Performance Evaluation and Router Design for Backtrack-to-the-Origin-and-Retry Routing in Hypercycle-Based Interconnection Networks

by

Don Radvan
B.Eng., University of Victoria, 1988

A Thesis Submitted in Partial Fulfillment of the
Requirements for the degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming to the
required standard

[redacted]

Dr. Nikitas J. Dimopoulos, Supervisor (Department of Electrical and Computer Engineering)

[redacted]

Dr. Kin F. Li, Departmental Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Gholamali C. Shoja, Outside Member (Department of Computer Science)

[redacted]

Dr. R.D. Rasmussen, External Examiner (Jet Propulsion Laboratory)

University of Victoria

Supervisor: Dr. Nikitas J. Dimpoulos

ABSTRACT

Hypercycles are a class of multidimensional graphs which are generalizations of the binary n-cube. These graphs are obtained by allowing each dimension to incorporate two or more vertices and a cyclic interconnection strategy. Hypercycles offer simple routing and the ability, given a fixed degree, to choose among a number of alternative size graphs. These graphs can be used in the design of interconnection networks for distributed computing systems tailored specifically to the topology of a particular application. Hypercycles use a routing policy, whereupon paths that block at intermediate nodes are abandoned and a new attempt is made using an alternate path. Intermediate nodes are chosen at random at each point from among the ones that form the shortest paths from a source to a destination.

This thesis presents a computer-based simulator for simulating all such Hypercycle networks. The results of this simulator present a favourable comparison of the above-mentioned routing policy with that of e-cube routing in terms of overall message delay. Furthermore, the implementation of a Hypercycle router is discussed in detail and a partial design suitable for VLSI implementation is presented along with results from hardware simulations performed on this design. This is done to show that the space-time complexity of the system can be reduced to the point where realization in current technology is feasable.

Examiners:

[redacted]

Dr. Nikitas J. Dimopoulos, Supervisor (Department of Electrical and Computer Engineering)

[redacted]

Dr. Kin F. Li, Departmental Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Gholamali C. Shoja, Outside Member (Department of Computer Science)

[redacted]

Dr. R.D. Rasmussen, External Examiner (Jet Propulsion Laboratory)

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

In the past 30 years the field of computer system design has seen an explosion of technological development. Improvements in computational speed, information storage capacity, system density, information transfer rates and other measurable performance parameters have been about an order of magnitude every five years. Until recently, the most economical route to improve system throughput was in changing the underlying technology in which a system was built. As an example, systems moved from vacuum tube-based to transistor-based to integrated circuit-based technologies primarily to improve computational performance and density. While implementation had changed, the overall architecture was comparatively stable. Most of these systems still embodied the Von Neumann approach to computational engines; that is, one instruction operates on one data item at any moment in time. Such devices are simple to understand and simple to construct. Furthermore, their performance is somewhat predictable. These qualities helped make the Von Neumann architecture the one of choice for most designers.

Early on, however, it was noted that architectures which departed from the basic Von Neumann approach could produce machines with improved performance parameters, given identical implementation technology. Pipelining and vector processing produced many successful products and indeed are used in many of the commercial systems today. Examples of these improvements include the IBM 3090 Vector Facility[25], the Motorola MC68020 microprocessor's instruction pipeline[22] and the Intel i860 microprocessor's instruction pipeline and vector floating point units[24]. Other departures from the classical Von Neumann architecture are still emerging into mainstream design. Harvard architectures, where code and data occupy separate physical and logical memory spaces, are only recently gaining widespread acceptance as seen in the Motorola 88000 reduced-instruction-set microprocessor[26] and Advanced Micro Devices AMD29000 embedded processor[23]. Several designs[22][26] try such changes as multiple functional units or replicated functional units in an effort to improve performance. While the results are positive, difficulty of design and complex behavior patterns have relegated most of these approaches to low volume mainframes and supercomputers.

More recently, through advances in VLSI, replicating entire processors (multiprocessor systems) and even entire systems (multicomputer systems) have been proposed and implemented[10, 12, 14, 17]. These latter systems, known as parallel or concurrent

computers, have shown great promise in solving computational problems which are inherently non-sequential. Problems such as matrix decomposition, sorting, image processing and atmospheric analysis can be solved advantageously using non-sequential systems. Furthermore, upgrading and expanding such a machine is less expensive and much more simplified. Unlike the enhancing of single processor systems, where the central processor is replaced by newer technology, concurrent systems can be designed to allow more, perhaps improved, processing nodes to be attached to the existing system while retaining the current nodes. Controlling system expansion becomes a separate issue from implementation technology, since the systems need only expand to their desired limits rather than the limits imposed by the updated technology, as in uni-processor systems.

Another benefit of concurrent systems is the possibility of improved fault tolerance[28] over sequential systems. Much work has been done in this area [27] and it continues to be a fruitful area of research. Uni-processor systems must, by definition, classify the processor as a critical resource since its failure would result in the failure of the entire computing system. Concurrent systems, however, need not classify the processing nodes as critical resources since the failure of any single node may not result in the complete failure of the system. Not all concurrent systems are fault tolerant. However, since

many such systems do replicate much if not all of their structure, the possibility for greater fault tolerance exists.

Non-sequential machines can be classified into several groups. Multiprocessors[31] are machines which replicate the processing elements (PE's) of a system. The PE's then share common memory subsystems via an interconnection network. Examples of such machines include the Ardent Titan[29]. Multicomputers[31] consist of replicated processing elements each with their own memory subsystem. The module which contains both the PE and its local memory is called a processing node. Each node is a self-contained computer in its own right and may function autonomously from any other node or, together with several other nodes or all other nodes, as a single entity. Examples of such machines include the Intel Personal Super Computer (iPSC) [38], Cosmic Cube[14], MAX[16] and various transputer-based products [32].

When working together on a single problem, it is often necessary for the nodes to exchange information. Message passing computers such as the Hypercube[10,15], Cosmic Cube and MAX interact via messages exchanged over communication channels and such are the systems which will be studied here.

There are many ways of interconnecting the computational nodes in a message passing machine. Bus-based interconnection systems have the nodes connected to a single, high-speed, backplane bus.

While such a system is simple to construct (it can use existing, bus-based, uni-processor components), its expansion capabilities are limited by the bandwidth of the bus. As more processing nodes are added to the system, the amount of communication between processors increases until the capacity of the bus is reached. Furthermore, the bus itself must be classified as a critical component since its failure would mean that none of the processing nodes could communicate with each other. Alternatively, fully-connected architectures require each node to support a connection to every other node. This would allow for a high data transfer rate since each communication channel would carry only messages for the two nodes to which it is connected. Expansion of such a system is limited by the number of channels each node supports.

It would be helpful if, should a single communication channel fail, alternate routes for messages which would otherwise use this channel could be found which are of similar length. Areas of interest in fault-tolerant, concurrent computer research include measuring system impact of processing node and communcation channel failure, fault detection and resolution, methods of "graceful" system degredation, and graph structures and routing algorithms which exhibit various degrees of resiliency in the presence of faults.

1.1 Designing Interconnection Networks

It is convenient to think of an interconnection network as a directed graph (digraph) where the processing nodes correspond to the vertices of the graph and the communication channels correspond to the edges. Given this analogy, one can use graph theoretic techniques to describe the properties and behavior of the network which the digraph represents.

A graph is called regular when the degree of each node is the same and when each node's view of the remaining subgraph is identical. Bus-based, binary n-dimensional cubes and fully-connected topologies are examples of regular graphs.

Architectures have employed the use of sparse graph structures, partially to gain higher communication bandwidth and fault tolerance while minimizing complexity. The Hypercube, Cosmic Cube, and the Connection Machine[16] have adopted regular interconnection patterns corresponding to graphs of binary n-dimensional cubes.

The binary n-dimensional cube (n-cube) is a popular interconnection topology. Many problems map directly or nearly directly to it and its complexity grows as $\mathbf{O}(n \log n)$. A number of successful commercial concurrent systems such as the iPSC[38] and the Connection Machine[16] have adopted this topology.

A significant amount of current research is directed at attempting extensions and generalizations of the basic properties of the n-cube.

Broder et. al. [4] have proposed the construction of product graphs[13] from small "basic" graphs. Their prime objective is to synthesize fault-tolerant networks with a given degree of coverage. In these multidimensional graphs, they define a single route from a source node to a destination node as the product of routes in each of the intervening dimensions. Routing is exhausted in each dimension before another dimension is considered. Bhunyan and Agrawal [3] have introduced the generalized hypercubes (GHC) which are also graph products of fully connected "basic" graphs. A mixed radix numbering system [2] is used to express the properties of these graphs and their routing. Wittie [17] gives a good overview and comparison of several interconnection networks including the spanning bus and dual bus hypercubes. These are essentially binary n-cubes with broadcast buses connecting the processors in each dimension.

There exist many advantages to having a regularly structured interconnection network, and these have been proven numerous times in their being incorporated in many recent designs such as the MARK II [15] and MARK III [10] Hypercube of Caltech/JPL, MAX [11,12], the Cosmic Cube [14] and the Connection Machine [16]. In these structures, simple routing [7] can be accomplished by locally computing each successive intermediate node —for a path that originates at a source node and terminates at a destination node— as a function of the current position and the desired destination. Many regular problems (such as the ones found in image processing, physics

etc.) have been mapped on regular structures such as binary n-cubes, and run on the corresponding machines exhibiting significant speedups. In contrast, embedded real-time applications [11, 12], tend to exhibit variable structures that do not necessarily map optimally on an n-cube. In addition, since the number of nodes (and hence the number of processing elements) of a binary n-cube is given as 2^n , it means that a particular configuration cannot be expanded except in predefined quantum steps. For example, if a given embedded application requires a system comprising 9 nodes, the next larger n-cube with 16 nodes must be chosen. This constitutes a significant increase in cost, especially in power-mass limited environments.

Hypercycles[8, 21] are product graphs of "basic" graphs that allow, as compared to the Generalized Hypercubes (GHC) [3], a richer set of component "basic" graphs ranging in complexity from simple rings to the fully connected ones used in the GHC. Also, contrary to Broder et al [4], the component graphs are defined and analytical expressions for routing are provided, the aim being twofold:

(a) To provide a general class of computer interconnection networks that match the node and communication density requirements of a given embedded system more closely than that which is possible using binary n-dimensional cubes.

(b) To increase the throughput of a given network by providing routing expressions that can be computed analytically (and hence

are candidates for VLSI implementation) and which provide a maximum number of optimal alternate paths from a source node to a destination node. The existence of alternate paths guarantees that a message will not be blocked waiting for its single route to be freed, but it would in turn search for the availability of alternate paths. This strategy also provides for limited fault protection, since a faulty path can be marked as permanently busy, and thus messages can be routed around it.

The Hypercycles, being regular graphs, retain the advantages of simple routing and topological regularity. Yet, since we are dealing with a class of graphs, rather than isolated graphs, we have the flexibility of adopting any particular graph (from the class) that closely matches the processing, communication and topology requirements of a given application.

This thesis supports the benefits of Hypercycles (as compared to the binary n-cube using e-cube routing) through the simulation of various Hypercycle networks. To accomplish this, a graphical-based simulator was built which could handle any Hypercycle network. Furthermore, in order to show that implentation of a router if feasabable, a discrete, medium-scale integration, Hypercycle hardware router has been designed and proves that quick routing decisions are attainable for Hypercycle based interconnection networks.

This work is divided into the following chapters. Chapter 2 introduces the mixed radix number system, basic graph terminology and specific notation, and the Hypercycles and their properties. Chapter 3 discusses network routing and presents results of work on evaluating the performance of several example Hypercycles using computer simulation. Chapter 4 discusses the partial design and testing of a Hypercycle router circuit as it would be implemented in discrete small- and medium-scale integration parts and in 1.5 micron-based VLSI. The thesis concludes and the results of the work are summarized in chapter 5 along with suggestions for future research.

Chapter 2. Introduction to Hypercycles

This chapter presents introductory and background material directly relevant to the simulation and design presented in the chapters which follow. Firstly, the mixed radix number system is introduced. This numbering system forms the basis of routing in Hypercycle graphs. This is followed by basic graph notation and terminology as it applies to computer interconnection network design. Finally, the class of Hypercycle graphs is formally introduced and their properties discussed.

2.1 Mixed Radix Number System

The mixed radix representation [2], is a positional number representation, and it is a generalization of the standard b-base representation, in that it allows each digit position to follow its own base independently of the other. This is convenient to use when describing multidimensional interconnection networks because it

permits the use of differing populations (bases) for each dimension (digit).

Given any decimal number M, it is possible to factor M into r factors such that

$$M = m_1 \times m_2 \times \cdots \times m_i \times \cdots \times m_r$$

Now, given any other number X satisfying $0 \leq X \leq M - 1$, one can represente X as an r-tuple $(x_1 x_2 \ldots x_r)$ where $0 \leq x_i \leq (m_i - 1)$; $i = 1,2,\ldots,r$. The x_i 's are digits in the r-tuple and we can assume (without loss of generality) that x_1 is the most significant digit and x_r is the least significant digit. Each digit x_i is associated with a weight w_i such that

$$X = \sum_{i=1}^{r} x_i w_i$$

where w_i is defined as

$$w_i = \frac{M}{m_1 m_2 \cdots m_i} \qquad i = 1,2,\ldots,r \ .$$

A number X[1] is then denoted as a sequence of digits in the following way:

$$(X)_{m_1 m_2 \ldots m_r} = x_1 x_2 \ldots x_r \mid_{m_1 m_2 \ldots m_r} .$$

We have used the subscripts to explicitly denote the radices involved.

---

[1] X will be given in base 10.

Example 1:

Let

$$M = 10 = 2 \times 5.$$

Then

$$m_1 = 2, m_2 = 5$$

and

$$w_1 = \frac{M}{m_1} = \frac{10}{2} = 5$$

$$w_2 = \frac{M}{m_1 m_2} = \frac{10}{2 \times 5} = 1$$

Therefore, the following values of X are denoted as

$$X = 0, (0)_{2\,5} = 0\,0\,|_{2\,5}$$

$$X = 1, (1)_{2\,5} = 0\,1\,|_{2\,5}$$

$$X = 4, (2)_{2\,5} = 0\,4\,|_{2\,5}$$

$$X = 5, (5)_{2\,5} = 1\,0\,|_{2\,5}$$

$$X = 7, (7)_{2\,5} = 1\,2\,|_{2\,5}$$

$$X = 9, (9)_{2\,5} = 1\,4\,|_{2\,5} .$$

It can be proven[2] that the maximum allowable value of X is

$$X = M - 1$$

and, moreover,

$$(M - 1)_{m_1\, m_2 \ldots m_r} = (m_1 - 1)\,(m_2 - 1) \ldots (m_r - 1)\,|_{m_1\, m_2 \ldots m_r} .$$

2.2 Graph Notation

Presented here is a general graph notation used in interconnection network theory as well as notation which is specific to Hypercycles. Such notation is often used to compare performance parameters of various interconnection networks. Terminology such as degree, diameter, walk, distance, average distance and topological regularity are defined here. These are defined in general as well as they specifically apply in Hypercycles.

Let $\mathcal{G}$, an undirected graph, be defined as the following tuple:

$$\mathcal{G} = (\mathcal{N}, \mathcal{E})$$

where $\mathcal{N}$ is the set of nodes defined as

$$\mathcal{N} = \{ a_i \quad ; i = 1,2,...M \}$$

and $\mathcal{E}$ the set of edges defined as

$$\mathcal{E} = \left\{ \varepsilon_{ij_i} = (a_i , b_{j_i}) ; j_i = 1, 2, \ldots, d_i \; ; i = 1, 2, \ldots, M \right\}$$

with $a_i , b_{j_i} \in \mathcal{N}$ and d_i the degree of node a_i. The degree of a node, d_i, is the number of edges incident on that node. A graph is regular, if all nodes have the same degree. This can be defined further by saying that a graph is regular if, after the removal of any single node, the resulting subgraph is identical in form regardless of the node removed. The degree of a graph, $d(\mathcal{G})$, is the maximum of the node degrees. Note that, in a regular graph, all nodes have the same degree and, therefore, the graph also has the same degree as any single node. A walk in $\mathcal{G}$ [5] is a sequence of edges $\varepsilon_1 \, \varepsilon_2 \ldots \varepsilon_l$, such that if $\varepsilon_i = (a_i, a_{i+1})$ then $\varepsilon_{i+1} = (a_{i+1}, a_{i+2})$ and $\varepsilon_i \in \mathcal{E}$. The distance, $dis(\gamma, \delta)$, between nodes γ and δ is defined as the shortest walk between γ and δ if any, otherwise, $dis(\gamma, \delta) = \infty$. The diameter, k, of a graph is the maximum distance between any pair of nodes. The terms connectivity and interconnectivity refer to the connection density between nodes. That is, a dense graph is said to have a high level of connectivity. We define both the connectivity of a graph and the density of a graph as being equal to the following ratio

$$\frac{n_l}{M}$$

where n_l is the number of edges in the graph and M is the number of nodes in the graph. We note that for regular graphs, this ratio is also the degree of the graph.

2.3 Hypercycles

It is now possible to introduce the Hypercycle class of graphs. As explained in Chapter 1, Hypercycles[8,21] is a class of multidimensional graphs with a richer topology than can be found in binary n-dimensional cubes. Hypercycles can be fully expressed using the terminology defined in the previous chapters and, indeed, this is what is done here.

Let an *r*-dimensional Hypercycle be defined as the following regular undirected[2] graph:

$$G_m^{\rho} = \left\{ \mathcal{N}_m^{\rho} , \mathcal{E}_m^{\rho} \right\}$$

where $m = m_1,m_2,m_3,...,m_r$ is an *r*-digit mixed radix number indicating the nodal population in each dimension and $\rho = \rho_1,\rho_2,...,\rho_r$; $\rho_i \leq m_i / 2$ is also an *r*-digit mixed radix called the connectivity vector, determining the connectivity in each dimension which ranges from a simple cycle

[2]Note that one can use an undirected graph to model an interconnection network with no loss of generality since the communication channels which connect processing nodes are considered to be singular and bidirectional in nature. Once a link is occupied for communication, it is allocated to the transaction in both directions.

($\rho_i = 1$) to a fully connected cycle ($\rho_i = m_i/2$). Figures 2.1 and 2.2 give some examples of single dimensional Hypercycles. Then the set of nodes is given as

$$\mathfrak{N}_m^{\rho} = \{0, 1, 2, ..., M\text{-}1\}.$$

Now, given

$$\alpha , \beta \in \mathfrak{N}_m^{\rho}$$

where α and β are r-digit mixed radix numbers, then

$$(\alpha , \beta) \in \mathfrak{E}_m^{\rho}$$

if and only if there exists a dimension $j : 1 \leq j \leq r$ such that

$$\beta_j = (\alpha_j \pm \xi_j) \bmod m_j$$

where $1 \leq \xi_j \leq \rho_j$ and $\alpha_i = \beta_i$ for all $i = 1, 2, ..., j - 1, j + 1, ..., r$. That is, nodes α and β are connected if and only if their distance in a single dimension i is less than or equal to ρ_j while the distance along all the other dimensions is exactly 0.

As explained in section 2.2, the degree of a graph depends on the number of vertices in the graph and the density or richness of vertex interconnection. In Hypercycles, both density and population may vary from one dimension to another, within the same graph. Therefore,

the degree of a Hypercycle graph is dependant on the connectivity ρ and population m in each dimension.

Theorem 2.1: The degree, d, of a Hypercycle is given by

$$d = \sum_{i=1}^{r} f(m_i, \rho_i)$$

where

$$f(m_i, \rho_i) = \begin{cases} 2\rho_i & \text{if } 2\rho_i < m_i \\ m_i - 1 & \text{if } 2\rho_i = m_i \end{cases} .$$

Proof 2.1: Given two integers ξ, ψ where $|\xi|, |\psi| < m/2$ and $|\xi| \neq |\psi|$, then

$$(\alpha + \xi) \bmod m \neq (\alpha + \psi) \bmod m$$

and

$$(\alpha + \xi) \bmod m \neq (\alpha - \xi) \bmod m$$

for $0 < \xi < m/2$.

Therefore, for each $\xi : 0 < \xi \leq \rho_i$, the r-tuples $\alpha_1 \alpha_2 ... (\alpha_i \pm \xi) \bmod m_i ... \alpha_r$ define two distinct nodes in dimension i of a Hypercycle graph for a total of $2\rho_i$ connections.

If $\xi = m_i/2$, then

$$
\begin{aligned}
(\alpha + \xi) \bmod m &= (\alpha + m_i/2) \bmod m \\
&= (\alpha + m_i/2 + m_i/2 - m_i/2) \bmod m \\
&= (\alpha - m_i/2) \bmod m \\
&= (\alpha - \xi) \bmod m
\end{aligned}
$$

Thus the same node is defined and, therefore, the total number of nodes reachable (connections) becomes m_i - 1. Therefore, for dimensions where $2\rho_i < m_i$ there are $2\rho_i$ connections per node and for dimensions where $2\rho_i = m_i$, there are m_i - 1 connections per node. Thus, since all dimensions are independent of each other, e.g. share no common ports, the degree of any node is simply the sum of the degrees of each of the constituent dimensions. ■

The diameter of a Hypercycle graph is the longest shortest-possible-path walk between any two nodes. Again, this is dependent on both the number of nodes in the graph, the connectivity of the nodes, and the number of dimensions in the graph.

