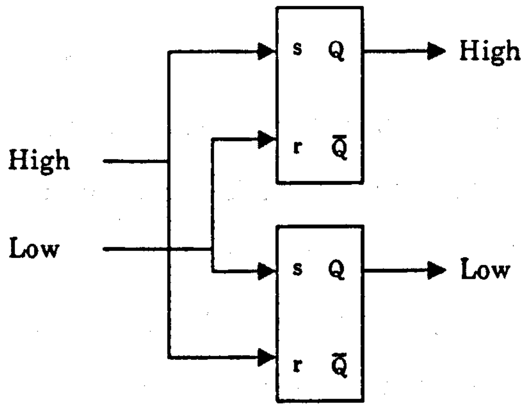


Figure 4.7: Asynchronous ternary memory element

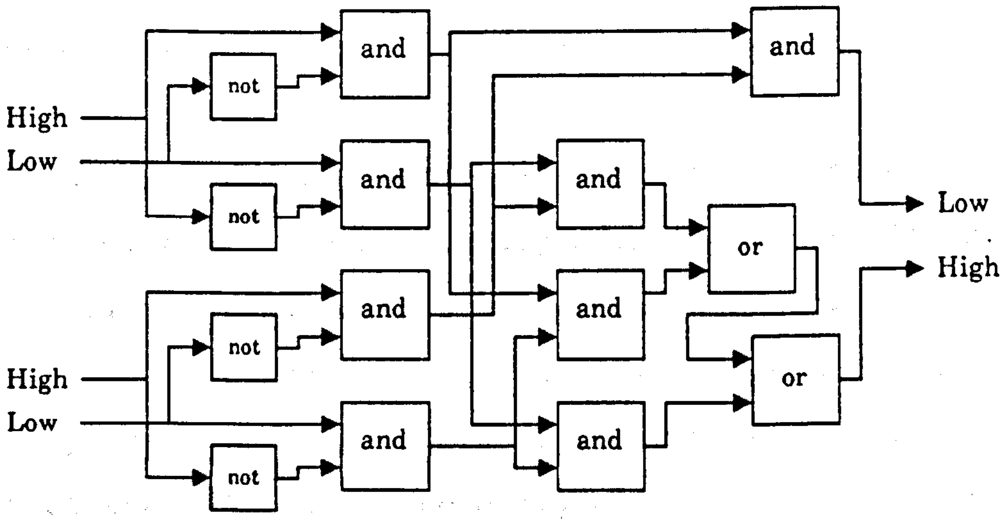


Figure 4.8: 2 input asynchronous ternary nand gate

4.2.4 The multiplexor

The multiplexor can be seen at the bottom of Figure 4.4 and Figure 4.5. Its purpose is to route the ternary output from the logic of the face-cell to the inputs of the following face-cell in the string and also to the 'face-cell-in-other-dimension' input of the next face-cell in the intra-Soup-cell ring architecture. In addition the multiplexor can route a ternary one, zero or undefined. The undefined signal is included to provide a means of preventing unused capacity in any of the multiplexors from interfering with the intended programming. This can be facilitated by routing the undefined signal to unused multiplexor outputs when circuits are specified. Also when a multiplexor is disabled it will give a ternary undefined signal on all outputs.

When the multiplexor is enabled it is controlled by the contents of the face-cell multiplexor program register. Some optimisation has been performed on the manner in which the program register controls the multiplexor in order to simplify the face-cell architecture. Consider routing every input of the multiplexor to every output. The inputs to the multiplexor are from the face-cells nand gate, memory, the input from one other face-cell within the same Soup-cell, and the constants 1, 0 and undefined. The outputs are to the next face-cells nand gate and memory, and to one other face-cell within the same Soup-cell. This is a total of six inputs and three outputs. In order to route any input to one output, a three bit binary register is required to select from the six possibilities. For all three of the multiplexors outputs a nine bit register would, therefore, be required. Each output of the multiplexor will also need six selector circuits controlled by decoding the appropriate three bits of the register. This would require a total of eighteen selectors in each face-cell. The feeling during the design of the face-cell was that the multiplexor was likely to dominate its design. The usefulness of having all inputs was therefore examined in more detail so that optimisation could be considered.

Consider the input to the face-cells memory element. Connecting a constant zero or one output from the previous face-cell to the input of the memory serves little purpose, as the face-cell multiplexor could be programmed to give this one or zero output directly. Consider connecting two consecutive memory elements together. Again this serves little purpose as the second memory element will act as a wire for the first. The useful inputs to memory are therefore likely to be the input from the nand gate or the input from other-dimension. (Either of these inputs could be used to route a 1, 0 or undefined signal to the memory element if required.) Hence in this architecture, only a 1 bit register is used to select between the two useful possibilities for the memory input. Similar reductions can be made for the output to the next face-cells nand gate and the output to the following face-cell in the same Soup-cell. The program mapping that was used can be seen in Appendix 1. The resultant register sizes required for this architecture are 2 bits long in each case. The number of register bits required to program the multiplexor has been reduced from nine to five. Penalties paid for the lack of regularity in the face-cell multiplexor will be discussed later in this chapter and also in chapter 5.

As an example of a multiplexor program consider the program '001101'. The most significant digit is zero, which specifies that the program is a normal multiplexor specification as opposed to a programming or locking cell (see later). The second digit specifies the input to the next face-cells memory. As it is a zero, it can be seen from Appendix 1 that the input to the next face-cells memory is to be the input to this face-cell from the previous face-cell in the intra-Soup-cell architecture. The third and fourth digits are both 1. These specify the input to the next face-cells nand gate, and in this case the input is the output from this face-cells nand gate. The fifth and sixth digits are 0 and 1 respectively. These specify the output to the next face-cell in the intra-Soup-cell architecture, and in this case are the output from this face-cells memory.

4.2.5 The program register

The multiplexor program register is six bits wide. (A full explanation of the mapping used is given in Appendix 1). If the most significant bit of a program is a zero then the remaining five bits are used to program the multiplexor as described in section 4.2.4. If the most significant bit is set then the cell can perform special programming and locking actions. For example, one special program is 111111, which traverses a face-cell string emptying all cells as it goes, and continues until either the far end of a string or a locked cell is reached. (An 'all-ones' program is detected by "AND"ing together all of the bits stored in the program register.) If a locked cell is reached then the lock is cleared and this face-cell is left in the empty state. This permits a Soup with cells in an initially random states to be initialised by the input of a stream of all-ones messages on all external face-cells of the machine. All face-cells should eventually be forced empty.

4.2.6 The control unit revisited

In addition to the full and empty states, the control unit has a locked state. Face-cells have two additional outputs which float to undefined if the most significant bit in the program register is a zero. If the most significant bit is a one then the face-cell can be used to lock or program, see Appendix 1 for the program codes used. The lock and program outputs are used to permit dynamic programming and can be used to control another face-cell within the same Soup-cell. For simplicity it was decided that the controlled face-cell would be the same cell that followed the face-cell in the intra-Soup-cell ring architecture already described. See Figure 4.9 and the top and bottom of Figure 4.5 for the locking topology used. A use of locking a face-cell is that a program in a locked face-cells program register will not move further along its string if the face-cell following becomes empty.

Cells may also act as programmer cells. A face-cell forced to act as a programmer effectively rotates up its program register one bit resulting in the least significant five bits becoming the most significant five bits for a new program for the face-cell which follows it in the string. A new least significant bit is generated using the other-dimension input, a transition of undefined to 1 or 0

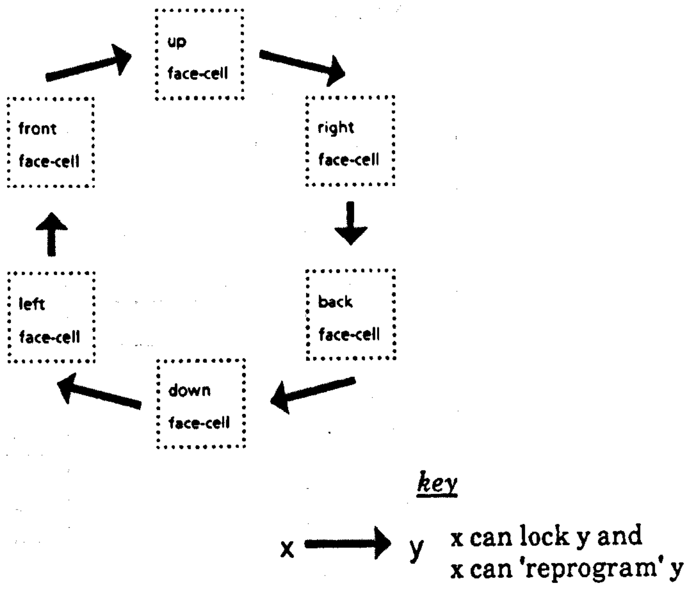


Figure 4.9: Face-cell locking and programming topology

on this input generates the programming event. A programmer face-cell forces the next face-cell in the string to accept the new program by means of a 'new program' signal from the control unit, see Figure 4.10. By abutting six such programmer cells a new six bit program can be generated, see Figure 4.11. Note that there is a potential problem in locations along this programmer chain as it would be possible to accidentally generate an all-ones program, which would then travel through the following face-cells and empty them. This is avoided by designing the control unit such that all all-ones messages are ignored when a cell is acting as a programmer.

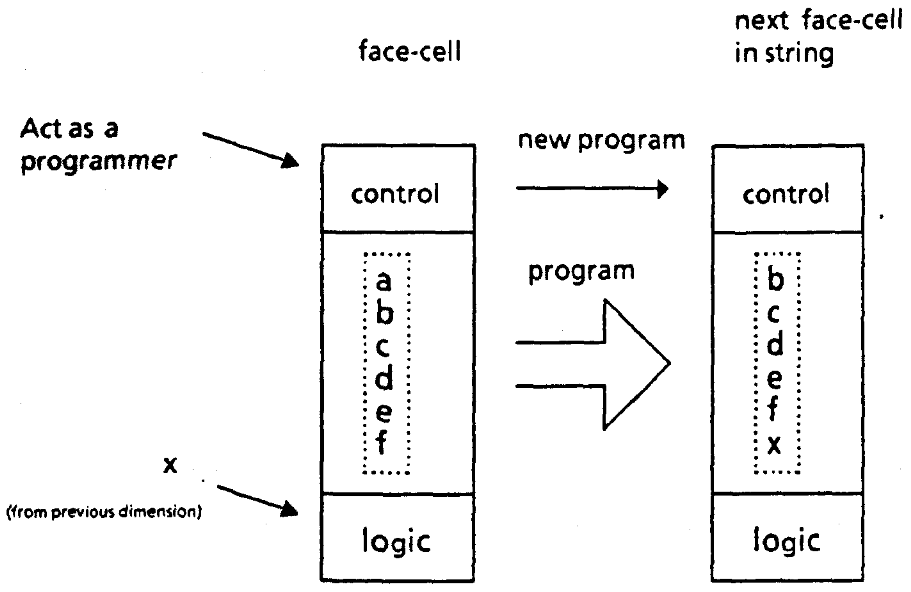


Figure 4.10: A face-cell generating a new program for the face-cell following in the string

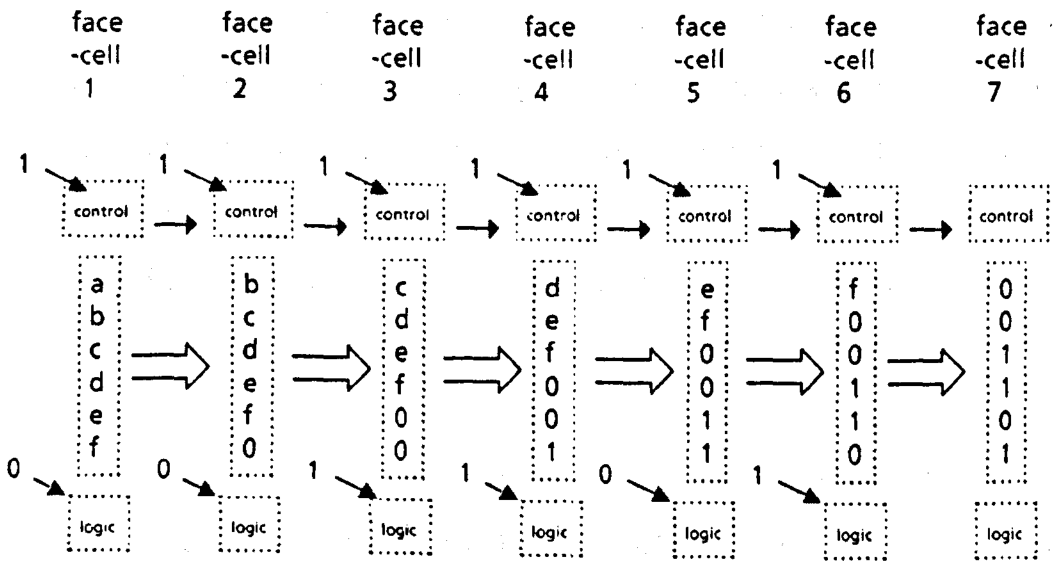


Figure 4.11: An example of the generation of a new 6-bit mutliplexor program

4.3 IS THIS AN ARCHITECTURAL SOUP?

The definition of a basis architecture for an Architectural Soup was given in chapter 2. Two of the main properties are seen as being homogeneity and isotropy. The cells of this architecture are homogeneous in that they are all identical. As each Soup-cell is functionally identical irrespective of which face-cell is being examined and the Soup-cells are in a regular lattice, they also exhibit isotropy. A requirement of the basis was that it was controllable and deterministic. This will be illustrated through simulation work in chapter 5. The Soup-cells in this architecture have the same inter-cell communication topology as the basis, i.e. six immediate neighbours abutted in a cubic near-neighbour mesh. Each face-cell contains the required primitives of, a wire, a one-bit memory, a simple processing element, and the ability to generate the constants zero and one. As each Soup-cell consists of six face-cells, it has in fact the required functional primitives of a basis cell six times over.

A Soup-cell in this implementation does not have the ability to place its functional primitives between any of the inputs to the Soup-cell and any outputs as required. There are restrictions within the Soup-cell as to which outputs it is possible to route data to. However, by combining a few of these implementation Soup-cells a cell with the required functionality in intra-cell routing could be emulated. As the penalty of such emulation should not be significant this architecture is an Architectural Soup.

4.4 ARCHITECTURE SUMMARY

A face-cell is a simple processing element consisting of logic (a nand gate and a 1 bit memory), a multiplexor, a program register and a control unit. A Soup-cell contains six independent face-cells which communicate in six different directions. These face-cells also have limited communication between the other five face-cells within the same Soup-cell. A Soup is a three-dimensional lattice of Soup-cells which have been abutted such that the face-cells in the same orientation lie in long straight strings which stretch from one side of the Soup to the other.

The control unit of a face-cell can be in one of four states;, empty, full, locked or programmer. A face-cell's control unit is in the empty state if the multiplexor program register does not contain a multiplexor program. Conversely, a full state denotes the presence of a program, unless the face-cell is in the locked or programmer states. If a face-cell is locked then its program (and all programs behind the locked cell in this face-cell string) is unable to move further along the face-cell string. Locks can only be cleared via an all-ones program. If a face-cell is in the programmer state then it is controlled by another face-cell within the same Soup-cell, in order to generate new face-cell programs. This new program is generated from the current program's five least significant bits shifted up one bit, and the other-dimension input generating a new least significant bit. Programmer cells will typically be used in blocks of six in order to generate the six bits required for a new multiplexor program.

A multiplexor program is a six bit program specifying which inputs to the face-cell should be routed to the face-cells outputs. An all-ones program is a special multiplexor program used for deleting unwanted multiplexor programs. An all-ones program will travel along a face-cell string and place each face-cell into the empty state. It will only stop travelling along the string when either the end of the string or a locked face-cell is reached. If a locked cell is reached then the lock is cleared.

Ternary Soup Logic is three-state logic which is used in the logic component of a face-cell. It has the traditional 1 and 0 of binary logic and an additional undefined signal. One particular use for an undefined signal is to provide a means of preventing any unused multiplexor capacity from generating unwanted output, which can interfere with other face-cells.

4.5 ARCHITECTURE ASSESSMENT

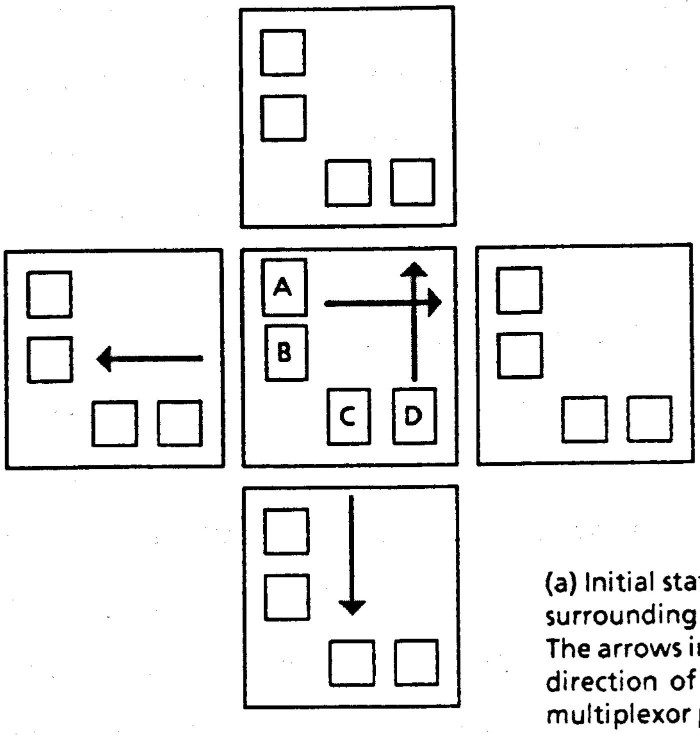
Considering topics of previous chapters, it can first be seen that the Soup architecture described can be programmed from an initial state through repeated generation of all-ones messages. The probability of the system being in a state from which it can not be initialised, as discussed in section 3.5, is small, and could be further reduced by making the restriction that all face-cells are forced into an empty state when the machine is switched on. The loading process is deterministic in that programs stay in the same order within their respective face-cell strings. It is possible to communicate from any face-cell to any other, although doing so will restrict the functionality of the face-cells along the communication path, as some functional capacity of the intermediate cells will be required for this path. An example communication path might be constructed out of a string of nand gates. Cells along the path would therefore lose the ability to use their nand gate. There is an element of dynamic programming capability in the architecture as described in chapter 2. (In the event of a face-cell program moving, the contents of the one bit memory does not move with it. The significance of this has not been examined).

The intra-Soup-cell architecture meets the criteria of chapter 2 of regularity and no global bus. Consequently it generalises for different face-cell interconnection topologies such as different compacting polyhedra as discussed in chapter 3. For example, an architecture could have only five face-cells in a Soup-cell, which are similarly connected to each other as a ring. For such an architecture it would be difficult to place face-cell strings in straight lines, as the Soup-cells would be pentahedral as opposed to cubic.

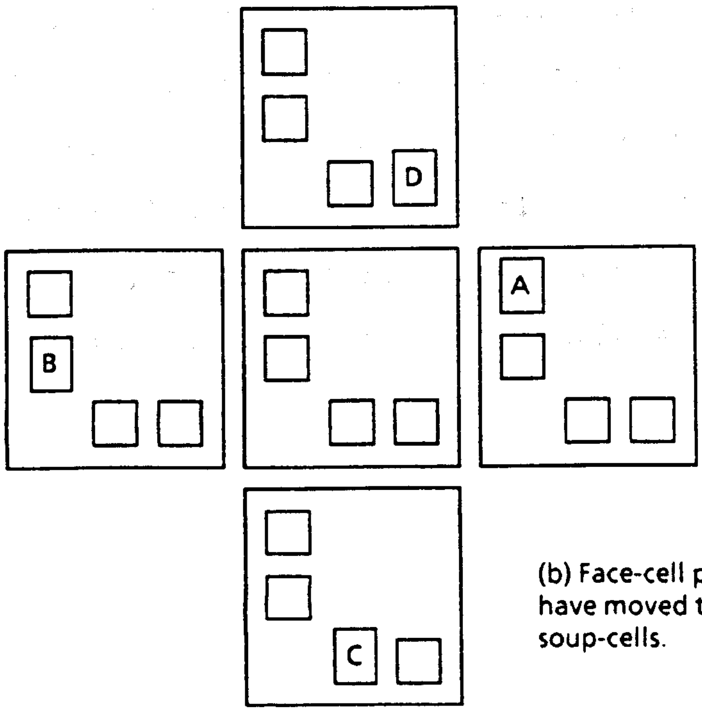
Before undertaking any simulation some design restrictions in the architecture can be identified. Firstly, there are problems with a string type architecture. The speed of program input to a string is restricted by the speed of the strings slowest elements. This should not be significant as all of the cells are identical but it emphasises the importance of the constructional difficulty of aiming for regularity throughout the machine. Secondly, the time taken for a circuit to stop moving is dependant upon the length of the string. For example, when a program is input to an empty string it must travel to the end. For larger Soups the time taken for a program to traverse the length of the string increases. Considering implementation issues, despite some optimisation, the size of the program register is still large in comparison to the size of the logic component of the cell. This suggests having a greater logic component in a face-cell, such as more nand gates or a larger memory. One possibility would be to have several replications of the functional elements using the same program register, but having different connection topologies to neighbouring cells. This would have the penalty of further complexity of programming, in addition to the increased complexity resulting from optimisations performed on the program register. There is some unused capacity in the program register which could perhaps be used, see Appendix 1. For example, it can be seen that there are several methods of obtaining the output from the nand gate of a face-cell. It is thought that there is likely to be much capacity for improvement in this multiplexor architecture.

There is a fault in the dynamic aspect of face-cells, concerning the mobility of circuits in the architecture. This prevented significant investigation into the ability of such machines to reprogram themselves using circuits dynamically created within the Soup itself. Consider an individual Soup-cell. It contains six face-cells with each having its own multiplexor program. If all the face-cells in the six Soup-cells abutting this Soup-cell are empty then all of the face-cell programs will move. The problem essentially is that each program will move in a different direction and to a different Soup-cell. For example, the right face-cell program will move to the Soup-cell on the right while the left face-cell program will move to the Soup-cell on the left. The intra-face-cell communication then has no meaning as the six face-cell programs are in six different Soup cells, see Figure 4.12. This suggests a need for some form of "directional" element in a Soup-cell to control the direction of motion of face-cell programs. This is not possible with the string architecture defined as the direction is inherent in its architecture.

Considering use of such a machine, there would likely be the requirement that if more than one task is executing within the machine, they do not corrupt each other. Unfortunately, with the current architecture this could happen. For example, a program which continuously generated all-ones messages could eventually reprogram the whole of a string, despite the possibility that part of the string might be being used by another task. This suggests having two types of lock, a user-lock and a system-lock. Users would not be able to set a system-lock, and a user-generated all-ones program would not remove this system-lock. The system-lock would be used for restricting tasks to regions within the



(a) Initial state with all surrounding face-cells empty. The arrows indicate the direction of motion of the multiplexor programs.



(b) Face-cell programs have moved to different soup-cells.

Figure 4.12: Illustration of the dynamic reprogramming problem (simplified to two dimensions)

Soup, and only cleared by an appropriate system-all-ones message which can not be generated by a user task. In the event of the system wishing to move a task it could temporarily clear all user-locks and then reset them on the new destination being reached.

Finally, consider the fault tolerance issues discussed in chapter 3. Although no fault tolerance circuitry is incorporated into the architecture there is capacity for it's inclusion. For example, the detection of a high=1 and low=1 on any ternary signal must indicate a fault. A problem with string architecture is that in the event of a face-cell being irrevocably damaged there is a potential for all following face-cells to become unusable, due to inaccessibility. With the architecture described it is possible to use these face-cells. They are reprogrammable by other face-cells via the internal other-dimension interconnect, as the source face-cells for the reprogramming input are in different face-cell strings. However, this will be a complex task.

Chapter 5: Simulation work

Ideas of previous chapters were illustrated through simulation. The simulation environment will be described. This will be followed by detail of the programming examples used to demonstrate the feasibility of the architecture, including justification for the choice of these examples. The work of this chapter should be read in conjunction with the material contained in Appendix 1 to Appendix 5. Results from this work include a critique of the simulation environment and of the specific implementation of Soup architecture. In light of this, suggestions for further work are made.

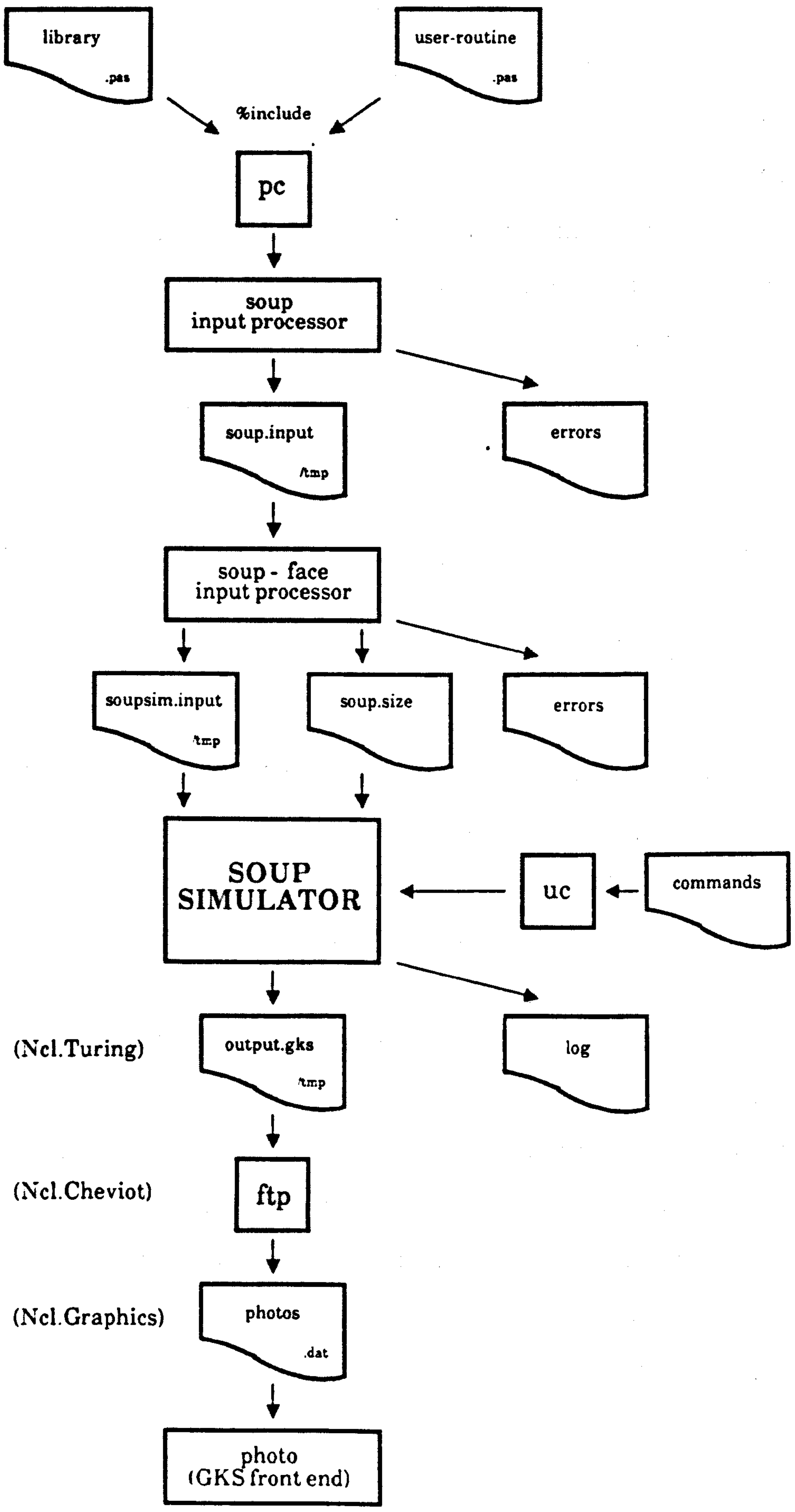
5.1 SIMULATION ENVIRONMENT

The simulation environment can be seen in Figure 5.1. This can be considered as four units, namely, the generation of an application circuit description to be simulated, a preprocessor to translate the description into suitable input for the Soup, the simulator itself, and graphical display of results. An overview of each of these sections will be given.

5.1.1 Circuit description

Circuits were described by means of Pascal procedure calls. A representation of a Soup which comprised of a rectangular array of cells was first initialised blank. The choice for a rectangular Soup was made taking into consideration the Pascal array data structure. (There are many alternatives to a cubic machine.) Each Soup-cell was represented by six face-cell multiplexor registers. It is these face-cell multiplexor programs that are defined through Pascal procedure calls to describe a circuit. The definition procedures provided permitted specification of each component in the multiplexor architecture, this having been described in section 4.2.4. The procedures are documented in

Figure 5.1 - Simulation Environment



Appendix 2. When the required circuit has been defined, remaining capacity in the multiplexors was padded out by circuitry likely to have low distractional effect to the graphical output method described in 5.1.4. The algorithm used to perform this was to attempt to route an undefined signal to any undefined face-cell outputs, although this was restricted by the capability of the multiplexor design. For example, the to-other-dimension output of any face-cell could be directly connected to a ternary undefined signal if the output was not being used. Other multiplexor outputs could either be connected to the from-other-dimension input or the previous nand input. A nand output could be undefined if either of its two inputs were undefined. The procedure used for performing this function can be seen in Appendix 2.

The relationship of the circuit description to the simulation environment can be seen at the top of Figure 5.1, circuit descriptions specified in a user-routine incorporated into a Pascal program, which in turn contained the library routines. The resultant program was compiled and then executed in order to generate a three-dimensional complete array of face-cell multiplexor definitions, each Soup cell being represented by six triplets. Each triplet describes the multiplexor program of one of the Soup-cells face-cells. As an example of a triplet consider NM1. From Appendix 1 it can be seen that the first character, N, defines the output from this face-cell to the memory element of the next face cell in the string to be the output from this face-cells nand gate. The second character specifies the output to one input of the following face-cells nand gate, and is the output from this face-cells memory in this case. The third character specifies the output to the intra-Soup-cell communication architecture (see section 4.2), and is a constant 1 in this case. Other examples of triplets can be seen in part 1 of Appendix 5 and an explanation of their meaning is given in Appendix 1. The Pascal library can be seen in Appendix 2 and the user-routines for the examples shown later in this chapter can be seen in Appendix 3.

5.1.2 Preprocessor

The Soup input processor took as input the circuit description array described in the previous section. This preprocessor translated the face-cell multiplexor triplets to the required six bit face-cell programs, the mapping is given in Appendix 1. For an example of such translation consider the face-cell multiplexor stub NM1 which was described in the previous section. The preprocessor takes this stub and converts it into the corresponding binary program which is suitable for input to a face-cell string.

From Appendix 1 it can be seen that in this case the program would be 010100.

One restriction of the Soup architecture is that in order to load a new circuit description into a Soup, input for a string of face-cells must occur at the start of the string. Each Soup-cell consists of six different face-cells, each face-cell is in a different face-cell string. As a consequence the input data for a Soup-cell has to be input from six different external faces of the Soup. For this reason the output of the input processor was the circuit array broken down into the appropriate order for external input to the faces of the Soup. The format of this output can be seen in part 2 of Appendix 5.