Theorem 2.2: Hypercycles have a diameter k [3] given as,

$$k = \sum_{i=1}^{r} \left\lceil \frac{\lfloor m_i / 2 \rfloor}{\rho_i} \right\rceil .$$

Proof 2.2: Given that there are r digits in a node address, clearly, the largest distance (longest walk of minimal length) would be

[3]The function $\lfloor \xi \rfloor$ denotes the largest integer smaller than or equal to ξ, while $\lceil \psi \rceil$ denotes the smallest integer larger than or equal to ψ.

such that all r digits between source and destination are different. Also, for each digit (since it is expressed in base m_i) the maximum distance[4] between any two points in dimension i is given as

$$d_{\max} = \lfloor m_i / 2 \rfloor$$

Thus, because each point is connected to points that are some distance $\xi : 1 \leq \xi \leq \rho_i$ away, any point can be reached in a maximum of

$$\lceil d_{\max} / \rho_i \rceil$$

step (where each step corresponds to a single channel). Therefore, for each cycle the maximum walk is

$$\left\lceil \frac{\lfloor m_i / 2 \rfloor}{\rho_i} \right\rceil$$

and this can be summed over all dimensions to obtain

$$k = \sum_{i=1}^{r} \left\lceil \frac{\lfloor m_i / 2 \rfloor}{\rho_i} \right\rceil .$$

■

Hypercycles are a broad class of graphs in that certain subclasses of Hypercycles include several popular graph structures. The 2-

[4]The distance between any two nodes is determined by the "greedy" strategy of equation 2.3.1.1

dimensional toroidal-mesh[18] with N processing elements in each dimension, for example, is a Hypercycle where

$$M = N \times N = N^2$$

and

$$\rho = 1\,1\,.$$

The binary n-dimensional cube can also be expressed as a subclass of Hypercycle graphs, where

$$M = 2 \times 2 \times \cdots \times 2 = 2^n$$

and

$$\rho = 1\,1\,1\,\ldots\,1.$$

Figure 2.1 shows a simple $\mathbf{G}_7^1$ single-dimensional Hypercycle graph. Contrasting this is Figure 2.2 which is a fully-connected Hypercycle $\mathbf{G}_7^3$ but still only containing a single dimension. Figure 2.3 shows an example of a Hypercycle $\mathbf{G}_{4\,4}^{1\,1}$ which is also a two-dimensional 16-node toroidal mesh. Figure 2.4 shows the graph of a binary 4-dimensional cube also called a Hypercycle $\mathbf{G}_{2222}^{1111}$.

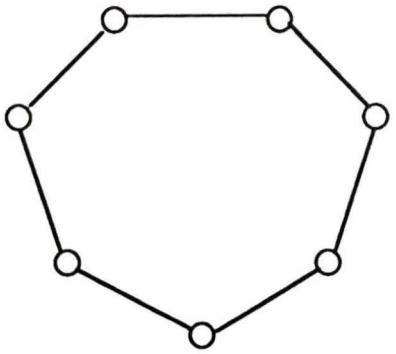


Figure 2.1 Hypercycle $\mathfrak{G}_7^1$

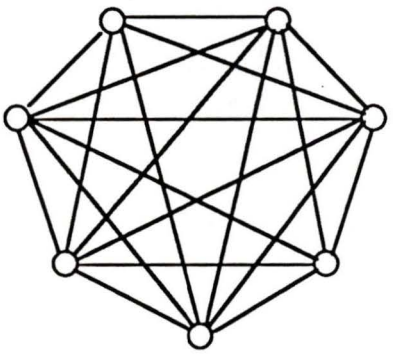


Figure 2.2 Hypercycle $\mathfrak{G}_7^3$

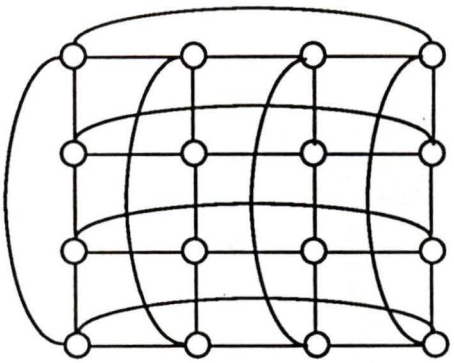


Figure 2.3 Hypercycle $G_{44}^{1\,1}$

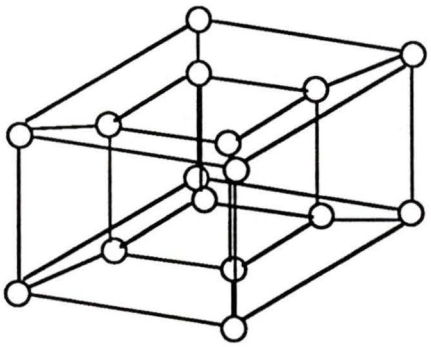


Figure 2.4 Hypercycle G_{2222}^{1111}

2.3.1 Routing

A key element in developing new interconnection networks is to provide a routing strategy that can supply (possibly several) shortest-length paths based on local information only (hence the desire for regular graph structures). Interconnection networks based on binary n-dimensional cubes usually route messages by comparing the

destination address of the message with the current node's address. A decision can then be made as to which of the node's neighbours the message should be forwarded. This can be done as follows. The digits in a node address are ordered from most-significant to least-significant. Starting at the most significant digit of the source nodes address, if this digit is different from the destination address, it is changed to match the destination addresses digit in this place. Note that since this is a binary cube all address digits are base 2. If this digit is not different the next most significant digit is used and so on. Once a digit is found to be different and changed, the resulting address will be the next one in the path of the message. Since this address differs from the original in one position only, it is implied from the definition of the Hypercube that the node with the newly formed address is connected to the node having the orginal address. Since such graphs are regular, each node would possess an identical algorithm. Hypercycles have routing properties that are similar to those of the binary n-dimensional cubes in that they use destination and current addresses to compute the next address in the walk. Given nodes

$$(\alpha)_{m_1\, m_2\, \ldots m_i\, \ldots m_r} = \alpha_1\, \alpha_2\, \ldots \alpha_i \ldots \alpha_r$$

and

$$(\alpha^*)_{m_1\, m_2\, \ldots m_i\, \ldots m_r} = \alpha_1\, \alpha_2\, \ldots \xi \ldots \alpha_r\,,$$

a walk, from node α to node α^*, can be constructed as follows:

$$\alpha_1\alpha_2...\alpha_i...\alpha_r$$

$$\alpha_1\alpha_2...\xi_1...\alpha_r$$

$$\alpha_1\alpha_2...\xi_2....\alpha_r$$

...

$$\alpha_1\alpha_2...\xi...\alpha_r\ .$$

according to the following[5]

$$\xi_{j_i+1} = \begin{cases} (\xi_{j_i} + \rho_i) \bmod m_i & \text{if } \left[(\xi - \xi_{j_i}) \bmod m_i \equiv |\xi_{j_i}, \xi|\right] > \rho_i \\ (\xi_{j_i} - \rho_i) \bmod m_i & \text{if } \left[(\xi_{j_i} - \xi) \bmod m_i \equiv |\xi_{j_i}, \xi|\right] > \rho_i \\ \xi & \text{if } |\xi_{j_i}, \xi| \le \rho_i \end{cases} \qquad (2.3.1.1)$$

$$\xi_0 = \alpha_i \qquad \xi_{l_{max}} = \xi$$

Equation 2.3.1.1 ensures that the shortest walk possible is taken since, when possible, the longest steps ($\xi_{j_i} \pm \rho_i$) are taken. This is known as the greedy strategy. We call the length l_{max} of such a walk, the distance along dimension i. While this distance is minimal, this walk is not the only minimal distance route available.

---

[5]Denote as $|\mu, \nu| = \min\{ (\mu - \nu) \bmod m, (\nu - \mu) \bmod m \}$, the distance between two integers modulo m.

2.3.2 Multidimensional Routing

Hypercycles are a multidimensional graph class and as such, must provide a routing scheme which can be used across all dimensions. Routing in more than one dimension is similar to routing in a single dimension. The major difference is that at each node in the path a decision is made as to which next eligible dimension will be chosen. This is shown as follows. Given an originating node

$$(\alpha)_{m_1\, m_2\, \ldots m_r} = \alpha_1\, \alpha_2 \ldots \alpha_r$$

and a destination node

$$(\beta)_{m_1\, m_2\, \ldots m_r} = \beta_1\, \beta_2 \ldots \beta_r$$

and if q_i denotes their distance along dimension i, their total distance, denoted as $\mathbf{dis}(\alpha,\beta)$ and defined as the sum of the individual distances[6] along all the dimensions, is given as

$$\mathbf{dis}(\alpha,\, \beta) = q = \sum_{i=1}^{r} q_i$$

For these nodes, there are a total of [21][7]

$$I = \begin{pmatrix} q \\ q_1, q_2, \ldots, q_r \end{pmatrix} = \frac{q!}{q_1!\, q_2! \cdots q_r!}$$

---

[6]This is defined by the "greedy" strategy of equation 2.3.1.1

[7]For the definition of a multinomial number, see [1] pp 32.

distinct walks of length q that connect them. These are constructed by sequentially modifying the source address, each time substituting a source digit by an intermediate walk digit, until the destination is reached. The following walk connects **source** to **destination.**

$$\textbf{source} = \alpha_1\ \alpha_2 \ldots \alpha_i \ldots \alpha_j \ldots \alpha_r\ ;$$

$$\alpha_1\ \alpha_2 \ldots \xi_1 \ldots \alpha_j \ldots \alpha_r\ ;$$

$$\alpha_1\ \alpha_2 \ldots \xi_1 \ldots \psi_1 \ldots \alpha_r\ ;$$

$$\alpha_1\ \alpha_2 \ldots \xi_2 \ldots \psi_1 \ldots \alpha_r\ ;$$

$$\alpha_1\ \alpha_2 \ldots \xi_2 \ldots \psi_2 \ldots \alpha_r\ ;$$

$$\ldots\ ;$$

$$\alpha_1\ \alpha_2 \ldots \xi_2 \ldots \beta_j \ldots \alpha_r\ ;$$

$$\ldots\ ;$$

$$\beta_1\ \beta_2 \ldots \beta_i \ldots \beta_j \ldots \beta_r\ = \textbf{destination}$$

Figure 2.5 gives an example of two distinct walks of equal length that connect a source to a destination for a Hypercycle. We define the average distance between any node and all other nodes in the network as the average length of all minimal distance routes between the node and all other nodes in the network.

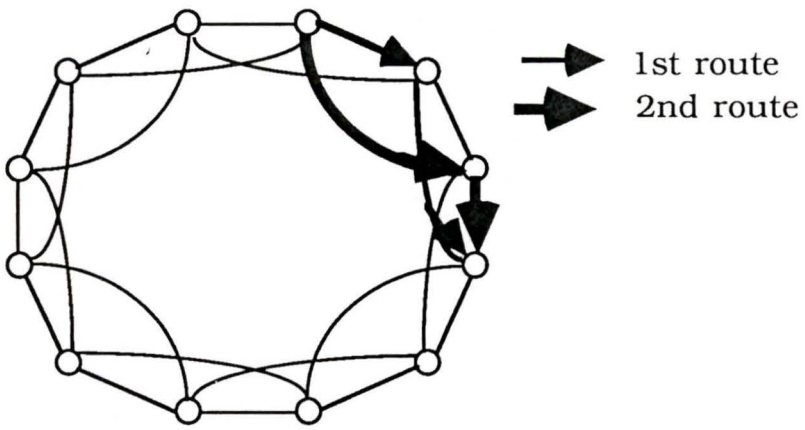


Figure 2.5 Two distinct walks of equal length from a single source to a single destination.

Theorem 2.3: The average distance[8], $\overline{d}$, between any node and all other nodes in an r-dimensional Hypercycle $\mathcal{G}_m^{\rho}$ can be calculated as

$$\overline{d} = \frac{\sum_{l=1}^{k} l\, n_l}{m_1 m_2 \cdots m_r - 1} .$$

Proof 2.3: Given an r-dimensional Hypercycle $\mathcal{G}$ and restricting ourselves in any single dimension i, we can calculate the number of nodes that are distance l from any source node

$$(\alpha)_{m_1 m_2 \ldots m_r} = \alpha_1\, \alpha_2 \ldots \alpha_r$$

as equal to the number of nodes ξ_i that satisfy the relation $(l - 1)\rho_i < |\alpha_{j_i}, \xi_j| \le l\rho_i$. This number is given as

$$n_l^i = \begin{cases} 2\rho_i & \text{if} \quad l\rho_i < \lceil m_i / 2 \rceil \\ (m_i - 1) - 2(l-1)\rho_i & \text{if} \quad \lceil m_i / 2 \rceil \le l\rho_i \le \dfrac{m_i - 1 + 2\rho_i}{2} \\ 0 & \text{otherwise} \end{cases}$$

Given a node

$$(\alpha)_{m_1 m_2 \ldots m_r} = \alpha_1\, \alpha_2 \ldots \alpha_r$$

[8]This is defined by the "greedy" strategy of equation 2.3.1.1

there are

$$n_l = \sum_{i=1}^{r} n_l^i$$

nodes at distance one, and in general

$$n_l = \sum_{\substack{l_1,l_2,\ldots,l_r \geq 0 \\ l_1+l_2+\ldots+l_r=l \;\; l_i \neq 0}} \prod_{i=1}^{r} n_{l_i}^i$$

nodes at distance l. Therefore, the average distance between any two nodes in a Hypercycle G_m^{ρ} can be calculated as

$$\overline{d} = \frac{\sum_{l=1}^{k} l\, n_l}{m_1 m_2 \cdots m_r - 1} .$$

■

Some typical distances and diameter of various Hypercycles are given in Table 2-1. Of particular interest here is the binary cube graphs. The first binary cube in the table is the G_{222222}^{111111}. If one was restricted to using the binary cube graph, the next larger size of binary cube is $G_{2222222}^{1111111}$, which contains twice as many nodes. However, with Hypercycles, there exist other graphs which have similar distance, degree and diameter properties to G_{222222}^{111111} with a wider choice of the number of nodes. As will be shown in later chapters, the degree and diameter of a graph influence the efficiency of the network. In

particular, one can find a degree six Hypercycle with 441 nodes, as shown, but the closest Hypercube, one with 512 nodes would need to be degree nine. Thus, a system built from degree six nodes can be expanded to at least 441 (since many other degree six Hypercycles exist) but a degree six binary cube is restricted to 64 nodes.

Table 2-1. Hypercycle distances and diameters

GRAPH	DEGREE	DIAMETER	NODES	AVERAGE DISTANCE PER NODE
$m = 62$ $\rho = 31$	6	2	12	1.586
$m = 53$ $\rho = 21$	6	2	15	1.683
$m = 522$ $\rho = 211$	6	3	20	1.994
$m = 333$ $\rho = 111$	6	3	27	2.157
$m = 3322$ $r = 1111$	6	4	36	2.469
$m = 32222$ $\rho = 11111$	6	5	48	2.781
$m = 222222$ $\rho = 111111$	6	6	64	3.095
$m = 2217$ $\rho = 112$	6	6	68	3.454
$m = 2237$ $\rho = 1111$	6	6	84	3.475
$m = 357$ $\rho = 111$	6	6	105	3.650
$m = 555$ $\rho = 111$	6	6	125	3.658
$m = 2222222$ $\rho = 1111111$	7	7	128	3.556
$m = 779$ $\rho = 111$	6	10	441	5.677

Chapter 3. Routing in Hypercycles

It was shown in chapter 2 how Hypercycles provide at least one minimal-distance path from a source node to a destination node. It now must be shown how one of these paths can be optimally chosen. This is the problem of routing in a network; that is, how to route messages such that:

a) The network is utilized efficiently.

b) Deadlocks do not occur or, if they do, they are resolved and the resolution does not incur a heavy penalty on message delay.

It must be noted that the routing strategy to be employed depends on the type of routing involved. Generally, there are two types of routing[38]; circuit switching and packet switching. In circuit switching, an entire source-to-destination path is reserved before any data is transferred. Once established, the path is released only after all the data has been transferred. In contrast, packet switching breaks the data into small, fixed length packets. These packets are then forwarded, one at a time, to the network. As packets are received at each node in the path, they are placed in a queue. When an

appropriate port becomes available, the packet at the head of the queue is then forwarded. Both methods have advantages and disadvantages. Circuit switching uses more of the network bandwidth as it attempts to establish a complete source-to-destination path. Packet switching only uses the bandwidth as it is needed. However, each node must spend time and memory queueing and dequeueing packets and if these processes are not to be handled by the CPU, then further intelligence is required in the routing circuits to handle this task. Furthermore, with packet switching, a network could be overwhelmed by a sudden surge of traffic, causing some nodes to lose packets due to insufficient queue space. This cannot happen in circuit switching since sufficient (non-preemptable) bandwidth is reserved in advance. In addition, circuit switching is more suitable for concurrent systems since no breaking and reassembling of messages is needed and thus messages can be delivered faster.

3.1 Deadlocks in Routing

Deadlock is a state of inaction which results when two or more uncompromising processes require resources held by one another. In interconnection networks this occurs when node-to-node communication channels are allocated such that the completion of a partial path requires a channel, or communication segment, which already has been allocated to a different partial path which, in turn, waits for a channel allocated to the first partial path. By partial path we mean the path constructed from channels allocated up until the

block was encountered. It is obvious that neither path can continue to full completion, and the only remedy is to break the already established and deadlocked partial paths and try again. Note that deadlocks occur only in circuit switched networks[1] since packet switched networks allocate only one channel at a time to messages.

There are four basic strategies which can be used when dealing with deadlocks[20]. The simplest strategy is to ignore deadlocks completely. The result of this strategy in a Hypercycle-based network would be total network gridlock. Since deadlocks are never resolved, the partially completed paths are never released and the network quickly grinds to a halt.

A more intelligent scheme is deadlock prevention. Certain routing algorithms (e.g. virtual channels, e-cube routing[7]) prevent deadlocks by ordering the resources (channels) to be allocated, making it impossible for deadlocks to occur. Thus a lower order resource cannot be allocated if a required higher order resource cannot be obtained. The process must first wait until the higher order resource has been made available, then obtain this resource before it can obtain the lower order resource. The disadvantage of this approach in an interconnection network is that it limits the number of paths connecting a source to a destination to exactly one, even though several alternate free paths of equal length may exist at a particular moment. Furthermore, should a segment fail, any processing node

[1]It is assumed here that the packet switching model will not deadlock from full message buffers since once a buffer overflows, further messages are discarded.

pairs which contain that failed segment in their path could not communicate with one another, making the system intolerant to faults.

Another approach is to permit deadlocks to occur but to resolve them when they do occur. This is called deadlock detection and resolution. While able to use the network more fully than deadlock prevention (since there is no restriction on path choice), additional complexity at each node is required to be able to detect deadlocks and resolve them. Moreover, the amount of network time lost while the deadlock is being detected and resolved can be costly in terms of network communication capacity, thus lowering network efficiency.

The fourth option is to avoid impending deadlocks. This could be accomplished by employing a two-phase locking method[20] of deadlock avoidance. The algorithm works as follows. A process requests each of the resources it needs one at a time, in any order, to accomplish the task. If any one of the resources cannot be granted (because some other process has ownership), all the previously granted resources are released and the process tries again. In Hypercycle routing, this could be realized by requiring a blocked path to backtrack to its origin and retry, thus releasing the previously held partial path and avoiding a potential deadlock situation.

3.2 Backtrack-to-the-Origin-and-Retry Routing

For Hypercycle-based interconnection networks, because of the existence of cycles in each dimension, the use of an e-cube type routing that prevents deadlocks is not possible without global knowledge of the state of the network. As an example, consider the following.

Example 3.1:

Figure 3.1 shows a 7 node $\mathbf{G}_7^1$ Hypercycle. There are two cases to be considered: unidirectional communication and bidirectional communication.

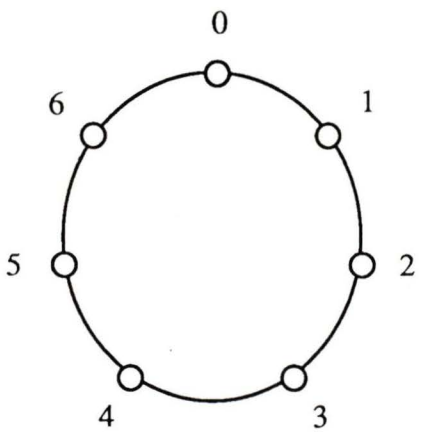


Figure 3.1 A $\mathbf{G}_7^1$ Hypercycle.

CASE 1: Links are requested in a single direction (e.g. clockwise)

Let M_{03} be a message originating at node 0 and bound for node 3. Now, assume that at the same time a message M_{21} originates at

node 2 bound for node 1. We will assume that links are requested clockwise about the graph. Message M_{03} captures the edges (0,1) and (1,2) while M_{21} captures the edges (2,3) and (3,4). Message M_{03} is now blocked at node 2 since it needs edge (2,3) to complete its path. Message M_{21} continues by capturing edges (4,5), (5,6) and (6,0) at which point it too is blocked since it requires edge (0,1) to complete its path. Therefore, the result is a deadlock, which has neither been prevented nor avoided.

CASE 2: Links can be requested in either direction.

Let M_{41} be a message from node 4 to node 1. At the same time, assume message M_{03}, a message originating at node 0 bound for node 3 arises. M_{41} will capture the edges (4,3) and (3,2) while M_{03} will capture the edges (0,1) and (1,2). Both messages are now waiting at node 2 for edges the other message holds. Again, a deadlock situation has arisen which has neither been prevented nor avoided. ■

Note that while there exist Hypercycles where e-cube routing is possible (such as binary n-cubes), the above example shows that this cannot be a general solution. What is proposed instead is a deadlock avoidance routing strategy. According to the backtrack-to-the-origin-and-retry (BTOR) routing [8,21], at each node all potential candidates for the next node in the partial path are identified. For all such identified nodes, the corresponding available ports that can be used in order to continue the path are also identified. One of these available

ports is then randomly selected, through which the subsequent link in the path is established. If no available ports are to be found at an intermediate node in the path, then a break notification is returned to the origin (through the already established partial path to the blocking node), the partial path is destroyed, and a new attempt for the creation of the required path is initiated. This routing strategy avoids deadlocks through the backtracking and releasing of segments, and also guarantees that the formed path will be of a minimum length, since each subsequent link is selected according to equation 2.3.1.1. In contrast, Hyperswitch routing [15] backtracks only to the previous node in the partial path. This has some disadvantages as compared to BTOR routing. Since the Hyperswitch backtracks only to the previous node, the remaining partial path may be blocking other messages attempting to route through.

3.3 Simulator

To study the behavior of backtrack-to-the-origin-and-retry routing in Hypercycle-based interconnection networks, a simulator has been constructed, using the Extend™ [2] graphical-based simulation environment for the Apple Macintosh computer, capable of simulating any Hypercycle based network. Both the BTOR and the e-cube routing strategies were implemented, although the e-cube routing can only be used for binary cube networks.

---

[2]Extend is a trademark of Imagine That!, Inc.

An example of the simulation environment is given in figures 3.2, 3.3 and 3.4. Extend provides a framework in which processes can be modelled and significant events trapped and dealt with. Each Hypercycle processing node can be modelled as a process which generates, routes and receives messages. For each node, a Poisson message generator is assumed which generates packets with a uniform distribution of destinations. Each packet carries the destination address which is used for routing along with the message creation time and a break indication. The simulation code is written in MoDL[33], which is a variation of the C language. MoDL is exclusive to the Extend simulator. The code for both backtrack-to-the-origin-and-retry routing and e-cube routing is given in Appendix A. Only minor variations in process code exist between different Hypercycle topologies.

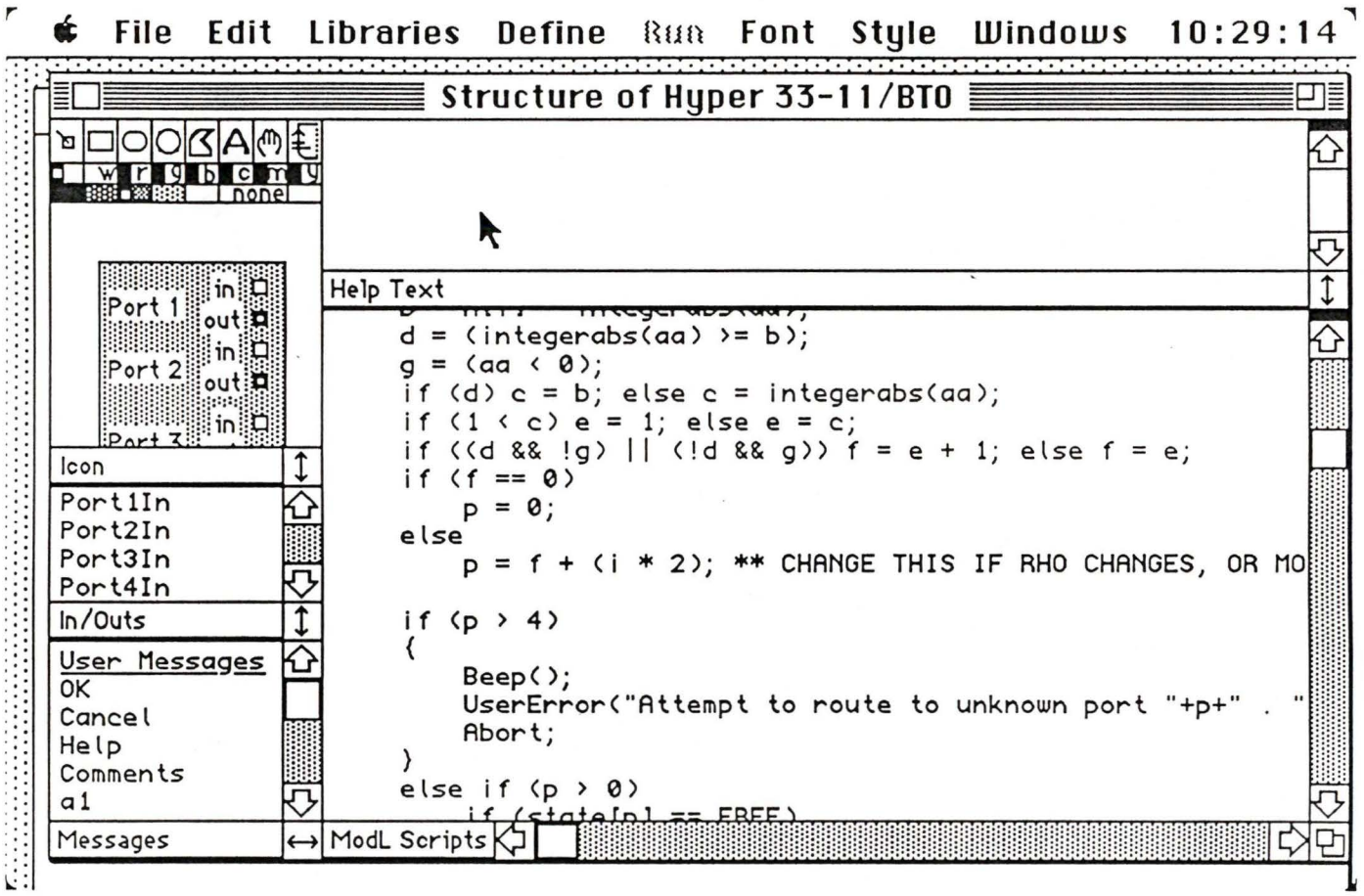


Figure 3.2 Extend Simulation Environment

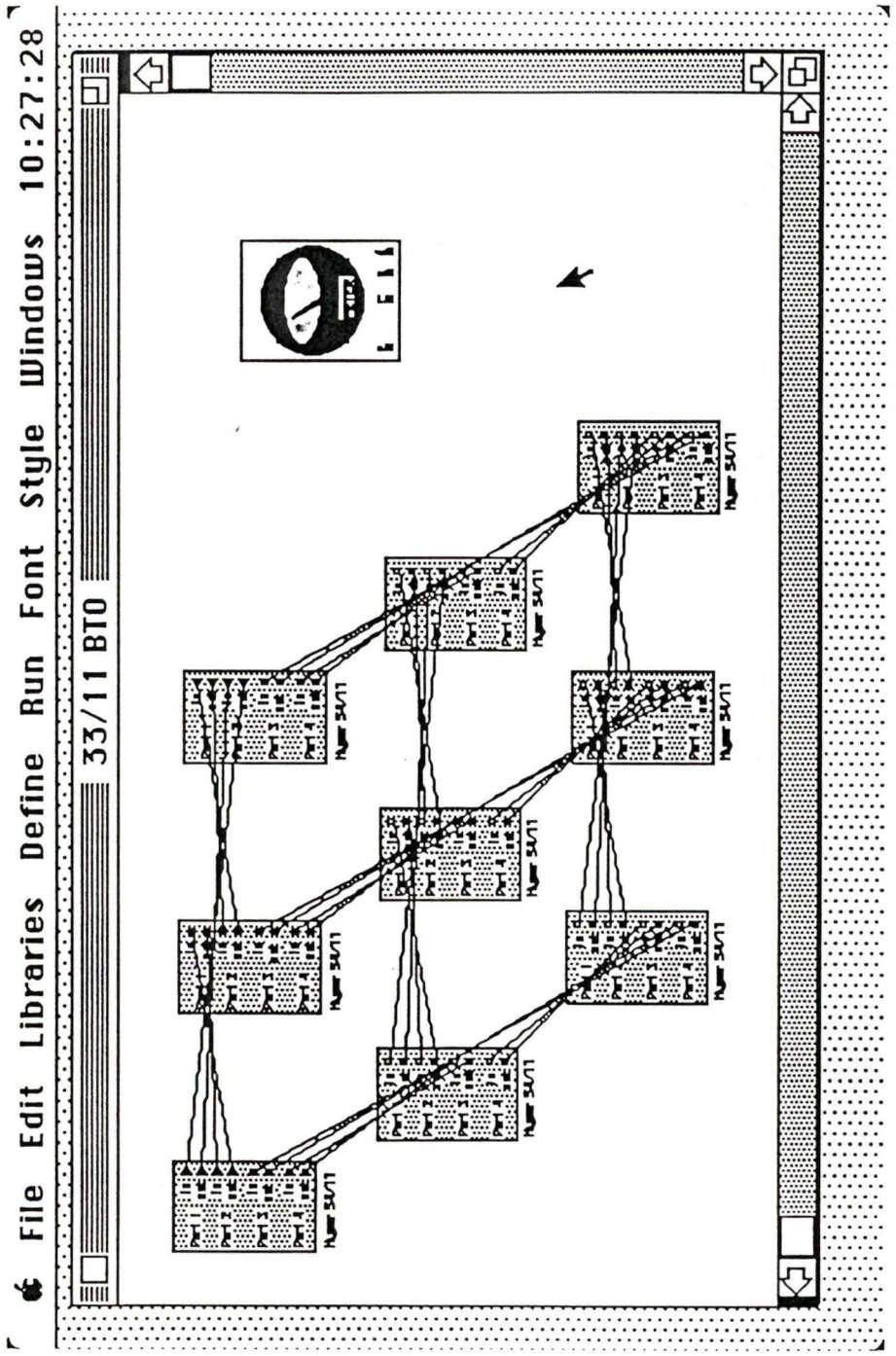


Figure 3.3 Graphical Representation of Hypercycle Network.

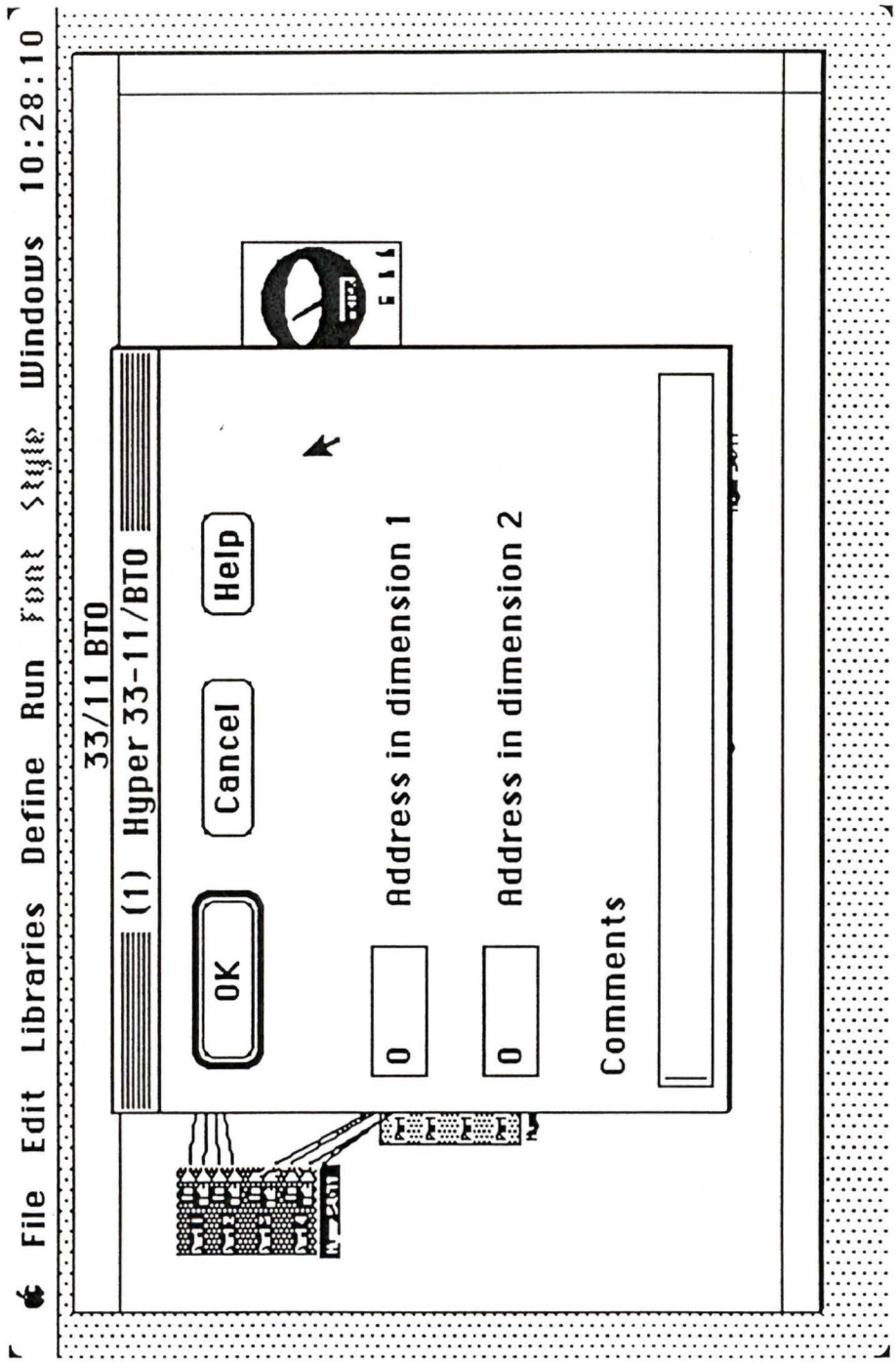


Figure 3.4 Dialog Box for Hypercycle Node.

Once the node processes have been encapsulated in MoDL code, icons representing the processing nodes are linked together as shown in figure 3.3. Links are assigned priorities so that collisions can be resolved. To insure uniform results alternate simulations were performed with reverse priorities assigned. Figure 3.4 shows how each node is given a unique address. Dialog boxes such as these are part of the MoDL language. Finally, an icon called Meter was created to access certain global properties of the system, such as message generation time and packet length. These properties could be easily moved to the node dialog boxes to permit experimentation in areas such as hot spotting.

While the simulator is running, each node collects statistics on the number of messages successfully routed and their average lifetime. When the simulation finishes each node's statistics are written to a data file.

3.4 Simulation Results

A packet transmission time (over an already established source to destination path) of 100 simulation clock ticks was assumed for every simulation. Throughput and delay characteristics for a number of Hypercycle networks with both e-cube and backtrack-to-the-origin-and-retry routing were calculated in terms of the load (input load). Both the input load and the throughput[3] were normalized in terms of

[3]Throughput is defined here as the rate of successful message transmission per unit time. That is, the number of paths which have achieved completion per unit time.

the maximum capacity of each network, taken to be proportional to the number of links in the corresponding graph. This is done as follows:

$$\text{input load} = \frac{\text{number of messages offered per unit time} \times \text{message length}}{n_l \times \text{overall simulation time}}$$

$$\text{throughput} = \frac{\text{number of successfully delivered messages} \times \text{message length}}{n_l \times \text{overall simulation time}}$$

where n_l is the number of links in the network.

The average delay was expressed in actual time units (simulation clock ticks) necessary to establish a source-to-destination circuit.

Figure 3.5 shows the throughput performance of a $G_3^{1\,1}{}_3$ Hypercycle. The results show that under increasing load, packets covering longer average distances exhibit lower throughput rates than packets of shorter average distances. This can be attributed to the fact that the packets of longer average distance find it increasingly difficult to obtain a complete path and, therefore, according to the backtrack-to-the-origin-and-retry algorithm, dissolve the partial path already created and attempt to find another. Figure 3.6 shows the delay performance for the same Hypercycle. The effects of increasing load are clearly seen here. The delay is an exponentially related function to the input load due to the increase in collisions and the increase in blocked paths.

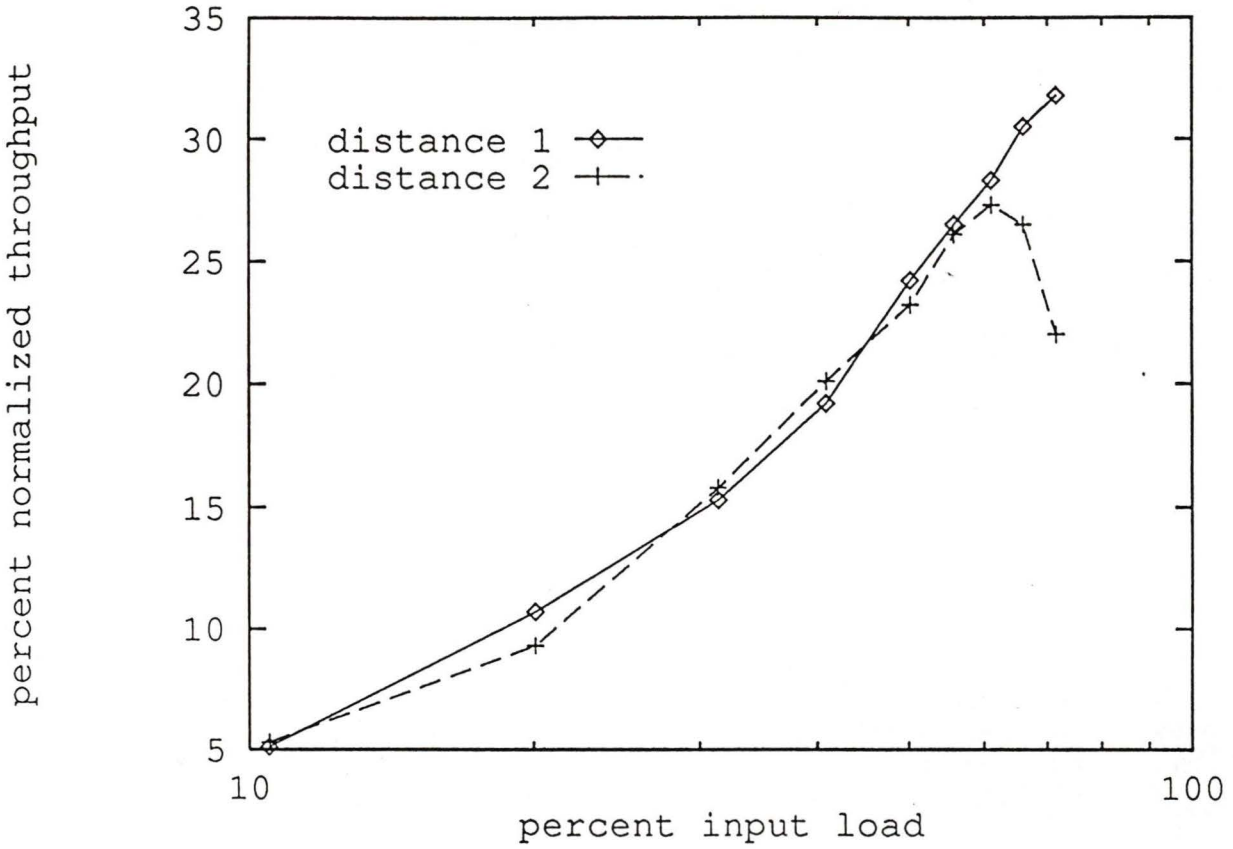


Figure 3.5 Throughput vs. input load for a G_{33}^{11} Hypercycle broken down by message distance.