5.1.3 The Simulator

The simulator treats each individual face-cell within the Soup as four separate components; logic, multiplexor, control and program register. If a component's state changed, events were scheduled in the other components of this face-cell, and appropriately in any components of neighbouring face-cells which might be effected. (For simplicity, the component units relative speeds were assumed to be approximately the same.) The three-dimensionality of the architecture to be simulated enforced a major restriction in the maximum possible simulation. The maximum size that could be represented on the computing facilities available was a 60 by 60 by 60 Soup-cell machine. This size had to be further reduced during simulation due to the requirement of memory to maintain a large event list.

The simulator could be controlled at a low level through a command language. Explanation of the simulator commands are given in Appendix 4. For example, it was possible to step though a number of individual simulation events and at any point examine the current state of the event list. Throughout this process it was possible to examine and "photo" part of the current Soup state, or obtain a more detailed description as to the state of any individual Soup-cell if required.

The circuit descriptions from the input processor were entered directly into the first cell of the appropriate face-cell strings. This avoided interface considerations. The period frequency between two consecutive load events to the same string was defined to be significantly greater than the duration of any other event in the simulator. This ensured a multiplexor program which was loaded first would move along the face-cell string before the next program was input.

5.1.4 Graphical display of results

One particular command available during a simulation was that of taking a snapshot of the memory elements contents in one region of the simulated Soup. When the simulation was complete these 'photos' were exported to a machine which possessed graphics capabilities. A display program plotted the photos by drawing an outline box for a memory element if it contained a 1. No box was drawn if the memory contained a zero. The position of this box was drawn relative to its position in the Soup, the three dimensional position in two dimensional space was calculated using a cabinet projection. Memory elements for face-cells which communicated in different directions were drawn in different colours. For example, if a face-cell communicated upwards and its memory element was set then a box was drawn in red. Example output from the display program can be seen in Plate 5.1 . The top portion of this Plate illustrates the use of different colours to show the direction of face-cell communication. The six different coloured boxes overlapping at the top of the Figure show how a single Soup-cell would be displayed if all six of its face-cell memory elements contained a 1. The bottom half of the Plate shows a Photo of a random Soup, such as might be observed when a Soup was switched on. The time stated at the bottom of each Photo is the total real

Illustration of direction of communication in face-cells

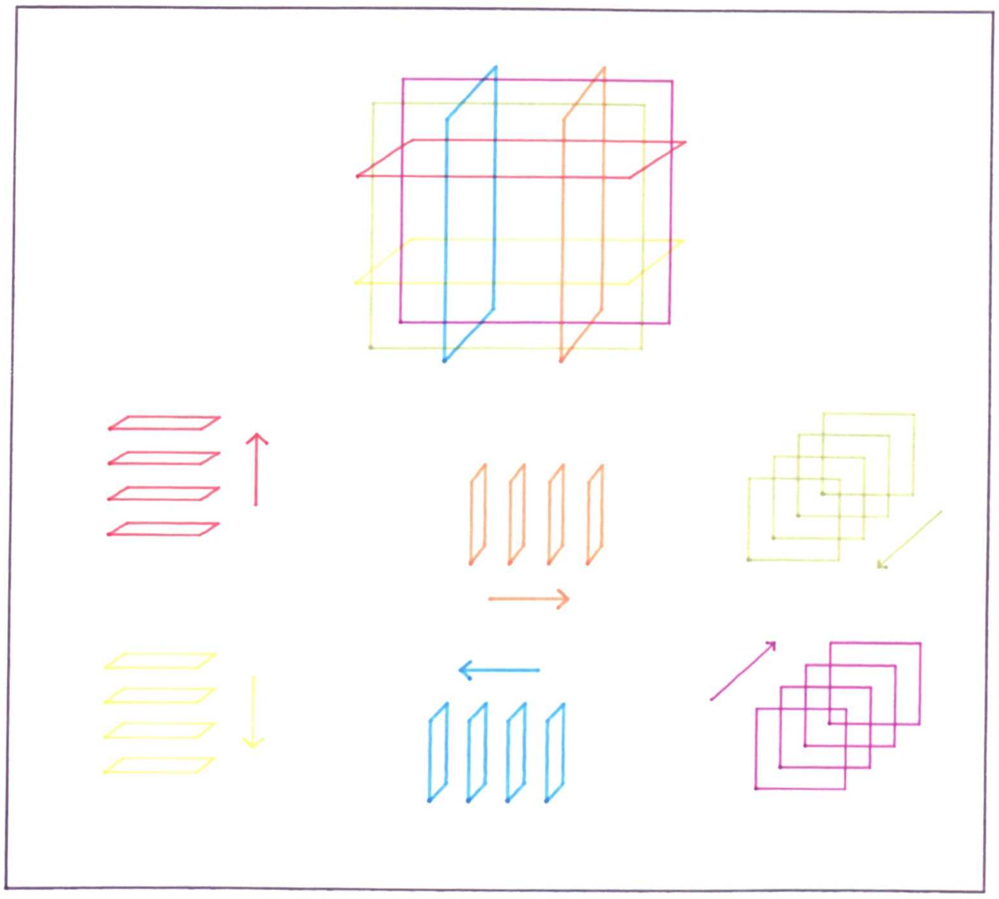
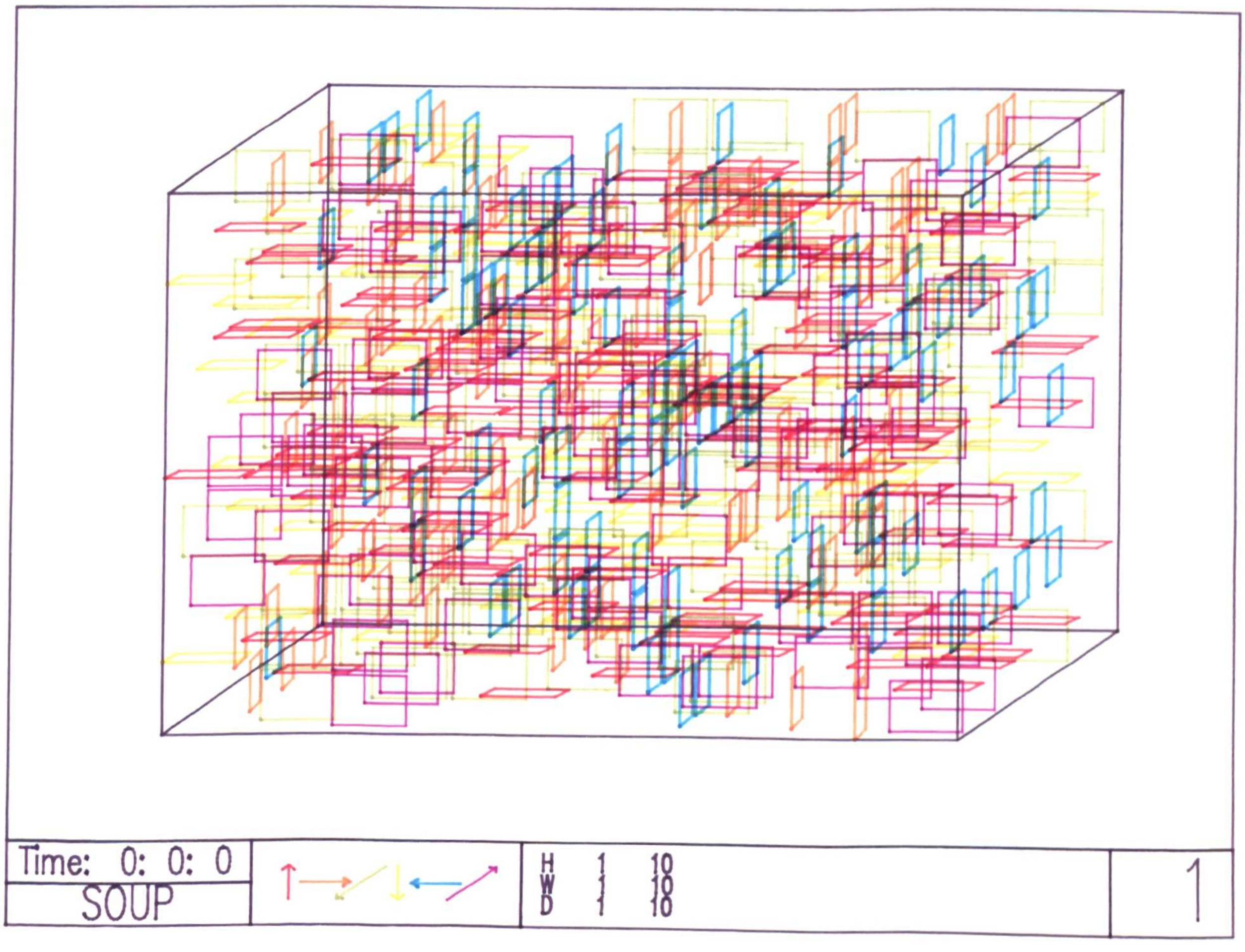


Photo of a random Soup



simulation time to the Photo being taken. In this case, the initial state, this is 0. The times are elapsed times taken using one processor on a lightly loaded Encore Multimax. (See [LEE87] for a description of the system, note that no use was made of the machines multiprocessing capability.) Percentage processor time afforded to the simulator rarely dropped below 95% thus the elapsed times can be used as indication of real processor time, hence the computational complexity of the simulations. To the right of the time is a box containing six arrows. The direction and colour of these arrows corresponds with the directional information in the diagram in the top half of this Plate. The arrows also serve to show which face-cell information is being displayed. For example, to reduce the amount of information displayed to aid illustration, it may be desired to display only the upward communicating processors, the up-arrow alone would be drawn in this box. To the right of these arrows are the height, width and depth of the region of a Soup being displayed, given in numbers of Soup-cells. The final digit at the right-hand side is a Photo index, in the eventuality of there being a sequence of Photos related to a particular simulation.

5.2 PROGRAMMING EXAMPLES

The aim of the simulation work is to illustrate ideas of previous chapters. In particular to show that it is possible to program an Architectural Soup, that it is possible to program at such a low level, and to program with the extra flexibility of a third dimension. The simulations performed include; an illustration of loading, an illustration of external reprogramming of the machines architecture, a parallel edge detection, and finally, dynamism through internally generated reprogramming. The program definitions associated with each of these examples can be seen in Appendix 3. This Appendix illustrates the level at which the machine was programmed.

5.2.1 An illustration of loading

This first example aims to show that from an initial state, with random memory elements and all face-cells containing no program, it is possible to load programs into each face-cell which clear all memory elements. (The assumption that all of the face-cells program registers can be initially forced into the empty state is inherent in the face-cell architecture design. This would be achieved through generation of a continuous stream of all-ones programs to every face-cell on the peripheries of the Soup.) The initial random state is shown by the Photo at the top of Plate 5.2 . Only the up, down, left, and right communicating processors are drawn (two arrows are missing at the bottom of the Photo). This is to make the photos clearer. What is happening in the processors drawn also happens in those not drawn.

The circuit consists of a three dimensional array, routing a zero to every face-cells memory element. The circuit description is Appendix 3.1 . When a face-cells program is input from the edge of the Soup, it will travel in a straight line along the face-cell string until reaching a face-cell that is either full or the opposite edge of the Soup. Obtaining a zero for input to a memory element requires two face-cells to be programmed, both at the same coordinate height width and depth, but in different face-cell strings. In the majority of cases it will take different times for the two face-cells programs to load to the required location. For example, one of the face cells might be input from the left-hand side and the other from the bottom. It is only when both programs arrive at the required location that both multiplexors will be enabled and the zero generated. The input to memory is undefined until

Illustration of loading: Initial random state

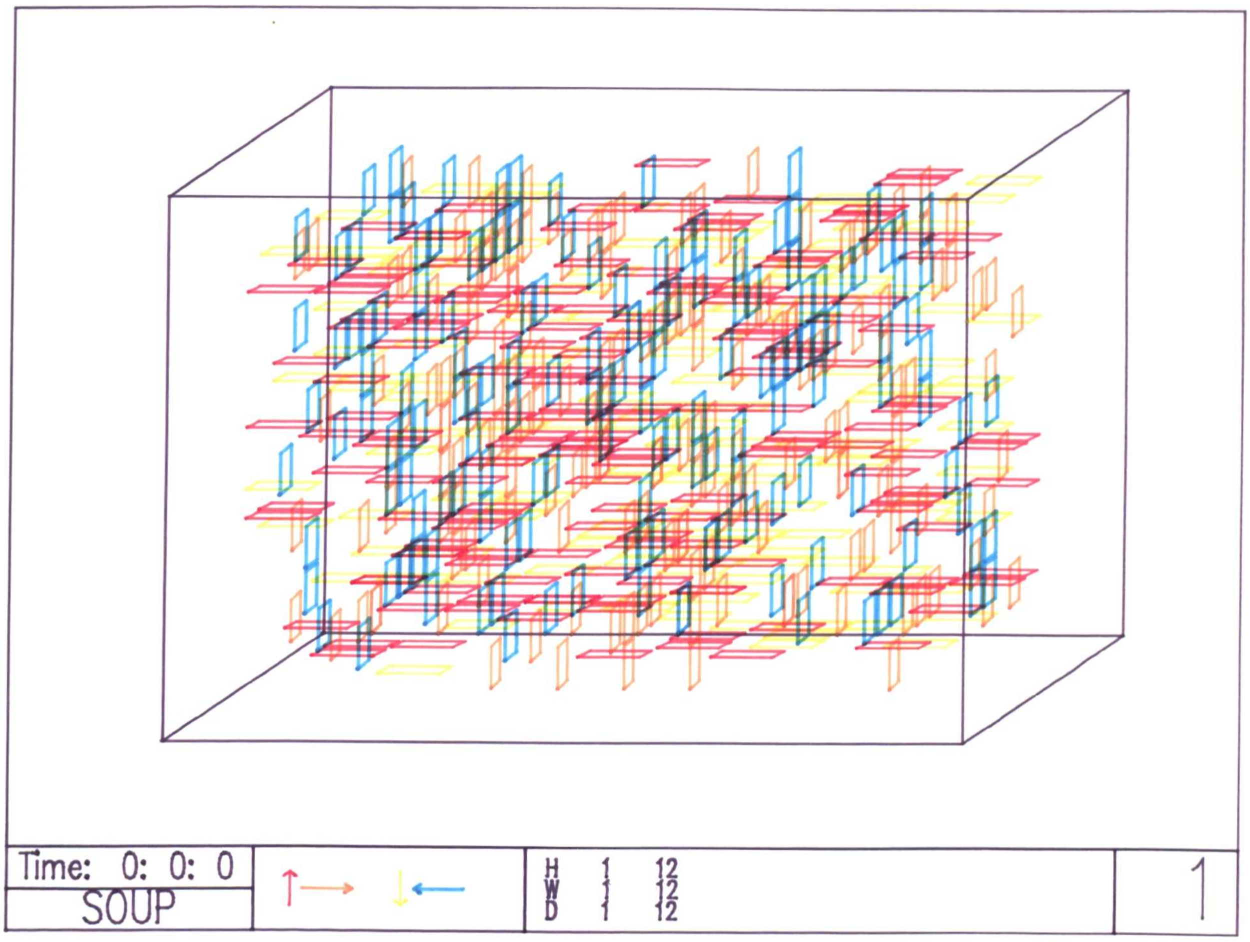
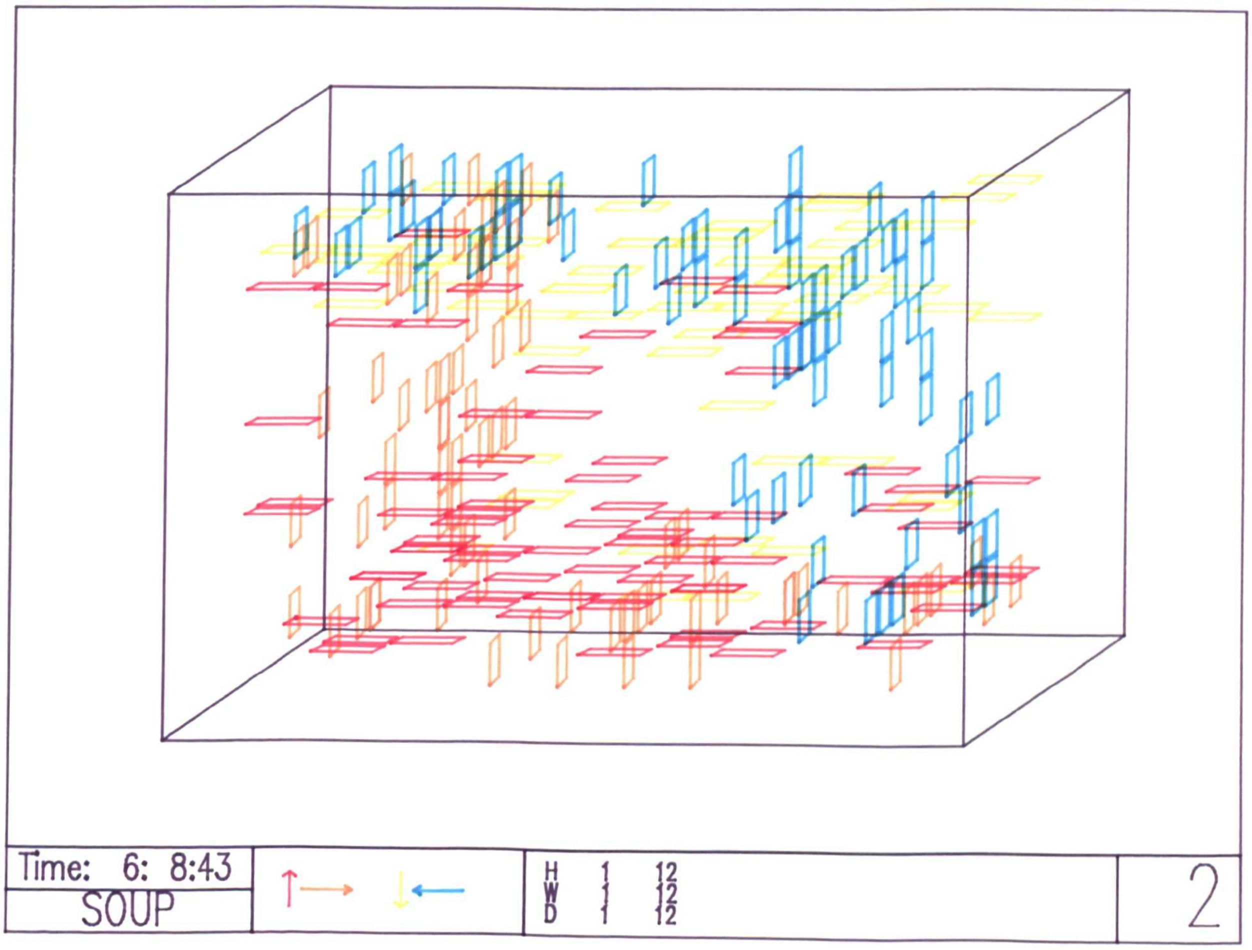


Illustration of loading: Partially completed load



this point, it will then be set to zero and the memory cleared.

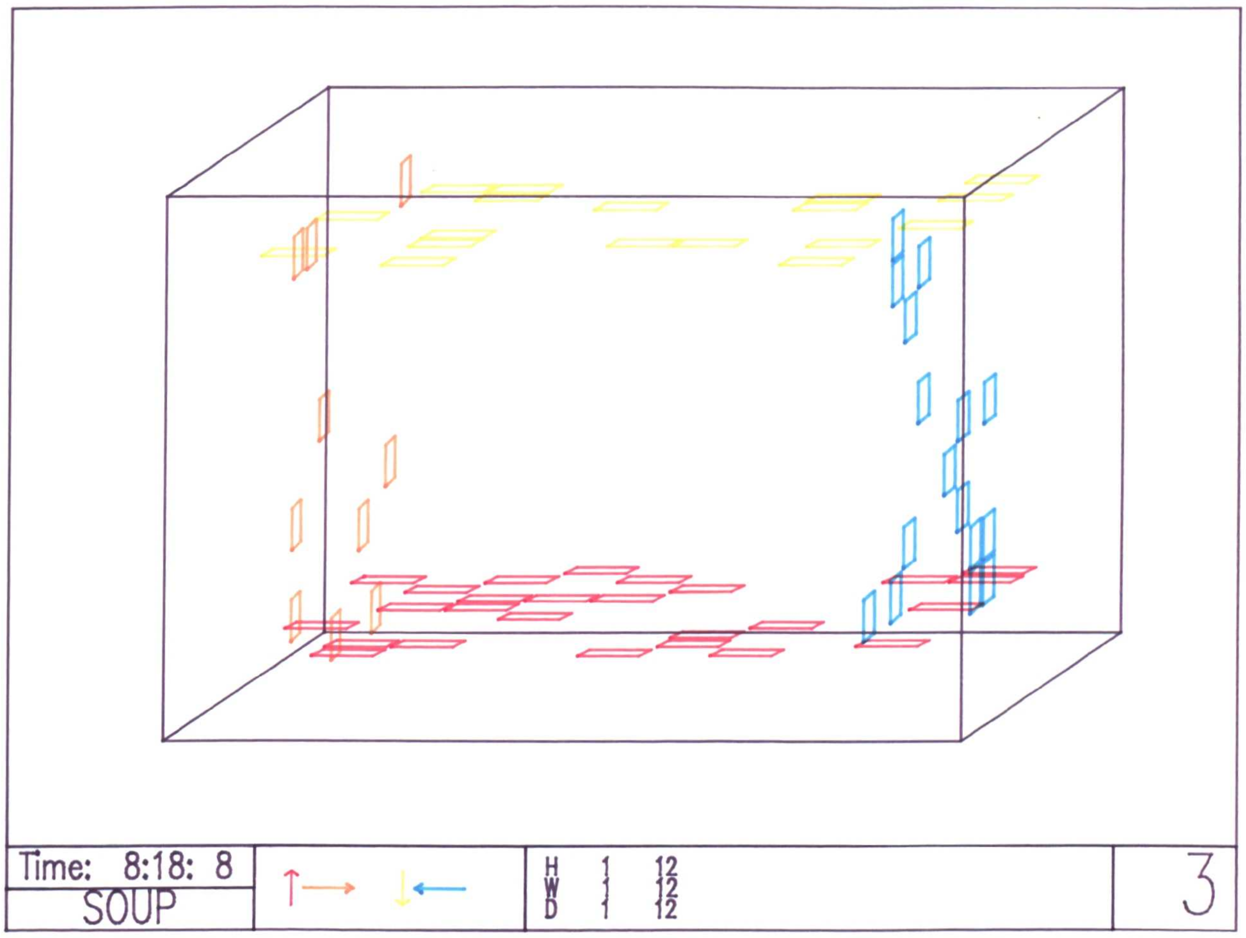
The loading process can be seen in the Photos at the bottom of Plate 5.2 and the top of Plate 5.3 . The first Photo is part way through the load, whilst the second is completed. The memory elements in the final image which still contain ones are at the start of strings ie. the face-cells have no predecessors and hence have had no input to their memory elements.

Since it has been shown that it is possible to set all of the memory elements to zero, it will be assumed that all memory elements are initially zero in the following examples. This assumption was made to decrease simulation effort.

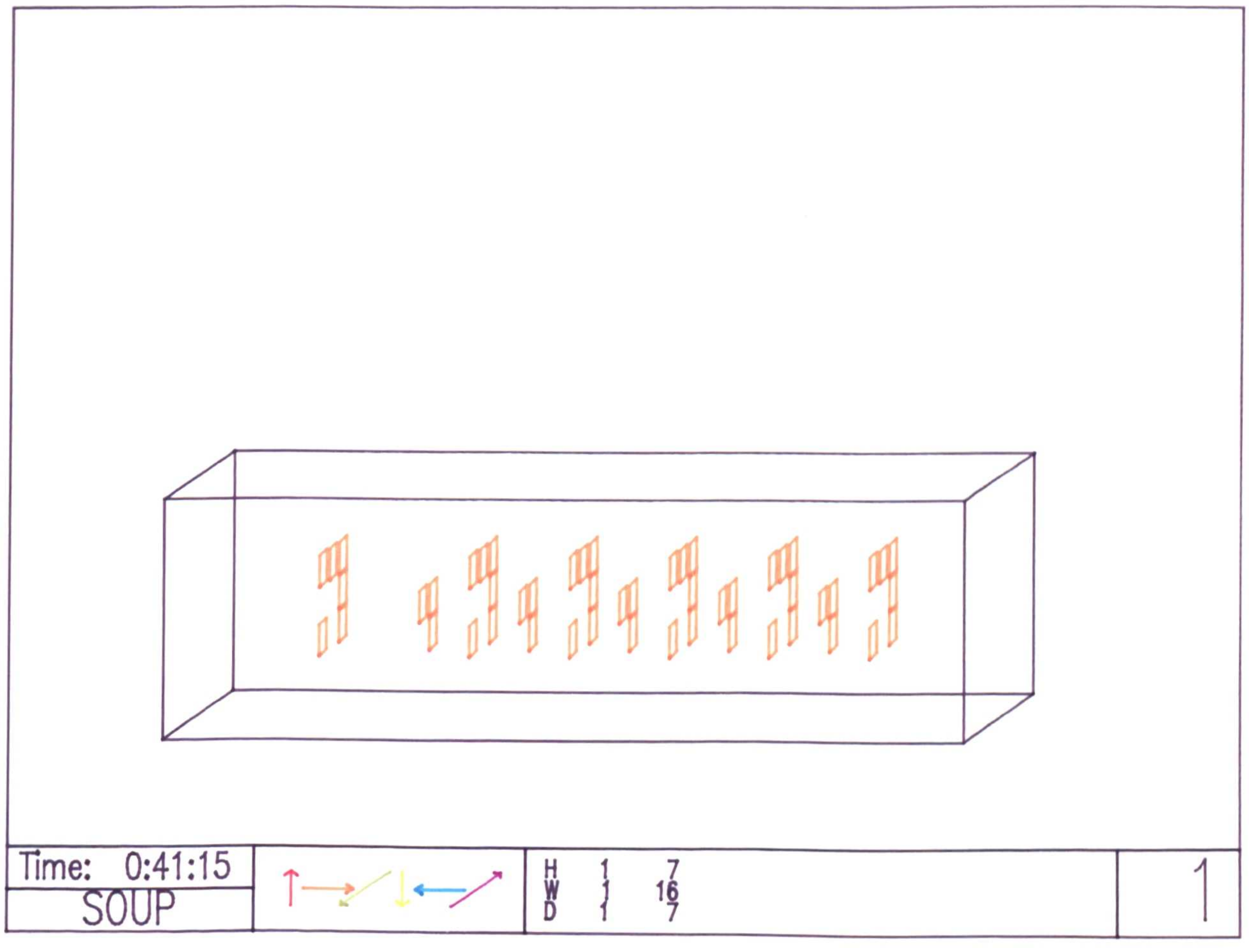
5.2.2 Inverting wires

This example aims to show two points. Firstly that a more complex circuit can be loaded and secondly to give understanding of the output format. A wire was simulated as a series of nand gates, see Figure 5.2 . It can be seen from the figure that at each stage along the wire the input will be inverted. The output from each stage is routed to the

Illustration of loading: Load complete



Inverting wires



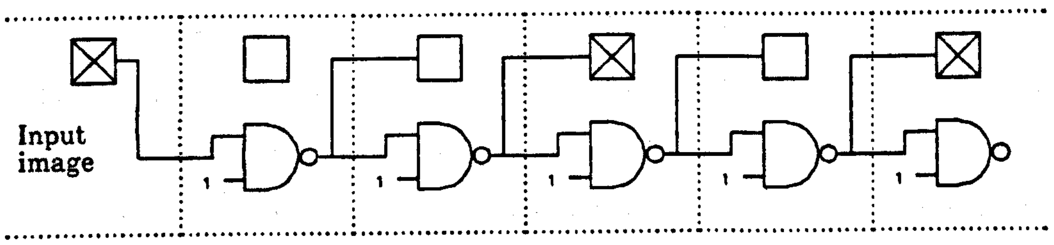


Figure 5.2: The inverting wire example

input to the next nand and also to the input of the next memory so as to display the result at each stage. Results can be seen in the lower Photo on Plate 5.3 . A 3 by 3 bit array of data was used for input. This can be seen on the left-hand side of the Photo. The 'wires' ran through the face-cell processors which communicate to the right. The input array can be seen to invert at the first stage of the wire and then invert back to the original at the second stage. In total there are five inverted stages and five non-inverted stages. The wider gap which can be observed between the input bit array and the first memory elements along the wire (on the left-hand side of the Photo) can be explained through examination of the interconnect topology. This is illustrated in figure 5.2 .

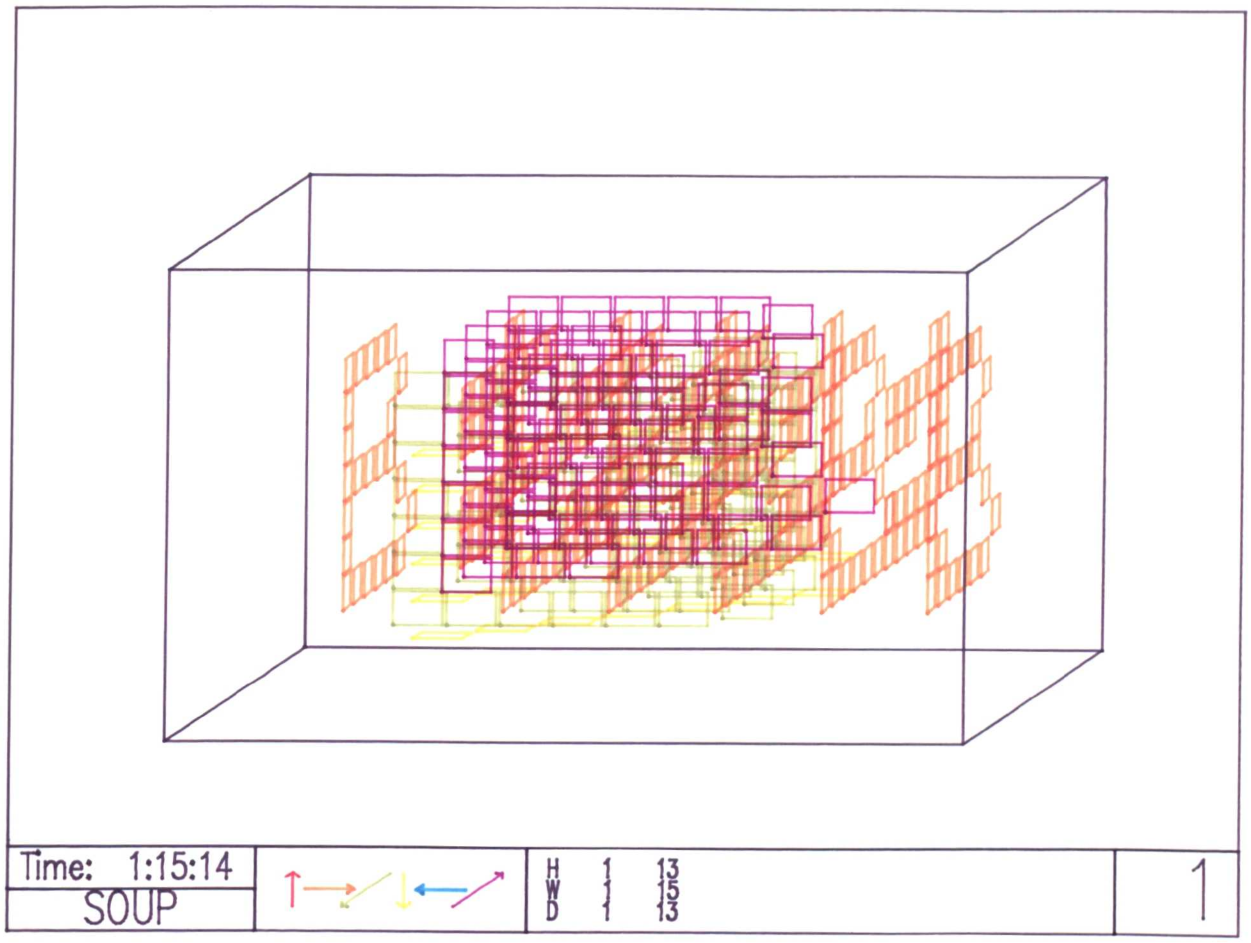
The first observation that can be made from this simulation is that it has been possible to deterministically load a specified circuit, albeit a simple one. Secondly, all of the face-cell memory elements are being drawn (all six arrows are displayed at the bottom of the Photo), and no memory cells have been displayed other than those expected, hence the circuit padding algorithm has not interfered with the display in this case. A third observation is that no multiplexor has become enabled before the programs have reached the required destination along the face-cell strings. This would have resulted in other memory elements becoming set. A significant saving in simulation effort was achieved in all of the following simulations by making the assumption that a face-cells multiplexor will not become enabled until both its predecessor and successor in the string contain a program.