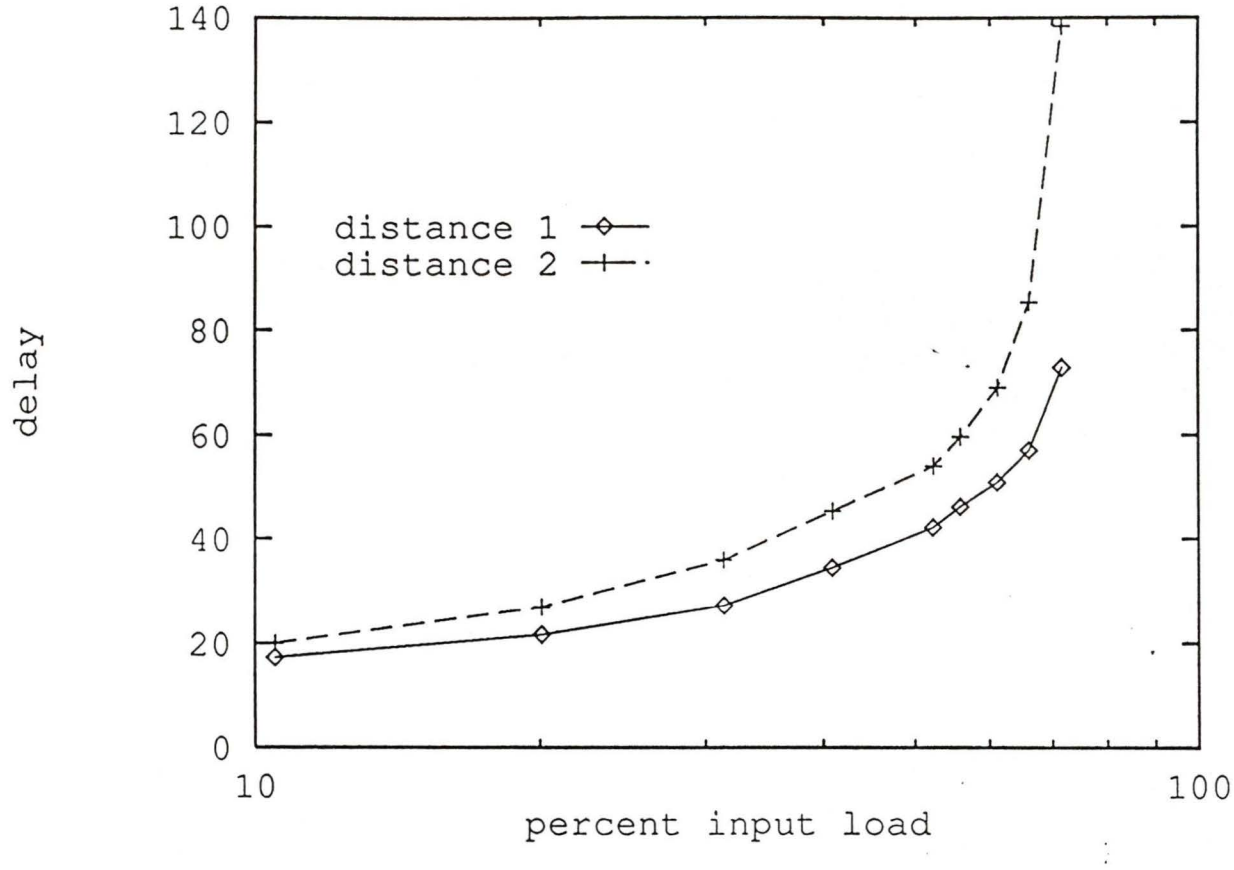


Figure 3.6 Delay vs. input load for a G_{33}^{11} Hypercycle broken down by message distance.

Figures 3.7, 3.8 and 3.9 show the delay performance of a binary 4-cube using both backtrack-to-the-origin-and-retry routing and e-cube routing. As it was expected, the performance of the backtrack-to-the-origin-and-retry is clearly superior to that of the e-cube. This is attributed to the fact that the backtrack-to-the-origin-and-retry can use alternative paths to the destination instead of the single path allotted by the e-cube routing which must wait for each segment to become available. Note the characteristic shapes of the delay curves for each routing method. E-cube routing, which waits at blocked paths, has a gradual but steady increase in delay as the input load is increased. Backtrack-to-the-origin tends to maintain its low delay times until the input load is such that the network has reached its capacity; then the delay escalates quickly.

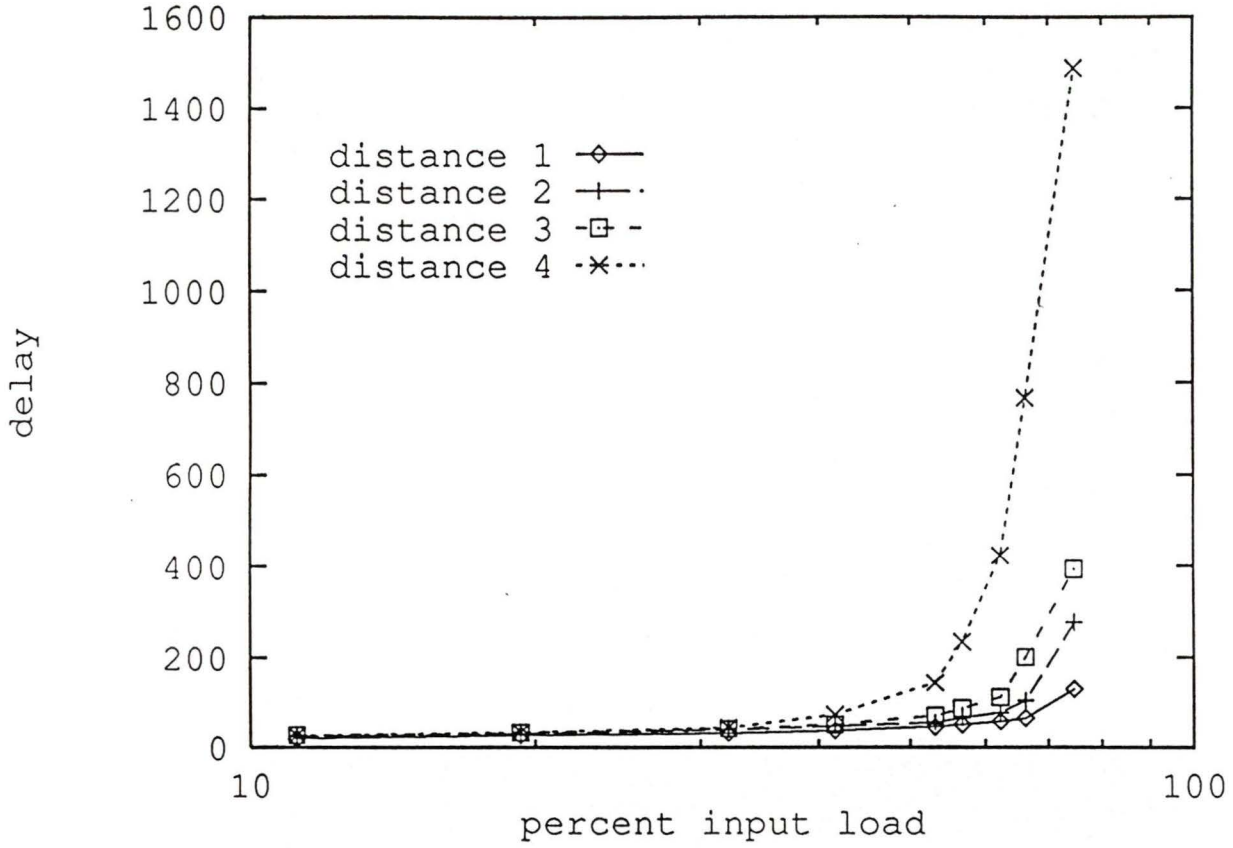


Figure 3.7 Delay vs. input load for a G_{2222}^{1111} Hypercycle using BTOR routing broken down by message distance.

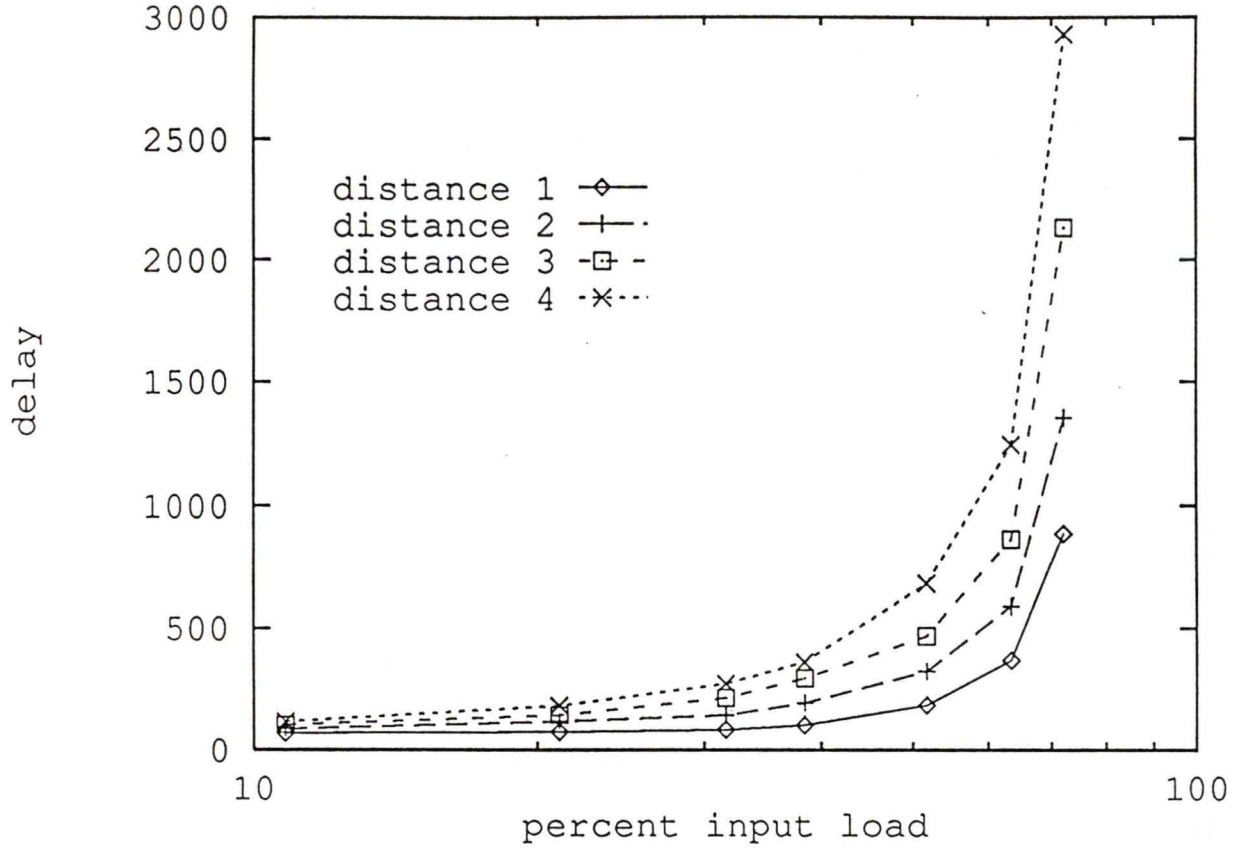


Figure 3.8 Delay vs. input load for a G_{2222}^{1111} Hypercycle using e-cube routing broken down by message distance.

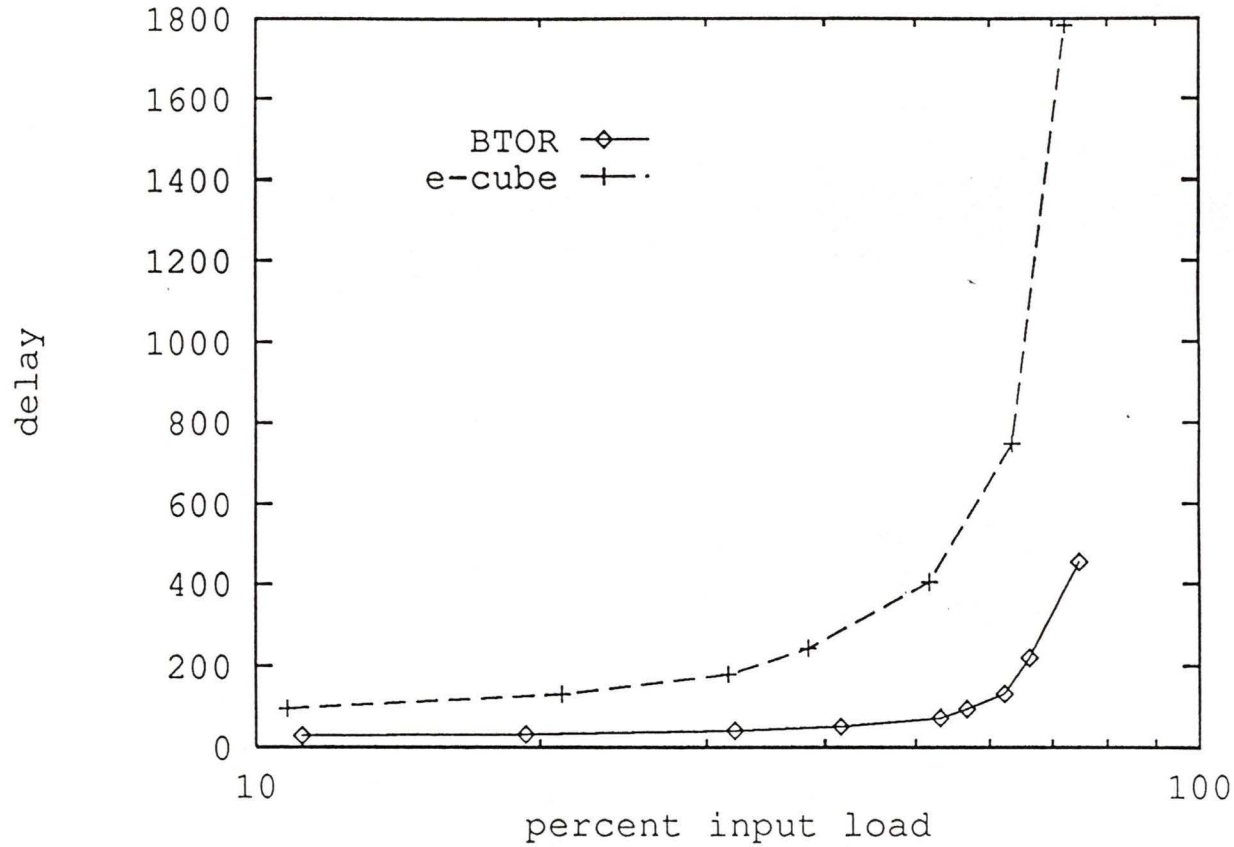


Figure 3.9 Delay vs. input load for a G_{2222}^{1111} Hypercycle broken down by routing algorithm.

Figures 3.10 and 3.11 show that under heavy loads, e-cube[4] routing has slightly higher throughput rates for a given load than backtrack-to-the-origin. As mentioned before, backtrack routing does not effectively route packets of longer average distances under heavy loads because the path is dissolved as soon as a blocked route is encountered, which has a high probability of occurrence. Thus the message spends its time looking for routes which do not exist (i.e., have a low probability of existing) thus never completing a successful route. As a further note, the delay performance of both routing algorithms at the loading levels required to exhibit this behaviour is so poor that such loading levels could be considered as overloading the network. Indeed, if one examines the delay graph of figures 3.9, one can see that the "knee" or critical loading point above which delay is exponential is below the point at which backtrack routing begins to exhibit lower throughput than e-cube routing. Nevertheless, under more realistic operating conditions, light to medium loads, BTOR is clearly superior to e-cube routing.

---

[4]Note that since a binary n-dimensional cube is a Hypercycle which contains no cycles in any of its dimensions, it is possible to use e-cube routing without the inherent deadlock problem encountered for the general Hypercycle as noted in chapter 2.

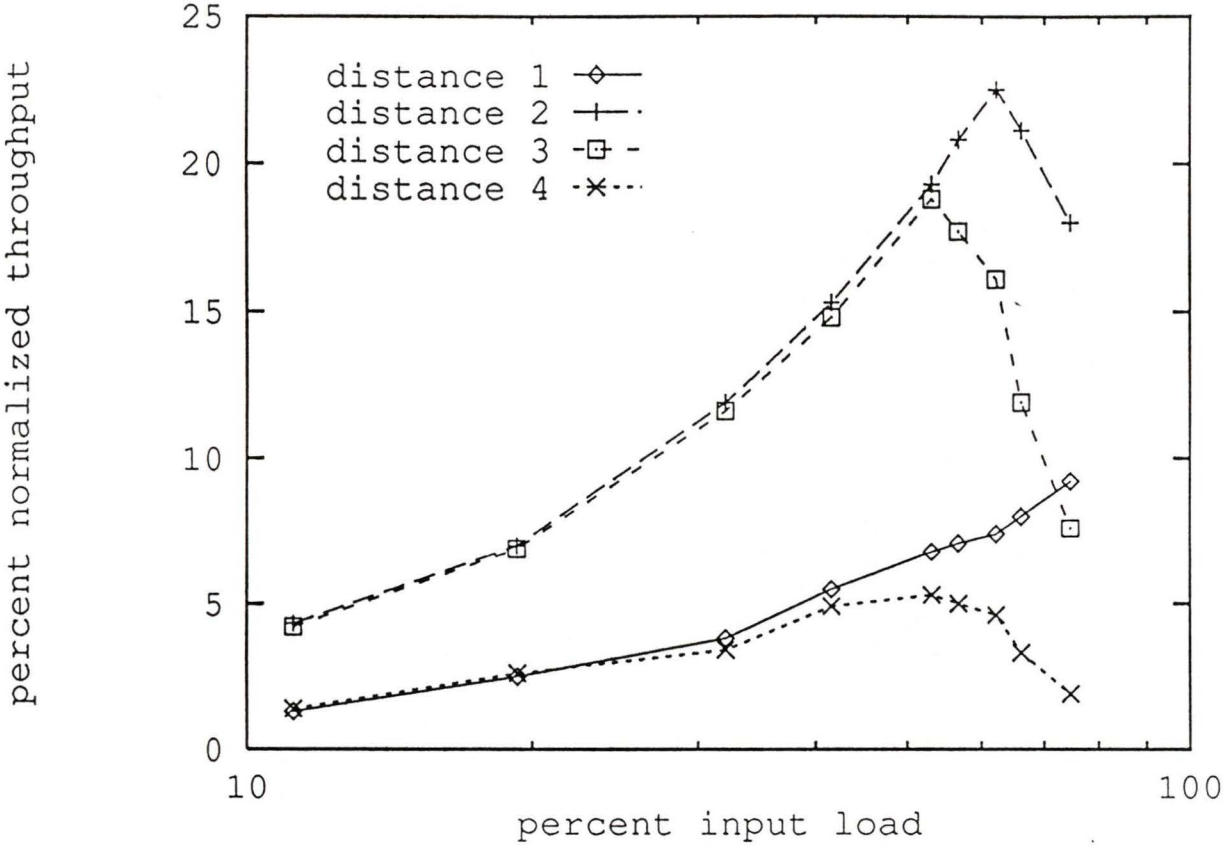


Figure 3.10 Throughput vs. input load for a G_{2222}^{1111} Hypercycle using BTOR routing broken down by message distance.

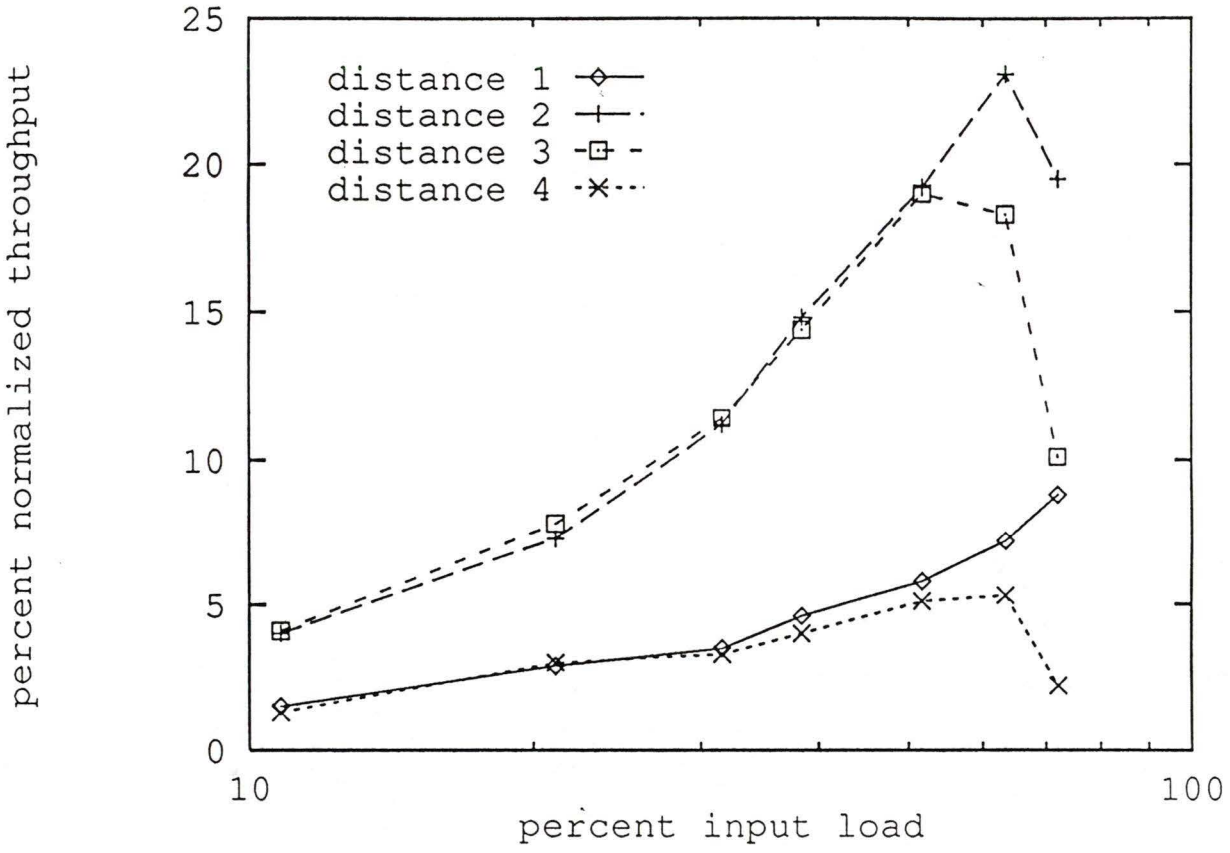


Figure 3.11 Throughput vs. input load for a G^{1111}_{2222} Hypercycle using e-cube routing broken down by message distance.

The comparison of binary n-cubes to alternate Hypercycles of similar size shows that other Hypercycle graphs exist which offer better performance. Consider Figure 3.12 which compares the throughput characteristics of the two previous 4-cubes, one with BTOR routing and one with e-cube routing, with a $G_3^{1\,1}{}_3$ and a binary 3-cube, both using BTOR routing. This clearly shows the expansion capabilities of Hypercycles. The increased throughput of the $G_3^{1\,1}{}_3$ can be attributed to the fact that this graph can use the higher interconnectivity of a degree 4 node whereas the binary 3-cube is restricted to degree 3. Further evidence of the superiority of a general Hypercycle graph structure is given in the delay characteristics of the above four topologies as shown in figure 3.13, which clearly shows the $G_3^{1\,1}{}_3$ has superior performance over the binary 3-cube even though both have the same routing algorithm (BTOR). Again, this is attributed to the ability of the $G_3^{1\,1}{}_3$ to use the higher degree of connectivity.

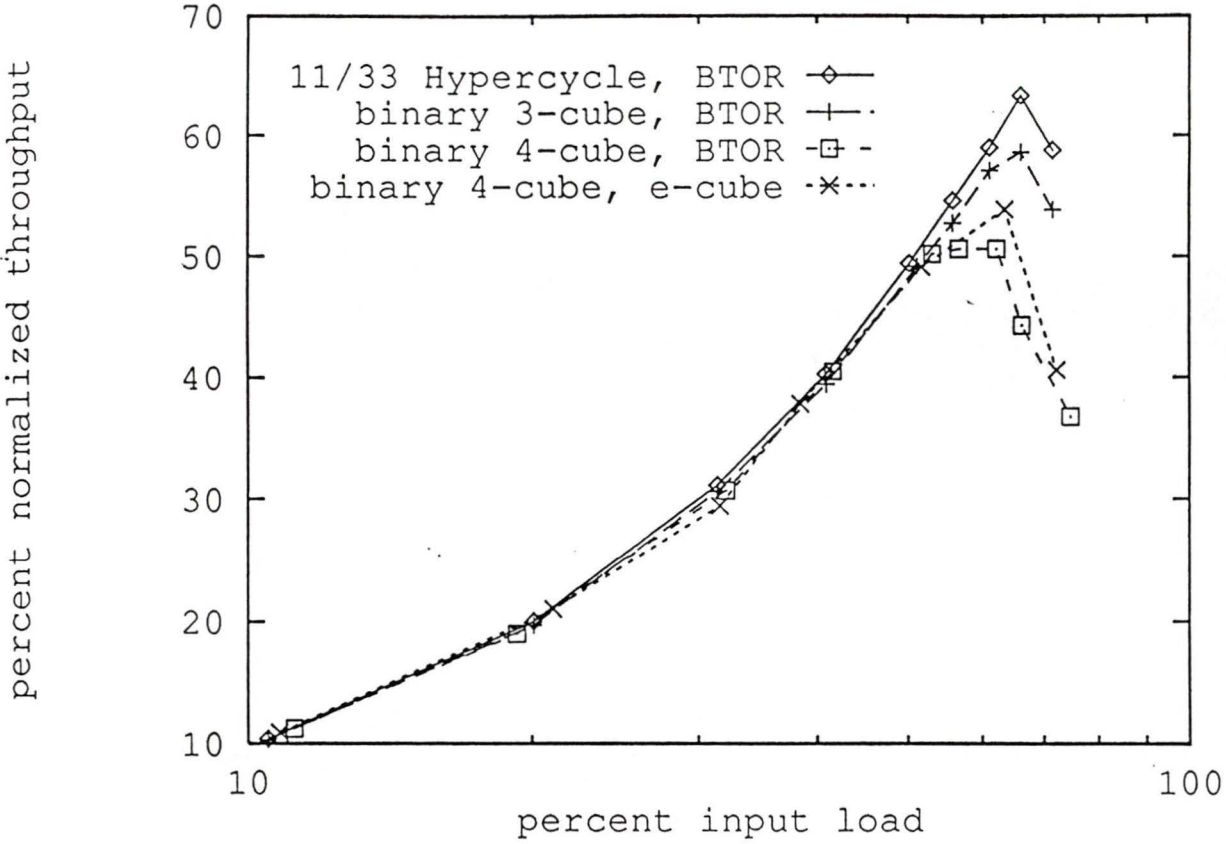


Figure 3.12 Throughput vs. input load for various Hypercycles.

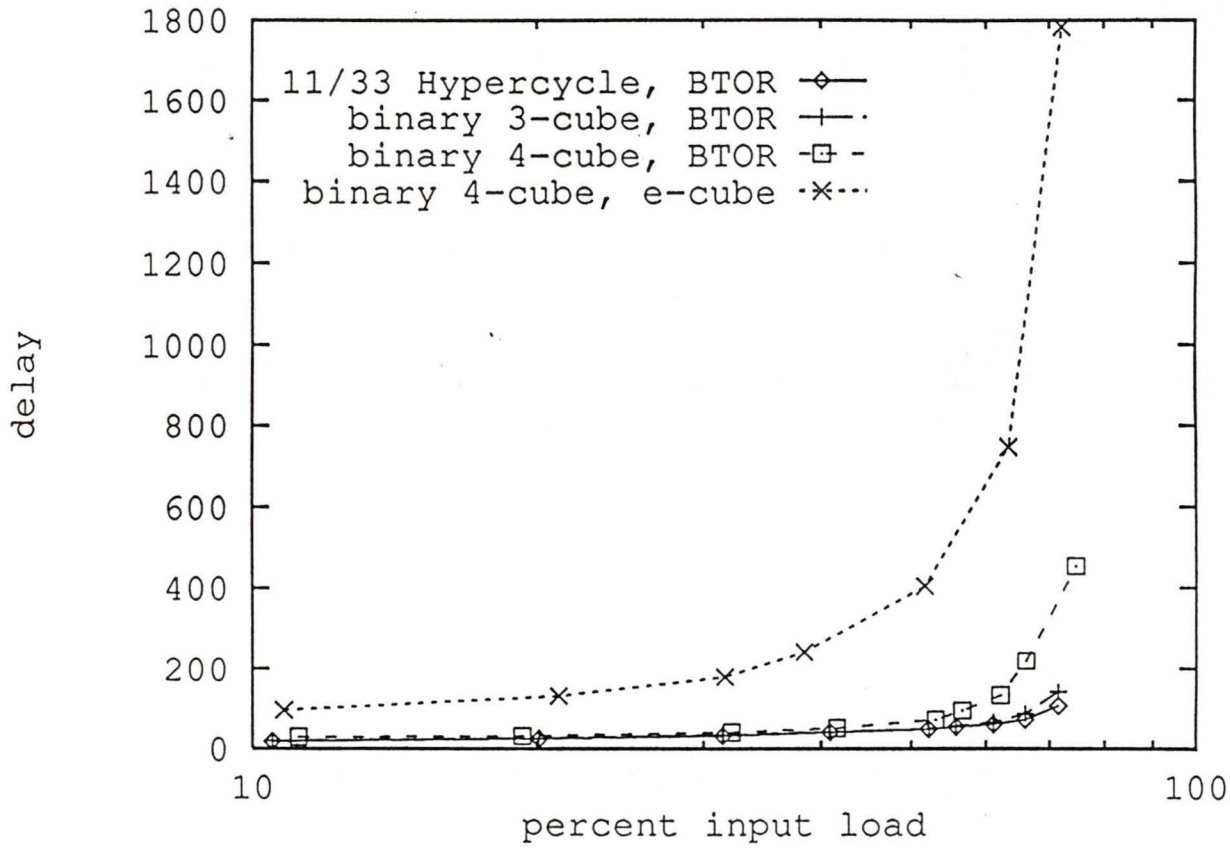


Figure 3.13 Delay vs. input load for various Hypercycles.

For graphs of higher degrees, figures 3.14 and 3.15 show the effect of alternate paths on system delay and throughput. The 7 node Hypercycle has only one path choice per source/destination connection (because it is a fully connected graph). This effectively reduces the system to e-cube style routing only. The 15 node $G_{5\,3}^{2\,1}$ and 16 node $G_{4\,4}^{2\,2}$ graphs offer two alternate routes between source and destination and therefore provide higher system throughput. Generally, system throughput and delay are functions of both average distance and the average number of alternate paths between any two nodes. Indeed, in [18] it was shown that, for minimal-distance routes in a mesh topology, the optimal path in terms of delay performance between a source and destination node is the one that maximizes the number of alternate paths at each intersection.

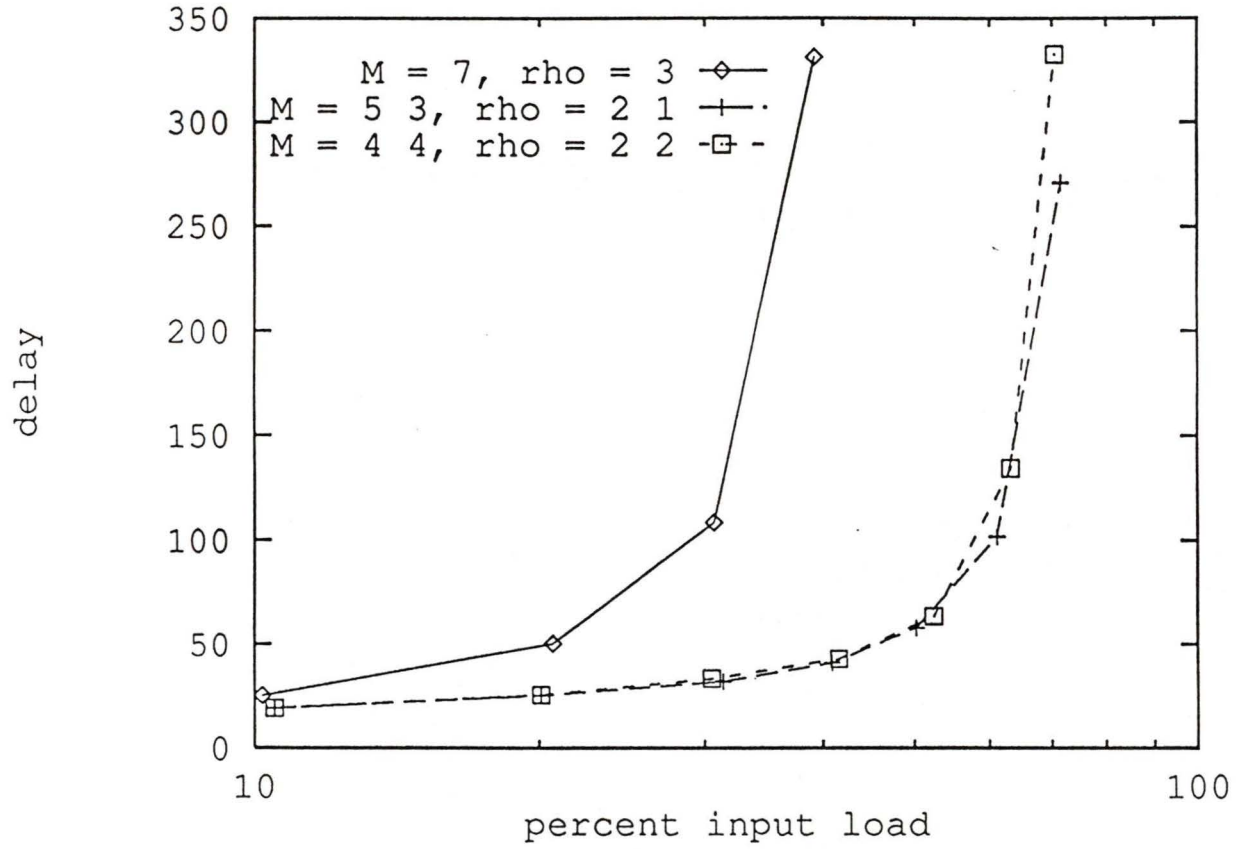


Figure 3.14 Delay vs. input load for various Hypercycles.

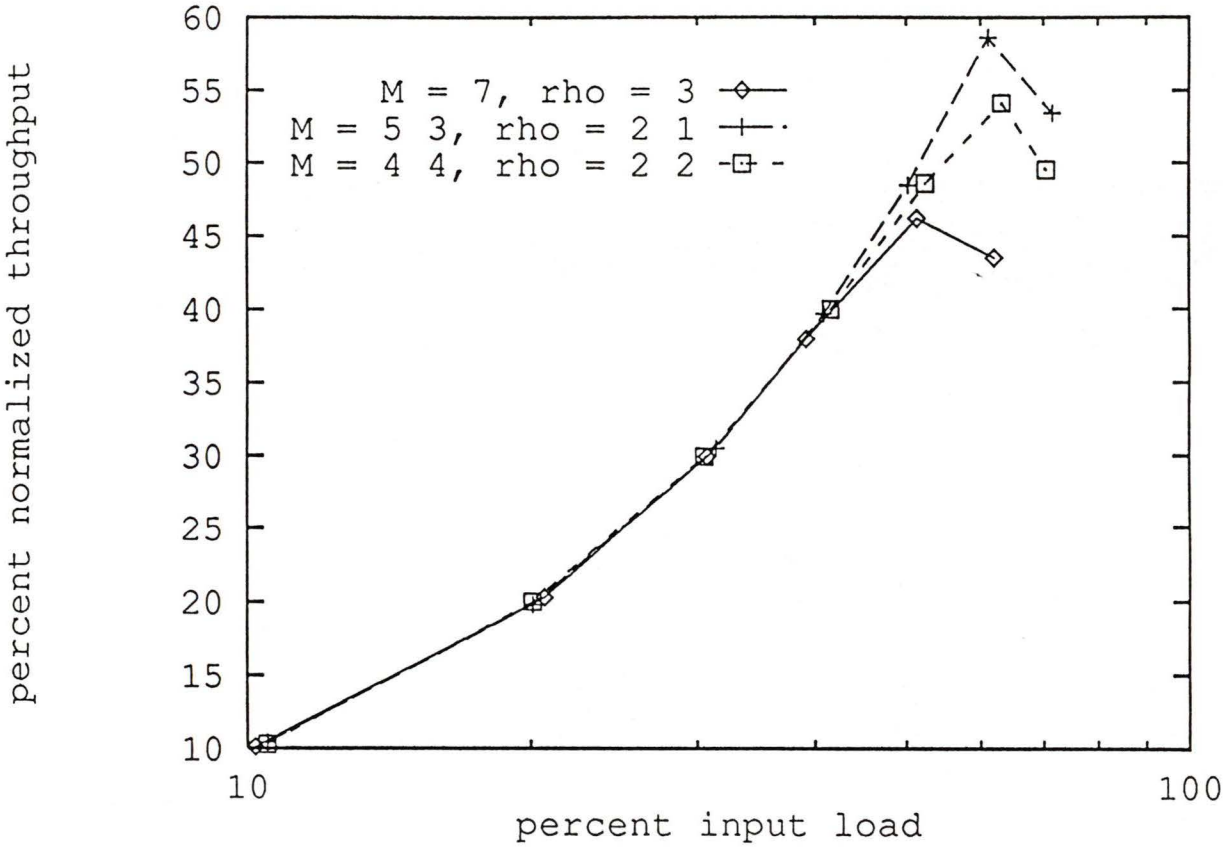


Figure 3.15 Throughput vs. input load for various Hypercycles.

When the simulator is working with networks subject to large input loads, it was shown that some messages have difficulty finding routes when using backtrack-to-the-origin-and-retry routing. When the simulation is halted, there may exist packets which have still not been routed. Even though the probability that these packets will complete successfully is small (because the probability of a free existing path is small), there still is some probability that these packets will eventually complete their paths. However, in order to insure that the results of the simulations are valid, these packets are assumed to not have completed their paths and thus are not included in the results for throughput and delay. Note that when the offered load is large enough to cause this phenomena, the delay approaches an asymptote whose slope is infinite, so that any additional offered load can be assumed to cause infinite delays. Furthermore, it is noted that e-cube routing approaches infinite asymptotic delay at throughput rates where backtrack-to-the-origin-and-retry routing delay is still small.

Chapter 4. Hardware Realization

With the properties of Hypercycles and backtrack-to-the-origin-and-retry routing examined and simulated, it is now possible to discuss the implementation of a Hypercycle router. Assuming that each node in an interconnection network corresponds to a vertex in the associated graph and has been numbered accordingly, and in addition edges correspond to communication channels which are attached to ports at each individual node, a Hypercycle router with backtrack-to-the-origin-and-retry-routing would need to possess the following capabilities:

- provide a set of possible next nodes according to equation 2.3.1.1 for a given destination address
- randomly choose a single port from the set of available ports
- generate break signals if no ports are available
- control switching elements (for communication ports)

These capabilities limit the technologies in which a router may be implemented. Implementation technology alternatives that are feasible include:

(a) Microcontroller with routing algorithm in software

(b) Read-Only-Memory (ROM) lookup tables

(c) Discrete components including Programmable Logic Arrays (PLA's) and gate arrays

(d) Very Large Scale Integration (VLSI)

The first alternative, a software algorithm executed by a dedicated microcontroller, would be relatively inexpensive and the most flexible option. However, with cycle times of 50 to 100 nanoseconds or more per instruction, such an implementation would yield a very slow router. ROM lookup tables could be used. However, each node would require its own, unique set of ROMs and each distinct Hypercycle would also require a different set of ROMs. This is not a general solution. Furthermore, control logic would still be required to validate possible ports as being available, randomly select ports, etc..

Hypercycles permit the routing to be computed analytically. As shown in chapter 2, the equations for routing involve only the simple arithmetic operations of addition, subtraction and modulus. It is possible, therefore, to implement these operations directly in hardware. Furthermore, since routing in each dimension is mutually exclusive from any other dimension, the time-space complexity can be

reduced. This makes options (c) and (d) possible candidates for implementation. Discrete components provide a rapid prototyping stage and can execute quickly. However, they result in large circuits with high component counts. This can be reduced somewhat by using large gate arrays which are now available[34]. The last option is VLSI. Circuits using such technology are fast and have a greatly reduced component count over discrete logic. The drawback here is the long and expensive development time.

The optimum solution would lie between options (c) and (d). If a discrete system could be designed and prototyped, and then moved into gate array or VLSI technology, the benefits of rapid design, high execution rates and low component count could be realized. Indeed this is what is being done. Using a schematic capture system, the initial discrete logic design can be simulated on a computer. Once finalized, a sufficiently fast technology can be chosen to implement the design.

4.1 System Placement

Processing nodes in a concurrent computer usually contain a central processing unit (CPU), local memory and some form of input and output hardware with which the node can communicate with the network. The Hypercycle hardware router is optimally placed alongside the communication hardware, as shown in figure 4.1. At this location, the router can control the messages entering and leaving the

node as well as messages passing through to other nodes via a crossbar switch. Note that the switch would be necessary to control communication channels regardless of the implementation technology of the router. Requests for network services are given by the CPU to the routing hardware in the form of a small data packet which contains the destination address. This packet is then either forwarded to the appropriate connecting node or returned to the CPU for requeueing. The router is configured by the node's CPU. Configuration information includes its local address ξ, the dimensional populations m and the dimensional connectivities ρ.