5.2.3 External reprogramming

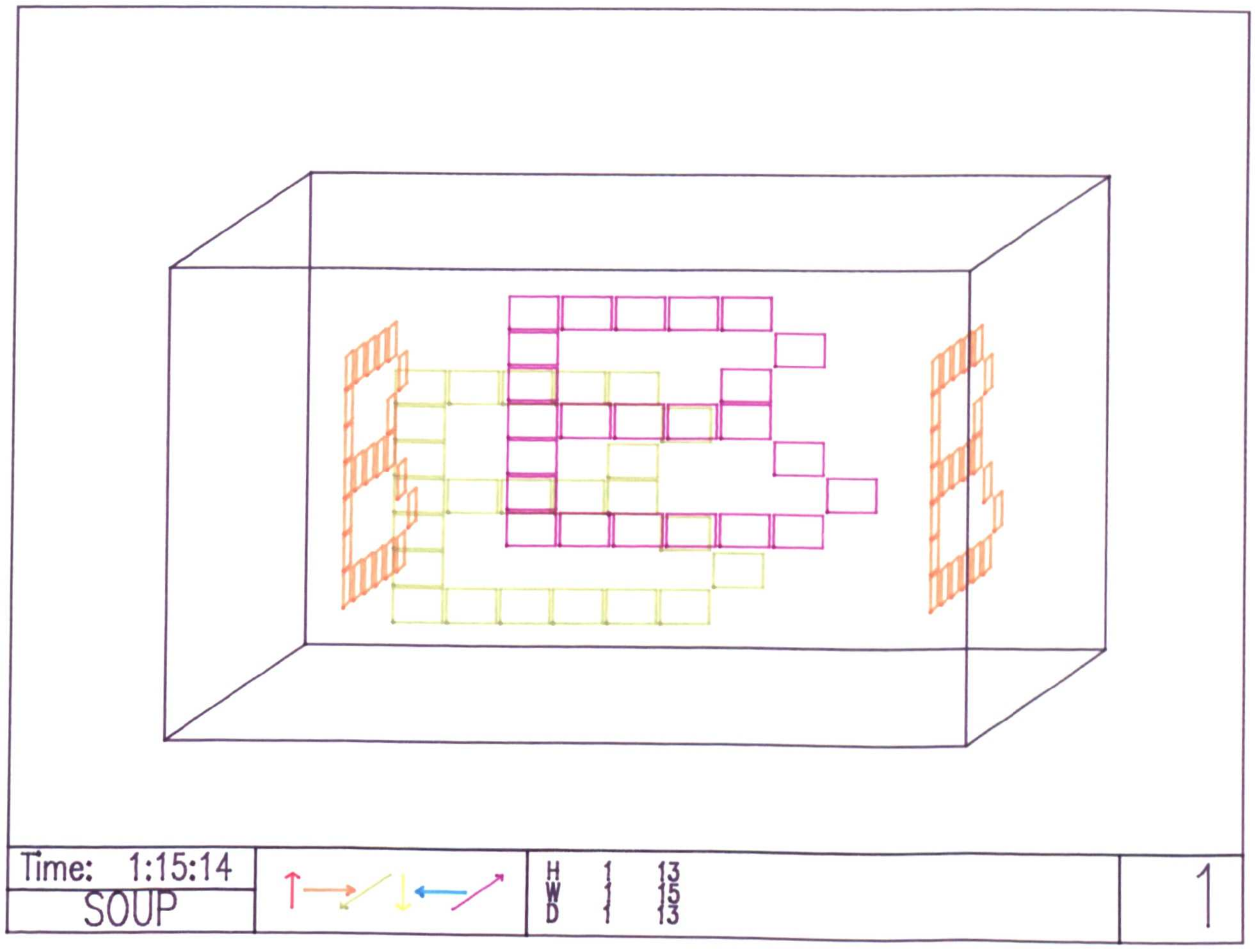
This example aims to show that it is possible to route information in directions other than a straight line, to completely reprogram the Soup externally, and also that information in memory elements is not lost during such a reprogramming operation.

The Photos in Plate 5.4 correspond to the initial circuit. The bottom Photo is drawn with the same data as the top but with the central portion removed. Essentially, this circuit

External reprogramming: 'B' copied front, back and right



Simplified version of above



is similar to the inverting wire example with the input bit array (the B on the left) being copied to the right-hand side of the Soup. Information is also routed towards the front and back face-cell strings along a diagonal in this routing area (see Figure 5.3 (a)). Information is then similarly wired to display areas at the front and back of the Soup (areas 4 and 5 in Figure 5.3 (a)).

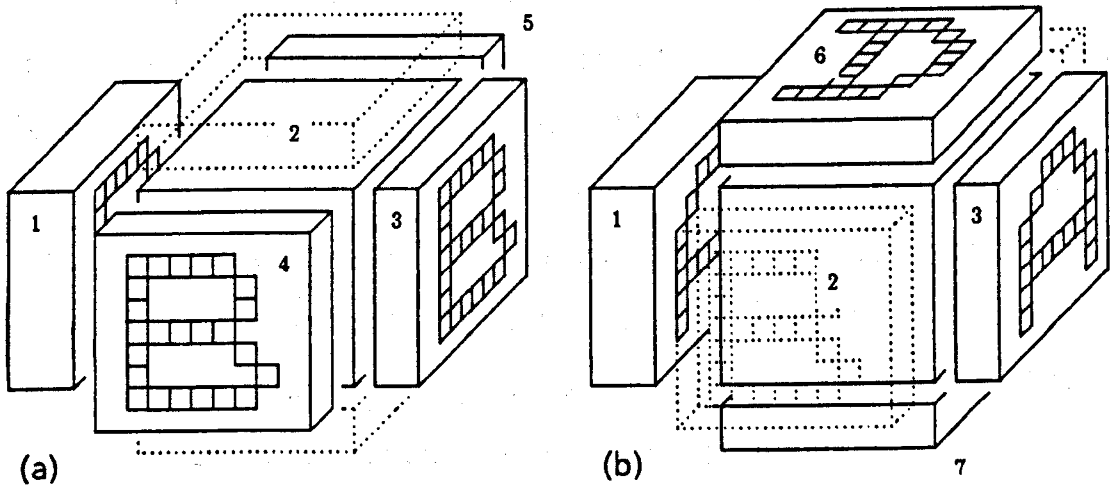


Figure 5.3: The external reprogramming example

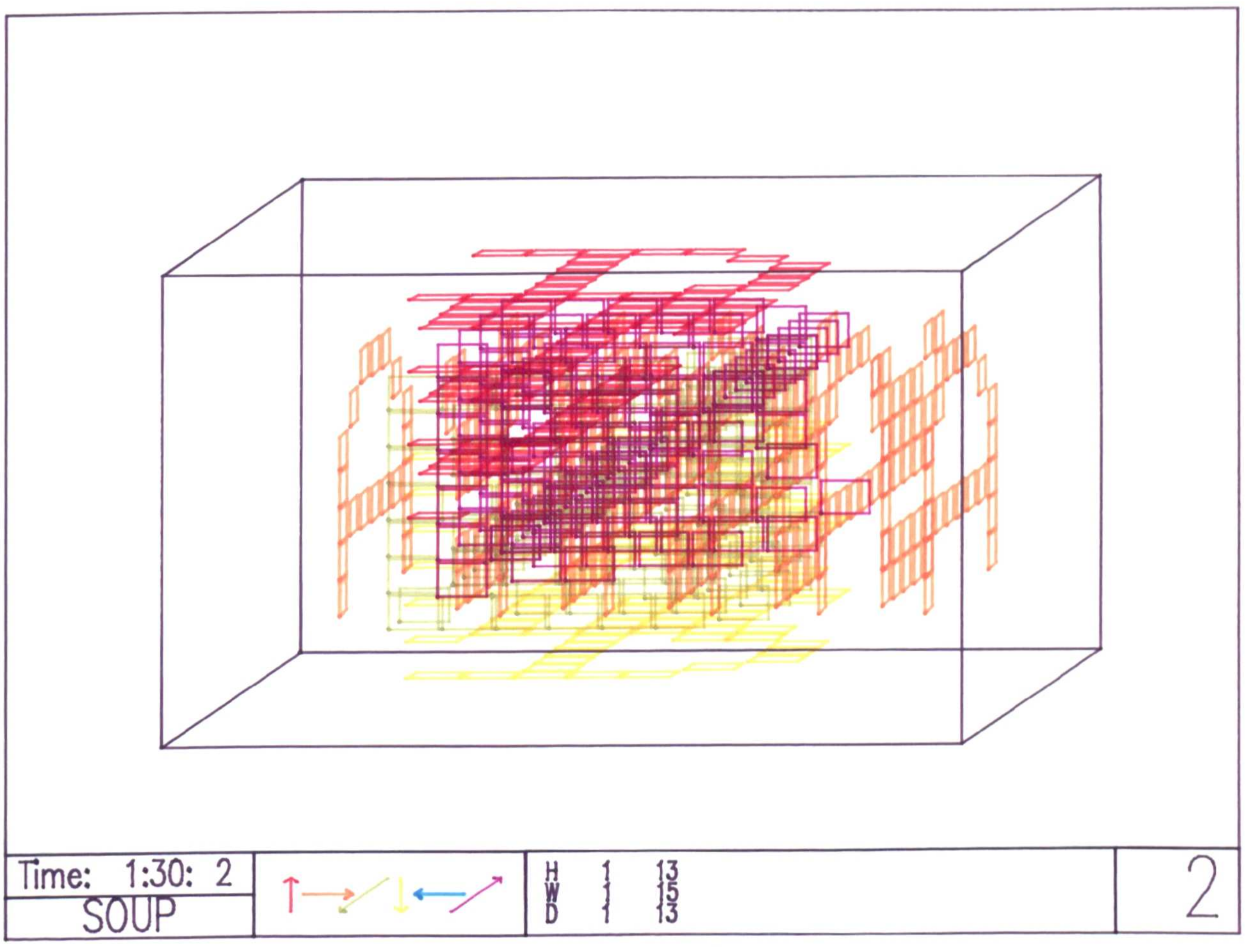
The circuit loading for this example took 15 minutes, real time, to simulate. The Soup was then emptied by placing a single 111111 program in every face-cell on the periphery of the Soup. One all-ones message is guaranteed to empty all multiplexor programs in a face-cell string as all cells are in the full state, none having been made locked. The Photos in Plate 5.4 correspond to the memory elements after the reprogramming has taken place. The memory elements have retained their contents despite all face-cells now being empty. (In the top photo in Plate 5.4 a few downward memory elements have been set. This is as a result of the padding algorithm.)

A second circuit definition was then input to the Soup. The results of this can be seen in Plate 5.5 . It is similar to the first circuit except that the bit array is now copied up and down as opposed to left and right, see Figure 5.3 (b). The B's remaining in the final image were from the initial circuit. The second circuit does not use these memory cells. Consequently they are padded to undefined inputs. This results in their contents remaining the same.

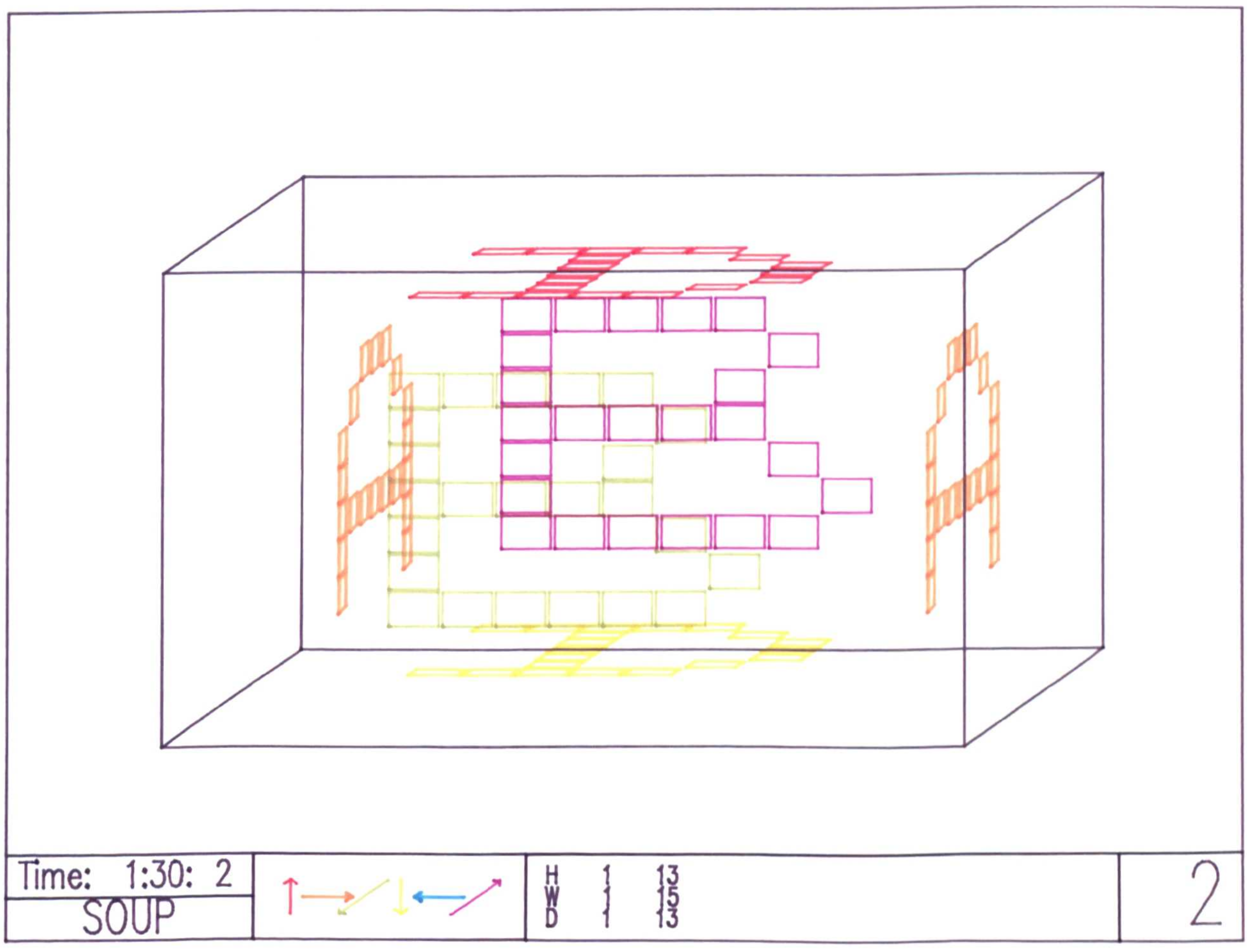
5.2.4 Edge detection

This example shows a parallel combinational logic task. The input image was a 2-D array of pixel information, a pixel being either 1 (set) or 0 (clear). The edge detection algorithm used compares each individual pixel with its four immediate neighbours. If a pixel was set and any

External reprogramming: 'A' copied up, down and right



Simplified version of above



neighbouring pixels were not set then the pixel was on an edge. The circuitry for this algorithm is in two sections.

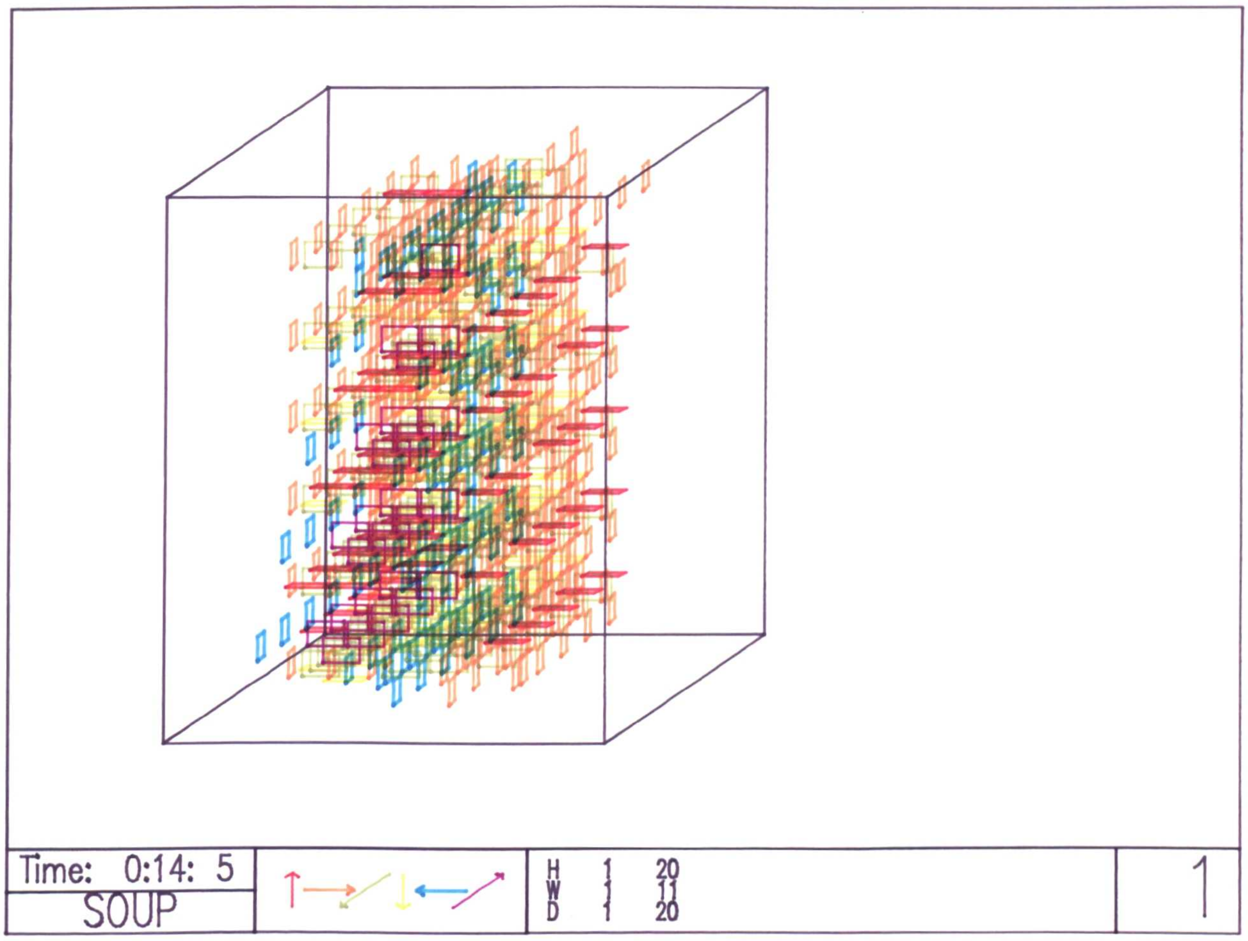
Firstly, a circuit to perform a one bit edge detection. By examining the central pixel at a location and its four neighbouring pixels, a circuit was derived to perform the above algorithm. Naming the central pixel as X and four neighbours as A, B, C and D, the following gives the result 1 if X is on an edge :-

(X nand (A nand B)) nand (X nand (C nand D))

Placing one such circuit for each pixel of the input image, the second problem is to route the pixels pertaining to the input image to the appropriate inputs of these one bit edge detectors. The routing required can be seen at the right of Figure 5.4 . If a pixel was on the outside boundary of the image, a zero would be routed in place of the missing neighbour information. The resultant circuitry for a one bit edge detector was 3 Soup cells high by 3 wide. The input and output images were therefore spaced accordingly.

Results can be seen in Plate 5.6 and Plate 5.7 . Plate 5.6 shows a simple triangle edge detection. The bottom Photo was drawn utilising the same information used for the top Photo. The central information is removed to show the input and output images only. Despite the smallness of this image, it

Edge detection: Circuit



Edge detection: Extraction of input and result

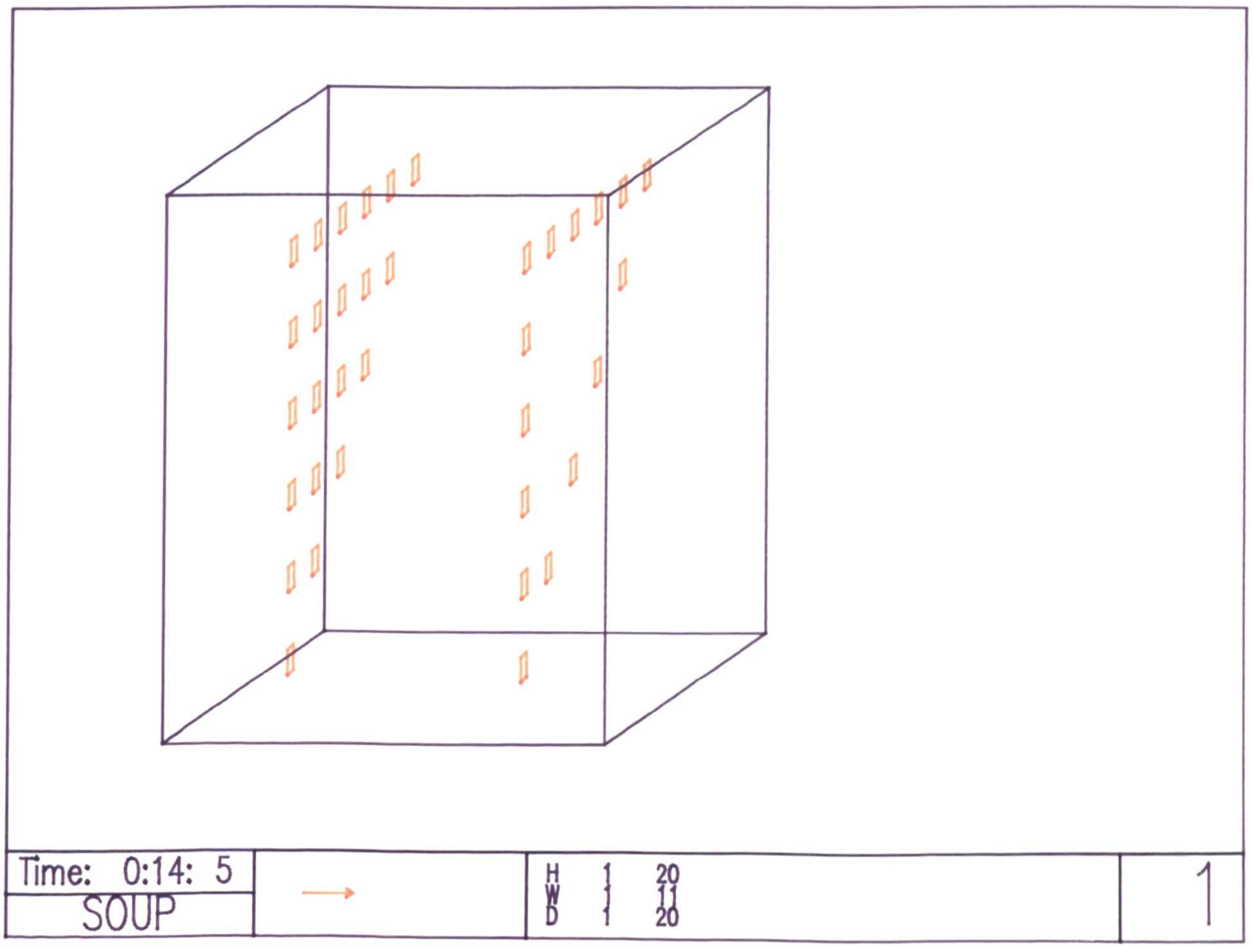
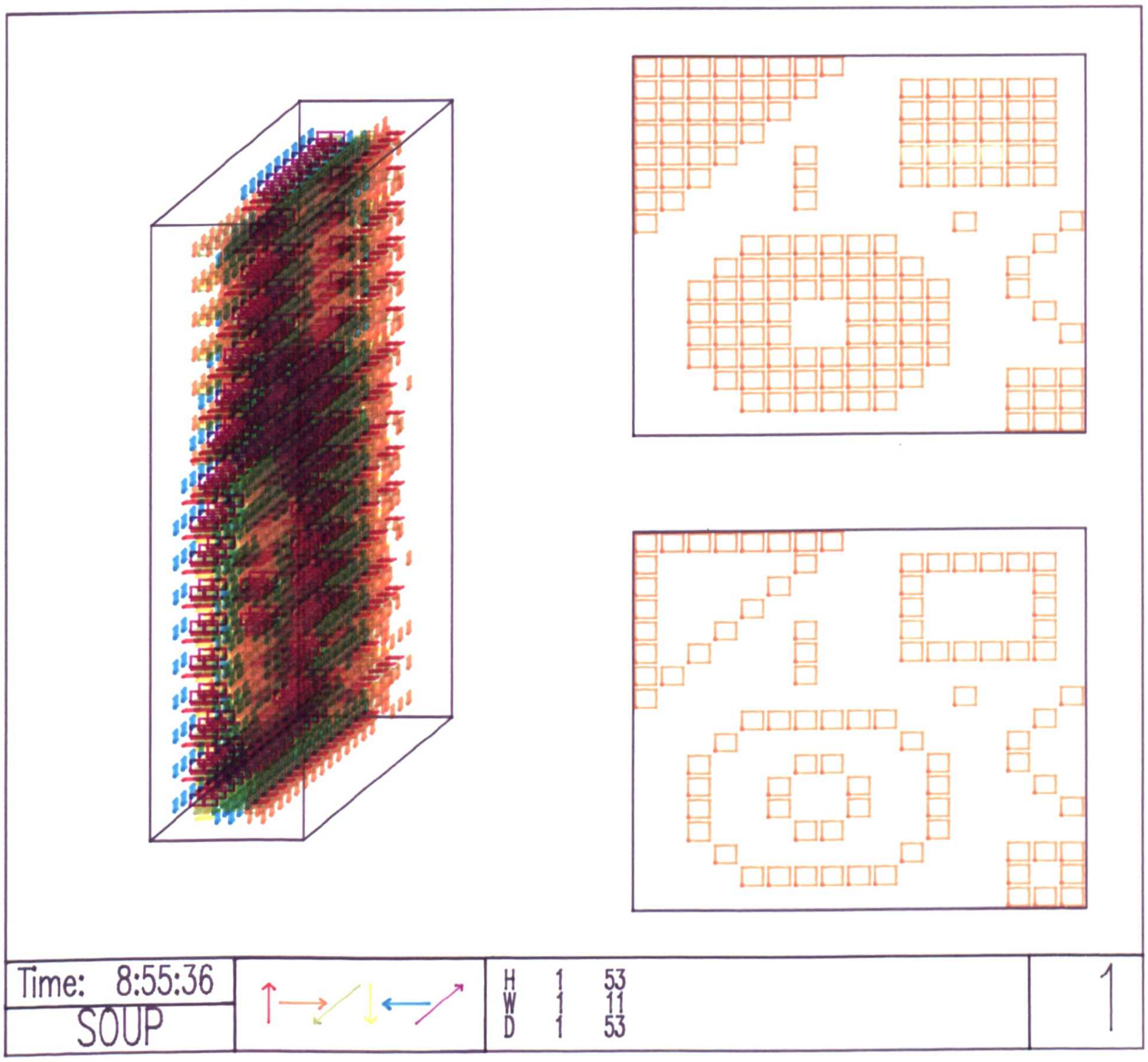
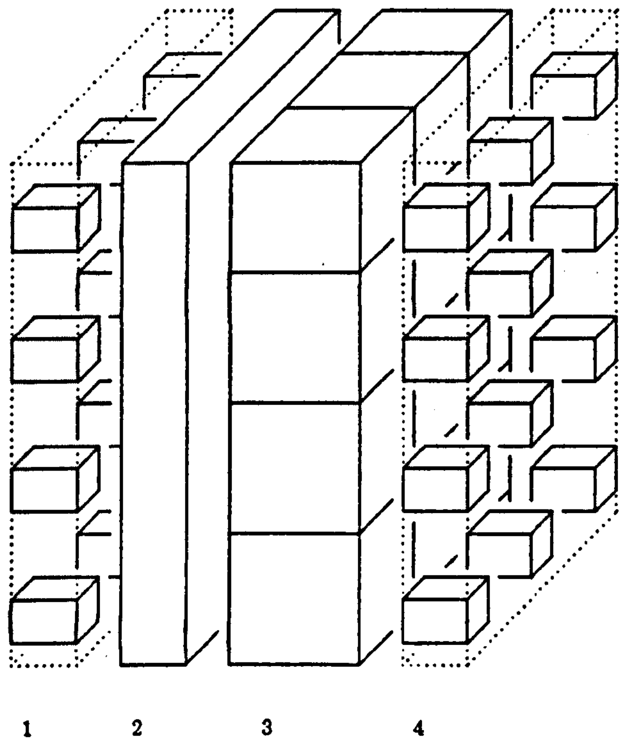
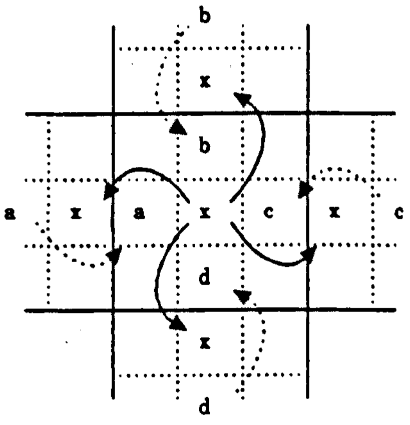


Plate 5.7

Edge detection: A more complex image







Pixel routing for a 1 bit edge detector

1. Input image
2. Routing circuitry
3. 1 bit edge detectors
4. Output image

Figure 5.4: The edge detection example

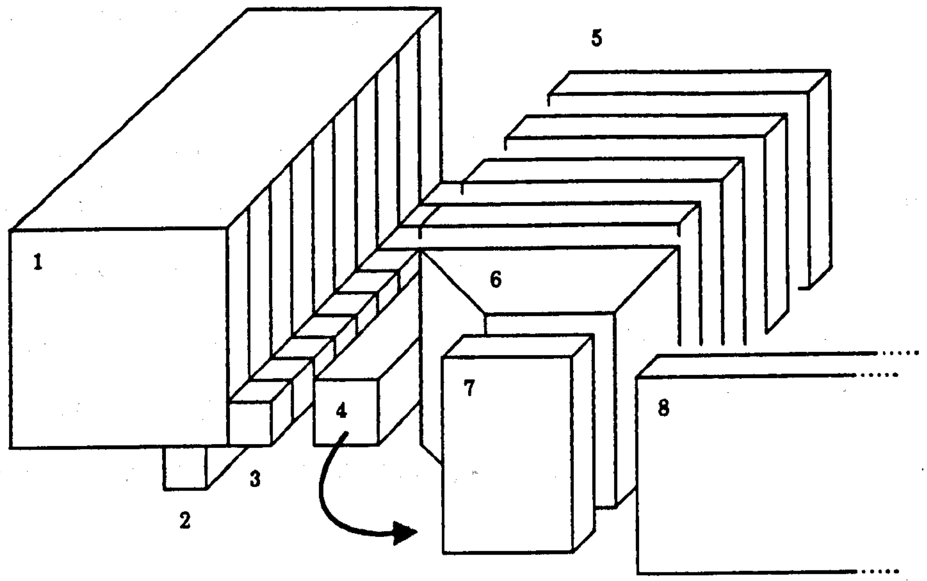
is hard to interpret the top Photo. This is exacerbated by the image size in Plate 5.7 . The two sub-frames in Plate 5.7 were obtained by reading data out of the bit-array used to plot the circuit on the left of the Photo. Note that the width of the circuit array is the same in both Plate 5.6 and Plate 5.7. As only height and depth change for this circuit, circuit size increases in direct proportion to image size. It is important that if such an algorithm were to be used on a Soup, it would show no discrimination in terms of speed for large or small images. (Note that there is a higher degree of padding interference this example.)