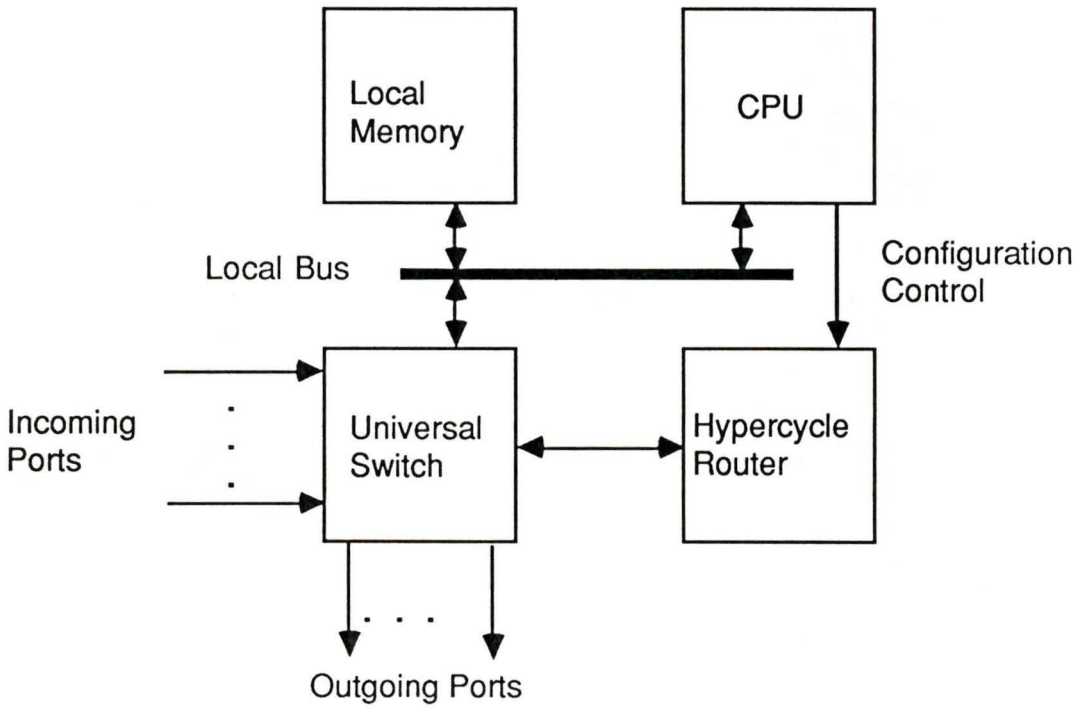


Figure 4.1 Configuration of a processing node with a Hypercycle routing circuit.

4.2 General Design

Given the requirements of a router outlined at the start of the chapter, one possible design is shown in figure 4.2. This circuit works as follows. A destination address from an incoming port or from the host node CPU is presented at the destination address register. The Next Port Generators then determine which ports can be used for the continuation of the path. The Port Validator ensures that the ports indicated are available for use. Then the port selector randomly chooses a single port to which the originating port is connected and to which the destination address is forwarded. The major functional units of the router, shown in figure 4.2, are described below:

a) Next Port Generator:

The subsystem termed the Next Port Generator provides the next port to which a message may be routed. This is done by using a modulo arithmetic circuit based on the algorithm of equation 2.3.1.1 to obtain a relative address with respect to the current address. In each dimension i, there are $2\rho_i$ ($2\rho_i$ -1 if $\rho_i=m_i/2$) possible edges. To each of these edges, there is a corresponding communication channel and a logical port. We number the logical ports from 1 to $2\rho_i$ ($2\rho_i$-1) so that the minimum step in the counterclockwise direction will correspond to logical port 1, the maximum step to logical port ρ_i,, while the minimum step in the clockwise direction will correspond to logical port ρ_i+1 and the maximum step to port $2\rho_i$ ($2\rho_i$-1). Finally,

this logical port number is mapped to a physical port number by accounting for all the ports used by previous dimensions and then adding this as an offset to the logical port.

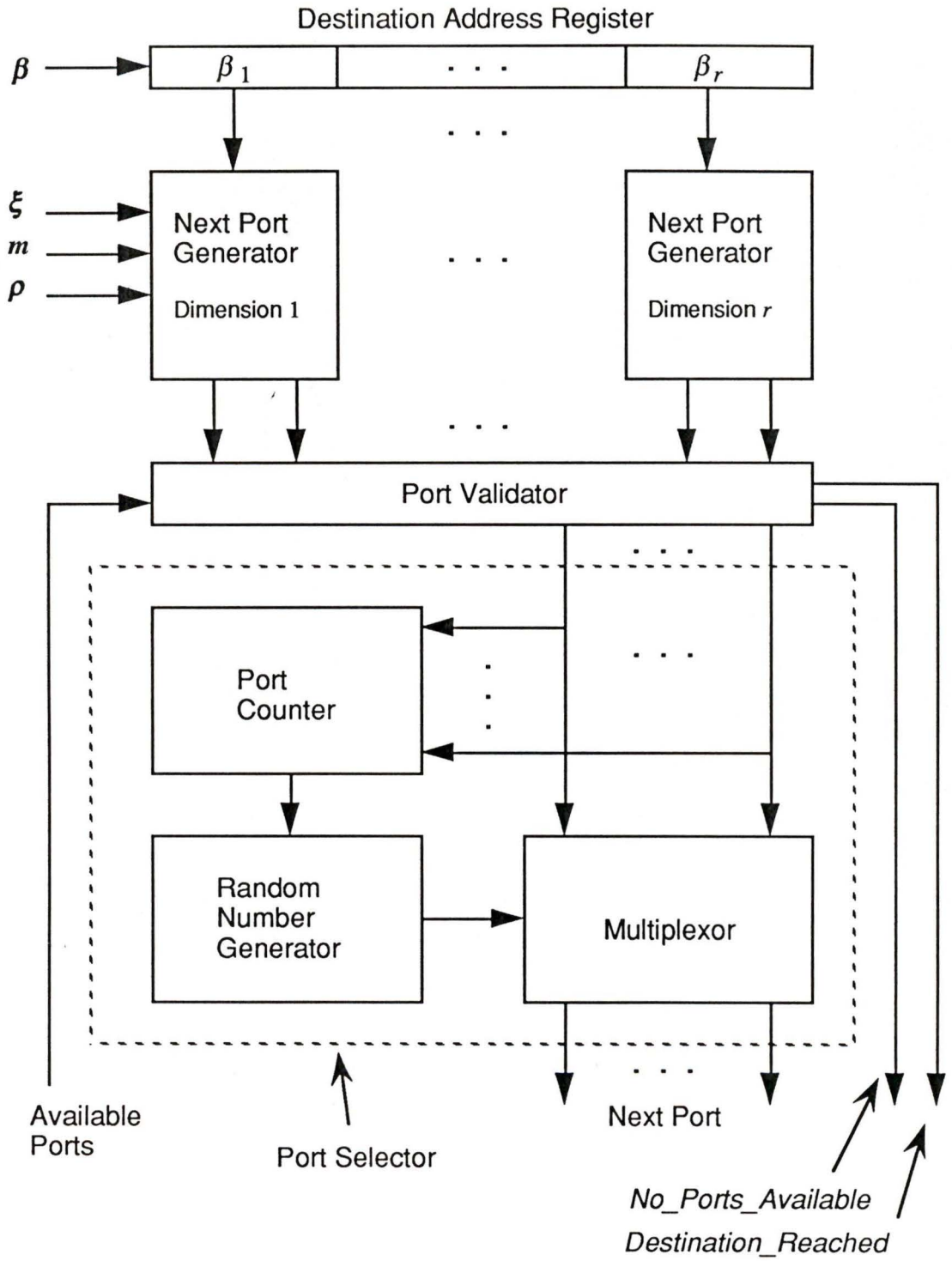


Figure 4.2 Details of the Hypercycle router circuit.

As an example, consider the case given in figure 4.3. If a message starts at node 2 bound for node 5, the first step is to determine which route is shortest; clockwise or counterclockwise. In this case the clockwise route is shortest and this yields a largest possible step of 2. Since the route is clockwise, the step size, 2, is offset by the connectivity, ρ_i, which produces a logical port of 4. If this were the first, or only, dimension of the Hypercycle, then this number would also correspond to the logical port. If other dimensions existed before this dimension, then the logical port would be offset by the number of ports used in all previous dimensions to yield a physical port number.

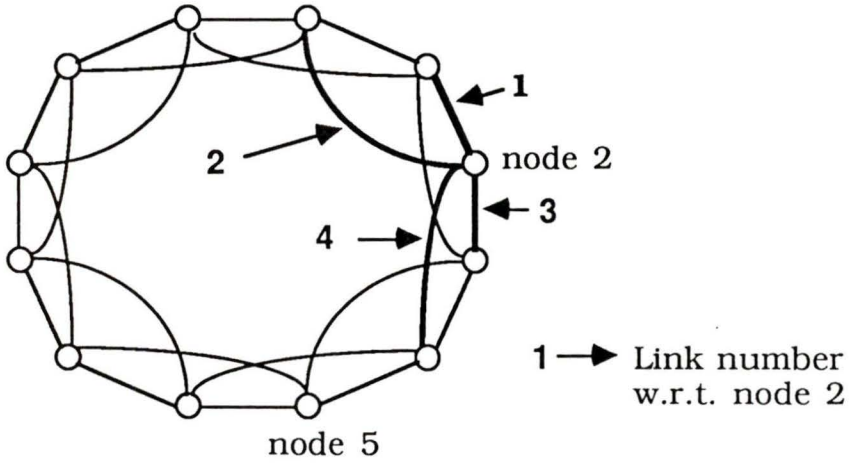


Figure 4.3 Determining the shortest route from node 2 to node 5.

This subsystem also indicates if the destination within any single dimension has been reached. In an r-dimensional Hypercycle, this unit is replicated r times in each router.

b) Port Validator:

The possible ports are fed from the Next Port Generator to the Port Validator which first decodes all r offered ports into one of the$d$, where d is the degree of the graph, physical ports. Note that one dimension may use many ports if its connectivity is high, although a message can be routed to only one of the ports[1]. It then matches these ports with those ports that are not currently in use. If there are any ports offered which are still available, these are forwarded. If there are no offered ports which are still available, then this unit can generate a break connection signal to be passed down the partially completed path. The Port Validator is composed of a demultiplexor followed by a logical ANDing of signals which indicates if the ports are available for routing (i.e. not already allocated) and the demultiplexed port signals.

c) Port Selector:

If the Port Validator has found available ports for routing, these are passed to the Port Selector. The Port Selector must accomplish the following tasks: count the number of valid ports, generate a random number between one and this count, select the port corresponding to the random number.

---

[1]This is true in this particular implementation, which is presented here for simplicity. However, it will be shown in section 4.3.3 how a single dimension can offer more than one port.

Other subsystems, primarily residing in the communication switch, would detect break signals and connect and disconnect communication links under the control of the router itself.

4.3 Design of NPG

4.3.1 Hardware Design

The Next Port Generator (NPG) implements the routing equations of chapter 2 and provides a physical port number to which the message can be routed. Since there is one NPG for each dimension in the system, routing for each dimension is done in parallel. Figure 4.4 shows a block diagram of a single NPG. The operation of the subsystem is as follows.

Given a local address ξ_i, a dimensional population m_i, and a destination address β_i, the minimum of

$$(\xi_i - \beta_i) \bmod m_i$$
$$(\beta_i - \xi_i) \bmod m_i$$

as required by 2.3.1.1 is calculated and forwarded. This essentially decides whether it is shorter to move clockwise or counterclockwise about the cycle and what the length of such a move would be. Note that it can be determined at this point if the destination address within this dimension has been reached since

$$(\xi_i - \beta_i) \bmod m_i = 0 \text{ if } \xi_i = \beta_i$$

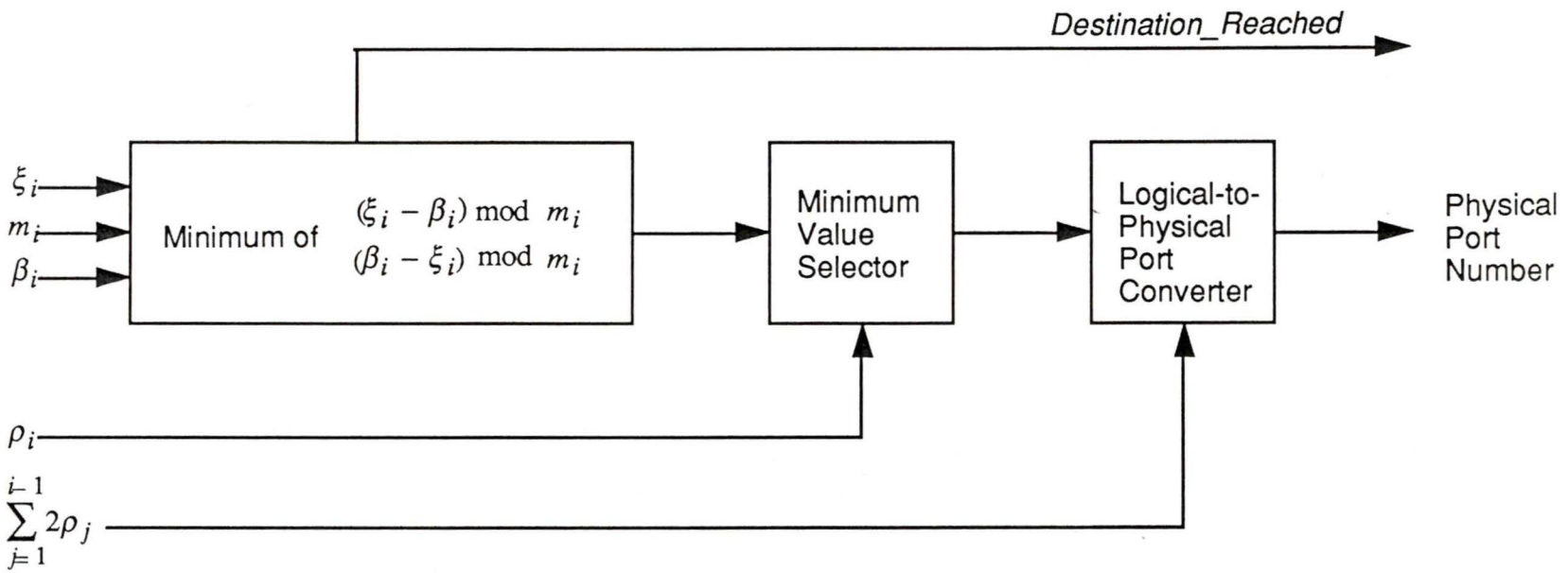


Figure 4.3 Block diagram of a Next Port Generator for dimension i.

The next step is to decide which is smaller, the connectivity ρ_i, which decides the maximum possible leap, clockwise or counterclockwise, or the destination itself. This must be done since a step size greater than ρ_i cannot be taken. The result of this choice is the optimum logical port in this dimension (assuming a greedy strategy, as is the one used here, is employed). This logical port is then forwarded to a circuit which converts it to a physical port, based on the number of ports which have preceded this dimension. It is assumed here that the lowest port numbers are used to route for dimension 1, the next lowest possible set for dimension 2, etc.. For example, in a $\mathfrak{G}_5^{2\ 1}{}_3$ Hypercycle, the first $2\rho_1 = 4$ ports are used for dimension 1 and the remaining $2\rho_2 = 2$ ports are used for dimension 2.

The operations of 4.4 can be simplified somewhat if one notes the following:

Given a number $n \in \Re^+$, then, by the definition of the modulo operation[1],

$$\lambda \bmod n = n - |\lambda| \quad \text{for } \lambda < 0 \text{ and } |\lambda| < n$$

and

$$\lambda \bmod n = \lambda \quad \text{for } 0 \le \lambda < n \ .$$

Therefore, it is sufficient to find the minimum of

$$|\beta_i - \xi_i| \qquad 4.1a$$

$$m_i - |\beta_i - \xi_i| \qquad 4.1b$$

instead of the minimum of

$$(\xi_i - \beta_i) \bmod m_i$$
$$(\beta_i - \xi_i) \bmod m_i \ .$$

Before the circuits of the Next Port Generator can be designed, several issues must be resolved. The maximum value of m_i would place a lower bound on the number of bits used to represent the various parameters m_i, β_i, ξ_i and ρ_i. Three bits per parameter permit up to eight nodes per dimension (since $2^3 = 8$). While this is acceptable, the use of four bits (16 nodes per dimension) to represent node addresses would permit the use of commercial, four-bit combinatorial logic circuits, such as adders, comparators and multiplexors.

With four bits to represent node addresses, nodes can be numbered from 0 to 15. However, if m_i is represented as a four bit number, then it is possible to provide populations of only 0 to 15 (since the binary number 1111 is 15 in base 10). Hence, the actual maximum allowable number of nodes per dimension is 15 using four bit addresses (0 to 15 nodes per dimension). Therefore, nodes must be numbered from 0 to 14 (0 being the first node in the cycle and 14 being the fifteenth node in the cycle). This is not a critical restriction. Note that since $m_{i_{max}} = 15$, the maximum value of ρ_i is $\rho_{i_{max}} = \lfloor m_{i_{max}} / 2 \rfloor = 7$. Thus, all possible values of ρ_i can be represented in three bits.

To capture the schematic, the CASE™ Technology Stellar Design Environment[35] was used. This design package permits the hierarchical design of circuits using industry standard components.

Once the design was completed, an interface to the SILOS [36] simulation program permits one to simulate the circuits. To complement the simulation interface, simulation files for 74LS small-scale integration (SSI) and medium-scale integration (MSI) technology and 1.5 micron VLSI technology were written.

Figure 4.5 shows the implementation of the block diagram of figure 4.4. All of the primitive components are assumed to be of the 74LS family [37]. The operation of this circuit is as follows.

Firstly, ξ_i is subtracted from β_i. This is done using the sign-bit subtractor component indicated by SUB1. The details of the sign-bit subtractor are given in Appendix B. The result of this subtraction is a positive number from 0 to 14 (since the maximum node address is 14) and a sign bit which is asserted if the result is negative. This positive number is then forwarded to another sign-bit subtractor, SUB2, which subtracts this number from m_i. The result of this subtraction is always positive since,

$$0 \le \beta_i < m_i,$$
$$0 \le \xi_i < m_i$$

therefore

$$\beta_i - \xi_i < m_i \ .$$

Since both subtractors produce positive valued results, equations 4.1a and 4.1b are satisfied.

REVISIONS						
EFF	AUTHORITY	ZONE	LTR	DESCRIPTION	DATE	APPROVED

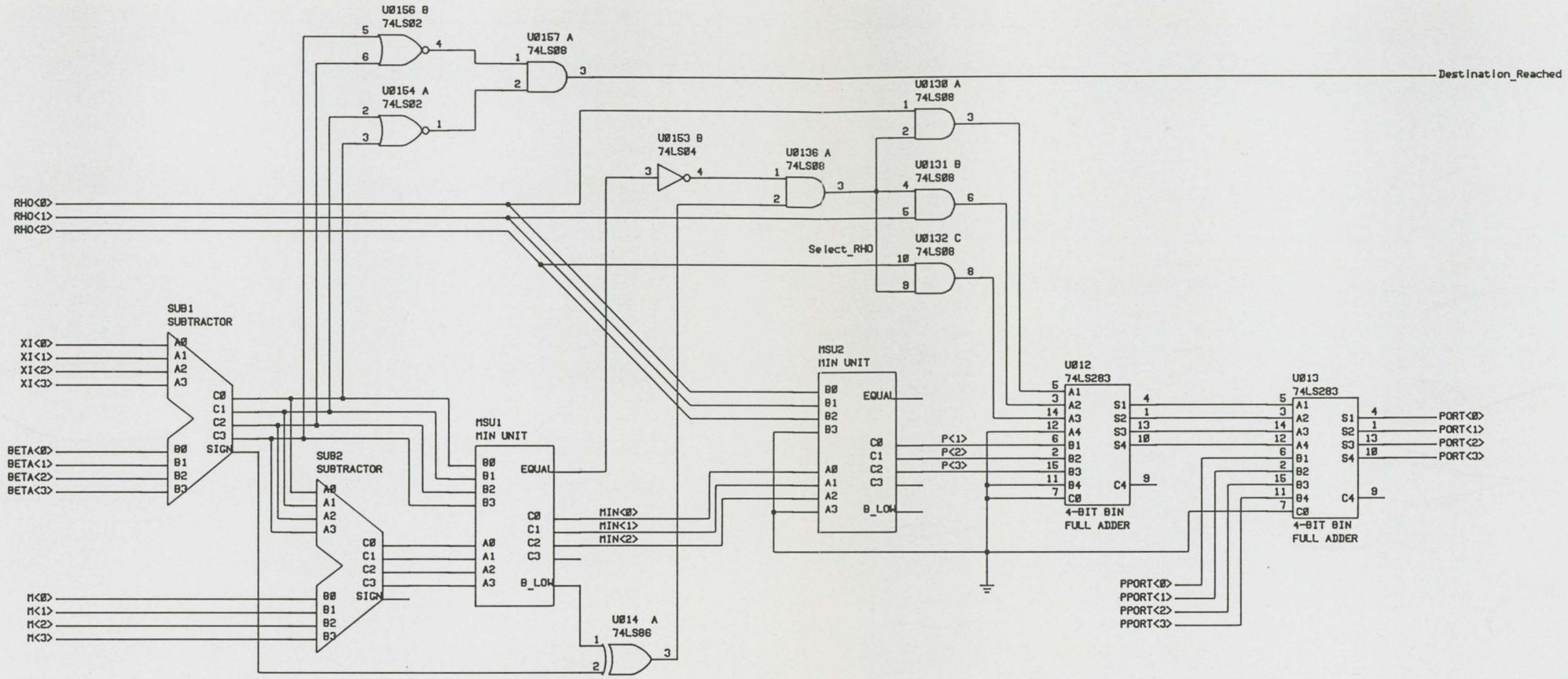


Figure 4.5 Implementation of the Next Port Generator

CONTRACT Hypercycle Router		University of Victoria
DR	/ /	Next Port Generator
CHK	/ /	
ENG	/ /	
APPD	/ /	SIZE D | FSCM NO. - | DWG NO. -
NEXT HIGHER ASSY. -		
14:41:45 26-Jun-90		FIG 4.5 | SHEET 1 OF 1

The two positive numbers are then compared at the minimum value selection unit MSU1. This circuit compares two four bit numbers and forwards the lowest valued one, also generating signals to indicate if the two numbers are equal in value, EQUAL, and if the second input, B, is the low value, B_LOW. The details of this circuit are also given in Appendix B.

The resulting signal emanating from MSU1 is the same as that leaving the first block in 4.4, namely the minimum distance to the destination. This can now be compared to ρ_i at MSU2. The minimum of these two values is the longest step possible given the configuration, clockwise or counterclockwise, to the destination. After MSU2, the direction of travel must be resolved. This is accomplished by exclusive-ORing the sign bit from SUB1 with the B_LOW signal from MSU1 to yield a signal which indicates the direction of travel for the minimum distance. This signal is valid in all situations except when both directions are valid (e.g. $|\beta_i - \xi_i| = m_i - |\beta_i - \xi_i|$). Such a situation occurs when m_i is even, $2\rho_i = m_i$ and a message from node γ_i is bound for node $(\gamma_i \pm m_i/2) \bmod m_i$. An example of such a situation is a binary n-dimensional cube in which $\rho_i = 2$ for all i and $m_i = 2$ for all i. In this case, the port from γ_i to $(\gamma_i \pm m_i/2) \bmod m_i$ must either be numbered as ρ_i or as $2\rho_i$ - 1. This is an arbitrary decision so the cross-cycle link was assigned to port ρ_i. Since this is the case, the signal derived above, which sends a message clockwise or counterclockwise, would have to be quantified with a signal which could detect the equality of the two

directions. Such a signal is available from MSU1. The EQUAL signal of MSU1 is asserted if both inputs are identical, as would be the case if $|\beta_i - \xi_i| = m_i - |\beta_i - \xi_i|$. This signal, when inverted, can be used as a gating trigger for the directional signal, and indeed, this is what is done at the AND gate U0136 A in figure 4.5. The resulting signal is itself a gate for ρ_i, which is simply added to the desired port if the direction is clockwise. The result of MSU2, now the largest possible step toward the destination in this dimension, is added to the gated ρ_i signal through U012. The result of this is a logical port number which can be mapped to a physical port number by adding it to the sum of all previous ports, as mentioned above.

4.3.2 Hardware Simulation Results

The CASE design environment includes an interface to the SILOS digital logic simulation program. This interface permits the setting and monitoring of any signal within the design. After the simulation is performed, CASE displays the result in a graphical fashion, as shown in figure 4.6. On the far left hand side of the diagram are the signal names which correspond directly to the schematic signal names. At the top of the diagram is the timing scale.

Figure 4.7 shows a simulation timing of the Next Port Generator of figure 4.5 as it would operate if the components where implemented in 74LS technology. The timing signals where taken from [37] and indicate maximum possible timings. Details of these timings are given

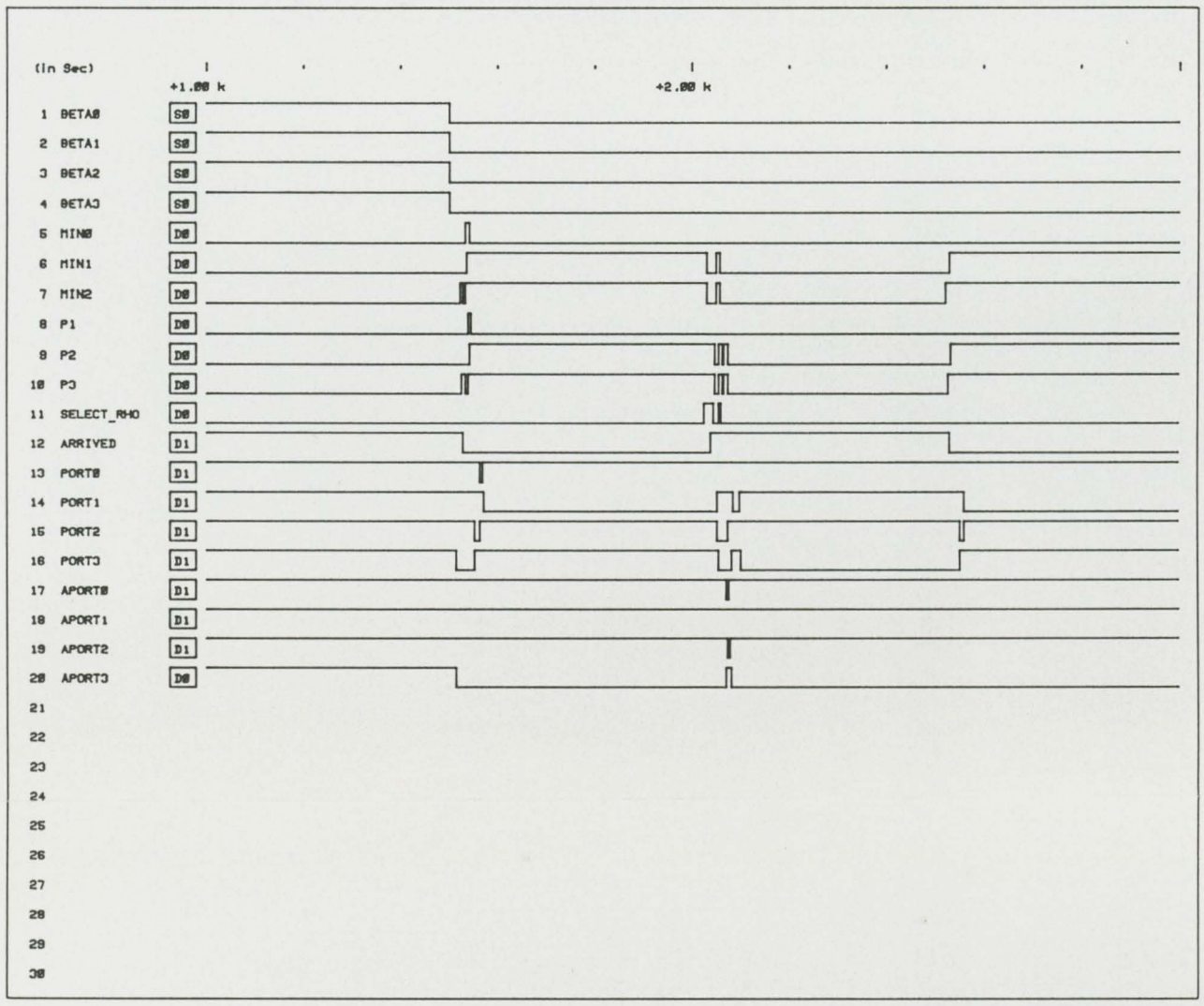


Figure 4.6

Example Simulation Results.

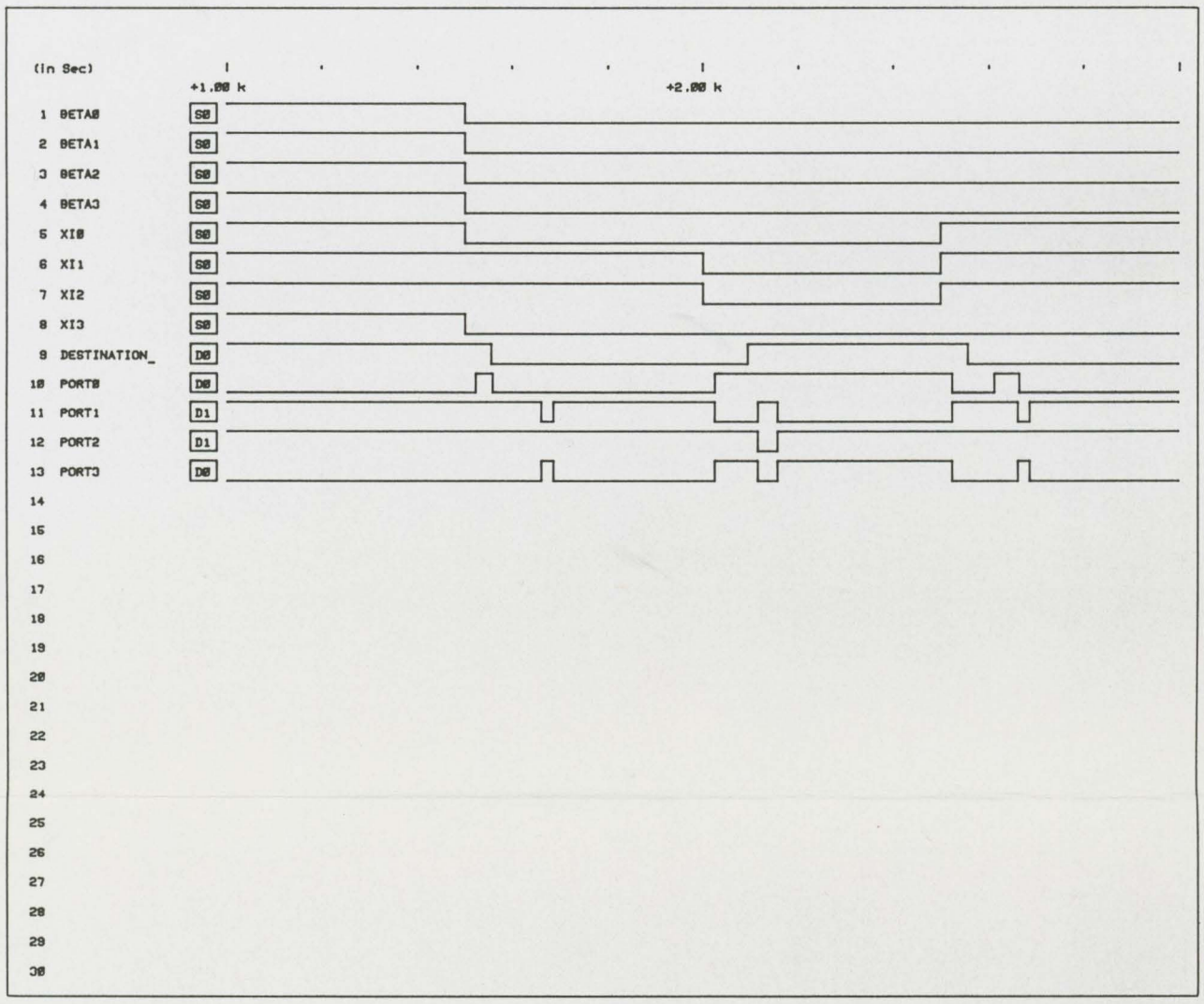


Figure 4.7

TTL Simulation Results.

in Appendix C. The initial 1000 nanoseconds of the simulation are used to let the logic reach steady state from power up. At 1500 nanoseconds the input signals shown in Table 4,1 are presented to the system.

Table 4.1 Input Signals for Figure 4.5

	bit 0	bit 1	bit 2	bit 3
ξ_i	0	1	1	0
β_i	0	0	0	0
m_i	1	1	1	1
ρ_i	1	1	0	
$PPORT_i$	0	0	0	0

After a further 190 nanoseconds, the network settles to produce the following outputs:

Table 4.2 Output Signals for Figure 4.5

	bit 0	bit 1	bit 2	bit 3
Destination_Reached	0			
$PORT_i$	0	1	1	0

Table 4.2 indicates the destination has not been reached in this dimension, which is correct since the current address is not equal to the destination address. Furthermore, the port offered for routing is port 6, which is also correct for a greedy strategy such as this one. Note that since $PPORT_i$ was 0, the logical offered port was 6, and thus the physical offered port is 6.

Figure 4.8 has the timing expanded in the area of 1700 nanoseconds to show how the signals actually settle.

The maximum timing of the subsystem can be determined by examining the critical timing path through the circuit. In this implementation, the critical path is through the carry-in inputs of the 74LS283 devices in the subtractors and through the AND gate U0136 in figure 4.5. This path can be activated by letting $\xi_i > \beta_i$. At 2500 nanoseconds, the inputs shown in Table 4.3 are presented to the system.

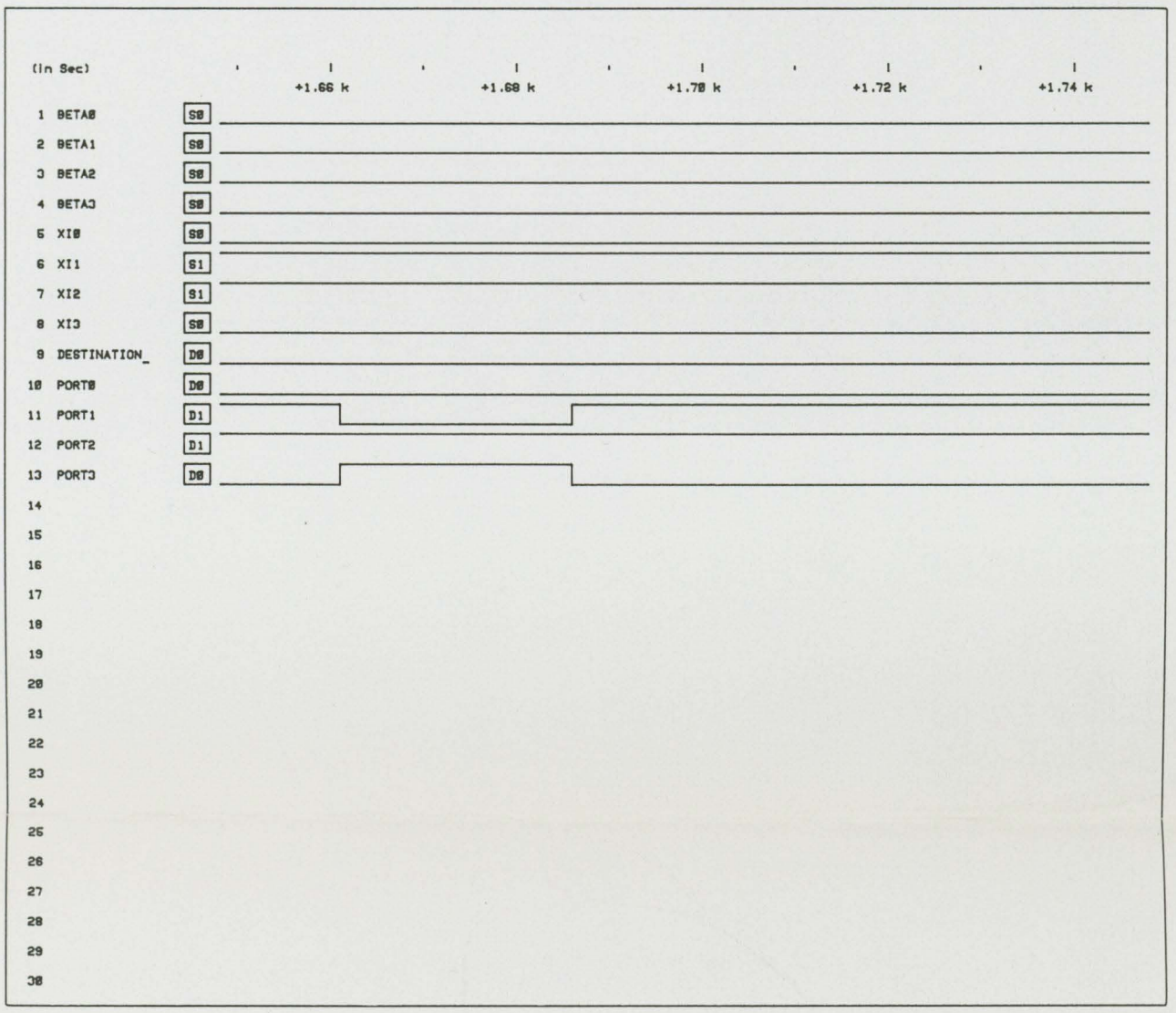


Figure 4.8

TTL Simulation Results Expanded Around 1000ns

Table 4.3 Input Signals for Figure 4.5 (at 2500 ns)

	bit 0	bit 1	bit 2	bit 3
ξ_i	1	1	1	0
β_i	0	0	0	0
m_i	1	1	1	1
ρ_i	0	1	0	
$PPORT_i$	0	1	0	0

After approximately 190 nanoseconds, the network settles to give the following outputs:

Table 4.4 Output Signals for Figure 4.5

	bit 0	bit 1	bit 2	bit 3
Destination_Reached	0			
$PORT_i$	0	1	1	0

Figure 4.9 shows the timing signals from figure 4.7 expanded in the area of final settling. As can be seen from the simulation results the network settles in approximately 190 nanoseconds. Since this is

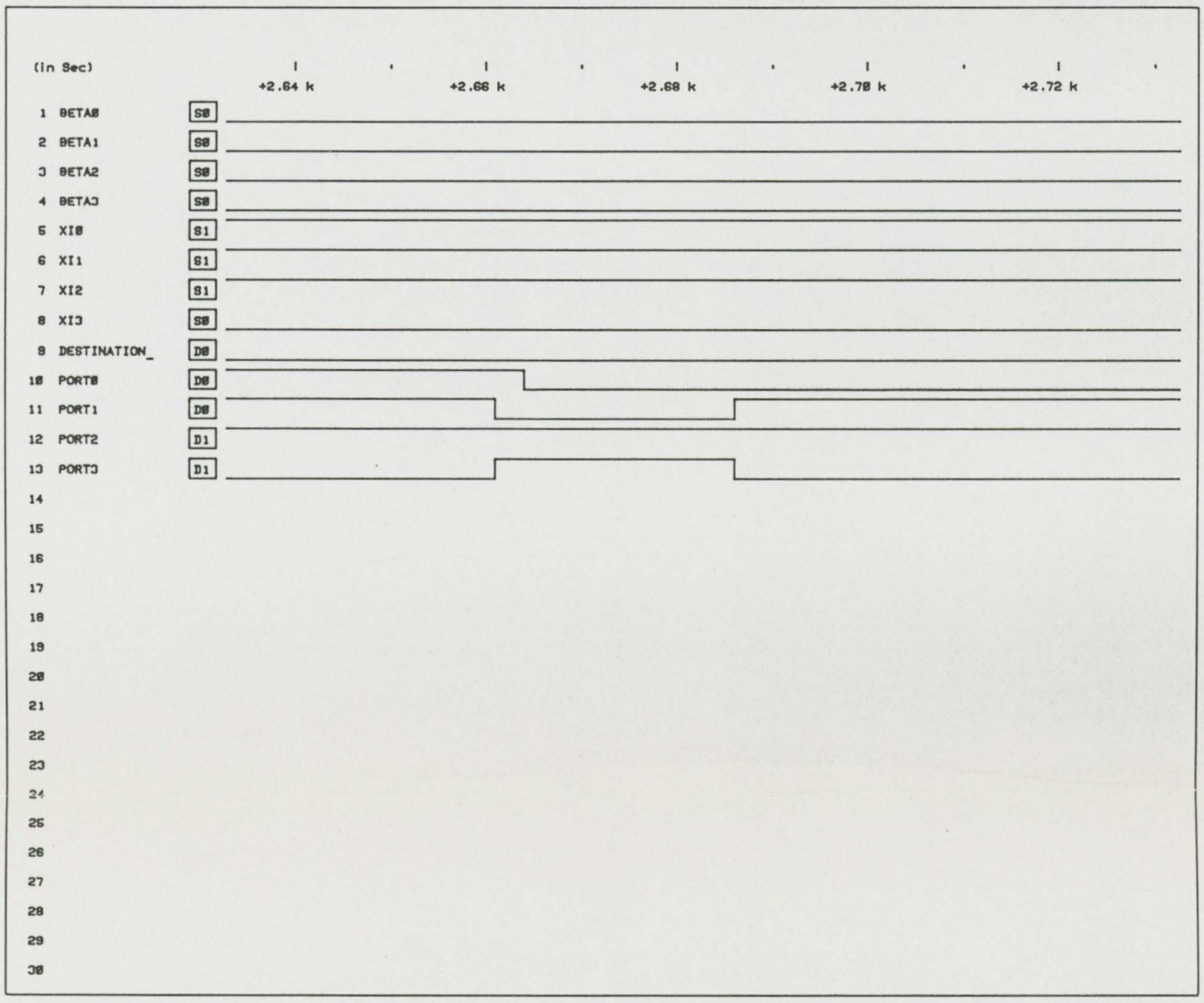


Figure 4.9

TTL Simulation Results Expanded Around 2000ns.

the worst possible case timing, all further possible routing combinations will produce valid results in 190 nanoseconds or less.