As an aside, although the examples used in this section involved static images, they could in theory be dynamic, with the output edge detected image changing as the input image changed. The algorithm would need to be examined for spikes due to this asynchronous behaviour.

5.2.5 Internal Reprogrammer

A final example shows an internally generated reprogramming, the use of locks, and the use of a counter. The circuit consisted of a counter selecting programs from a 6 bit wide memory, and using these programs to reprogram an area of the Soup. The architecture can be seen in Figure 5.5 .

The counter consists of a series of delay lines comprising long chains of memory elements. An assumption made is that all memory elements will initially contain zero. A one placed at the start of the chain will travel through each memory element. Examining this chain at regular intervals gives a crude counter, with all of the bits below a point set to 1, and all of those above it to 0. This is the basis of the register in Figure 5.5. The counter can be made to



1. Delay lines
2. Feedback path
3. Register
4. Register decode (store selection and programmer signals)
5. Program stores
6. Store routing
7. Programmer cells
8. Display area

Figure 5.5: The internal reprogrammer example

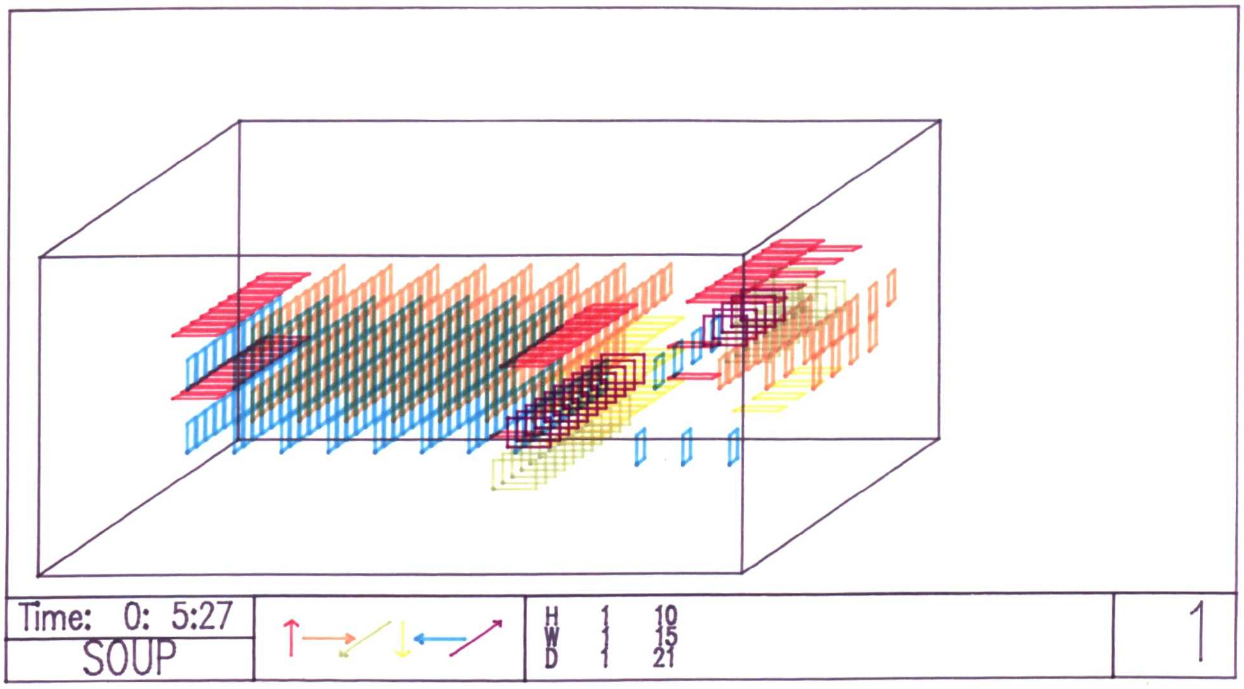
reset by taking the output signal from the far end of the chain, inverting it, and feeding it back to the input. This counter-register can be decoded to act as a memory selector and to send an enable signal to the programmer cells when the memory is being read. The algorithm used was that if bit 2n is set and bit 2n+2 is clear, then the memory plane at column n is selected. If bit 2n+1 is set and bit 2n+2 is clear, then the memory is assumed to have been read. As a result a programmer enable signal is sent to generate the required programming event.

The top Photo on Plate 5.8 shows a small counter and associated register decode. The regular component making up the majority of this image is the delay line counter. The line of green squares at the bottom is the feedback path. The less regular component to the right of the Photo is the register decode circuitry.

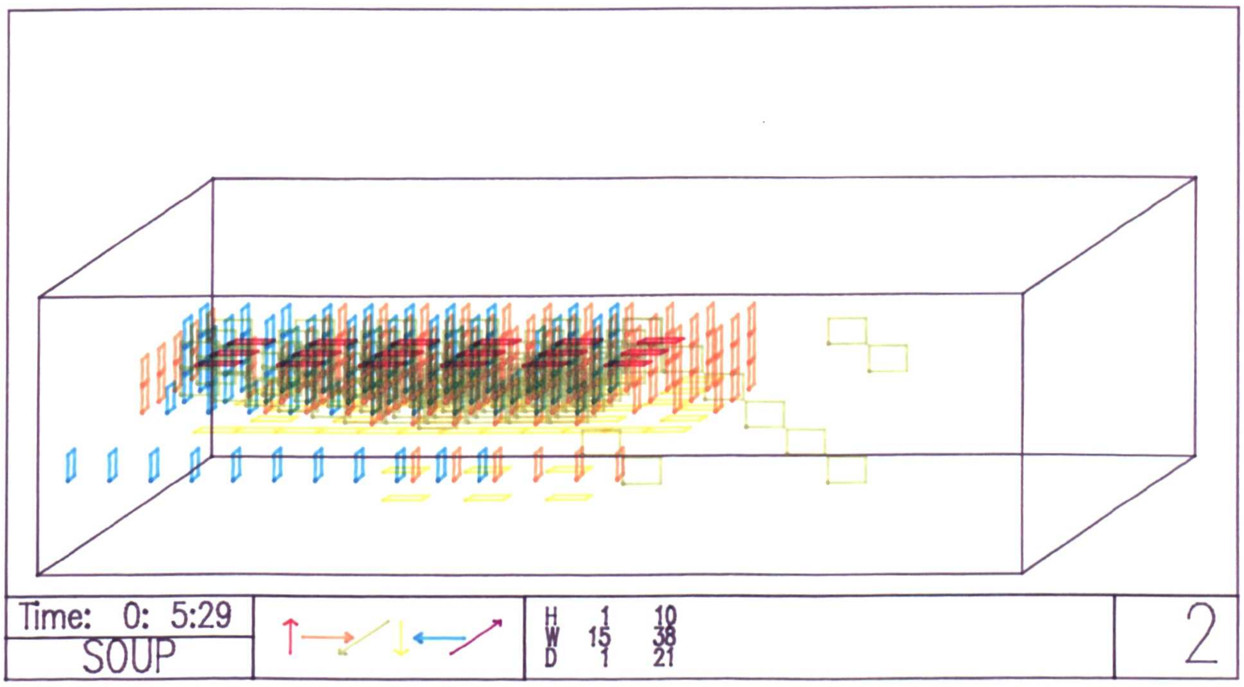
In order to illustrate the reprogramming, a series of programs were stored in memory, which, if programmed in correct sequence, would send either a 1 or a 0 to the cell in next dimension. This data is used to build up an image in order to illustrate the reprogramming progression. In this case the ones and zeros were generated by the face-cells which communicate to the right, and the face-cells in the next dimension to these are those at the front. A 111111 message was initially generated to empty the cells up to the next locked cell, making space for the image circuit.

The middle Photo on Plate 5.8 illustrates the 6 bit wide memory which can be seen at the back left-hand side. (The padding algorithm has made interpretation difficult.) In front of this is an area of green, which corresponds to the program memory store and associated routing. The implementation used for a 1 bit memory circuit was 2

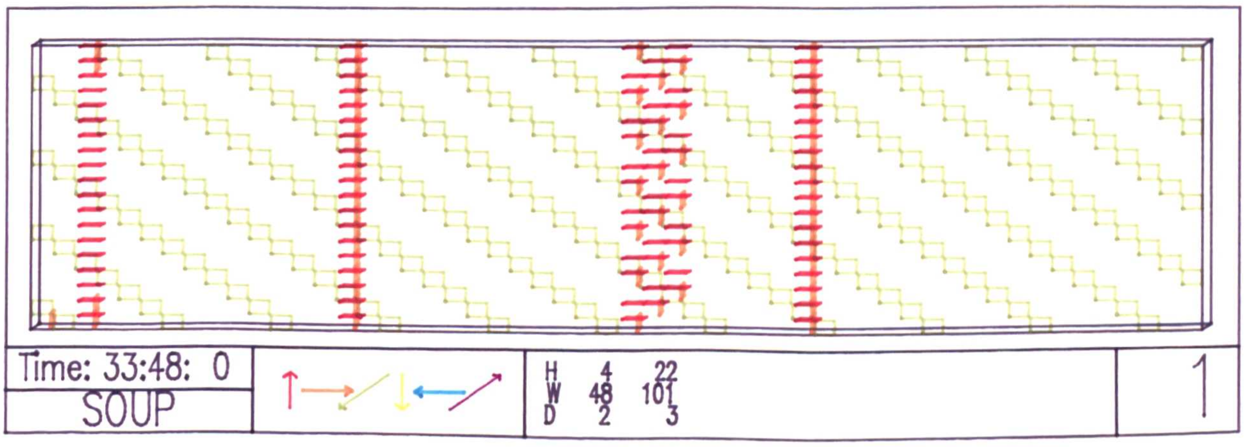
Internal reprogrammer: Part A; delay line counter and decode



Internal reprogrammer: Part B; program memory and routing



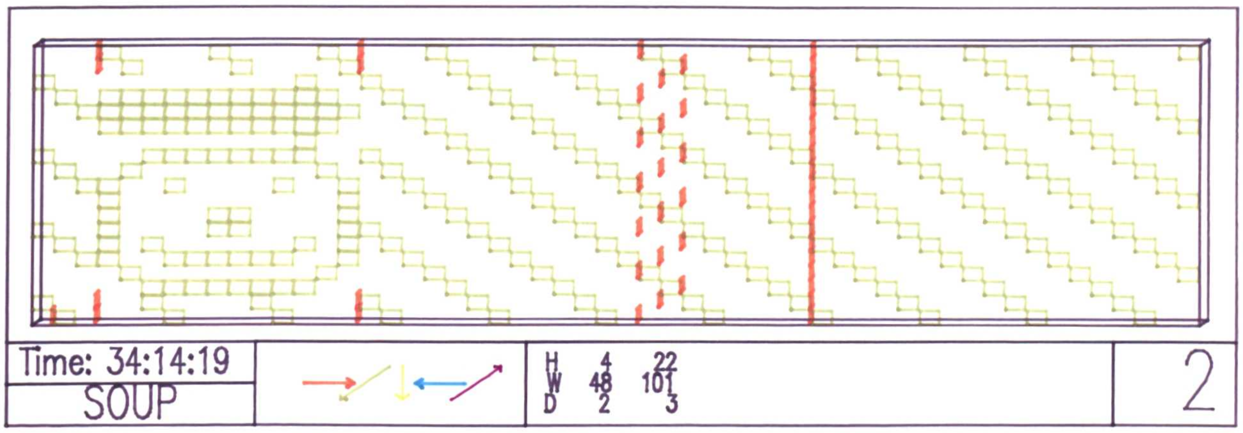
Internal reprogrammer: Part C; display area and locked cells

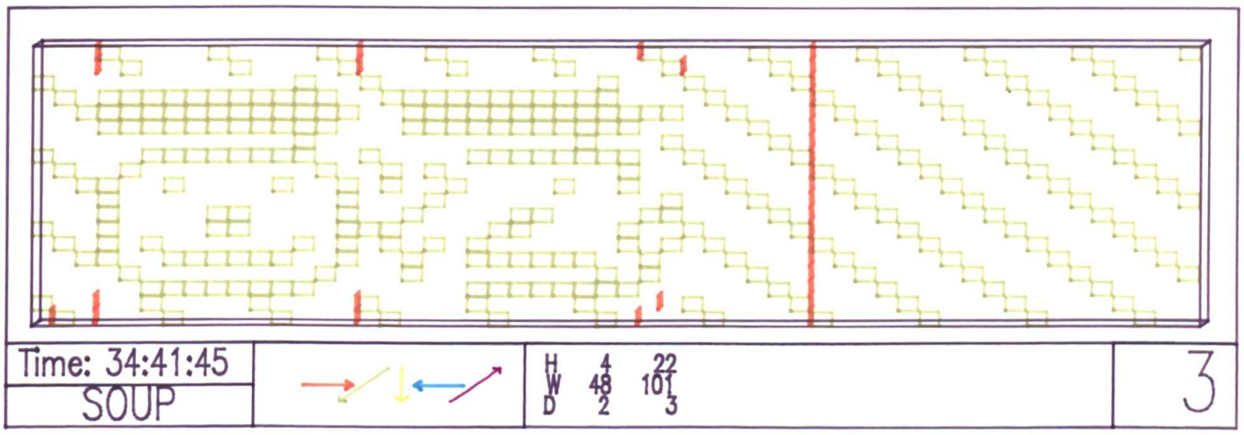


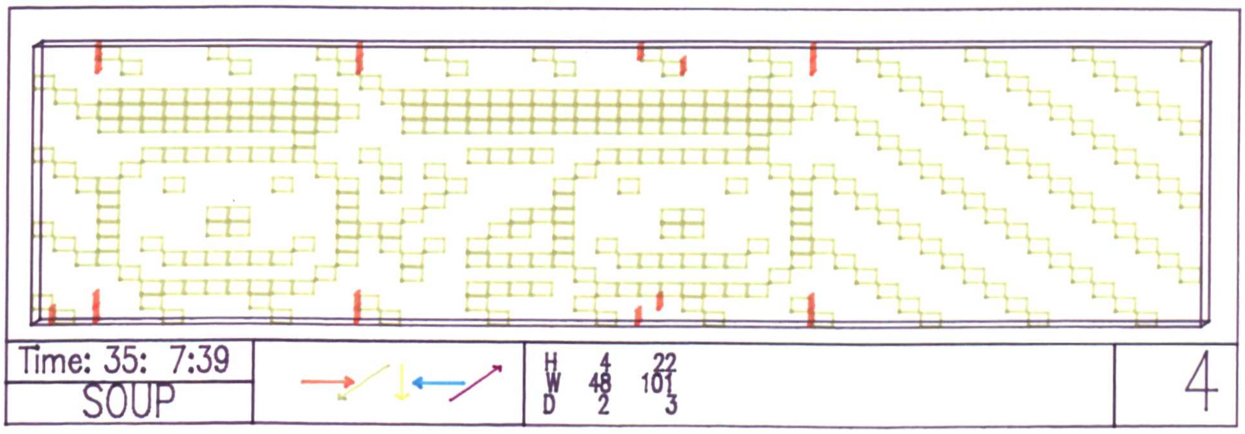
Soup-cells wide and so required some routing to connect the outputs to 6 consecutive Soup cells for the programmer cells. The path of the programmer enable signal can be seen as the line of blue cells in the foreground. Note that the signal actually travels in the opposite direction to the memory cells highlighted. These memory elements have only been set as a result of the padding algorithm. The left edge of the display area can be seen as the series of diagonal green lines on the right hand side of the Photo. The diagonal lines were the initial pattern chosen.

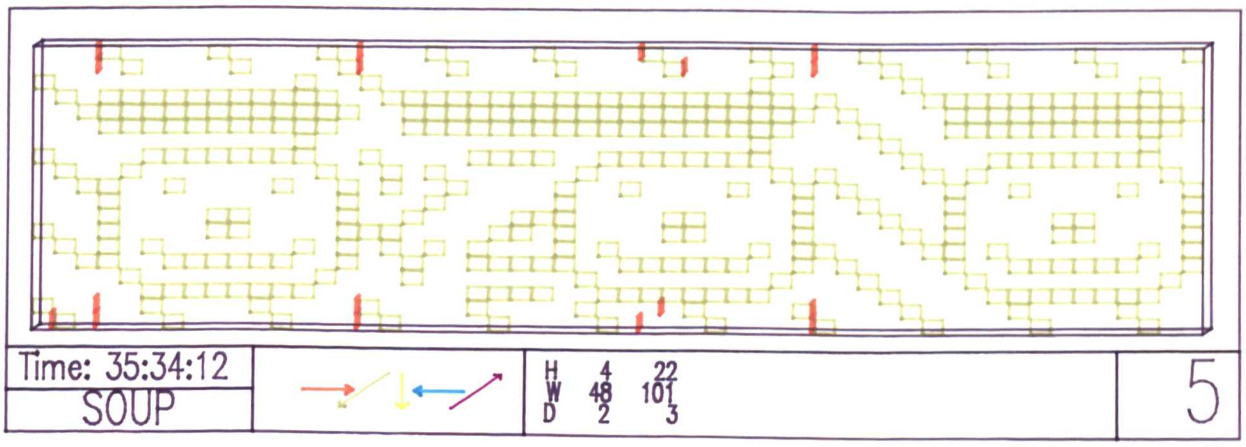
The circuit was executed for a larger data set, a 12 by 15 bit-array consisting of a smiling face and an arrow. The initial pattern of the display area is shown in the bottom Photo on Plate 5.8. The orange cells are the locked cells, the red cells are the bits set in memory from the previous dimension in order to set the lock. There was initially a line of locks on the far left-hand side of the image, but these were removed by an initial generation of a 111111 message. Next, there is a straight line of locks. The generation of an all-ones message on the right-communicating face-cells will travel as far as this lock, clear the lock, and then stop. On one full cycle through the memory all bits of the image are generated in sequence. Each will travel to the right until the multiplexor of the face-cell following is full. The resultant display can be seen in the top Photo on Plate 5.9 .

Internal reprogrammer: Stages of a repeated reprogramming action









The next band of locks are not a straight line, and so the image will be harder to understand. This can be seen in the second Photo in Plate 5.9 (numbered 3). Note that the multiplexor programs have passed through the face-cells which were used to generate the first image, as these face-cell multiplexors are now empty. The fourth band of locks are again in a straight line, thus a similar image to that in the top Photo of Plate 5.9 is generated. Here the image has partially overlapped the previous image, so some memory cells have been reprogrammed for a second time.

There are now no locks remaining, so further generation of all-ones messages will result in the emptying of the face-cells up to the edge of the Soup. This can be seen in the bottom Photo of Plate 5.9. The circuit will count ad infinitum and so keep programming the appropriate code for the image. These programs will keep travelling to the far edge of the Soup but will display the same image. The display will no longer change.

5.3 AN ASSESSMENT OF THE SIMULATION ENVIRONMENT

It has been shown that it was possible to design circuits using the Pascal procedure call method described. For the more complex circuits it was necessary to design in small sections to ensure each component functioned as expected. The computation effort required for the circuit description stage was not high for these (small) examples. It is suggested that more time could reasonably be afforded to better padding algorithms. The padding algorithm used was simple but proved effective for most of the examples. A more advanced algorithm could examine the surrounding circuitry in detail to make more use of the undefined signals available. Such padding would be aided by the availability of more undefined signals in the face-cell multiplexor architecture.

The pre-processor for the simulator typically required more computation effort than the circuit description stage due to the quantity of input and output involved. The size of the output file was typically an order of magnitude larger than the size of the circuit description array. (The sizes of both could be reduced by using a more compact data format.) Note that the preprocessor is performing a simple repetitive task which is a good candidate for parallel programming.

The simulator was capable of simulating a small Soup but was slow, which was not surprising considering the speed of two-dimensional VLSI simulators written in a similar manner. An avenue for speed improvement is in the insertion of events into the event list. A profiling of the simulator revealed that 95% of the computation effort was spent on event list manipulation. This is the expected result for such a low-level simulator written using a single event list. The events themselves are simple and so require little effort to perform, whereas maintenance of a long and ordered event list requires more effort. The simulator reduced effort by only generating new events in a local environment to any events which have generated change, not over the whole machine. This is of undoubted benefit towards the end of a simulation, when the number of events is much smaller than the total number of multiplexors in the machine. Large event lists occur initially when the machine is loaded, however, which suggest that it might be beneficial to test and perform the events for every cell in the machine, in order to avoid the cost incurred in maintaining this list. Another possibility for increasing the simulator efficiency may be to partially sort an event list by locality with the aim of reducing any high paging demands.

The simulation is a highly parallel task. A method of implementing a parallel version would be to consider the Soup as a collection of sub-Soups and simulate each one on a different processor. There would be a high degree of inter-processor communication for the information at the boundaries of each simulated sub-Soup but this should be local to the processors of the simulating machine. Due to the small size of the Soup architecture processing elements the processors of the simulator would need to be tightly coupled with respect to their simulation clocks in order to allow for all interaction between neighbouring sub-Soups to complete. If the majority of calculation becomes localised to one or two processors, a large number of processors will become idle for a large percentage of the time. In this case the parallel version would only benefit in reducing initialisation work.

The graphical output is a significant improvement on an earlier display routine, which had attempted to draw a true three dimensional circuit diagram. This diagram rapidly became unintelligible as circuit size was increased. Even this new version is difficult to understand for much more than a 10 by 10 by 10 circuit unless the application circuitry has regular structure. The graphical output photos were static. A real-time display of the memory elements as they changed was considered. This would give more insight into the behaviour of an application but would require significant computation to perform. Possible benefit of a display of this form would be as a debugger in synchronisation problems, to identify areas where values in memory are changing and areas where they are not, hence, which components of the application circuit appear to be functioning.

5.4 AN ASSESSMENT OF THE ARCHITECTURE IN LIGHT OF SIMULATION

The simulations have shown that from an initial random state the architecture can be externally programmed (inverting wire and edge detection), externally reprogrammed (the A and B example), and internally reprogrammed in a limited capacity (smiling face example). A sequential task can be seen as the counter in the internal reprogramming example. A parallel task is the edge detection. The display area in the internal reprogramming example is an example of different tasks being executed in different dimensions. On these grounds it has been shown to be possible to program an Architectural Soup, and to program with the extra flexibility of a third dimension. The three dimensional programming of this architecture proved less difficult than had been anticipated. (Once a set of circuits providing the required primitive function have been designed, the application circuit design task need not be at such a low level. Application circuits could be created by generating the required interconnection between a selection of appropriate primitives, similar to a program calling standard library routines.) Some of the factors not shown by these simulations include several independent tasks in progress at the same time, the ability to dynamically move such tasks around a Soup, and surface interface considerations.

The programmability of the face-cells was made more complex as a result of the lack of orthogonality in the multiplexor. For example, the output to-next-dimension could not be the same as the output to-next-memory. Programming the nand gate was often confusing as the two inputs come from different face-cells in different Soup cells. A possible simplification would be to take the from-other-dimension input from the same face-cell as the other nand gate input.

Cells locking whilst loading in the internal reprogrammer was a particular problem for this prevented the external load from completing as expected. A suggested solution is to connect to a common enable all those cells which are to be locked together, and make the generation of this enable from the last face-cell programs to be loaded. Similarly, components of circuits starting execution whilst other parts are still loading (especially circuits containing counters) will be error prone. This suggests the size of any equichronic region should be smaller than a counter, so preventing circuits starting early by means of the equichronic region communications protocol.

The circuit design method can subsequently lead to inefficient circuit design as much of the Soups multiplexor capacity may be unused. Some form of algorithmic circuit compression would result in circuits utilising fewer Soup cells. A detrimental effect would be increased difficulty in identifying what was going on in a circuit if it were compressed. This is similar to trying to understand the code produced by an optimising compiler.

5.5 SOME SUGGESTED FURTHER DESIGN AND SIMULATION WORK

One particular area requiring examination is that of circuits which dynamically change their configuration during computation depending on the course the computation is taking. (This topic was discussed in section 2.3.3 .) This could not be performed using the architecture illustrated due to the mobility design fault discussed in 4.5 . A suggested illustration for a suitably designed Soup would be a dynamic stack which claimed and released Soup-cells as demands on the stack varied. Synchronisation issues also need examination. I suggest implementation of self-timed protocols, such as described in [YAK85] and the Muller C-element [MEA80] [YAK85] as illustration. Another area not examined was multitasking. The architecture illustrated was not appropriate for several independent tasks that executed simultaneously, especially if they started and finished at different time, since loading a new circuit would interfere with any existing circuit. Some form of simulation of a multitasking environment or of several dynamic stacks competing for space would act as illustration.

There has been no examination of input and output issues from the machine. These will be technology dependent. However, simulation work could perhaps give insight into the forms of real-world interface which might be favoured. Particularly useful would be an examination of the trade-off between the time it takes a circuit to load and the time it takes to execute.

Given an improved architecture in light of the above, one aim would be to illustrate a non-trivial task, such as the implementation of a functional language. A functional language would be an interesting candidate due to the parallelism inherent, and also because of the small grain size of this parallelism [JON87]. [JON87] examines current research work in parallel implementations of functional languages. The implementation problem is identified as ensuring a system is feasible to program, that it is highly concurrent, and that communication is minimised. Concurrency is inherent in the Soup architecture in that the machine is highly parallel. The ability to exploit the parallelism in a task is only restricted by the communications protocols and the size of any isochronic regions used. Communication ability is similarly inherent in that the restrictions are created by the method of implementation. The point of contention is the feasibility of programming the Soup architecture. Combination primitives of the functional language (such as adder circuits) could be implemented in a similar manner to the examples described in this chapter. Dynamic creation and interconnection of such primitives is a more complex task and will require more examination. This problem could be simplified by emulating a functional architecture ([JON87]) on an Architectural Soup. Solving the dynamic allocation issues of a functional language at the low level of a Soup will be complex. It is more feasible to emulate a higher level machine which is better suited to this task.

There is a need for some form of examination into the utilisation of the Soup-cell's component parts, in order to assess their relative importance. This again will be technology dependent as component timings depend on their method of implementation. An interesting simulation would be of the Soup emulating itself, to estimate speed reduction and the number of Soup-cells in the emulated machine. Once utilisation estimates have been examined, more efficient face-cells can be designed.

Other aspects of the architecture which can be varied, are the face-cell interconnect, the Soup-cell interconnect, and the external shape of the Soup. The external shape of the Soup has been rectangular in all of the examples of this chapter. This need not be the case, in fact a perfectly rectangular machine is unlikely due to the material restrictions discussed in chapter 3. Programming issues involved in other shapes such as pyramids or spheres require consideration. The Soup-cell interconnect is based on the conclusion from chapter 1 that the near-neighbour mesh is becoming predominant. Other connection architectures could be examined, but they are likely to result in more complex face-cells or more complex programming. The interconnection of face-cells within a Soup-cell, however, could be altered with little complexity increase. Figure 5.6 (a) has the logic communication in the opposite direction to the program and lock communication. This may provide for some subtle programming differences. Figure 5.6 (b) shows a less regular interconnection topology. Figure 5.6 (c) has two separate program and lock loops, with very different topology. Figure 5.6 (d) is partially disjoint, and has some interesting possibilities in that one of the face-cells can never be locked and so could act as some form of supervisor to the

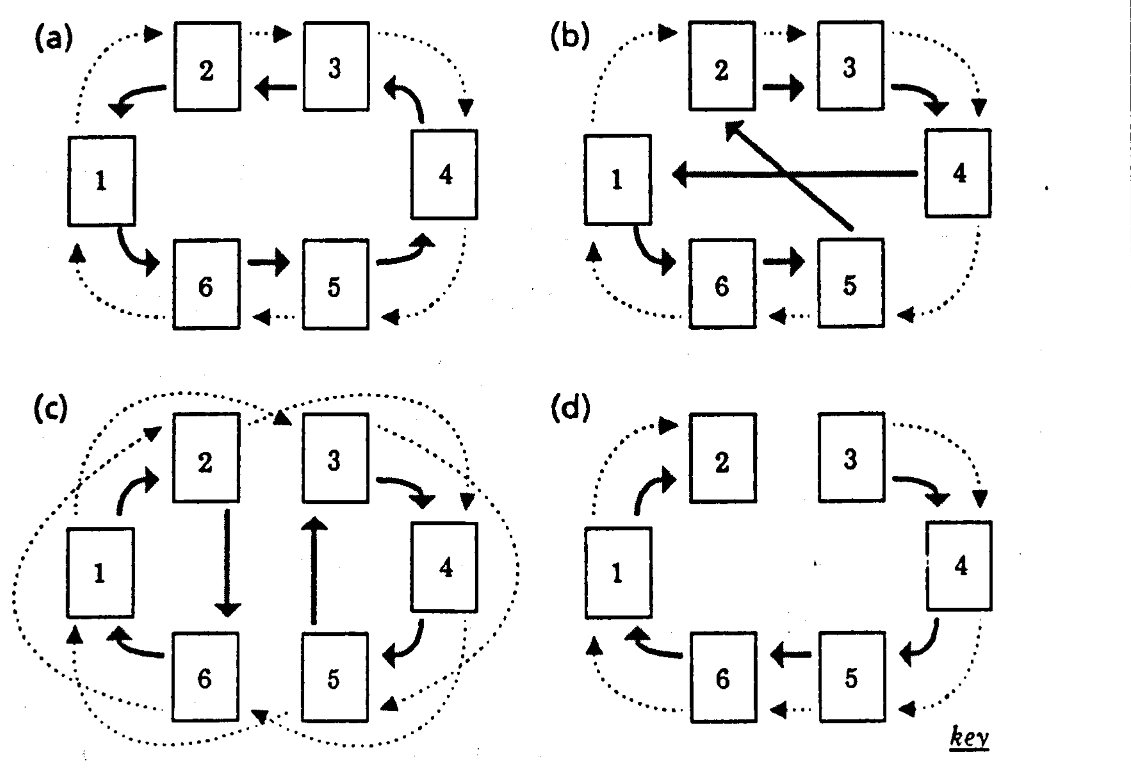


Figure 5.6: Some alternative interconnections

other face-cells. The more irregular a topology becomes, the less general purpose it is likely to be, since it represents a programming complication.

5.6 SUMMARY AND CONCLUSION

The work described in this chapter was aimed at a consolidation of the ideas already discussed, and an examination of the feasibility and difficulties involved. The simulator simulated a Soup of microscopic proportions, (the size will be estimated in chapter 6), but it served to show that this architecture could be programmed. A degree of internal programming was possible but the architecture was generally not appropriate for this. A criticism of the simulation work is that no real time timing estimates were obtained. It is not possible, therefore, to assess the likely speed of this machine with respect to other general purpose machines. However, the parallelism inherent in the architecture is unquestionable. The assessment of timing performance is seen as one of the most important factors requiring examination.

This chapter concludes by considering whether the architecture described would be a good candidate for a first architectural Soup. Some positive aspects of the architecture are its functionality, that it can be loaded, and that it has been possible to specify small circuits which execute on it. The dynamic aspects of the machine, in the form of program and locking circuitry, were not so successfully designed. If suitable material and manufacturing technology became available to build a Soup then it is proposed that this architecture would be a good initial candidate without the dynamic circuitry. (Omission of this would have the additional benefits of simpler program register and control circuitry resulting from decreased face-cell complexity.) Such a machine could then be used as an emulator for testing any new Soup architectures.

Chapter 6: On a VLSI implementation

This chapter illustrates a VLSI design for one face-cell of the architecture described in chapter 4. A further architecture assessment resulting from this design is performed, and estimates based on this are made for the size of best possible designs.

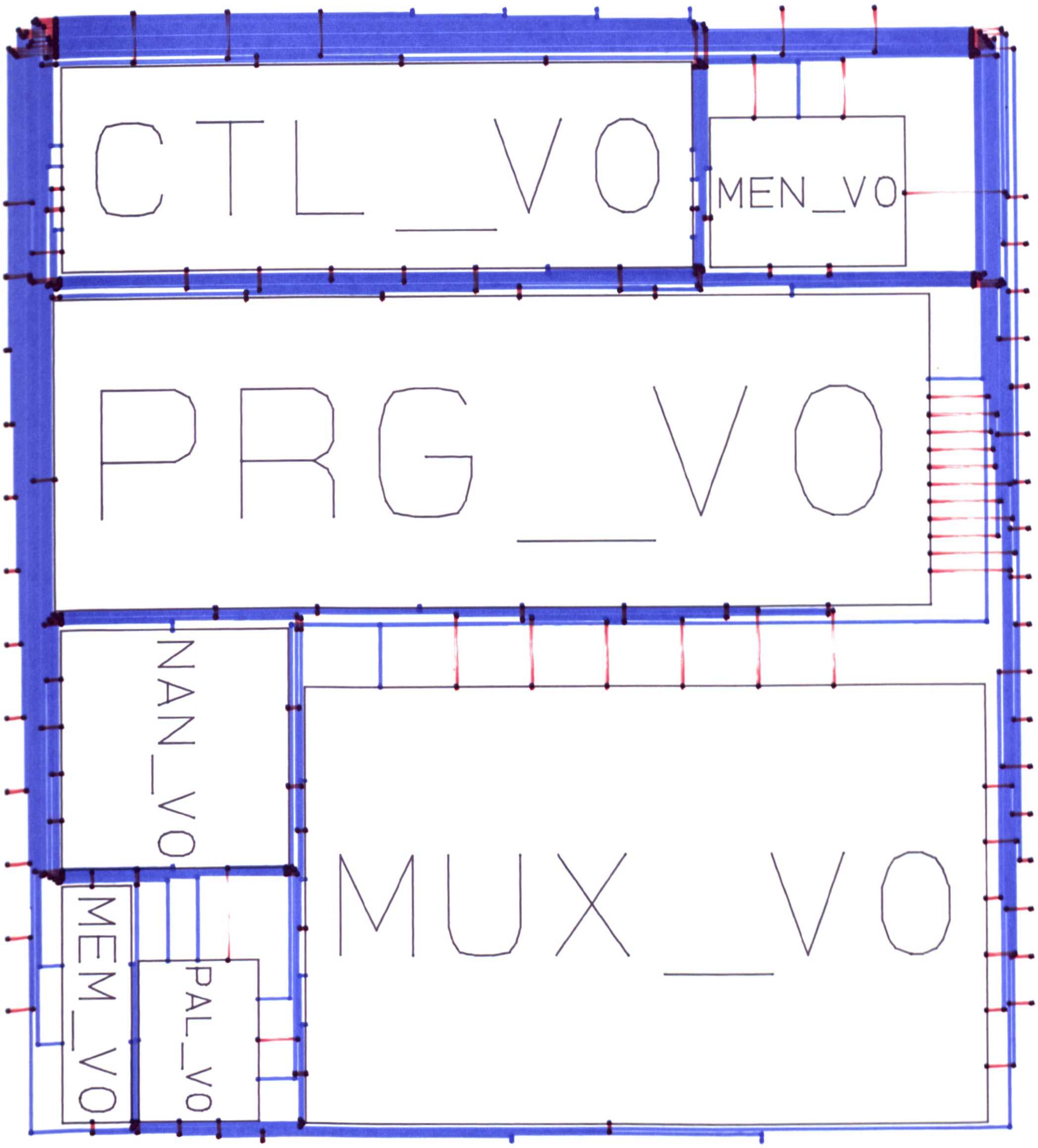
It was not the aim of the VLSI implementation to produce a working design, but to produce a close approximation of a face-cell, and illustrate the simplicity of its design. The implementation identifies the components of the design likely to represent greater cost in terms of silicon area and hence the greater complexity. Area estimates will be used to obtain an overall estimate as to the number of processing elements in a futuristic machine. This requires relating a two-dimensional implementation of a face cell to some of the opportunities and problems of full three-dimensional implementation.

6.1 THE DESIGN

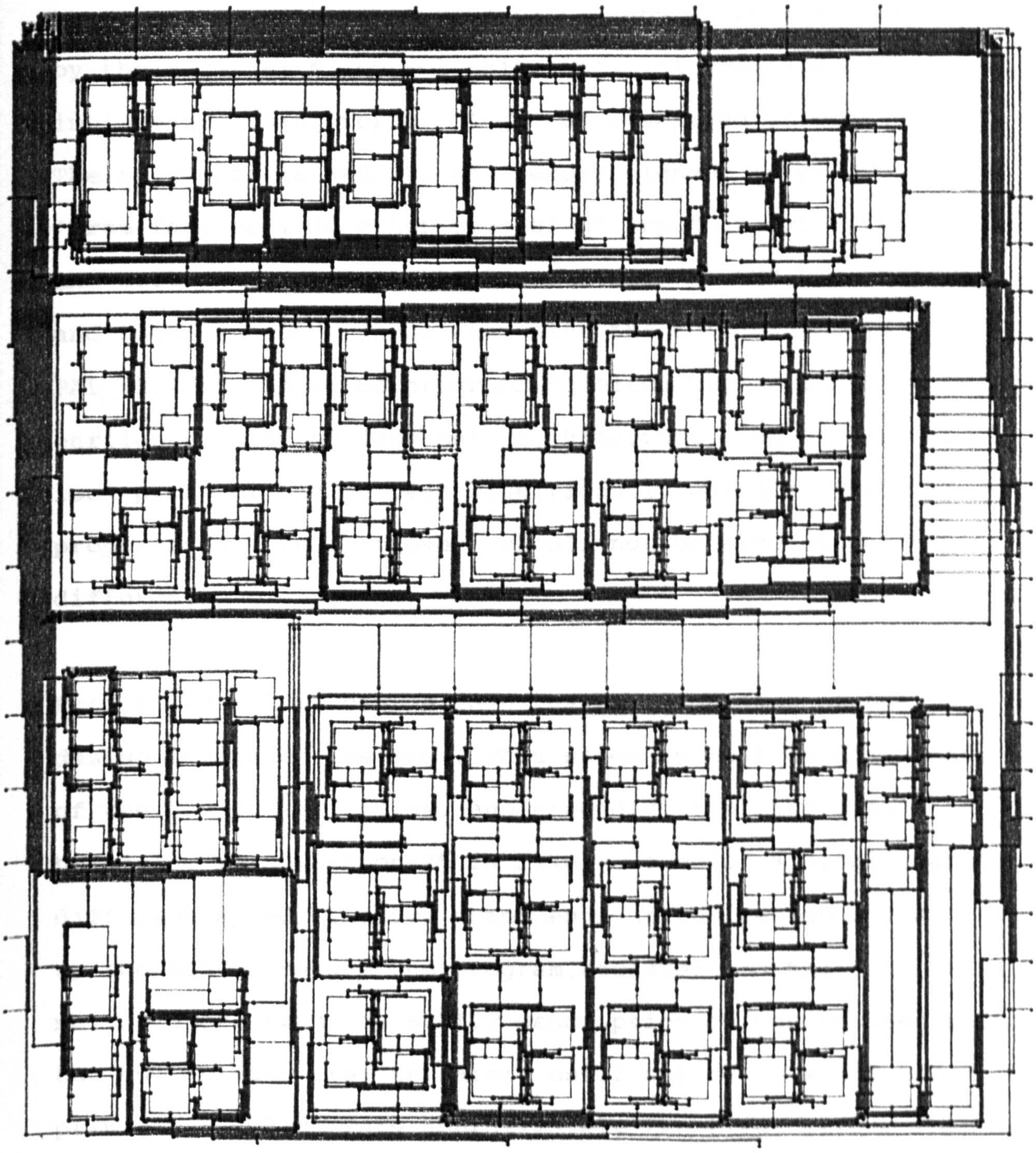
The face-cell was designed using STRICT [CAM85]. The STRICT description is Appendix 6. The resultant design can be seen in Plates 6.1 and 6.2. Plate 6.2 is drawn using the same description as Plate 6.1 except it is expanded to the level of individual 'and' 'or' and 'pass' gates.

The control section in Plate 6.1 comprises of CTL_V0 which is the main control unit, and MEN_V0 which is the multiplexor enable circuitry. The control unit is essentially three SR flip-flops, which act as the state-machine. There is associated circuitry to examine the current state and the environment, to determine if a change of state is required. The circuit described does not include timing circuitry. Timing circuitry would be required to ensure the flip-flops switch deterministically. This is unlikely to significantly effect area. The multiplexor enable circuitry enables the multiplexor outputs if the face-cell multiplexor program register is full, and the previous face-cell in the face-cell string has acknowledged receiving a full signal. The block PRG_V0 is the multiplexor program register. Each bit of the register is implemented by

Silicon design for one face-cell



Silicon design in detail



an SR flip-flop with a pass transistor on the input and output. There is also circuitry to shift the program register up one bit in the event of the cell being used as a programmer. Input and output pass transistors are controlled by the face-cells control unit. Logic of the face-cell is implemented using ternary logic as described in chapter 4. The nand gate is NAN_VO, the memory element is MEM_VO, the 'from-face-cell in other-dimension' input is two wires into the multiplexor. The multiplexor itself is MUX_VO. Program and lock circuitry associated with the multiplexor is PAL_VO. Note that the programming circuitry is based on an earlier version of face-cell architecture, the program and lock protocol having been significantly changed. The proportional area, however, should not be significantly different.

From Plate 6.1 it can be seen that no component of the design dominates the area. This suggests that no component of the design is complex, and secondly, that the design is balanced, so no component is likely to be of a significantly different speed. The two components of the design which take up the most area are the program register and the multiplexor. From the STRICT description it can be seen that the multiplexor area is mainly built out of selector block primitives. These were approximately ten times larger than they need to be (see later). The program register is built up from six one-bit registers, each consisting of an SR flip-flop, two inverters, three pass transistors and an 'or' gate. A functionally similar circuit is given in [MEA80] consisting of two pass transistors and two inverters only, thus there is much scope for optimisation. If the design were optimised, it would be expected that the proportionate size of the multiplexor would become approximately the same as the control section, and the area of the program register would be less.

6.1.1 Design omissions

As stated above, there is a need for a local clock in order to maintain sharp edges within the control unit. The distribution of such a clock was discussed in chapter 3. Also, circuitry has not been examined for spikes, in particular, a spike which generated a high=1 and low=1 on a ternary output wire from the multiplexor has potential catastrophic result. This again suggests some form of synchronisation, such as a local clock or some form of self-timed mechanism. An estimate will be made in section 6.3 for the proportion of silicon area such synchronisation mechanisms might require.

6.2 A FURTHER ARCHITECTURAL ASSESSMENT

Accepting that the STRICT description is a good approximation to a face-cell, an important result is that no part of the design takes up significantly different amounts of space. The design is not optimum, but despite this, it has still resulted in a believably sized processor chip which could be manufactured with existing technology. Two major area costs appear to be the control unit and the multiplexor. Much of the area associated with the control circuitry is to do with the program and lock operations. Area would be reduced if this were improved. (It was suggested in chapter 5 that it could be omitted completely.) The multiplexor circuitry would be reduced if it were not dealing with a two wire ternary system, by passing three signals along a single wire, (see [INT87] for experimental work on three state transistors), or by designing a cell which was not ternary.

Designing with the aim of fabricating a face-cell, in order to obtain true timing estimates, would be beneficial in that relative timing costs of face-cell functional components can be similarly assessed. Consider the internal reprogramming of cells. The frequency of use of this capability will determine the required speed and ease of use associated with it. There will be a series of trade-offs of this form between all component units, to obtain highest utilisations.

6.3 AREA ESTIMATES

The chip was designed using nmos design rules and lambda of 3 microns. It measures 4.9 mm by 5.3 mm. This gives a chip area of 26 mm^2. As stated previously the chip design does not include any timing circuitry. The cell is essentially asynchronous so the proportion of timing circuitry will be small. It is estimated that it would generate a maximum of a 25% increase in area. This gives :-

Face-cell area ~ 32.5 mm^2

By examining the design in Plate 6.2 it is clear that there is unused silicon area and a high proportion of routing. This design was produced whilst the STRICT design system was still in development, and using only 45 minutes of processor effort on a MicroVax. As the face-cell is to be replicated many millions of times, large processor effort can be justified for optimisation. Given many thousands of hours spent on optimisation, with best possible place and route algorithms, the resultant design is estimated to be 50% smaller, thus :-

Best implementation of this design ~ 16 mm^2

Reductions in complexity will result in reductions in the numbers of components and the amount of routing between them. There are two complexity issues to consider. Firstly, the face-cell which has been implemented was not optimal due to limitations on the time available for production of the design. There are unnecessary repetitions of Boolean functions within the different components of the machine and the design is primarily "and" and "or" gate oriented as opposed to "nor" gates which take less silicon area. An example of poor use of area is the selector cell, (there are 18 in the design), which was implemented using an or-gate, an inverter and two pass transistors. The resulting size of the selector is of the order of 10 times larger than a selector primitive could be. See [MEA80] Plate 7 for an example. Secondly, as previously discussed, the face-cell design has a percentage of unused capacity in its multiplexor. Taking this into account, we can consider a smaller face-cell implementation for a different face-cell architecture of equal functionality. For these factors combined, I allow a reduction of area of at least another 50%, thus :-

Area of best face-cell design ˜ 8 mm^2

6.3.1 Long term estimates from computer technology predictions

Silicon feature size is decreasing. Estimates for the smallest feature sizes possible vary between 0.3 microns [MEA80], and 0.03 microns implied by [BAT88]. ([POR87] states that feature sizes may shrink to 0.3 lambda by 1997.) The figure of 0.1 microns was chosen as an approximate estimate. The electrical properties of components will change as the feature size is reduced [MEA80], but it is assumed that a direct scaling down of the chip design will give an approximate estimate of the size of a minimum feature size functionally equivalent circuit. The design previously discussed uses a lambda of 3 microns. Lambda is a measurement of length. Halving the size of lambda would result in a chip area of a quarter of the original size. Consequently :-

Long term best face-cell design ˜ (8^0.5 * (0.1/3))^2
= 0.01 mm^2

As design processes improve it will become possible to build larger chips, so permitting the inclusion of more components within a single chip. See [JES86] and [GRI84] for examples of wafer scale integration. Current computers typically consist of 1 m^2 of silicon [JES86]. If a continuous perfect chip of this size is assumed then the face-cell architecture is such that they will abut into long strings. Several strings could be laid in parallel across the chip as an estimation of face-cell numbers in the same orientation on such a chip. (A machine with this architecture would be of little use. A generalisation will follow for a machine with the architecture discussed in chapter 4.)

The number of face-cells of one particular orientation in 1 m^2 is :-

$$\sim 1\text{m}^2 / 0.01\text{mm}^2$$
$$= 100000000$$

A bolder assumption, introduced in chapter 1, is that future technologies will be three-dimensional. Experimental work is currently being performed for three dimensional silicon, such as [ROS83] [NUD85] [GRI84]. For simplicity, it is assumed (and this is a massive assumption) that the dimensionality problems, such as power distribution and flaws in the implementation materials, will be surmounted. Generalising from the estimate of 1m^2 for current machines, a volume of 1 m^3 is taken as a speculation of possible volume of a future machine.

A Soup-cell consists of six face-cells in different orientations, left, right, front, back, up and down. Consider laying six two-dimensional face-cells in a stack. These can be interconnected in the third dimension, using the 'to-face-cell in other-dimension' connections, to form the ring between the six face-cells in a Soup-cell. Similarly, connections can be made between the inputs and outputs of the face-cells and the inputs and outputs of corresponding face-cells in neighbouring Soup-cells, to form the string topology described in Chapter 4. The left, right, front and back face-cells could, in fact, abut to the neighbouring Soup-cells to form the required topology, as each of these communicates in the horizontal plane, only in different direction. The up and down face cells communicate in the vertical plane. The circuitry for these face-cells has been placed in the horizontal plane, but the interconnection can be provided by routing wires up and down. (This wiring should not have significant effect on the area estimation, so is ignored in the volume calculation below.) In summary, a Soup-cell is now being considered as six two-dimensional face-cell circuits, interconnected in the third dimension. Assuming that the silicon design rules used in two dimensions (see [MEA80]) can be used as an estimate of required width between circuits in a third dimension, the estimate is made that face-cells can be spaced 6 lambda apart :-

```
    Height of one Soup-cell ~ 6 * 0.1 * 6
                            = 3.6 microns
```

So, in 1 m^3 of silicon it is estimated that there will be of the order :-

```
    100000000 / 3.6*10^-6
    ~ 3*10^13 Soup-cells.
```

6.4 SUMMARY AND CONCLUSIONS

The silicon chip design gave insight to the complexity of the face-cell design. It also showed the likely relative costs of each of the face-cell component units. The ternary logic component takes up less than a quarter of the design. Firmer conclusions could be made through production of a working design, ideally, a chip containing a number of face-cells intercommunicating. Relative timing statistics would aid in determination of function cost.

An estimate has been made for the number of Soup-cells in a future silicon machine. These figures will vary depending on the technology being considered. In particular, consider feature sizes :-

Minimum feature size number of cells in 1m^3 using different lambda :-

0.3 microns [MEA80]	1 * 10^12
0.1 microns	3 * 10^13
0.03 microns [BAT88]	1 * 10^15

There will be much debate of such figures, particularly since many factors have been taken into consideration. For example, three dimensions routing takes less area due to the extra flexibility [ROS83] [PRE83]. This could make Soup-cells smaller. This benefit is likely to be more than offset by the three dimensional building complications such as power distribution (heat).

As an aside, one of the largest simulations described in chapter 5 used 30000 Soup cells to perform an edge detection on a 12 by 15 pixel image. It took 9 hours to simulate on one processor of an Encore Multimax [LEE87]. On the above volume estimate for a 0.1 micron feature size this represents a volume of Soup of less than 1 mm^3.

Chapter 7: Summary and conclusions

This chapter comprises of a summary of the work contained in this thesis and suggestions for general further work. (Specific further work was detailed in the sections to which it referred.) The chapter concludes with remarks on the usefulness and validity of the work performed.

7.1 SUMMARY

Chapter 1 made distinction between dedicated purpose and general purpose architecture. Dedicated computers are usually easier to design, as the problems for which they are intended can be defined. A capability for future general purpose computers was identified as the ability to emulate many different dedicated architectures.

A consequence of Von Neumann architecture has been that the silicon in machines is being under-utilised. Hardware is essentially parallel but the architectures imposed on it are sequential. It is argued that low level reconfigurability could increase utilisation.

Existing architecture was examined and trends were identified. The number of processing elements in general purpose machines is increasing. These processors have simpler function. Memory is being incorporated into each individual element. The large number of elements is enforcing a simple communication topology with a trend towards near-neighbour mesh. There is a trend towards multiple instruction stream and multiple data stream (MIMD) architecture.

Trends in technology suggest that future machines are likely to permit much larger numbers of components. It is also likely that machines will be three-dimensional. In order to simplify the design and production complexity which will be involved with such technology, it will likely be necessary to have simple and regular architectures.

Chapter 2 proposed a class of architectures which can be programmed at very low levels. These were termed 'Architectural Soups' due to the myriad of 'potential' architectures available from within a machine. Soups were classified through closeness to a basis architecture. This basis was shown to follow the architecture and technology trends identified in Chapter 1. It therefore follows that any machine with similar architecture to this basis must also follow these trends and aims.

Each cell of the basis machine communicates with its six immediate neighbours using a three dimensional near neighbour mesh topology. The functionality of each cell is provided through the ability to program a nand gate, or a one-bit memory, or a wire between any inputs and outputs of any cell within the machine. It is also possible to program a constant binary one or zero to any output of a cell. Other properties of the basis architecture are that it is; homogeneous, isotropic, controllable and deterministic.

Some methods of programming a machine of this style were examined. A pessimistic view was that it would be similar to the design of VLSI silicon chips, whose method of design has not yet been perfected. In addition, use of the third dimension must now be considered. An alternative, more optimistic view, is that such programming is at a level just below machine code. For the majority of tasks on existing machines it is now assumed a compiler will generate efficient machine code. A compiler could similarly generate circuit descriptions for the Soup. It was argued that as the trends identified in Chapter 1 suggest this style of architecture, it is conceivable to be both possible and useful to program a Soup. (A benefit identified was that it could be efficient for both parallel and sequential algorithms.)

Some programming methods were illustrated, the main variation being in the dynamism of the ability to change the machine architecture. The usefulness of dynamic change will determine which style of architecture is the most practical.

Possible benefits from Soup architecture were identified as; increased flexibility, increased speed of solution, decreased system size required, and decreased system design costs and software costs. In conclusion, it was felt that it may be unreasonable to design a machine with processing elements at this gate level, although such a machine would be of interest from the point of view of examining such flexibility. As trends project towards this style of architecture, it is suggested that future machines would essentially be of this form, with the exception of having slightly more complex cells. Investigation of Architectural Soup architecture may therefore reveal capability and limitation of future general purpose computers.

Chapter 3 examined some implementation issues. A diversity of topics must be considered in implementing Architectural Soup styles of architecture. It was not possible to perform a thorough investigation into all issues.

There are theoretical limitations on machine design. Example work illustrated was mathematical theory in compacting solids. Properties of silicon style technology were then examined. Design for large three dimensional machines will be complex. A benefit of the regularity in Architectural Soup architecture is the ability to model a small component and make generalisations for large machines. The surface of the machine presents technology considerations due to the discontinuity of the material. The surface also presents programming complexity due to discontinuity of the Soup cells. Problems with interfacing compatibility and the related problem of initialisation were discussed. Synchronisation issues were considered on two levels, firstly, for deterministic function of the machine, and secondly, for appropriate synchronisation mechanisms between the component parts of application circuits. The importance of fault tolerance in circuit design was emphasised and suggestions made. In particular, an unreliable Soup could emulate a more reliable one. Finally some alternative technology such as optical and biological machines were examined.

Chapter 4 described a specific architecture in detail. The primary aim was to illustrate issues of previous chapters, in particular, a possible control mechanism. The architecture consisted of the repetition of a simple cell. A cell consisted of, a control unit, program register, multiplexor and ternary logic.

This architecture was shown to be an Architectural Soup. Some important characteristics of the architecture were, that it had no global bus, that it was possible to initialise and deterministically program the machine, and that the design has a degree of internal reprogramming capacity. An assessment of the design was given, in particular the dynamic aspect of the cell design did not function in the way one would like.

Chapter 5 described the simulation of a small portion of a machine with the architecture of chapter 4. The complexity of performing such a simulation on existing machines was highlighted and the environment used detailed. Some of the difficulties were, in the size of the machine which could be simulated, the amount of processor effort required to perform the simulation, the representation of the simulated machines internal state, and extraction of results from the example calculations. Simulation work was described. Application programming of the architecture was through a Pascal procedure call method. The Soup did not prove to be as difficult to program as had been expected. The simulations firstly showed an architecture in an initial random state, then the ability to deterministically load and execute a circuit description, perform several different types of task, and the ability to both externally and internally reprogram the machine architecture. An assessment of the architecture and of the simulation environment were made. The chapter concluded by examining whether, in light of simulation, the architecture used could be considered as a good candidate for the first architecture to be implemented when such three dimensional technology is available. The conclusion was, with exception of the dynamic aspect of the design (the internal reprogramming capacity), the architecture would be a good candidate.

Chapter 6 described how a fraction of the machine (one sixth of a cell) was designed in silicon. It is not possible currently to implement a full Soup due to its size and three dimensionality. The purpose of the design given, was to illustrate simplicity. It was found that the design was of a believable size for implementation on a silicon chip.

The size of the chip design was used to obtain rough estimates for the number of processing elements in future machines. A speculation for the number of cells in a large three dimensional machine with simple processing elements was calculated as being of the order of 10 to the power 13 cells.

7.2 FURTHER WORK

The majority of this thesis comprises a feasibility study for Architectural Soup architecture. Consequently, and as with any feasibility study, further work can be performed. Specific work was detailed in the sections where relevant. A general aim for further work is to consolidate the completeness of the work of this thesis, an example being, a more thorough investigation of the topics considered in chapter 3. A topic which was not discussed in chapter 3 is the possible use of analogies from material structures. Due to the small grain size of the processing elements of an Architectural Soup, there is a similarity to the molecules found in materials. Study of molecular bonding may give insight into the complexity of cells and identify communication topologies which may be feasible. A study of flaw types, such as dislocations, which occur naturally in material, may give insight into fault tolerance considerations.

A major simulation goal for the future was identified as the full implementation of a functional language. It may also be of benefit to examine alternative Soup architectures to the one described in chapter 4. An observation which can be made is that in all aspects of Soup design there is a large design space of possible configurable architectures. Motivations for choice within this design space is required.

Further work should also assess the viability and efficiency of Soup machines in relation to the potential performance of other architectures. Assessment in the long term could perhaps be through full three-dimensional implementations. Currently it would be possible to examine a surface of soup-cells, for example, using Wafer Scale Integration techniques.

7.3 CONCLUDING REMARKS

This thesis examined some architectures for general purpose computers in the very long term, based on projections of existing architecture and technology trends. It has examined the concept of a computer which is re-configurable at a very low level. The architecture is simple and repetitive yet capable of performing complex tasks. This architecture was shown to have potential to emulate both parallel and serial machines. A limitation is the control overhead which may be incurred to control this flexibility.

This thesis has examined the potential usefulness, and viability in principle, of an Architectural Soup. It has shown that it has been possible to design a simple regular architecture for large three-dimensional technology. A cell for a small-grain reconfigurable architecture could be designed. It has been possible to simulate a small portion of the architecture and it has also been possible to program the machine. The flexibility offered by the architecture suggests that it would be a reasonable initial candidate for implementation in the event of suitable technology becoming available.

The concepts of Architectural Soup serve as an illustrative tool for the sorts of capability that will be possible in general purpose machines in the very long term. For example, a conclusion which can be drawn from this work is that a single ultimate general purpose computer, or one ultimate general purpose architecture, is unlikely due to the diversity of architecture and technology choices which can be made.

The concepts involved in Architectural Soup architecture may be worthy of more detailed long term research. This would serve to collate a diversity of related material. As trends have suggested that future machines may be of this form, collation of material would prove useful in the design of any similar machines. Work should ideally remain general as it is difficult to assess factors which will be of greatest concern in future machines. Factors which currently have low significance may become increasingly important.

This thesis began by stating that computers are bound by physical size, the speed of light, and the properties of the technology and the construction methods. Any other restrictions that a computer may have are due to the imposition of man-made architectural decisions.

Architectural Soup is an architecture which reduces the man-made restriction through provision of a flexible underlying architecture. The usefulness of such flexibility may be offset by increased control. For Architectural Soup machines it is possible that too much flexibility is available, and too much control is required. Future general purpose machines will likely lie in between current architecture, and the architecture of an Architectural Soup.

REFERENCES

[ACM84]

The Fifth Generation Challenge, ACM 1984 annual conference, ISBN 0-89791-144-x

[ABU87]

Y.S. Abu-Mostafa & D. Psaltis, Optical Neural Computers, Scientific American, March 1987, 66-73

[AND87]

G. Anderla & A. Dunning, Computer Strategies 1990-9: Technologies-Costs-Markets, 1987, ISBN 0-471-91585-8

[BAR84]

H.G. Barrow, Proving the Correctness of Digital Hardware Designs, VLSI design, July 1984, 64-77

[BAS77]

F. Basket & T.W. Keller, An Evaluation of the Cray-1 Computer, in [KUC77] 71-84

[BAT80]

K.E. Batcher, Design of a Massively Parallel Processor, IEEE Transactions on Computers, Volume C-29, Number 9, September 1980, 836-840

[BAT88]

R.T. Bate, The Quantum-Effect Device: Tomorrow's Transistor?, Scientific American, March 1988

[BEA85]

P. Beadle & J. Wiles & L.M. Goldschlager, Implementation of an ALU by a Parallel Machine, 1985, in [BER85] 153-165

[BER85]

P. Bertolazzi & F. Luccio, VLSI: Algorithms and Architectures, Proceedings from international workshop on parallel computing and VLSI, Amalfi, Italy, May 23-25 1984, ISBN 0-9510708-0-0

[BRA85]

H.N. Brady, A MIMD Organisation for the Execution of Interconnection Routing Algorithms, IEEE circuits and devices magazine, March 1985, 39-43

[BUC83]

I.Y. Bucher, The Computational Speed of Supercomputers, Proceedings ACM sigmetrics conference on measurement and modelling of computer systems, August 1983, 151-165, also in [HWA84] 74-88

[CAM85]

R. Campbell & A. Koelmans & M. McLauchlan, STRICT: A Design Language for Strongly Typed Recursive Integrated Circuits, IEE proceedings, March 1985, 108-115

[CAR80]

J.E. Carroll, Physical Models for Semiconductor Devices, 1980, ISBN 0-7131-3308-2

[CHA68]

Chambers Encyclopaedia, 1968

[CHE83]

S. Chen, Large-scale and High-Speed Multiprocessor System for Scientific Applications, CRAY X-MP-2 series, proceedings NATO advanced research workshop on high speed computing, June 1983, revised in [HWA84] 46-58

[CHI82]

P.M. Chirlian, Analysis and Design of Integrated Electronic Logic Circuits, Volume 2: Digital Electronics, 1982, ISBN 0-06-318215-7

[COD68]

E.F. Codd, Cellular Automata, ACM monograph series, 1968

[DAL84]

C.U. Smith & J.A. Allen, Future Directions for VLSI and Software Engineering, Lecture notes in computer science: VLSI engineering - beyond software engineering, 1984

[DAV84]

R. Davis & D. Thomas, Systolic Array Chip Matches the Pace of High-Speed Processing, Electronic design, October 31 1984. Related articles in next three issues.

[DEN80]

J.B. Dennis, Data Flow Supercomputers, Computer, November 1980, 48-56

[ECK79]

R.E. Eckhouse & R.L. Morris, Minicomputer Systems, 1979, ISBN 0-13-583922-x

[ENC86]

Multimax Technical Summary, Encore computer corporation, 726-01759 rev c, September 1986

[FEN81]

T. Feng, A Survey of Interconnection Networks, Computer, December 1981, 12-27

[FER85]

D.K. Ferry, Interconnection Lengths and VLSI, IEEE circuits and devices magazine, July 1985, 39-41

[FOU85]

T.J. Fountain, Plans for the CLIP7 Chip, Integrated technology for parallel image processing, 1985, ISBN 0-12-444820-8, 199-214

[FRI83]

G. Fritsch & W. Kleinoeder & C.U. Linster & J. Volkert, EMSY85-The Erlangen Multi-Processor System for a Broad Spectrum of Applications, Proceedings 1983 international conference on parallel processing, 325-330

[FUN77]

L. Fung, A Massively Parallel Processing Computer, in [KUC77] 203-204

[GIB83]

J.R. Gibson, Electronic Logic Circuits, 1983, ISBN 0-7131-3491-7, Chapter 7

[GOA82]

G.B. Goates & T.R. Harris & R.E. Oettel & H.M. Waldron, Storage/Logic Array Design: Reducing theory to practice, VLSI design, July 1982, 56-62

[GOL85]

A.V. Goldberg & S.S. Hirschorn & K.J. Lieberherr, Approaches Toward Silicon Compilation, IEEE circuits and devices magazine, May 1985, 29-38

[GOT82]

A. Gottlieb & R. Grishman & C.P. Kruskal & K.P. McAuliffe & L. Rudolph & M. Snir, The NYU Ultracomputer -- Designing a MIMD Shared-Memory Parallel Machine, IEEE 0149-7111/82/0000/0027

[GRI84]

J. Grinberg & G.R. Nudd & R.D. Etchells, A Cellular VLSI Architecture, COMPUTER, January 1984, 69-81

[HAD64]

H. Hadwiger & H. Debrunner & V. Klee, Combinatorial Geometry in the Plane, 1964, LCCCN 64-10297

[HAY78]

J.P. Hayes, Computer Architecture and Organisation, 1978, ISBN 0-07-027363-4

[HEW80]

C. Hewitt, The Apiary Network Architecture for Knowledgeable Systems, 1980

[HIC83]

P.J. Hicks, Semi-Custom IC design and VLSI, 1983, ISBN 0-86341-011-1

[HIL85]

W.D. Hillis, The Connection Machine, 1985, ISBN 0-262-08157-1

[HIL87]

W.D. Hillis, The Connection Machine, Scientific American, June 1987, Volume 256, Number 6, 86-93

[HUT87]

L.D. Hutcheson & P. Haugen & A. Husain, Optical Interconnects Replace Hardwire, IEEE spectrum, March 1987, 30-35

[HWA84]

K. Hwang, Supercomputers: Design and Application, IEEE computer society press, 1984, ISBN 0-8186-0581-2

[HWA87]

K. Hwang, Advanced Parallel Processing with Supercomputer Architectures, Proceedings of the IEEE, Volume 75, Number 10, October 1987

[INM85]

INMOS, Transputer, INMOS publication 72-TRN-010-002

[INT87]

Special Issue on Multi-Valued Logic Systems, International journal of electronics, Volume 63, Number 2, August 1987

[JAN85]

P.A. Janson, Operating Systems - structures and mechanisms, 1985, ISBN -12-380230-X

[JAS77]

Z.D. Jastrebski, The Nature and Properties of Engineering Materials, 2nd Edition, 1977, ISBN 0-471-02859-2

[JES86]

C.R. Jesshope & W. Moore, Wafer Scale Integration, proceedings of a workshop held at the University of Southampton, July 1985, ISBN 0-85274-497-8

[JON87]

S.P. Jones, The Implementation of Functional Programming Languages, 1987, ISBN 0-13-453325-9

[JOS84]

M. Joseph & V.R. Prasad & N. Natarajan, A Multiprocessor Operating System, 1984, ISBN 0-13-605170-7

[KAH77]

G. Kahn & D.B. MacQueen, Coroutines and Networks of Parallel Processes, IFIP congress proceedings, 993-998

[KAT84]

M.G. Katevenis, Reduced Instruction Set Computer Architectures for VLSI, ACM doctoral dissertation award 1984, ISBN 0-262-11103-9

[KEL85]

S.H. Kelem, A Method for the Automatic Translation of Algorithms from a High-Level Language into Self-Timed Integrated Circuits, IEEE circuits and devices magazine, March 1985, 17-21

[KOE86]

A. Koelmans, System Structure for Asynchronous Fault Tolerant VLSI Circuits, SRM 428, University of Newcastle upon Tyne Computing Laboratory, 1986

[KOS84]

A. Koster, Compiling Prolog Programs for Parallel Execution on a Cellular Machine, in [ACM84] 167-178

[KOZ80]

E.W. Kozdrowicki & D.J. Theis, Second Generation of Vector Supercomputers, Computer, November 1984, 71-83

[KRA81]

G.D. Kraft & W.N. Toy, Microprogrammed Control and Reliable Design of Small Computers, 1981, ISBN 0-13-581140-6

[KUC77]

D.J. Kuck & D.H. Lawrie & A.H. Sameh, High Speed Computer and Algorithm Organisation, Symposium on high speed computer and algorithm organisation, University of Illinois, 1977, ISBN 0-12-427750-0

[KUN80]

H. T. Kung & C.E. Leiserson, Algorithms for VLSI Processor Arrays, in [MEA80] Section 8.3, 271-292

[KUN86]

S.Y. Kung, VLSI Array Processors, 1986, in [MOO86] 7-24

[LAE71]

A.E. Laemmel, General Purpose Cellular Computers, Presented at the symposium on computers and automata, Polytechnic Institute of Brooklyn, April 13-15, 1971

[LEE87]

P.A. Lee, Parallel Processing on the Multimax Computer System, PPM/001 Computing Laboratory, University of Newcastle upon Tyne, August 1987

[LIN82]

N.R. Lincoln, Technology and Design Tradeoffs in the Creation of a Modern Supercomputer, IEEE transactions on computers, Volume C-31, Number 5, May 1982, 349-362

[LIP87]

G.J. Lipouski & M. Malek, Parallel Computing - theory and comparisons, 1987, ISBN 0-471-82262-0

[LOE76]

A.L. Loeb, Space Structures - Their Harmony and Counterpoint, 1976, ISBN 0-201-04650-4

[LPM87]

R.P. Lippmann, An Introduction to Computing with Neural Nets, IEEE ASSP Magazine, April 1987, 4-22

[MAN82]

B.B. Mandelbrot, The Fractal Geometry of Nature, 1982, ISBN 0-7167-1186-9

[MCC86]

J. McCanny & J. McWhirter, The Derivation and Utilisation of Bit Level Systolic Array Architectures, 1986, in [MOO86] 47-59

[MEA80]

C.A. Mead & L.A. Conway, Introduction to VLSI systems, 1980, ISBN 0-201-04358-0

[MEL86]

R. Melhem, Irregular Wavefronts in Data-Driven Data-Dependent Computations, 1986, in [MOO86] 303-312

[MOL84]

D. Moldovan, An Associative Array Architecture Intended for Semantic Network Processing, in [ACM84] 212-221

[MOO86]

W. Moore & A. McCabe & R. Urquhart, Systolic Arrays, International workshop on systolc arrays, July 1986, ISBN 0-85274-826-4

[MUK86]

M. Mukaidono, Regular Ternary Logic Functions - ternary logic functions suitable for treating ambiguity, Transactions on computers, Volume c-35, Number 2, February 1986

[NUD85]

G.R. Nudd & R.D. Etchells & J. Grinberg, Three-Dimensional VLSI Architecture for Image Understanding, Journal of parallel and distributed computing, Number 2, 1985, 1-29

[OED33]

The Oxford English Dictionary, 1933

[PEA85]

J. Pearl, Heuristics - intelligent search strategies for computer problem solving, 1985, ISBN 0-201-05594-5

[POR87]

W.A. Porter & J.L. Aravena, Orbital Architectures with Dynamic Reconfiguration, IEE proceedings, Volume 134, part E, Number 6, November 1987

[PRE83]

F.P. Preparata, Optimal Three-Dimensional VLSI Layouts, Mathematical systems theory 16, 1983, 1-8

[PRE84]

K. Preston & M.J. Duff, Modern Cellular Automata, 1984, ISBN 0-306-41737-5

[PRO87]

F.P. Prosser & D.E. Winkel, The Art of Digital Design - an introduction to top-down design, 2nd edition, 1987, ISBN 0-13-046673-5

[ROB84]

F.S. Roberts, Applied Combinatorics, 1984, ISBN 0-13-039313-4

[ROS82]

A. Rosenfeld & A. Wu, Cellular Computers for Parallel Region-Level Image Processing, Pattern recognition, Number 15, 1982, 41-60

[ROS83]

A.L. Rosenberg, Three-Dimensional VLSI: A Case Study, Journal of the ACM, Volume 30, Number 3, July 1983, 397-416

[RUZ86]

W.L. Ruzzo, Simple Universal Parallel Computers Based on Hypercube Interconnections (detailed abstract), Computer Science Department, University of Washington, Seattle, 1986

[SCI71]

Computers and Computation, readings from Scientific American, ISBN 0-7167-0936-8

[SLO71]

D.L. Slotnick, The Fastest Computer, in [SCI71]

[SOX76]

Supplement to The Oxford English Dictionary, 1976, ISBN 0-19-861123-4

[SMI81]

B.J. Smith, Architecture and Applications of the HEP Multiprocessor Computer System, Real Time Signal Processing IV, Proceedings SPIE, 1981, 241-248, also in [HWA84] 231-238

[SMI82]

K.F. Smith & T.M. Carter & C.E. Hunt, Structured Logic Design of Integrated Circuits using the Storage/Logic Array (SLA), IEEE journal of solid-state circuits, Volume sc-17, Number 2, April 1982, 395-406

[SNY82]

L. Snyder, Introduction to the Configurable, Highly Parallel Computer, COMPUTER, January 1982, 47-56

[STO77]

R.A. Stokes, Burroughs Scientific Processor, in [KUC77] 85-89

[SU86]

S. Su & M. Cutler & M. Wang & K. Saluja, Self-Diagnosis of Linear and Mesh Systolic Arrays by Signature Comparison, 1986, 217-227 in [MOO86]

[TOT64]

L.F. Toth, Regular Figures, International series of monographs on pure and applied mathematics, Volume 48, 1964, LCCCN 63-10121

[TRE82]

P.C. Treleaven & D.R. Brownbridge & R.P. Hopkins, Data-Driven and Demand-Driven Computer Architecture, Computing surveys Volume 14, Number 1, March 1982

[VAN70]

L.H. Van Vlack, Materials Science for Engineers, 1970, LCCN 74-91151

[VAN73]

L.H. Van Vlack, A Textbook of Materials Technology, 1973, ISBN 0-201-08066-4

[WOL86]

S. Wolfram, Theory and Applications of Cellular Automata, 1986, ISBN 9971-50-123-6

[YAK85]

A. Yakovlev, Designing Self-Timed Systems, VLSI systems design, September 1985, 70-90

[YUE86]

C.K. Yuen, Essential Concepts of Operating Systems, 1986, ISBN 0-201-12917-5

APPENDICES

index

Program-stub to program mapping

This appendix shows the possibilities for programming a face-cell multiplexor. It also shows the mapping between the higher level program stubs and lower level six-bit program for an individual face-cell multiplexor. (A face-cell is one sixth of a soup-cell.) The architecture upon which this Appendix is based is described in Chapter 4. A program used in the simulation environment, which performs the translation of stubs into six-bit programs, is the 'soup face input processor' in Figure 5.1.

Multiplexor program stubs usually consist of three characters. The first two characters of an individual stub define the information to be routed to the corresponding face-cell in the next soup-cells memory and half of the input for the corresponding face-cells nand gate. (The other half of the input to the nand gate comes from within the soup-cell itself, see Chapter 4.) The third character defines which information is to be routed to the next face-cell within the same soup-cell. Multiplexor program stubs may be other than three characters long when used to specify one of the face-cells programming or locking functions.

Multiplexor programs are six bits long. The most significant bit is used to determine if the face-cell is to be used in the normal manner or one of the special locking or programming modes. If the former is the case then the second-most-significant bit of the program is used to define the information to the following face-cells memory, the third and fourth most significant bits specify the information to be routed to the nand gate and the fifth and sixth bits specify the input to the next face-cell within the same soup-cell. The mapping used for this and also the mapping used for the programming and locking functions is explained over leaf.

Program-Stub to Program Mapping

Stub:	**Program:**	
N..	01????	Input to next cells *memory* is output from this cells nand.
F..	00????	Input to next cells *memory* is the input to this cell from the cell in the previous dimension.
.N.	0?11??	Input to next *nand* is the output from this cells nand.
.F.	0?10??	Input to next *nand* is the input to this cell from the cell in the previous dimension.
.M.	0?01??	Input to next *nand* is the output from this cells memory.
.0.	0?00??	Input to next *nand* is a zero.
..M	0???01	Output to the cell in *next dimension* the value in memory.
..1	0???00	Output to the cell in *next dimension* a one.
..N ..F	0???10	Output to the cell in *next dimension* the input to the next cells nand or the input from the cell in the previous dimension BUT the *stub component* must not be the same as the input to next cells memory i.e. the stubs N?N and F?F do not occur.
..?	0???11	Output to the cell in *next dimension* is disabled.
P	11abcd	Program until a lock is found. Leave all cells in the empty state after passing through them until either the last cell in the soup is reached, or a locked cell is found. If so then clear the lock. 'abcd' is an index which is used for debugging; to show where the programming message was generated. For example the simulators input processor always uses 110001.
L	10????	Use a cell in *lock* mode. A cell has the capability to lock the cell in the following dimension. Once locked, a cell's program cannot move along the soup. Unlocking can only be achieved via a 11???? reprogramming. To facilitate easy abutment of cells of the types below, the multiplexor outputs to nand and memory are predefined as 1 and the current value in memory respectively.
LK	10??1?	*Set lock* on the cell in next dimension if the value in memory is 1, or as soon as it becomes 1.
LP	10???1	Use as a *programmer*. Shift up the program register of the cell in next dimension 1 bit, when the input to the programmer cell from previous dimension goes from undefined to 1 or 0 use this as a new least significant program bit and hence obtain a new program for the successor. The successor is forced into the empty state, a transition on memory from high to low at the programmer cell sets the state to full. A programmer cell automatically locks the cell in next dimension on it's first attempt to program.
LKP	10??11	Use as a programmer with lock explicitly set from memory.

Pascal users library

This appendix shows the library of routines provided to aid users in circuit definition. This library program is compiled using a pascal compiler and 'includes' the required user routine in the form of a procedure called 'main'. The purpose of the library is described in more detail in Chapter 5 and its position in the simulation environment can be seen at the top left of Figure 5.1.

It can be seen from the program over leaf that the library provides pre-defined constants such as indices to the six face-cells of a soup-cell and three multiplexor outputs of a face-cell. Some 'types' were also defined in the library. For example, soup_type defines a cubic soup of maximum size 30 by 30 by 30 soup cells. (This size can be increased but the program performance will be effected for larger soups. Most of the simulations performed did not require this maximum size to be altered.) The main procedures provided are

CLEAR	to initially blank out the soup array.
P_SOUP	to program one particular multiplexor component within a soup.
PAD_OUT	to fill in any unused multiplexor capacity once a circuit description is finished so that the description is 'complete'.
WRITE_OUT	print out the padded circuit array in suitable form for input into the stub filter (soup-face input processor).

Some library some routines provide diagnostic information. The procedure P_SOUP will flag an attempt to program an element of a multiplexor which has already been programmed by a previous call to P_SOUP. The procedure PAD_OUT reports the number of multiplexor elements which have required padding.

```
{PROGRAM: The PASCAL library for the users application routine. The users routine is          }
{         %included, and is initially passed an empty circuit array.                           }
{         The output from this program is one or more three-dimensional 'complete' arrays of   }
{         'stubs' which describe the required circuit. The output must be run through a filter}
{         to break down the stubs into the required 6-bit binary programs before input to the  }
{         simulator.                                                                           }

PROGRAM input_processor(input,output,soup_out,user);
                                      {INPUT, OUTPUT : Used for diagnostics.                 }
                                      {SOUP_OUT      : Output of the three-D circuit arrays.}
                                      {USER          : Predefined for use by the %included   }
                                      {                routine if required.                  }
CONST max_height =  30;               {The maximum height, width and depth of the circuit    }
      max_width  =  30;               {arrays. Larger values require longer initialisation.  }
      max_depth  =  30;
      no_of_faces = 6;                {Number of faces in a cubic cell                       }

      up    = 1 ;  down  = 4 ;  left  = 2 ; {The indices to the faces, corresponding with the }
      right = 5 ;  front = 3 ;  back  = 6 ; {indices used in the stub filter and simulator.   }

      to_memory     = 1;              {Multiplexor output indices. Output 1 goes to the      }
      to_nand       = 2;              {following cells 1-bit memory, output 2 to one input   }
      to_other_dim  = 3;              {of the following cells NAND, and output 3 to the cell}
                                      {in the next dimension.                                }
      output_from_nand      = 'N';    {The names defining which signal to place on which     }
      output_from_other_dim = 'F';    {multiplexor output. For example 'N' on output 1 means}
      output_from_memory    = 'M';    {place the NAND output on the input to the following   }
      output_undefined      = '?';    {cells memory.                                         }
      {also have 0 and 1 outputs '0','1'}

      cell_length = 3;                {The number of outputs from the multiplexor, defining  }
                                      {the maximum stub length.                              }
TYPE  cell_type = PACKED ARRAY[1..cell_length] OF char; {One stub.                           }
      soup_type = ARRAY[1..max_height,1..max_width,1..max_depth,1..no_of_faces] OF cell_type;
                                      {A 3-d array of cubic cells.                           }
VAR   soup:soup_type;
      soup_out,user:text;
      height,width,depth,i:integer;
      pr,su:ARRAY[1..no_of_faces] OF 1..no_of_faces; {Initialised as 'predecessor' and      }
                                      {'successor' arrays for the face indices. For example }
                                      {the cell in dimension following 'up' is su[up]        }
{PROCEDURE: To initialise the circuit array blank.                                           }

PROCEDURE clear(VAR soup:soup_type);
VAR   h,w,d,f:integer;
BEGIN
 writeln('Initialisation begins ...');
 FOR h:=1 TO max_height DO
  FOR w:=1 TO max_width DO
   FOR d:=1 TO max_depth DO
    FOR f:=1 TO no_of_faces DO soup[h,w,d,f]:='   ';
 writeln('          ...   initialisation ends.');
END;

{PROCEDURE: To find the largest actual height width and depth that were used by the user, to }
{           determine how big a circuit needs to be simulated. For most of my simulations this}
{           was not required as calculations could easily be made to determine size.         }

PROCEDURE inquire_size(VAR max_h,max_w,max_d:integer;VAR soup:soup_type);
VAR   h,w,d,f:integer;
BEGIN
 max_h:=1;max_w:=1;max_d:=1;
 FOR h:=1 TO max_height DO
  FOR w:=1 TO max_width DO
   FOR d:=1 TO max_depth DO
    FOR f:=1 TO no_of_faces DO
     IF soup[h,w,d][f]<>'   ' THEN
      IF h>max_h THEN max_h:=h
       ELSE IF w>max_w THEN max_w:=w
        ELSE IF d>max_d THEN max_d:=d;
END;

{PROCEDURE: The standard procedure for programming part of a cell stub. This procedure will  }
{           give error diagnostics for illegal program combinations and any attempt to       }
{           reprogram a multiplexor output which has already been defined.                   }

PROCEDURE p_soup(VAR face:cell_type;mux_index:integer;data:char);
VAR   i:integer;
BEGIN
 FOR i:=1 TO cell_length DO             {Lock cells have ALL their mux outputs predefined, so }
  IF (face[i] IN ['L','K','P']) THEN    {attempting to program them must be an error.         }
   writeln('ERROR: ',face,' <- attempted to reprogram a ''lock'' cell.');

 IF (face[mux_index]=' ') OR (face[mux_index]=data) THEN
   BEGIN
    IF (face[mux_index]=data) THEN writeln('?',mux_index,' ',data,':',face);
                                      {Attempting to reprogram a mux output with the value   }
                                      {it has already been programmed to.                    }
    face[mux_index]:=data;
    IF (face[to_memory]=face[to_other_dim]) AND (face[to_memory]<>' ') THEN
                                                                  writeln('ERROR: ',face);
                                   {The mux output to memory must be different to the mux}
```

```
                                            {output to the other dimension.                    }
   END
  ELSE writeln('Attempt to reprogram :''',face,''' ',mux_index,' ',data);
 END;

 {PROCEDURE: To take a users circuit description array and make it 'complete' by padding out  }
 {           the stub. Note that the method used below is not the only way of padding, a good  }
 {           algorithm needs to program all of the unused multiplexor capacity without         }
 {           interfering with the user circuit.                                                }

 PROCEDURE pad_out(VAR soup:soup_type;height,width,depth:integer);
 VAR   h,w,d,f,pad_count,laid_count,i:integer;
 BEGIN
  writeln('padding ...');
  pad_count:=0; laid_count:=0;
  FOR h:=1 TO height DO
   FOR w:=1 TO width DO
    FOR d:=1 TO depth DO
     FOR f:=1 TO no_of_faces DO
      IF NOT (soup[h,w,d][f][1] IN ['L','P']) THEN {mux outputs predefined; no padding required}
       BEGIN
        FOR i:=1 TO cell_length DO
         IF soup[h,w,d][f][i]=' ' THEN pad_count:=pad_count+1
                                  ELSE laid_count:=laid_count+1;

        IF soup[h,w,d][f][to_memory]=' ' THEN {Pad memory input as NAND output if possible.     }
         IF soup[h,w,d][f][to_other_dim]='N' THEN soup[h,w,d][f][to_memory]:='F'
                                             ELSE soup[h,w,d][f][to_memory]:='N';
        IF soup[h,w,d][f][to_nand]=' '      THEN soup[h,w,d][f][to_nand]:='F'; {poss. undefined}
        IF soup[h,w,d][f][to_other_dim]=' ' THEN soup[h,w,d][f][to_other_dim]:='?'; {undefined }
       END;
  writeln(laid_count,' things placed'); writeln(pad_count,' pads performed');
 END;

 {PROCEDURE: To output the circuit description.                                                 }

 PROCEDURE write_out(VAR soup:soup_type;height,width,depth:integer;wait_time:integer);
 VAR   h,w,d,f:integer;
 BEGIN
  pad_out(soup,height,width,depth);
  writeln(soup_out,height,width,depth);
  FOR h:=1 TO height DO                {The ordering of these loops must correspond with the }
   FOR w:=1 TO width DO                {ordering in the proceeding filter.                   }
    FOR d:=1 TO depth DO
     BEGIN
      FOR f:=1 TO no_of_faces DO write(soup_out,soup[h,w,d][f],' ');
      writeln(soup_out);
     END;
  writeln(soup_out,wait_time);
 END;

 {PROCEDURE: To perform a complete reprogram of the machine. This is broken down by the filter }
 {           into a series of reprogram codes ('111111'), one for each cell on an outside face.}

 PROCEDURE reprogram(height,width,depth,wait_time:integer);
 VAR   h,w,d:integer;
 BEGIN
  writeln(soup_out,-1*height,-1*width,-1*depth);
  writeln(soup_out,wait_time);                 {Wait_time is the number of 'load' cycles to wait    }
 END;                                          {whilst the soup settles before continuing.         }

 {PROCEDURE: Places a cell in lock mode iff it has had no multiplexors programmed.              }

 PROCEDURE use_in_lock_mode(VAR face:cell_type);
 BEGIN
  IF face<>'      ' THEN writeln('ERROR: attempt to use defined cell as a lock cell : ',face)
              ELSE face:='L   ';
 END;

 {PROCEDURE: Use lock iff the cell is in lock mode, error if attempt to program lock bit twice.}

 PROCEDURE use_lock(VAR face:cell_type);
 BEGIN
  IF (face='L   ') OR (face='LP ') THEN IF face='LP ' THEN face:='LKP'
                                                    ELSE face:='LK '
                                  ELSE writeln('ERROR: cannot use_lock on ''',face,'''');
 END;

 {PROCEDURE: Use prog iff the cell is in lock mode, error if attempt to program prog bit twice.}

 PROCEDURE use_programmer(VAR face:cell_type);
 BEGIN
  IF (face='L   ') OR (face='LK ') THEN IF face='LK ' THEN face:='LKP'
                                                    ELSE face:='LP '
                                  ELSE writeln('ERROR: cannot use_programmer on ''',face,'''');
 END;

 {PROCEDURE: To route between two cells at the same height width and depth but in a different  }
 {           dimension by wiring through any intermediarry cells at this location.             }

 PROCEDURE route_between(h,w,d:integer;from_face:integer;from_mux_output:char;to_face:integer;
                                                                         VAR soup:soup_type);
```

```
BEGIN
 p_soup(soup[h,w,d][from_face],to_other_dim,from_mux_output);
 REPEAT
  from_face:=su[from_face];
  IF from_face<>to_face THEN p_soup(soup[h,w,d][from_face],to_other_dim,output_from_other_dim);
 UNTIL (from_face=to_face);
END;

{PROCEDURE: A stub for an inverting wire of length 1 cell. Abut cells for longer wires.       }

PROCEDURE wire1(h,w,d:integer;face:integer;VAR soup:soup_type);
BEGIN
 p_soup(soup[h,w,d][pr[face]],to_other_dim,'1'); {Make one input to the NAND a 1.              }
 p_soup(soup[h,w,d][face],to_nand,output_from_nand); {Pass NAND result to next cells NAND gate}
END;

%include user_routine

BEGIN
 writeln; writeln('SOUP input routine processor      V2.00'); writeln; writeln;
 rewrite(soup_out,'/tmp/soup.input');
 clear(soup);
 height:=0; width:=0; depth:=0;
 FOR i:=2 TO no_of_faces-1 DO  BEGIN  pr[i]:=i-1; su[i]:=i+1;  END;
 pr[1]:=no_of_faces; su[1]:=2; pr[no_of_faces]:=no_of_faces-1; su[no_of_faces]:=1;
 main(soup,height,width,depth);        {Call the users routine.                               }
END.
```

User routines

This appendix shows the user routines which were used for the illustrations in Chapter 5, namely for the simple loading example, the external reprogramming example, the edge detection example and the internal reprogramming example. The routines were combined with, and make extensive use of, the library routines in Appendix 2. In the simulation environment described in Chapter 5, the user routine can be seen at the top right of Figure 5.1.

The first routine is used to illustrate the external loading process associated with the 'string' architecture that was implemented. In this example the circuit array set up by the user routine was identical for every multiplexor. It consisted of generating a zero on the input to the multiplexors and routing this to the memory element which directly follows each multiplexor. The order of loading could be observed by examining the order in which the memory elements of the soup were set to zero.

The second routine is used to show external reprogramming, by using the circuit of Figure 5.3. The procedure set_up_picture_data is used to set up a two dimensional input image in a plane of memory elements. The procedure copy_to_faces copies this image to the right of the soup and also either front and back or up and down depending upon the parameters the procedure is called with. The main program calls the procedure twice to generate the two circuit descriptions that are required, Plate 5.4 and Plate 5.5.

The third routine is the edge detection circuit. The procedure set_up_picture_data is similar to the previous example except that the pixels of the image are spaced further apart to allow for the size of the edge detection circuitry required, see Figure 5.4. The procedure place_1bit_detect is the circuitry for a one bit edge detector using the algorithm

(X nand (A nand B)) nand (X nand (C nand D))

as described in Chapter 5. The procedure route_image provides the required routing between the pixels of the input image and the edge detectors.

The fourth routine is a more complex example which shows internal reprogramming. The circuit is illustrated in Figure 5.5. The procedure delay_block creates the required delay line using long chains of memory elements. It also generates the feedback path and the register. The register decode is performed by the procedure register_decode, the read-only memory and associated address lines and routing of data by memory_block, the actual programmer circuits and appropriate paths for control signals are set up by programmer_circuit, and the display area is set up by the procedure display_area. The main part of the user routine positions these elements with the required relative spacings according to the size of the input image.

```
{                                                                                            }
{USER ROUTINE: Used to illustrate a 'load'. Every cell places a zero on it's output to the   }
{              following cells memory.                                                       }

PROCEDURE main(VAR soup:soup_type;VAR sheight,swidth,sdepth:integer);
CONST max_h= 10; max_w= 10; max_d= 10;  {For a 10 by 10 by 10 circuit.                      }
VAR   x,y,z,h:integer;
BEGIN
 FOR x:=1 TO max_h DO                   {For every cube in the soup ...                     }
  FOR y:=1 TO max_w DO
   FOR z:=1 TO max_d DO
    FOR h:=1 TO 6 DO                    {For every face processor in the cube ...           }
     BEGIN
     p_soup(soup[x,y,z][h],to_memory,output_from_other_dim);  {Make next cells memory input }
                                        {the input to this cell from the previous dimension. }
      p_soup(soup[x,y,z][pr[h]],to_nand,'0');  {Send a zero from the previous dimension.     }
      p_soup(soup[x,y,z][pr[h]],to_other_dim,output_from_nand);
     END;
 write_out(soup,max_h,max_w,max_d,0);   {Output this circuit array.                         }
END;



{                                                                                            }
{USER ROUTINE: The external reprogramming example. Two images are read from a data file and  }
{              routed to different areas within the soup.                                    }

PROCEDURE main(VAR soup:soup_type;VAR sheight,swidth,sdepth:integer);
CONST indent = 2;
VAR   hl,wl,dl:integer;

{FUNCTION: To read one 'bit' of data from the input file.                                   }
FUNCTION read_pixel(VAR f:text):boolean;
VAR  ch:char;
BEGIN
 IF EOLN(f) THEN read_pixel:=FALSE
           ELSE BEGIN read(f,ch);
                      IF ch<>' ' THEN read_pixel:=true  ELSE read_pixel:=false;
                END;
END;

{PROCEDURE: To set up and display the input image on the left hand side of the soup.         }
PROCEDURE set_up_picture_data;
VAR   x,y,max_x,max_y,h,w,d:integer;
      image:PACKED ARRAY[1..100,1..100] OF boolean;
BEGIN
 readln(user,max_x,max_y);              {The height and width of the image.                 }
 FOR y:=max_y DOWNTO 1 DO               {Read in the data.                                  }
  BEGIN FOR x:=1 TO max_x DO image[x,y]:=read_pixel(user);
        readln(user);
  END;
                                        {For each column ...                                }
 FOR x:=1 TO max_x DO
  BEGIN                                 {For each row ...                                   }
   FOR y:=1 TO max_y DO
    IF image[x,y]=TRUE THEN             {If the image is a 1 ...                            }
     BEGIN
      h:=y+indent; d:=x+indent; write('.'); {Calculate appropriate height and depth in soup. }
      p_soup(soup[h,1,d][left],to_memory,output_from_other_dim);
      p_soup(soup[h,1,d][pr[left]],to_other_dim,'1'); {Display a 1 on 'left' proc. at (h,2,d).}
      p_soup(soup[h,2,d][left],to_nand,output_from_memory);
     END  ELSE  BEGIN                   {Otherwise the image is a 0 so ...                  }
      h:=y+indent;d:=x+indent; write(' '); {Calculate appropriate height and depth in soup.  }
      p_soup(soup[h,2,d][left],to_nand,output_from_memory);
      p_soup(soup[h,1,d][left],to_memory,output_from_other_dim);
      p_soup(soup[h,1,d][pr[left]],to_other_dim,output_from_nand);
      p_soup(soup[h,1,d][pr[left]],to_nand,'0'); {Display a zero.                            }
     END;
   writeln;
  END;                                  {Calculate the maximum soup height, width and depth }
 sheight:=max_x+2*indent;  swidth:=2;  sdepth:=max_y+2*indent; {required by this procedure. }
END;

{PROCEDURE: To copy an image from the left of the soup to the right, and also either up and  }
{           down faces or back and front faces.                                              }
PROCEDURE copy_to_faces(f1,f2:integer); {The valid combinations are f1=up   & f2=down  or   }
VAR   h,w,d,width_lo,width_hi,i:integer;{                                   f1=back & f2=front}
BEGIN
 width_lo:=swidth;                      {The starting width is the input images width.      }
 IF sheight>sdepth THEN width_hi:=sheight-2 ELSE width_hi:=sdepth-2; {To provide enough space }
                                        {to route up and down as well as left and right.    }
 IF ((width_hi MOD 2)=0) THEN width_hi:=width_hi+1; {An odd value would result in an inverted }
                                        {image as routing is via chains of inverting NOR gates}
 FOR h:=indent+1 TO sheight-indent DO   {For every pixel in the image ...                   }
  FOR d:=indent+1 TO sdepth-indent DO
   FOR w:=1 TO width_hi DO              {For every point in the routing area ...            }
    BEGIN
     wire1(h,w+width_lo,d, left, soup); {Copy input to the right.                           }
     IF ((d-indent=w) AND (f1=back)) THEN {If on the diagonal then route to the front and back}
      BEGIN
       route_between(h,w+width_lo+1,d, left, output_from_memory,f1,soup);
```

```
        p_soup(soup[h,w+width_lo+1,d][f2],to_nand,output_from_other_dim);
        p_soup(soup[h,w+width_lo+1,d][f1],to_nand,output_from_other_dim);
        FOR i:=1 TO (w-1) DO wire1(h,w+width_lo+1,d-i, f2, soup); {Route to the front.          }
        FOR i:=1 TO (width_hi-1-w-1) DO wire1(h,w+width_lo+1,d+i, f1, soup); {Route to back.    }
        p_soup(soup[h,w+width_lo+1,2][pr[f2]],to_other_dim,'1'); {Display at front.             }
        p_soup(soup[h,w+width_lo+1,2][f2],to_memory,output_from_nand);
        p_soup(soup[h,w+width_lo+1,sdepth-1][pr[f1]],to_other_dim,'1'); {Display at back.      }
        p_soup(soup[h,w+width_lo+1,sdepth-1][f1],to_memory,output_from_nand);
       END
      ELSE
       IF ((h-indent=w) AND (f1=up)) THEN {If on the diagonal then route up and down.           }
        BEGIN
         route_between(h,w+width_lo+1,d, left, output_from_memory,f1,soup);
         p_soup(soup[h,w+width_lo+1,d][f2],to_nand,output_from_other_dim);
         p_soup(soup[h,w+width_lo+1,d][f1],to_nand,output_from_other_dim);
         FOR i:=1 TO (w-1) DO wire1(h-i,w+width_lo+1,d, f2, soup); {Route to the bottom.       }
         FOR i:=1 TO (width_hi-1-w-1) DO wire1(h+i,w+width_lo+1,d, f1, soup); {Route to top.   }
         p_soup(soup[2,w+width_lo+1,d][pr[f2]],to_other_dim,'1'); {Display at the bottom.      }
         p_soup(soup[2,w+width_lo+1,d][f2],to_memory,output_from_nand);
         p_soup(soup[sheight-1,w+width_lo+1,d][pr[f1]],to_other_dim,'1'); {Display at the top.}
         p_soup(soup[sheight-1,w+width_lo+1,d][f1],to_memory,output_from_nand);
        END
  END;                                      {Width is the input image width plus the routing area }
  swidth:=swidth+width_hi+2;                {plus the right-hand display area.                    }
  FOR h:=indent+1 TO sheight-indent DO      {The right-hand display area ...                      }
   FOR d:=indent+1 TO sdepth-indent DO
    BEGIN  p_soup(soup[h,swidth-1,d][left],to_memory,output_from_nand);
           p_soup(soup[h,swidth-1,d][pr[left]],to_other_dim,'1');
    END;
END;

BEGIN
 reset(user,'pic.dat');
 set_up_picture_data;                       {Read in the first image into the soup.               }
 copy_to_faces(back,front);                 {Route it to the back and to the front.               }
 write_out(soup,height,width,depth,20);
 reprogram(height,width,depth,1);          {After a delay (20 load times) place a 111111 message }
                                           {on the face of every cell on the edge of the soup. As}
                                           {no cells have been 'locked' this has the effect of   }
                                           {completely reprogramming the machine.                }
 clear(soup);                              {Clear the circuit array ready for the next circuit   }
 set_up_picture_data;                      {description. } {Set up the second image.             }
 copy_to_faces(up,down);                   {This time copy it up and down, but not back or front.}
 write_out(soup,height,width,depth,0);
END;



{                                                                                                  }
{USER ROUTINE: Edge detection. This has four main components: the input, a routing section,       }
{              the edge detection section, and displaying the results (done by 'edge_detect').}

PROCEDURE main(VAR soup:soup_type;VAR sheight,swidth,sdepth:integer);
CONST indent = 0;  pix_height=3;  pix_depth =3; {The required pixel spacing of the input image}
VAR   h1,w1,d1,max_y,max_x:integer;

FUNCTION read_pixel(VAR f:text):boolean; {As previously explained}
VAR  ch:char;
BEGIN
 IF EOLN(f) THEN read_pixel:=FALSE
            ELSE BEGIN read(f,ch);
                       IF ch<>' ' THEN read_pixel:=true  ELSE read_pixel:=false;
                 END;
END;

{PROCEDURE: As previously explained, but note the changes in calculation of pixel h and d.    }
PROCEDURE set_up_picture_data;
VAR   x,y,h,w,d:integer;
      image:PACKED ARRAY[1..100,1..100] OF boolean;
BEGIN
 readln(user,max_x,max_y);
 FOR y:=max_y DOWNTO 1 DO
  BEGIN FOR x:=1 TO max_x DO image[x,y]:=read_pixel(user);
        readln(user);
  END;

 FOR x:=1 TO max_x DO
  BEGIN
   FOR y:=1 TO max_y DO
    IF image[x,y]=TRUE THEN
     BEGIN write('.');
      h:=(y-1)*pix_height + (pix_height+1) DIV 2 +indent;
      d:=(x-1)*pix_depth  + (pix_depth+1 ) DIV 2 +indent;
      p_soup(soup[h,1,d][left],to_memory,output_from_other_dim);
      p_soup(soup[h,1,d][pr[left]],to_other_dim,'1');  {Output is a 1.                           }
     END  ELSE  BEGIN write(' ');
      h:=(y-1)*pix_height + (pix_height+1) DIV 2 +indent;
      d:=(x-1)*pix_depth  + (pix_depth+1 ) DIV 2 +indent;
      p_soup(soup[h,1,d][left],to_memory,output_from_other_dim);
      p_soup(soup[h,1,d][pr[left]],to_other_dim,output_from_nand);
      p_soup(soup[h,1,d][pr[left]],to_nand,'0');  {Output is a zero.                          }
      END;
     writeln;
    END;                                  {As before, except height and depth are now bigger.    }
  sheight:=max_x*pix_height+2*indent;  swidth:=2;  sdepth:=max_y*pix_depth+2*indent;
 END;

 {PROCEDURE: Performs the edge detection.                                                       }
 PROCEDURE edge_detect;
 VAR   h,w,d:integer;

   {PROCEDURE: A one bit edge detector. The output is a 1 iff X is a 1 and one or more of the  }
   PROCEDURE place_1bit_detect(h,w,d:integer);  {neighbouring pixels (a,b,c,d) are zero.        }
   VAR    hi,wi,di:integer;
   BEGIN
    writeln('Placing 1 bit detector at ',h,w,d);                   {Diagnostic.                 }
    p_soup(soup[h+1,w  ,d+2][left],to_nand,output_from_memory);  {Route A and B together and   }
    p_soup(soup[h+1,w+1,d+2][pr[left]],to_other_dim,'1');         {NAND them to give  A NAND B. }
    p_soup(soup[h+1,w+1,d+2][left],to_nand,output_from_nand);
    route_between(h+1,w+1,d+2,left,output_from_nand, front,soup);
    p_soup(soup[h+1,w+1,d+2][front],to_memory,output_from_other_dim);
    p_soup(soup[h+1,w+1,d+1][front],to_nand,output_from_memory);
    p_soup(soup[h+1,w+1,d][pr[front]],to_other_dim,'1');
    p_soup(soup[h+1,w+1,d][front],to_nand,output_from_nand);
    route_between(h+1,w+1,d,front,output_from_nand, left,soup);
    p_soup(soup[h+1,w,d][left],to_nand,output_from_memory);
    p_soup(soup[h+1,w+1,d][left],to_nand  ,output_from_nand);
    p_soup(soup[h+1,w+2,d][left],to_memory,output_from_nand);
    p_soup(soup[h+2,w  ,d+1][left],to_nand,output_from_memory);  {Route C and D together and   }
    p_soup(soup[h+2,w+1,d+1][pr[left]],to_other_dim,'1');         {NAND them to give  C NAND D. }
    p_soup(soup[h+2,w+1,d+1][left],to_nand,output_from_nand);
    route_between(h+2,w+1,d+1,left,output_from_nand, down,soup);
    p_soup(soup[h+2,w+1,d+1][down],to_memory,output_from_other_dim);
    p_soup(soup[h+1,w+1,d+1][down],to_nand,output_from_memory);
    p_soup(soup[h,w+1,d+1][pr[down]],to_other_dim,'1');
    p_soup(soup[h,w+1,d+1][down],to_nand,output_from_nand);
    route_between(h,w+1,d+1,down,output_from_nand, left,soup);
    p_soup(soup[h,w,d+1][left],to_nand,output_from_memory);
    p_soup(soup[h,w+1,d+1][left],to_nand  ,output_from_nand);
    p_soup(soup[h,w+2,d+1][left],to_memory,output_from_nand);
    p_soup(soup[h+1,w  ,d+1][left],to_nand,output_from_memory); {move out X to this width.     }
    p_soup(soup[h+1,w+1,d+1][pr[left]],to_other_dim,'1');
    p_soup(soup[h+1,w+1,d+1][left],to_memory,output_from_nand);
    route_between(h+1,w+2,d+1,left,output_from_memory,down,soup);
    p_soup(soup[h+1,w+2,d+1][front],to_nand  ,output_from_other_dim); {NAND X with the two     }
    p_soup(soup[h+1,w+2,d+1][down ],to_nand  ,output_from_other_dim); {previous results to give}
    p_soup(soup[h  ,w+2,d+1][pr[down]],to_other_dim,'1');              { (X NAND (A NAND B)) and}
    p_soup(soup[h,w+2,d+1][down],to_nand,output_from_nand);            { (X NAND (C NAND D)).   }
    route_between(h  ,w+2,d+1,down,output_from_nand, left,soup);
    p_soup(soup[h+1,w+2,d  ][pr[front]],to_other_dim,'1');
    p_soup(soup[h+1,w+2,d  ][front],to_nand,output_from_nand);{for route_between}
    route_between(h+1,w+2,d  ,front,output_from_nand, left,soup);
    route_between(h  ,w+3,d+1,left,output_from_memory, up,soup);      {NAND these two values.    }
    p_soup(soup[h  ,w+3,d+1][up],to_memory,output_from_other_dim);
    route_between(h+1,w+3,d+1,up,output_from_memory, front,soup);
    p_soup(soup[h+1,w+3,d+1][front],to_memory,output_from_other_dim);
    route_between(h+1,w+3,d,front,output_from_memory, left,soup);
                                        {Display the edge detection result:                  }
                                        {  (X NAND (A NAND B) NAND (X NAND (C NAND D))        }
   p_soup(soup[h+1,w+2,d][left],to_nand  ,output_from_nand);
   p_soup(soup[h+1,w+3,d][left],to_memory,output_from_nand);
  END;

  {PROCEDURE: Route the input image data to the 1 bit edge detectors as required.              }
  PROCEDURE route_image(h,w,d:integer);

    FUNCTION in_soup(h,d:integer):boolean; {Returns true if (h,d) is part of the image.        }
    BEGIN
     IF (h>=1) AND (h<=sheight) AND (d>=1) AND (d<=sdepth) THEN in_soup:=true
                                                            ELSE in_soup:=false;
    END;

    {PROCEDURE: Forces a zero into the memory of cell [h,w+1,d][left].                         }
    PROCEDURE zero_left(h,w,d:integer);
    BEGIN
      p_soup(soup[h,w,d][pr[left]],to_nand,'0');
      p_soup(soup[h,w,d][pr[left]],to_other_dim,output_from_nand);
      p_soup(soup[h,w,d][left],to_memory,output_from_other_dim);
    END;

  BEGIN
   route_between(h,w,d,left, output_from_memory, left,soup);  {Copy input pixel to the output }
   p_soup(soup[h,w,d][left],to_memory,output_from_other_dim); {directly opposite.             }
   route_between(h,w+1,d,left, output_from_memory, left,soup);
   p_soup(soup[h,w+1,d][left],to_memory,output_from_other_dim);
   IF in_soup(h+2,d) THEN
     BEGIN                              {If there is a pixel above then route this data up for}
      p_soup(soup[h,w,d][up],to_nand,output_from_other_dim); {use by it's 1 bit edge detector.}
      p_soup(soup[h+1,w,d][pr[up]],to_other_dim,'1');
      p_soup(soup[h+1,w,d][up],to_memory,output_from_nand);
      route_between(h+2,w,d,up, output_from_memory, left,soup);
      p_soup(soup[h+2,w,d][left],to_nand  ,output_from_other_dim);
      p_soup(soup[h+2,w+1,d][pr[left]],to_other_dim,'1');
```

```
       p_soup(soup[h+2,w+1,d][left],to_memory,output_from_nand);
      END  ELSE zero_left(h+1,w+1,d);    {If not then place a zero on the top edge of the data.}
    IF in_soup(h-2,d) THEN
      BEGIN                               {If there is a pixel below then route this data down. }
       p_soup(soup[h,w+1,d][down],to_nand,output_from_other_dim);
       p_soup(soup[h-1,w+1,d][pr[down]],to_other_dim,'1');
       p_soup(soup[h-1,w+1,d][down],to_nand,output_from_nand);
       p_soup(soup[h-2,w+1,d][pr[down]],to_other_dim,'1');
       p_soup(soup[h-2,w+1,d][down],to_nand,output_from_nand); {for route_between}
       route_between(h-2,w+1,d,down, output_from_nand, left,soup);
       p_soup(soup[h-2,w+1,d][left],to_memory,output_from_other_dim);
      END  ELSE zero_left(h-1,w+1,d);    {If not then place a zero on the bottom edge.         }
    IF in_soup(h,d+2) THEN
      BEGIN                               {If there is a pixel to the right (deeper in the soup)}
       p_soup(soup[h,w,d][back],to_nand,output_from_other_dim); {then route further back.      }
       p_soup(soup[h,w,d+1][pr[back]],to_other_dim,'1');
       p_soup(soup[h,w,d+1][back],to_memory,output_from_nand);
       route_between(h,w,d+2,back, output_from_memory, left,soup);
       p_soup(soup[h,w,d+2][left],to_nand  ,output_from_other_dim);
       p_soup(soup[h,w+1,d+2][pr[left]],to_other_dim,'1');
       p_soup(soup[h,w+1,d+2][left],to_memory,output_from_nand);
      END  ELSE zero_left(h,w+1,d+1);    {If not then place a zero on the righthand edge.      }
    IF in_soup(h,d-2) THEN
      BEGIN                               {If there is a pixel to the left (shallower) then     }
       p_soup(soup[h,w+1,d][front],to_nand,output_from_other_dim); {route towards the front.   }
       p_soup(soup[h,w+1,d-1][pr[front]],to_other_dim,'1');
       p_soup(soup[h,w+1,d-1][front],to_nand,output_from_nand);
       p_soup(soup[h,w+1,d-2][pr[front]],to_other_dim,'1');
       p_soup(soup[h,w+1,d-2][front],to_nand,output_from_nand); {for route_between}
       route_between(h,w+1,d-2,front, output_from_nand, left,soup);
       p_soup(soup[h,w+1,d-2][left],to_memory,output_from_other_dim);
      END  ELSE zero_left(h,w+1,d-1);    {If not then place a zero on the lefthand edge.       }
  END;

BEGIN
 FOR h:=1 TO max_y   DO                  {For every pixel in the image ...                      }
  FOR d:=1 TO max_x  DO
   BEGIN
    route_image((h-1)*pix_height+2,swidth,(d-1)*pix_depth+2);
    place_1bit_detect((h-1)*pix_height+1,swidth+2,(d-1)*pix_depth+1);
   END;
 swidth:=swidth+2+5 ;                    {Width is image width plus routing width (2) plus the }
END;                                     {edge detector and display area width (5).            }

BEGIN
 reset(user,'pic.dat');
 set_up_picture_data;
 edge_detect;
 write_out(soup,height,width,depth,0);
END;




{                                                                                              }
{USER ROUTINE: The internal reprogramming example. The major components of this are a delay   }
{              line counter, a register, decode, memory, and internal reprogramming cells.    }

PROCEDURE main(VAR soup:soup_type;VAR sheight,swidth,sdepth:integer);
{NOTE: must be loaded using FORCELOAD because of 'lock' cells. (Explained in main text).       }
TYPE  image_type=PACKED ARRAY[1..100,1..100] OF boolean;
VAR   image:image_type;
      x,y,m_x,m_y,max_x,max_y,max_z,pso_h,pso_w,pso_d:integer;

{PROCEDURE: To copy the input signal in the memory cell on face 'f' to the next cells memory. }
PROCEDURE r_straight(h,w,d,f:integer);       {route_straight}
BEGIN  route_between(h,w,d,f,output_from_memory,f,soup); {Route through all 6 faces.          }
       p_soup(soup[h,w,d][f],to_memory,output_from_other_dim); {Foreward the answer.           }
END;

{PROCEDURE: To route the input signal from memory at (h,w,d,f) up 1 cell and back in the      }
PROCEDURE r_up(h,w,d,f:integer);             {route_up}                  { opposite direction. }
BEGIN  route_between(h,w,d,f,output_from_memory,up,soup); {route from 'f' to 'up'.            }
       p_soup(soup[h,w,d][up],to_memory,output_from_other_dim);
       route_between(h+1,w,d,up,output_from_memory,pr[pr[pr[f]]], soup); {route from 'up' back}
       p_soup(soup[h+1,w,d][pr[pr[pr[f]]]],to_memory,output_from_other_dim); {in opposite      }
END;                                                                    {direction to 'f'.    }

{PROCEDURE: To create n abutted delay blocks. Delay is obtained through long chains of memory }
PROCEDURE delay_block(n,h_lo,h_hi,w_lo,w_hi,start_depth:integer);                 {elements.  }
VAR   h,w,d:integer;
BEGIN
 IF ((h_hi-h_lo) MOD 2)=0 THEN h_hi:=h_hi-1; {Ensure output is in opposite direction to input.}
 p_soup(soup[h_lo,w_hi-1,start_depth-2][pr[back]],to_other_dim,'1'); {Place a 1 on an input to}
 p_soup(soup[h_lo,w_hi-1,start_depth-2][back],to_nand,output_from_other_dim); {a NAND gate.    }
 p_soup(soup[h_lo,w_hi-1,start_depth-1][back],to_memory,output_from_nand);
 route_between(h_lo-1,w_hi-1,start_depth-1,front, output_from_memory,up,soup); {Place feedback}
 p_soup(soup[h_lo-1,w_hi-1,start_depth-1][up],to_memory,output_from_other_dim);{on the other  }
 route_between(h_lo,w_hi-1,start_depth-1,up, output_from_memory,back,soup);    {input. This   }
 route_between(h_lo,w_hi-1,start_depth+n,back, output_from_memory,down,soup);  {creates an    }
 p_soup(soup[h_lo,w_hi-1,start_depth+n][down],to_memory,output_from_other_dim);{infinite loop.}
 route_between(h_lo-1,w_hi-1,start_depth+n,down, output_from_memory,front,soup);
 p_soup(soup[h_lo-1,w_hi-1,start_depth+n][front],to_memory,output_from_other_dim);
 FOR d:=start_depth TO (start_depth+n-1) DO
  BEGIN
   r_straight(h_lo-1,w_hi-1,d,front);   {The feedback path.                            }
   route_between(h_lo,w_hi-1,d,back, output_from_memory,right,soup); {The input signal to this}
   p_soup(soup[h_lo,w_hi-1,d][right],to_memory,output_from_other_dim); {delay line.           }
   route_between(h_lo,w_hi,d,down, output_from_memory,right,soup);   {The output signal from  }
   p_soup(soup[h_lo,w_hi,d][right],to_memory,output_from_other_dim);   {this delay line.      }
   route_between(h_lo,w_hi-1,d,right, output_from_memory,back,soup); {The delay line ...      }
   p_soup(soup[h_lo,w_hi-1,d][back],to_memory,output_from_other_dim);
   r_straight(h_hi,w_hi-1,d,left);
   route_between(h_hi,w_hi,d,left, output_from_memory,down,soup);
   p_soup(soup[h_hi,w_hi,d][down],to_memory,output_from_other_dim);
   FOR h:=h_lo+1 TO h_hi-1 DO r_straight(h,w_hi,d,down);
   p_soup(soup[h_lo+2,w_hi,d][left],to_nand,output_from_other_dim);   {Output the answer: the }
   p_soup(soup[h_lo+2,w_hi+1,d][pr[left]],to_other_dim,'1');          {delay line register.   }
   p_soup(soup[h_lo+2,w_hi+1,d][left],to_nand,output_from_nand);
   p_soup(soup[h_lo+2,w_hi+2,d][pr[left]],to_other_dim,'1');
   p_soup(soup[h_lo+2,w_hi+2,d][left],to_memory,output_from_nand);
   FOR h:=h_lo TO h_hi DO                {Main body: long chains of memory elements.        }
    IF (((h-h_lo) MOD 2) = 0) THEN
     BEGIN
      FOR w:=w_hi-2 DOWNTO w_lo+1 DO r_straight(h,w,d,right);
      r_up(h,w_lo,d,right);
      FOR w:=w_lo+1 TO w_hi-2 DO r_straight(h+1,w,d,left);
      IF (h<>h_hi) AND (h<>h_hi-1) THEN r_up(h+1,w_hi-1,d,left);
     END;
  END;
END;

{PROCEDURE: The display area.                                                              }
PROCEDURE set_up_display_area(h_lo,h_hi,w_lo,d,pic_width:integer);
VAR   x,y,z,h,w_hi,rel_w:integer;

    {PROCEDURE: To set a lock at cell (x,y,z).                                             }
    PROCEDURE loc(bottom:boolean;x,y,z:integer);
    BEGIN                               {Put a 1 on the lock enable line if required.      }
     IF bottom THEN BEGIN  p_soup(soup[x-1,y,z][pr[up]],to_other_dim,'1');
                           p_soup(soup[x-1,y,z][up],to_memory,output_from_other_dim);
                     END;
      use_in_lock_mode(soup[x,y,z][up]);
      use_lock(soup[x,y,z][up]);
    END;

BEGIN
 w_hi:=w_lo+4*pic_width+5; z:=d+1;      {Make display wide enough for 4 images.            }
 writeln('Display :');  writeln(' hlo,hhi',h_lo:3, h_hi:3); {Diagnostics.                  }
 writeln(' wlo,whi',w_lo:3, w_hi:3);  writeln(' dlo,dhi',d:3, d+1:3);
 FOR x:=h_lo TO h_hi DO
  BEGIN
   FOR y:=w_lo TO w_hi DO               {For every cell in the display area ...            }
    BEGIN
     p_soup(soup[x,y,z][front],to_memory,output_from_other_dim); {Copy input to memory.    }
     IF (((x+y) MOD 5)=1) THEN p_soup(soup[x,y,z][pr[front]],to_other_dim,'1') {Place a 1 on }
                          ELSE p_soup(soup[x,y,z][pr[front]],to_other_dim,'?');{the output to }
    END;                                {every 5th diagonal. NB pr[front] = left           }

   loc(x=h_lo,x,w_lo+2,z);              {Place locks at width=2 -this is because of a spike in}
                                        {the circuit sending a 111111 message at switch-on. }
   loc(x=h_lo,x,w_lo+pic_width+2,z);    {Place a column of locks wide enough for one picture. }
   loc(true,x,w_lo+2*pic_width+2+3-2+(x MOD 3),z); {Place the jagged column.               }
   loc(x=h_lo,x,w_lo+((5*pic_width) DIV 2)+2+3,z); {Place another straight column.         }
  END;
END;

{PROCEDURE: To set up a bank of programmer cells.                                          }
PROCEDURE programmer_circuit(bottom:boolean;h,w,d,pso_h,pso_w,pso_d:integer);
VAR i:integer;

  {PROCEDURE: To route the enable signal from the register decode.                         }
  PROCEDURE prog_signal_route(h1,w1,d1,h2,w2,d2:integer);
  VAR h,w,d:integer;  {ordering of routing is front then down then right(through 'left' procs)}
  BEGIN
   IF (h1<h2) OR (w1>w2) OR (d1<d2) THEN writeln('Routing fails'); {Assumptions made about the}
                                                  {location of the register decode block.  }
   FOR d:=d1 DOWNTO d2+1 DO r_straight(h1,w1,d,front); {Route signal to the front.          }
   route_between(h1,w1,d2,front, output_from_memory,down,soup);
   p_soup(soup[h1,w1,d2][down],to_memory,output_from_other_dim);
   FOR h:=h1-1 DOWNTO h2+1 DO r_straight(h,w1,d2,down); {Route signal downwards.           }
   route_between(h2,w1,d2,down, output_from_memory,left,soup);
   p_soup(soup[h2,w1,d2][left],to_memory,output_from_other_dim);
   FOR w:=w1+1 TO w2 DO r_straight(h2,w,d2,left); {Route signal to the right.              }
  END;

BEGIN
 IF bottom THEN                         {Connect the programmer enable signal.             }
  BEGIN  prog_signal_route(pso_h,pso_w,pso_d,h-2,w-1,d);
         p_soup(soup[h-2,w-1,d][left],to_nand,output_from_memory);
  END;
 FOR i:=0 TO 5 DO                       {There are six bits in a the program word.         }
  BEGIN
   IF bottom THEN
```

```
     BEGIN
      p_soup(soup[h-2,w+i,d][pr[left]],to_other_dim,'1'); {Route the enable signal to all of    }
      p_soup(soup[h-2,w+i,d][left],to_nand,output_from_nand); {the cells ...                      }
      route_between(h-2,w+i,d,left, output_from_nand, up,soup);
      IF (i MOD 2)=0 THEN
       BEGIN
        p_soup(soup[h-2,w+i,d][up],to_nand,output_from_other_dim);
        p_soup(soup[h-1,w+i,d][pr[up]],to_other_dim,'1');
        p_soup(soup[h-1,w+i,d][up],to_memory,output_from_nand);
       END  ELSE  BEGIN
        p_soup(soup[h-2,w+i,d][up],to_memory,output_from_other_dim);
        route_between(h-1,w+i,d,up, output_from_memory, up,soup);
        p_soup(soup[h-1,w+i,d][up],to_memory,output_from_other_dim);
       END;
     END;
    use_in_lock_mode(soup[h,w+i,d][up]); {Define as a programmer cell, programmers should be    }
    use_lock(soup[h,w+i,d][up]);                                                       { locked. }
    use_programmer(soup[h,w+i,d][up]);
    route_between(h,w+i,d,front, output_from_memory , up,soup); {Route in the cells data from   }
   END;                                                             { the program store. }
  END;

  FUNCTION read_pixel(VAR f:text):boolean;{To read 1 'bit' of data from the input file.         }
  VAR  ch:char;
  BEGIN
   IF EOLN(f) THEN read_pixel:=FALSE
              ELSE BEGIN read(f,ch);
                         IF ch<>' ' THEN read_pixel:=true  ELSE read_pixel:=false;
                   END;
  END;

  {PROCEDURE: To read in the input image into the boolean array 'image'.                       }
  PROCEDURE set_up_picture_data(VAR image:image_type;VAR m_x,m_y:integer);
  VAR   x,y:integer;
  BEGIN
   readln(user,m_x,m_y);
   FOR y:=m_y DOWNTO 1 DO
    BEGIN  FOR x:=1 TO m_x DO image[x,y]:=read_pixel(user);
           readln(user);
    END;
  END;

  {PROCEDURE: To create a bank of memory holding the required sequence of programs to generate }
  {           a display of memory elements identical to the image.                           }
  PROCEDURE memory_block(h_offset,w_offset,d_offset:integer;image:image_type ;
                                                        image_width,image_height:integer);
  CONST obr_h = 1 ;  obr_w = 2 ;  obr_d = 2 ; {Dimensions of 'one_bit_register' block.         }
  VAR   x,y,z,p,y_pos,picture_width:integer;

    {PROCEDURE: A one bit register which places it's data on the output if it is selected,      }
    {           if not, it copies the output from the previous cell to it's output.             }
    PROCEDURE one_bit_register(data:integer;first,last:boolean;h,w,d:integer);
    BEGIN                              {No more than 1 cell in chain must be selected at a time}
     IF first THEN p_soup(soup[h,w,d][front],to_memory,output_from_nand); {Output the result.   }
     p_soup(soup[h,w,d][front],to_nand,output_from_nand); {Invert result.                        }
     p_soup(soup[h,w,d][pr[front]],to_other_dim,'1');
     route_between(h,w+1,d+1,up,output_from_memory, up,soup);  {The enable signal.               }
     p_soup(soup[h,w+1,d+1][up],to_memory,output_from_other_dim);
     p_soup(soup[h,w+1,d+1][right],to_nand,output_from_other_dim);
     IF data=1 THEN                    {The data to be stored. Note that it is inverted.        }
       BEGIN  p_soup(soup[h,w,d+1][pr[right]],to_nand,'0');
              p_soup(soup[h,w,d+1][pr[right]],to_other_dim,output_from_nand);
       END  ELSE p_soup(soup[h,w,d+1][pr[right]],to_other_dim,'1');
     p_soup(soup[h,w,d+1][right], to_nand,output_from_nand); {For route_between.                }
     route_between(h,w,d+1,right, output_from_nand, front, soup); {NAND previous output with    }
     p_soup(soup[h,w,d+1][front],to_nand,output_from_nand);       {this value.                  }
     IF last THEN                      {Put a 1 at the beginning of the chain.                  }
      BEGIN  p_soup(soup[h,w,d+2][pr[front]],to_other_dim,'1');
             p_soup(soup[h,w,d+2][front],to_nand,output_from_other_dim);
      END;
    END;

  BEGIN
   picture_width:=6;                     {A six bit program, so have a 6 bit wide store.        }
   FOR x:=1 TO image_height DO           {For every 'one_bit_register' about to be placed,      }
   FOR z:=0 TO picture_width-2 DO        {route in the select (enable) signal from the register}
   BEGIN                                 {decode block.                                         }
   y_pos:=picture_width*2-1;
   FOR y:=picture_width DOWNTO 1 DO
    BEGIN
    IF (y>z+1) THEN
     BEGIN y_pos:=y_pos-1;
     route_between(h_offset+x,w_offset+y_pos,d_offset+z,front,output_from_memory,front, soup);
     p_soup(soup[h_offset+x,w_offset+y_pos,d_offset+z][front],to_memory,output_from_other_dim);
     END  ELSE  BEGIN  y_pos:=y_pos-2;
     route_between(h_offset+x,w_offset+y_pos,d_offset+z,front,output_from_memory,left, soup);
     p_soup(soup[h_offset+x,w_offset+y_pos,d_offset+z][left],to_memory,output_from_other_dim);
     route_between(h_offset+x,w_offset+y_pos+1,d_offset+z,left,output_from_memory,front, soup);
     p_soup(soup[h_offset+x,w_offset+y_pos+1,d_offset+z][front],to_memory,output_from_other_dim);
     END;
   END;
   END;
  d_offset:=d_offset+picture_width-1;     {The store routing area: Each one_bit_register is one }
  FOR z:=0 TO image_width DO              {cell high, but two cells wide. This code routes the  }
   FOR x:=0 TO picture_width*obr_w-1 DO   {outputs into abutting blocks one cell high by one    }
   BEGIN                                  {cell wide.                                           }
    p_soup(soup[h_offset,w_offset+x,d_offset+1+z*obr_d][pr[left]],to_other_dim,'1');
    p_soup(soup[h_offset,w_offset+x,d_offset+1+z*obr_d][left],to_nand,output_from_nand);
    route_between(h_offset,w_offset+x,d_offset+1+z*obr_d,left,output_from_nand,up,soup);
    p_soup(soup[h_offset,w_offset+x,d_offset+1+z*obr_d][up],to_memory,output_from_other_dim);
   END;
  h_offset:=h_offset+1;                   {Place the one_bit_registers as required.             }
  FOR z:=0 TO image_width DO
  FOR y:=0 TO image_height-1 DO
   FOR p:=0 TO picture_width-1 DO
    IF z=0 THEN                           {Start with a 111111 program.                         }
      one_bit_register(1,true,z=image_width,h_offset+y*obr_h,w_offset+p*obr_w,d_offset+z*obr_d)
     ELSE
     IF (p=4) THEN
      IF image[image_width-z+1,y+1] THEN  {000000 prog; sends a 1 to cell in the next dimension}
      one_bit_register(0,false,z=image_width,h_offset+y*obr_h,w_offset+p*obr_w,d_offset+z*obr_d)
      ELSE                                {000010 prog; sends a 0 to cell in the next dimension}
      one_bit_register(1,false,z=image_width,h_offset+y*obr_h,w_offset+p*obr_w,d_offset+z*obr_d)
     ELSE
     one_bit_register(0,false,z=image_width,h_offset+y*obr_h,w_offset+p*obr_w,d_offset+z*obr_d);
 END;

 {PROCEDURE: To decode the delay line register signal into memory select and programmer enable.}
 PROCEDURE register_decode(h,w,d,image_depth:integer;VAR pso_h,pso_w,pso_d:integer);
 VAR   i,d_i,i_depth:integer;
 BEGIN
  i_depth:=(image_depth+1)*2;             {Spacing of memory select is every 2 delay registers. }
  FOR i:=0 TO i_depth DO
   IF (i MOD 2)=0 THEN
    BEGIN                                 {Firstly do memory select:                            }
     route_between(h,w,d+i,left,output_from_memory, left,soup);    {Select iff delay register[i]}
     p_soup(soup[h,w,d+i][left],to_memory,output_from_other_dim); {is 1 and delay register[i+2]}
     p_soup(soup[h,w,d+i][left],to_nand,output_from_memory);      {is zero.                    }
     p_soup(soup[h,w+1,d+i][left],to_nand,output_from_nand);
     p_soup(soup[h,w+2,d+i][pr[left]],to_other_dim,'1');
     p_soup(soup[h,w+2,d+i][left],to_memory,output_from_nand);
     route_between(h,w+1,d+i,left,output_from_memory, front,soup);
     p_soup(soup[h,w+1,d+i][front],to_nand,output_from_other_dim);
     p_soup(soup[h,w+1,d+i-1][pr[front]],to_other_dim,'1');
     p_soup(soup[h,w+1,d+i-1][front],to_memory,output_from_nand);
     route_between(h,w+1,d+i-2,front,output_from_memory,left,soup);
     p_soup(soup[h,w+3,d+i][left],to_nand,output_from_memory);    {The memory select result.   }
                                                                {Now do programmer enable signal:}
     p_soup(soup[h,w,d+i][up],to_nand,output_from_other_dim);{(routed above)}
     p_soup(soup[h+1,w,d+i][pr[up]],to_other_dim,'1');            {Send a 1 iff delay register }
     p_soup(soup[h+1,w,d+i][up],to_nand,output_from_nand);        {[i] is a 1 and delay        }
     route_between(h+1,w,d+i,up,output_from_nand,left,soup);      {register [i+1] is a zero.   }
     p_soup(soup[h+1,w,d+i][left],to_nand,output_from_other_dim);
     p_soup(soup[h+1,w+1,d+i][pr[left]],to_other_dim,'1');
     p_soup(soup[h+1,w+1,d+i][left],to_nand,output_from_nand);
     route_between(h+1,w+1,d+i,left,output_from_nand,front,soup);
     route_between(h,w,d+i+1,left,output_from_memory, up,soup);
     p_soup(soup[h,w,d+i+1][up],to_nand,output_from_other_dim);
     p_soup(soup[h+1,w,d+i+1][pr[up]],to_other_dim,'1');
     p_soup(soup[h+1,w,d+i+1][up],to_nand,output_from_nand); {for route_between}
     route_between(h+1,w,d+i+1,up,output_from_nand,left,soup);
     p_soup(soup[h+1,w,d+i+1][left],to_memory,output_from_other_dim);
     route_between(h+1,w+1,d+i+1,left,output_from_memory,front,soup);
     p_soup(soup[h+1,w+1,d+i+1][front],to_nand,output_from_other_dim);
     p_soup(soup[h+1,w+1,d+i][front],to_nand,output_from_nand); {for route_between}
     route_between(h+1,w+1,d+i,front,output_from_nand,up,soup);
     p_soup(soup[h+1,w+1,d+i][up],to_memory,output_from_other_dim);
     IF i=(2*image_depth) THEN p_soup(soup[h+2,w+1,d+i][front],to_nand,'0'); {Place a zero at  }
     IF i<(2*image_depth) THEN                                          {the start of the chain.  }
      BEGIN
       p_soup(soup[h+2,w+1,d+i+1][pr[front]],to_other_dim,'1');   {Combine with the previous    }
       p_soup(soup[h+2,w+1,d+i+1][front],to_nand,output_from_nand); {result.                    }
       route_between(h+2,w+1,d+i,up,output_from_memory,front,soup);
       p_soup(soup[h+2,w+1,d+i][front],to_nand,output_from_nand);
       IF i=0 THEN
        BEGIN  pso_h:=h+2; pso_w:=w+1; pso_d:=d-1; {The location of the result.                 }
         p_soup(soup[h+2,w+1,d+i][front],to_memory,output_from_nand);
        END;
      END;
    END;
 END;

 BEGIN
  reset(user,'pic.dat');
  set_up_picture_data(image,m_x,m_y);    {Read in the image into boolean array.                 }
  delay_block((m_x+3)*2,2,m_y+4,1,2*m_x+5,3+6);  {Delay lines and register.                     }
  register_decode(2+2,2*m_x+5+3,3+6,m_x+1,pso_h,pso_w,pso_d); {Abut register decode.            }
  memory_block(2+2,2*m_x+5+3+4,3,image,m_x,m_y); {Abut ROM memory.                              }
  FOR y:=0 TO m_y-1 DO programmer_circuit(y=0,2+2+1+y,2*m_x+5+3+4+5,3-1,pso_h,pso_w,pso_d);
                                         {Number of programmers required = height of the image.}
  set_up_display_area(2+2+1-2,2+2+1+1+m_y,2*m_x+5+3+4+5+6,3-1-1,m_x); {Abut display area.       }
  inquire_size(max_x,max_y,max_z,soup);  {Find out how big this circuit is.                     }
  write_out(soup,max_x,max_y,max_z,0);
 END;
```

Simulator commands

This appendix shows the commands which could be issued to the Soup Simulator described in Chapter 5. These commands control simulation through features such as breakpoints, stepping through events, and detailed examination of the simulators event list. Macros of commands could be built using 'routines'.

Simulator Commands

Variables:	
DEFINE name [value]	Define a variable called 'name' with initial value 'value' or zero otherwise. Names must be in upper case, 10 characters or less.
UNDEFINE name	Delete most recently defined variable called 'name'.
SHOW-VARS	List all variables.
INCREMENT name [value]	Add 'value' or 1 to variable.
DECREMENT name [value]	Subtract 'value' or 1 from variable.
Predefined variables:	
UP	1 The indices of the face processors.
DOWN	4
LEFT	2
RIGHT	5
FRONT	3
BACK	6
HEIGHT	Hieght, width and depth of the simulation.
WIDTH	
DEPTH	
RUN-TIME	Elapsed real time in seconds.
SIM-TIME	The current value of the simulation clock.
Diagnostics:	
DISPLAY name1 [name2 ...]	Print the value of the variables named.
DISPLAY CELL h w d	Print information about the cell at (h,w,d).
DISPLAY EVENT-LIST n	Print the first n elements of the event list.
COUNT-LIST	Print the number of elements in the event list.
COUNT-LIST n	Print the number of elements in the event list scheduled for before time 'n'.
Directives:	
SIMULATE	Simulate until a breakpoint is reached or the event list is empty.
SIMULATE n	Simulate for 'n' events.
BREAKPOINT t	Set a breakpoint at time 't'.
PHOTO hl .. hh wl .. wh dl .. dh	Take a snapshot of the memory elements for later graphical display.
STOP	End the simulation.
Routines:	
ROUTINE name [p1 .. pn]	Parameters are the variables p1 to pn.
[statements]	
NEXT-IF var op var	Execute the next statement iff condition is
statement	true. Routines may be called recursively.
[statements]	
END	
op	= < > <= >=
STOP	Exit all routines, return to command line processor.
SHOW-RTN	List all routine names.
SHOW-RTN name	List the lines of the named routine.

Examples of program output

This appendix shows some examples of program output from within the simulation environment. The simulation environment is described in Chapter 5 and can be seen in Figure 5.1.

The first output is from the input processor. The input processor is a combination of the standard library of Appendix 2 and a user routine such as in Appendix 3. The output generated is a three-dimensional (complete) array of program stubs. An explanation of an individual stub is given in Appendix 1.

The second output is from the stub filter program (soup-face input processor in Figure 5.1) which has broken down the stubs according to the mapping in Appendix 1, and presented the expanded array in a suitable ordering so that it can be input at the six separate external faces of the cubic soup, to form the required circuit array within the Soup.

The third output is part of a 'photo' taken during a simulation run. The data forms a three-dimensional bit array showing which memory elements of the soup contained a 1 and which were 0 at the instant the photo was requested.

Examples of Program Output

1. Sample output stubs from the input processor (users routine):

```
...

NF?   NF?   NF?   NF?   NF?   NF?
NFF   FFM   NFF   NFF   NFF   NFF
F0N   FF?   NF?   NF?   NF?   NF?
NFF   FFM   NFF   NFF   NFF   NFF
NFF   FF1   FNN   NFF   NFF   NFF
NFF   FF?   NF1   FNN   NFF   NFF
NF1   FNN   NFF   FF?   NF?   NF?
NFF   NN?   NF1   FNN   NFF   NFF
FF?   NFM   NFF   NFF   NFF   NFF
NFF   FFM   NFF   NFF   NFF   NFF
F0N   FF?   NF?   NF?   NF?   NF?
NF1   NF1   NN?   NF?   NF?   NF?
NFF   NF?   NFM   NFF   NFF   NFF
NFM   NFF   FF?   NF?   NF?   NF?

...
```

Each line contains the program stubs for one cubic soup cell i.e. six face processors.

The output lines have a strict ordering corresponding with the input ordering of the stub filter program.

2. Sample output from the stub filter (soup-face input processor):

```
...

4 3 3 011010        ←
4 6 3 010111
1 2 3 011000
1 4 2 000010
2 2 3 011111
2 3 2 011100
2 3 4 001110
2 6 2 011100
3 6 8 001011
0                   ←
4 6 3 001011
2 2 6 010111
2 3 3 001001
2 3 4 001000
2 3 5 011000

...
```

Instruction to load the program 011010 at face 4 (down) position (3,3).

Every program line has a 1:1 correspondence with a stub from the input file.

Marks the end of the list of load information to be processed within the next load cycle period.

3. Sample output from the simulator for graphical display:

```
 1  8  1  11  1  8        ←
    0      0   23         ←
...
000000000000000000000000000000000000000000000000000000000000000000000000000
000000020000000006100002100000000202022200000000200400240000000200060200020
000000000000000000000000000000000000000100000100000000020000020000000000010
000010000000000000000000000000000000000000000000000000000000000000000000000
0000220000220020000000002020222000000000201400241000000200060200020000000

...
```

Height width and depth information.
Time in hours, mins and secs.
The array of memory elements, 3 bits of information are stored in each digit:

STRICT cell description

This appendix shows the STRICT description upon which Chapter 6 is based. The aims were to show the simplicity of the architecture and to obtain a crude estimate for the number of cells in a futuristic machine. It should be noted that at the time of this work the STRICT design system was not complete so was unusable for simulation or fabrication purposes.

In STRICT descriptions, 'blocks' are built through instancing a combination of other blocks and primitives, and specifying the interconnects between them. The positioning of these sub-blocks within a block can be influenced through a 'place' statement. Inputs and outputs to the block can be specified using a 'having' statement.

The blocks CTL (main control unit), PRG (six bit program register), NAN (ternary nand gate), MEM (ternary memory element), MUX (multiplexor), MEN (the additional control circuitry for the multiplexor enable) and PAL (the additional control circuitry for the program and lock functions) are CTL_V0, PRG_V0, NAN_V0, MEM_V0, MUX_V0, MEN_V0 and PAL_V0 in the silicon design for one face-cell, Plate 6.1. Plate 6.2 shows the same circuit but at the level of individual STRICT primitives. From the STRICT description these primitives can be seen to be a 1 bit inverter gate, a pass transistor, and two input 'or', 'nor' and 'and' gates.

```
!ARCHITECTURAL SOUP VERSION 1                                    APRIL 1988
! Description for ONE face processor, this design has not been tested and
! therefore will likely contain errors. The purpose of this STRICT
! description was to obtain area estimates for a silicon chip, and hence
! for a futuristic machine.
!
! Technology: 3 micron nmos, 2 metal layer routing
! The following were inherited from a test library
!   og2  - two input OR
!   ag2  - two input AND
!   nrg2 - two input NOR
!   ntg  - one input INVERTER
!   pass - a pass transistor
BUILD
{
INSTANCE                        ! One face processor.
   as1: prc
USING
   as1(program_inputs, from_o_dim, nand_in, mem_in, acting_as_prog, lock,
       empty_in, force_reprog, reprog_in, previous_mux_enabled,
       previous_empty_in, reprog_done)
MAKE
   program_output ::= as1.prog_outputs     ! Output of 6 bit program
   out_to_nand ::= as1.mux_out1            ! Outputs from multiplexor ...
   out_to_mem ::= as1.mux_out2
   out_to_other_dimension ::= as1.mux_out3
   prog_and_lock_info ::= JOIN (WIRE [2] | as1.out_act_as_prog, as1.out_lock)
   control_out_info ::= as1.ctl_prog_info  ! Control outputs ...
   empty_out ::= as1.empty_out
   force_reprog_out ::= as1.force_reprog_out
   reprog_out ::= as1.reprog_out
   mux_enabled ::= as1.mux_enabled
   to_previous_empty ::= as1.to_prev_empty
   reprog_done_out ::= as1.reprog_done_out
}
GIVEN
   ! One face processor
   BLOCK prc
      HAVING (prog_inputs @W: WIRE [6]  from_other_dim @N, nand_in @N,
              mem_in @N: WIRE [2] acting_as_programmer @N, lock @N: WIRE [1]
              empty_in @E, force_reprog @W, reprog_in @W,
              previous_mux_enabled @W, previous_empty_in @W,
              reprog_done @E: WIRE [1]): (prog_outputs @E: WIRE [6]
              mux_out1 @E, mux_out2 @E, mux_out3 @E: WIRE [2]
              out_act_as_prog @S, out_lock @S: WIRE [1]  empty_out @W,
              force_reprog_out @E, reprog_out @E, mux_enabled @E,
              to_prev_empty @E, reprog_done_out @W: WIRE [1])
      USE STRUCTURE
         {  INSTANCE
               control: ctl   prog_reg: prg   nand: nan   memory: mem
               multiplexor: mux   mux_enable: men   prog_and_lock: pal
               ! The components of one processor.
            PLACE
               (control; mux_enable) / prog_reg /
                              ((nand / (memory; prog_and_lock)); multiplexor)
            USING
               control(acting_as_programmer, lock, empty_in, reprog_done,
                               force_reprog, reprog_in, prog_reg.reprog_done,
                                     prog_reg.reprog_request, from_other_dim)
               prog_reg(prog_inputs, control.to_prog_reprog,
                                        control.prog_next, from_other_dim[1])
               nand(nand_in[0], nand_in[1], from_other_dim[0],from_other_dim[1])
               memory(mem_in[1], mem_in[0])
               multiplexor(from_other_dim[0], from_other_dim[1], nand.out_lo,
                                    nand.out_hi, memory.lo_out, memory.hi_out,
                                        prog_reg.prog, mux_enable.enable_mux)
               mux_enable(control.full_out, previous_mux_enabled,
                                      previous_empty_in, control.disable_mux)
               prog_and_lock(prog_reg.prog[0], prog_reg.prog[1],
                        mux_enable.enable_mux, nand_in[1], from_other_dim[1])
            MAKE
               prog_outputs ::= prog_reg.outs
               mux_out1 ::= multiplexor.out_to_nand
               mux_out2 ::= multiplexor.out_to_mem
               mux_out3 ::= multiplexor.out_to_other_dim
               out_act_as_prog ::= prog_and_lock.acting_as_a_programmer
               out_lock ::= prog_and_lock.lock
               empty_out ::= control.empty_out
               force_reprog_out ::= control.force_reprog_out
               reprog_out ::= control.reprog_out
               mux_enabled ::= mux_enable.enabled
               reprog_done_out ::= control.reprog_done_out
               to_prev_empty ::= control.empty_out }
   END
   ! Processors control unit.
   BLOCK ctl
      HAVING (acting_as_prog @N, lock @N, empty_in @E, reprog_done @E,
              force_reprog @W, reprog @W, from_prog_reprog @S,
              reprog_control @S: WIRE [1]  f_o_dim @S: WIRE [2]):
             (empty_out @W, reprog_done_out @W, to_prog_reprog @S,
              prog_next @E, force_reprog_out @E, reprog_out @E,
              disable_mux @S, full_out @S: WIRE [1])
```

```
  USE STRUCTURE
      {  INSTANCE  ! ffp's form a state machine. The other components are
            ! to generate the required control signals.
            lockd, empty, full: ffp   pass1, pass2: pass
            and1, and2, and3, and4, and5, and6: ag2
            or7, or8, or9, or10, or11, or12, or_f_o_dim, or_lock: og2
            not1, not2, not3: ntg
         PLACE
            (not1 /or8 / or9); (not2 / pass1 / pass2);
            (not3 / and2 / and3); (or12 / or11 / or10);
            (and1 / or_lock); (lockd; empty; full);
            (and6 / or7 / and4); (and5 / or_f_o_dim)
         USING
            or_f_o_dim(JOIN (WIRE [2] | f_o_dim[1],f_o_dim[0]))
            or_lock(JOIN (WIRE [2] | lock,acting_as_prog))
            not1(or8)   not2(lockd.q)   not3(acting_as_prog)
            and1(JOIN (WIRE [2] | acting_as_prog,reprog_control))
            and2(JOIN (WIRE [2] | not3.out,reprog_control))
            and3(JOIN (WIRE [2] | not3.out,reprog_done))
            and4(JOIN (WIRE [2] | full.q,empty_in))
            and5(JOIN (WIRE [2] | reprog_control,full.q))
            and6(JOIN (WIRE [2] | and1.out,or_f_o_dim.out))
            or7(JOIN (WIRE [2] | and6.out,and5.out))
            or8(JOIN (WIRE [2] | not2.out,force_reprog))
            or9(JOIN (WIRE [2] | reprog,force_reprog))
            or10(JOIN (WIRE [2] | empty.q,lockd.q))
            or11(JOIN (WIRE [2] | pass1.out,lockd.q))
            or12(JOIN (WIRE [2] | and3.out,and2.out))
            pass1(from_prog_reprog, or8.out)
            pass2(from_prog_reprog, or8.out)
            empty(or11.out, or12.out)   full(pass2.out, or10.out)
            lockd(or_lock.out, reprog_control)
         MAKE
            empty_out ::= empty.q
            reprog_done_out ::= from_prog_reprog
            to_prog_reprog ::= or9.out
            prog_next ::= acting_as_prog
            force_reprog_out ::= or7.out
            reprog_out ::= and4.out
            disable_mux ::= or7.out
            full_out ::= full.q }
END
!The processors 6 bit program register
BLOCK prg
   HAVING (ins @W: WIRE [6]  reprog @N,prog_next @N,
            f_o_dim_hi @S: WIRE [1]): (outs @W,prog @S: WIRE [6]
            reprog_done @N, reprog_request @N: WIRE [1])
   USE STRUCTURE
       {  INSTANCE
             sr1, sr2, sr3, sr4, sr5, sr6: ffp   reprog_control: ag2
             not1, not2, not3, not4, not5, not6: ntg
             in_p1, in_p2, in_p3, in_p4, in_p5, in_p6: pass
             out_p1, out_p2, out_p3, out_p4, out_p5, out_p6: sel
          PLACE
             reprog_control; (((in_p1 / not1); sr1) / (out_p1)); (((in_p2 /
             not2); sr2) / (out_p2)); (((in_p3 / not3); sr3) /
             (out_p3)); (((in_p4 / not4); sr4) / (out_p4)); (((in_p5 /
             not5); sr5) / (out_p5)); (((in_p6 / not6); sr6) / (out_p6))
          USING
             in_p1(ins[0], reprog)   in_p2(ins[1], reprog)
             in_p3(ins[2], reprog)   in_p4(ins[3], reprog)
             in_p5(ins[4], reprog)   in_p6(ins[5], reprog)
             not1(in_p1.out)   not2(in_p2.out)   not3(in_p3.out)
             not4(in_p4.out)   not5(in_p5.out)   not6(in_p6.out)
             sr1(in_p1.out, not1.out)   sr2(in_p2.out, not2.out)
             sr3(in_p3.out, not3.out)   sr4(in_p4.out, not4.out)
             sr5(in_p5.out, not5.out)   sr6(in_p6.out, not6.out)
             out_p1(sr1.q, sr2.q, prog_next)
             out_p2(sr2.q, sr3.q, out_p1.select_out)
             out_p3(sr3.q, sr4.q, out_p2.select_out)
             out_p4(sr4.q, sr5.q, out_p3.select_out)
             out_p5(sr5.q, sr6.q, out_p4.select_out)
             out_p6(sr6.q, f_o_dim_hi, out_p5.select_out)
             reprog_control(JOIN (WIRE [2] | sr1.q,sr2.q))
          MAKE
             outs ::= JOIN (WIRE [6] | out_p1.out,out_p2.out,out_p3.out,
                                       out_p4.out,out_p5.out,out_p6.out)
             prog ::= JOIN (WIRE [6] | sr1.q,sr2.q,sr3.q,
                                                 sr4.q,sr5.q,sr6.q)
             reprog_done ::= reprog
             reprog_request ::= reprog_control.out }
 END
 !Processors asynchronous ternary nand function
 BLOCK nan
    HAVING (i1h @W, i1l @W, i2h @W, i2l @W: WIRE [1]):
            (out_hi @E, out_lo @E: WIRE [1])
    USE STRUCTURE
        {  INSTANCE
              and1, and2, and3, and4, and5, and6, and7, and8: ag2
              or1, or2: og2   not1, not2, not3, not4: ntg
           PLACE
              (not1 / not2 / not3 / not4); (and1 / and2 / and3 / and4);
          (and5 / and6 / and7 / and8); (or1 / or2)
       USING
          not1(i1l)   not2(i1h)   not3(i2l)   not4(i2h)
          and1(JOIN (WIRE [2] | i1h,not1.out))
          and2(JOIN (WIRE [2] | not2.out,i1l))
          and3(JOIN (WIRE [2] | i2h,not3.out))
          and4(JOIN (WIRE [2] | i2l,not4.out))
          and5(JOIN (WIRE [2] | and1.out,and3.out))
          and6(JOIN (WIRE [2] | and2.out,and3.out))
          and7(JOIN (WIRE [2] | and1.out,and4.out))
          and8(JOIN (WIRE [2] | and2.out,and4.out))
          or1(JOIN (WIRE [2] | and6.out,and7.out))
          or2(JOIN (WIRE [2] | and8.out,or1.out))
       MAKE
          out_hi ::= or2.out
          out_lo ::= and5.out }
END
!processors 1 bit ternary memory
BLOCK mem
   HAVING (hi_in @W, lo_in @W: WIRE [1]): (hi_out @E, lo_out @E: WIRE [1])
   USE STRUCTURE
      {  INSTANCE hiff, loff: ffp(collapse)
         PLACE    hiff / loff
         USING    hiff(hi_in, lo_in)   loff(lo_in, hi_in)
         MAKE     hi_out ::= hiff.q
                  lo_out ::= loff.q }
END
! The multiplexor. Routes from the 'logic' parts of this cell to the
! inputs of the following cell and the cell in other dimension.
BLOCK mux
   HAVING
        (f_o_dim_lo @W, f_o_dim_hi @W, nand_lo @W, nand_hi @W, mem_lo @W,
         mem_hi @W: WIRE [1] prog @N: WIRE [6] enable @N: WIRE [1]):
        (out_to_nand @E, out_to_mem @E, out_to_other_dim @E: WIRE [2])
   USE STRUCTURE
      {  INSTANCE
            out1, out2, out3, out4, out5, out6: ag2
            smem1, smem2, snand1, snand2, snand3, snand4, snand5, snand6,
                                       stod1, stod2, stod3, stod4: sel
            todim_nand_a, todim_and: ag2   todim_nand_n: ntg
         PLACE
            (smem2 / smem1 / snand1); (snand2 / snand3 / snand4);
            (snand5 / snand6 / stod1); (stod2 / stod3 / stod4);
            (todim_nand_a / todim_nand_n / out1 / out2 / out3);
            (out4 / todim_and / out5 / out6)
         USING
            smem1(nand_lo, f_o_dim_lo, prog[1])
            smem2(nand_hi, f_o_dim_hi, prog[1])
            snand1(mem_hi, nand_hi, prog[2])
            snand2(mem_lo, nand_lo, prog[2])
            snand3(f_o_dim_hi, 0, prog[2])  snand4(f_o_dim_hi, 1, prog[2])
            snand5(snand1.out, snand3.out, prog[3])
            snand6(snand2.out, snand4.out, prog[3])
            stod1(mem_hi, 0, prog[4])   stod2(mem_lo, 1, prog[4])
            stod3(stod1.out, f_o_dim_hi, prog[5])
            stod4(stod2.out, f_o_dim_lo, prog[5])
            todim_nand_a(JOIN (WIRE [2] | prog[4],prog[5]))
            todim_nand_n(todim_nand_a.out)
            out1(JOIN (WIRE [2] | enable,snand5.out))
            out2(JOIN (WIRE [2] | enable,snand6.out))
            out3(JOIN (WIRE [2] | enable,smem1.out))
            out4(JOIN (WIRE [2] | enable,smem2.out))
            todim_and(JOIN (WIRE [2] | todim_nand_n.out,enable))
            out5(JOIN (WIRE [2] | todim_and.out,stod3.out))
            out6(JOIN (WIRE [2] | todim_and.out,stod4.out))
         MAKE
            out_to_nand ::= JOIN (WIRE [2] | out1.out,out2.out)
            out_to_mem ::= JOIN (WIRE [2] | out3.out,out4.out)
            out_to_other_dim ::= JOIN (WIRE [2] | out5.out,out6.out) }
END
!Multiplexor enable, dependant on this cells state, and the previous
!cells state.
BLOCK men
   HAVING (control_full @N, previous_mux_enabled @W, previous_empty @W,
          locked_but_not_programmer @N: WIRE [1]):
         (enable_mux @S, enabled @E: WIRE [1])
   USE STRUCTURE
      {  INSTANCE
            an1, an2: ag2   nt1, nt2: ntg   nr1: nrg2   sr1: ffp
         PLACE
            (an2 / an1 / nt1); sr1; (nr1 / nt2)
         USING
            an1(JOIN (WIRE [2] | previous_mux_enabled,previous_empty))
            an2(JOIN (WIRE [2] | control_full,an1.out))
            nt1(control_full)   sr1(an2.out, nt1.out)   nt2(nr1.out)
            nr1(JOIN (WIRE [2] | sr1.q,locked_but_not_programmer))
         MAKE
            enabled ::= sr1.q
            enable_mux ::= nt2.out }
END
!programmer and lock circuitry, acts on the cell in next dimension.
BLOCK pal
```

```
    HAVING (p1 @N, p2 @N, enabled @N, hi_fo_mem @W, p_nand_out_hi
          @W: WIRE [1]):(acting_as_a_programmer @S, lock @S: WIRE [1])
    USE STRUCTURE
       {  INSTANCE
             and1, and2, and3, and4: ag2   not2: ntg
          PLACE
             not2 / (and1; and2) / (and3; and4)
          USING
             not2(p2)   and1(JOIN (WIRE [2] | p1,not2.out))
             and2(JOIN (WIRE [2] | and1.out,enabled))
             and3(JOIN (WIRE [2] | and2.out,p_nand_out_hi))
             and4(JOIN (WIRE [2] | and2.out,hi_fo_mem))
          MAKE
             acting_as_a_programmer ::= and4.out
             lock ::= and3.out }
END
!sr flip-flop
BLOCK ffp
   HAVING (s @W, r @W: WIRE [1]):(q @E, q_ @E: WIRE [1])
   USE STRUCTURE
      {  INSTANCE nor1, nor2: nrg2
         PLACE    nor1 / nor2
         USING    nor1(JOIN (WIRE [2] | s,nor2.out))
                  nor2(JOIN (WIRE [2] | r,nor1.out))
         MAKE     q  ::= nor2.out
                  q_ ::= nor1.out }
END
!Selector, to choose between inputs a and b
BLOCK sel
   HAVING
      (a @W, b @W, select @N: WIRE [1]):(out @E, select_out @S: WIRE [1])
   USE STRUCTURE
      {  INSTANCE p1, p2: pass   not1: ntg   or1: og2
         PLACE    (not1 / p1); (p2 / or1)
         USING    not1(select)   p1(a, select)    p2(b, not1.out)
                  or1(JOIN (WIRE [2] | p1.out,p2.out))
         MAKE     out ::= or1.out
                  select_out ::= select }
END
```