A timing file for SILOS using 1.5 micron VLSI technology was constructed from timings given in [19]. For the medium-scale integration circuits 74LS85, 74LS157 and 74LS283, the internal small-scale integration designs[37] where used to build simulation models of the larger circuits. As before, timings where taken to be maximum. In this case, that meant an output load of 5 picofarads per logic gate. At an average of 0.8 picofarad load per input, this meant that each gate is simulated as if it had a fan out of 6. This is considered to be worst case since most fan outs in the circuit are far less than this and none are more. Figures 4.10 and 4.11 give the results of the 1.5 micron implementation of the NPG. The longest timing signal is 100 nanoseconds given by the inputs in Table 4.5 and seen in figure 4.10 at 2500 nanoseconds.

Table 4.5 Input Signals for Figure 4.5 (1.5 micron)

	bit 0	bit 1	bit 2	bit 3
ξ_i	1	1	1	0
β_i	0	0	0	0
m_i	1	1	1	1
ρ_i	0	1	0	
PPORT_i	0	1	0	0

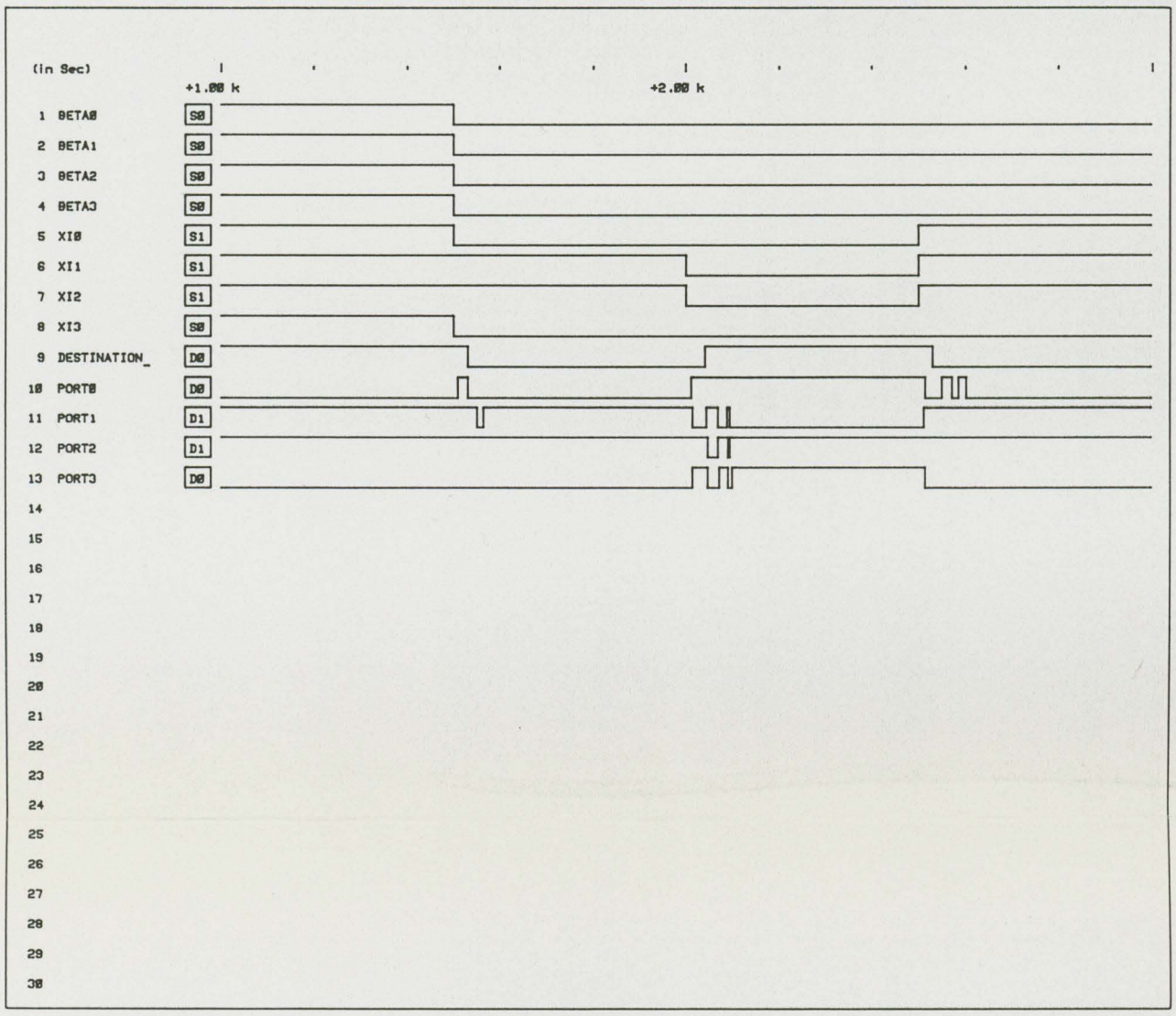


Figure 4.10

1.5 Micron Simulation Results.

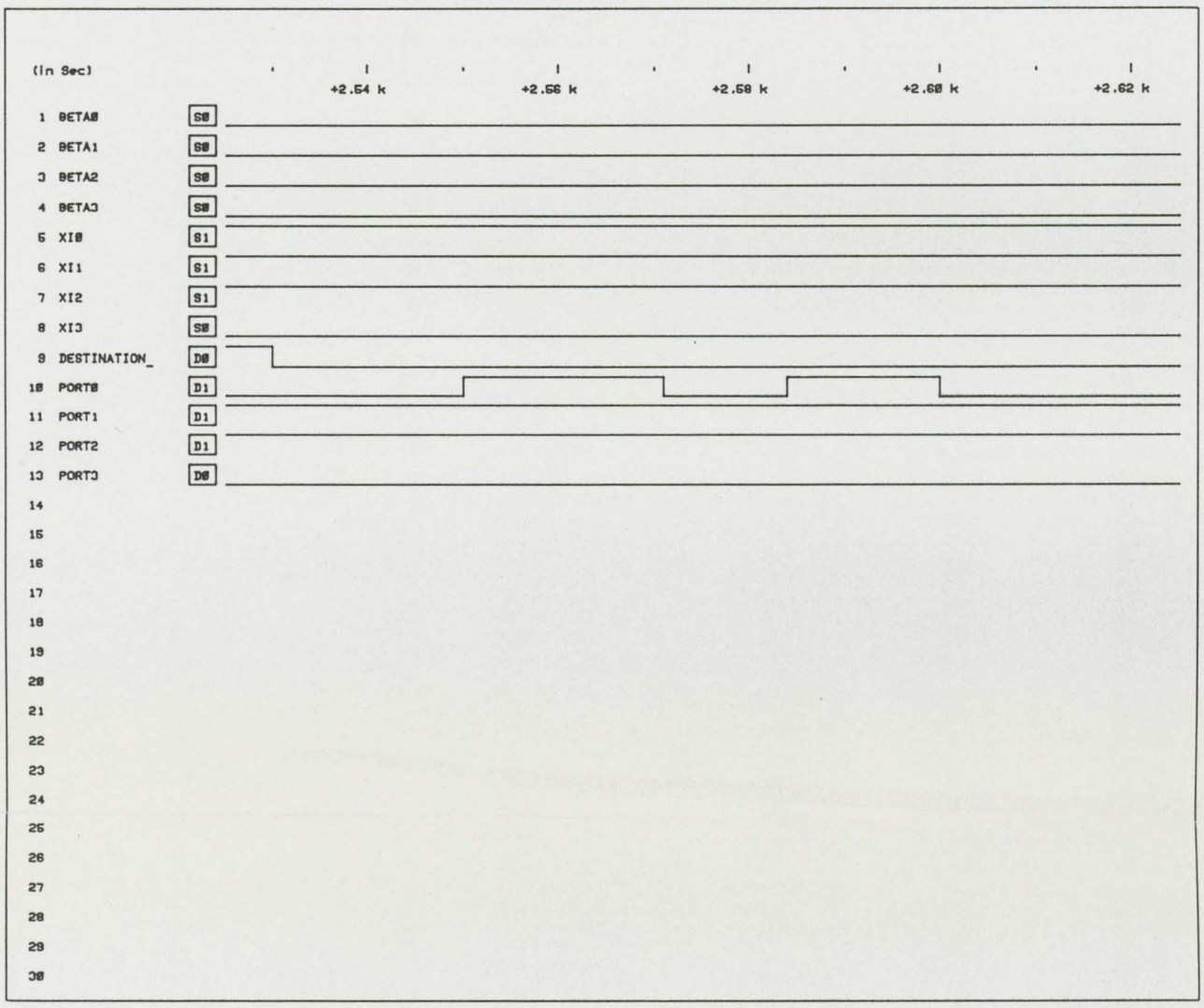


Figure 4.11

1.5 Micron Simulation Results Expanded.

Figure 4.11 shows the expansion of the resulting signals at 2600 nanoseconds. As can be seen, stable results are obtained in exactly 100 nanoseconds. Since this is the worst case (critical path) signal, this figure is an upper bound on calculation time.

As can be seen from the timing signals in figure 4.7, the signal named Destination_Reached, which indicates if the message has reached its destination port in this dimension, settles at approximately 90 nanoseconds. For 1.5 micron technology, this signal settles in 40 nanoseconds. Each dimension forwards this signal to the Port Validator where, by logical ORing all such signals, it can be determined if routing is necessary or if the destination has been reached. Therefore, such a decision can be made in 40 nanoseconds plus the time for logical ORing of the signals (≈5 nanoseconds in 1.5 micron technology). This would allow for a higher throughput of the routing circuitry if this signal where allowed to preempt the generation of offered ports (since the outcome is known already).

Figure 4.13 is a photograph of a functioning Next Port Generator implemented in 74LS logic. Experiments have shown that worst case routing times are approximately 125 nanoseconds. These results agree with those of the hardware simulator (190 nanoseconds worst case). The difference in times can be attributed to the fact that the simulator uses worst case delays through each gate, not typical delays as seen in the actual imp[imentation.

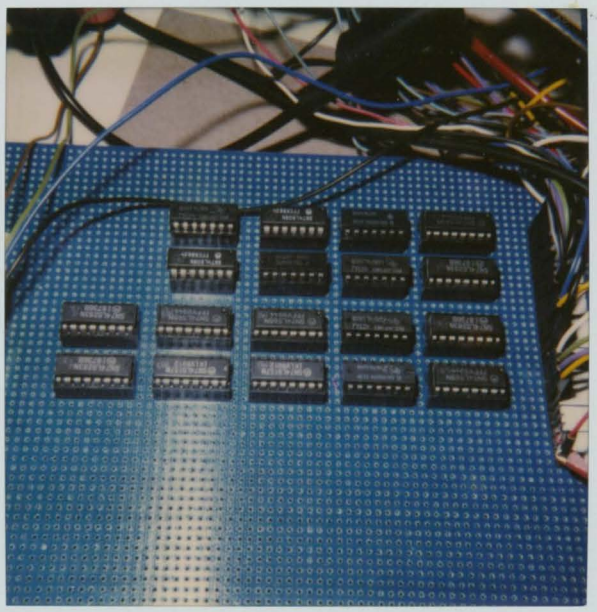


Figure 4.13 Photograph of Next Port Generator in 74LS Series Logic.

4.3.3 Router With Alternate Port Output

It is interesting to note that if m_i is even and $2\rho_i < m_i$[2], then if a message at node γ_i is bound for node $(\gamma_i \pm m_i/2) \bmod m_i$, that is, directly opposite the cycle, there are two possible minimal distance routes; one clockwise, the other counterclockwise (hence the ± sign). To further increase the number of optional ports to which a message can route, it is desirable to offer an alternate port within this dimension when such a situation arises. This can be simply accomplished by using the two EQUAL signals from the minimum selector units as control logic for a multiplexor which specifies an alternate port, as shown in figure 4.12. Since this situation calls for a maximum leap in either direction, the logical port offerings will be ρ_i and $2\rho_i$ since the offset is itself ρ_i.

4.3.4 Generating the PPORT Signals

The signal PPORT_i, indicating the number of preceding ports to dimension i, can be generated from information of previous dimensions. Since this is done only once, during configuration, delays in this circuit do not affect the execution time of the Next Port Generator.

---

[2]Note that the case where $2\rho_i = m_i$ was covered in section 4.3.1.

REVISIONS						
EFF	AUTHORITY	ZONE	LTR	DESCRIPTION	DATE	APPROVED

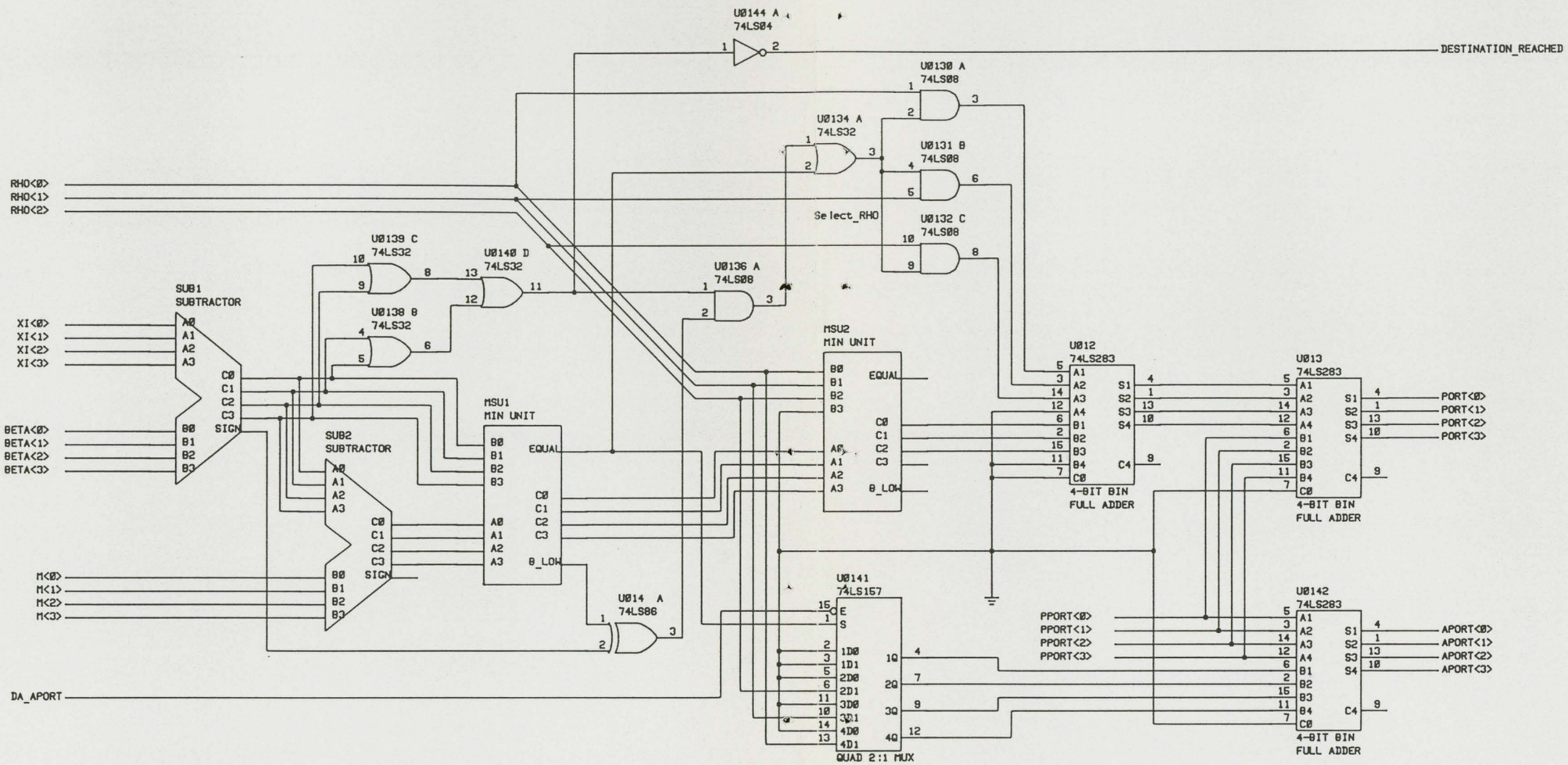


Figure 4.12 Next Port Generator with Alternate Port Output

CONTRACT Hypercycle Router		University of Victoria
DR	/ /	Next Port Generator
CHK	/ /	
ENG	/ /	
APPD	/ /	SIZE D | FSCM NO. - | DWG NO. -
NEXT HIGHER ASSY. -		
15:29:24 26-Jun-90		FIG 4 11 | SHEET 1 OF 1

Chapter 5. Conclusions and Discussion

The development of new graph structures for concurrent computer interconnection networks has raised the issue of the relative performance of these graphs to one another. The binary n-dimensional cube has emerged as a popular interconnection network due to its performance capabilities, especially in large systems. These networks do, however, have major drawbacks in the areas of expandability, fault tolerance and topological flexibility. Furthermore, the routing algorithms usually employed on such networks do not fully utilize the capacity of the networks since they will wait for and use only a single minimal distance path from a source node to a destination node while other such paths may exist.

Hypercycles, a new class of multidimensional graphs for concurrent computer interconnection networks which are essentially generalizations of the binary n-dimensional cube, show promise in overcoming some of the drawbacks of binary n-cubes.

Although these graphs are not the densest possible, they are attractive, because of their simple routing. Similarly to the n-cube, the destination address is used to sequentially route a message through intermediate nodes as outlined in chapter 2. Also, since the node addresses are represented in a mixed radix as a sequence of r-digits, each one of these digits is processed independently and in parallel with the remaining digits. Thus the hardware involved in the routing can be made fast (because of the parallelism) and simple (since each module need only handle arithmetic **mod** m_i, as compared to arithmetic **mod** $m_1m_2...m_r$ needed when all the address digits are necessary as is the case with such networks as the chordal rings [9], or the cube connected cycles [5]). Furthermore, Hypercycles are generalizations of some well known graphs such as the binary n-cube, 2- and 3-dimensional meshes, and rings, which are included as special cases so application programs written for such topologies may be directly applied to Hypercycles.

A computer-based simulator capable of simulating any Hypercycle network was constructed to validate the assumption that the backtrack-to-the-origin-and-retry routing employed in Hypercycles does indeed utilize the network more efficiently than e-cube routing. This simulator is graphical based and can be easily modified for alternate routing strategies and to simulate faulty systems. In order to verify that backtrack-to-the-origin-and-retry routing is indeed superior to e-cube routing in Hypercube-based networks, the two routing strategies where implemented on binary 4-dimensional cubes

in the simulator. The results of simulating these networks do show that backtrack-to-the-origin-and-retry routing is superior to e-cube routing, particularly in situations of moderate load. The delay characteristics of backtrack-to-the-origin are, at times, an order of magnitude better than that of e-cube and are never worse. The throughput characteristics show that, under very heavy loads, BTOR routing cannot route messages of long distances, and so falls below the performance of e-cube routing. However, the delay performance at these load levels is asymptotic to infinity so such load levels should be considered as overloading the particular network and, therefore, never reached. BTOR has the further advantage that it can run on any Hypercycle-based interconnection network. It is not limited to Hypercube networks like e-cube routing is.

A circuit design for the Next Port Generator subsystem of a Hypercycle router suitable for VLSI implementation has been offered. It has been shown that such a design can offer ports in 100 nanoseconds or less for 1.5 micron technology using the routing algorithm of chapter 2. This is comparable to the speed of most wired communication links which transfer information at a rate of approximately 1 bit per 100 nanoseconds (even an optical fibre link capable of transmiting at 100 million bits per second transmits only 10 information bits in 100 nanoseconds). Thus we can consider the process of routing as being a non-significant part of the overall message transmission time, especially as the size of the message grows (since routing time is independent of the message size).

5.1 Discussion of Further Work

The work presented in this thesis represents the early stages of interconnection network development. Since Hypercycles have only recently been introduced, there exist many areas open to further study and research.

In Hypercycles configured as binary n-cubes, it may be possible to alleviate the problems of throughput encountered in BTOR routing during periods of heavy load by employing an e-cube style routing to permit messages with long routes to complete their initial attempts.

It would also be interesting to implement a routing method that is a cross between Hyperswitch and backtrack-to-the-origin-and-retry. Recall that in Hyperswitch routing, the message backtracks only to the previous node when it encounters a block. It is certainly possible for a message to relinquish any amount of its partial path as opposed to all (BTOR) or one (Hyperswitch). This would allow a greater number of paths to choose from than by just releasing a single segment. Under heavy loads, it owuld be of interest to see if this sort of routing could be more effective for packets of longer distances than BTOR since the message need not free up its entire "hard won" partial path.

Such a scheme may be implemented so as the amount of backtrack as well as the next channel to be chosen are completely random

quantitites. In a random implementation, no state information needs to be carried with the message.

One of the advantages of Hypercycles is that a minimal-distance path can be computed analytically. While this is optimal under most conditions, a non-minimal length routing algorithm should also be considered. Although these are not optimal paths, and hence not as efficient in using the network, this type of routing would permit nodes to route messages around faulty links which previously constituted the only path from source to destination, as in nearest neighbour pairs. It is possible to use the Hypercycle simulator developed here to simulate networks under various conditions of faults. This would test the fault tolerance of different Hypercycle graphs and different routing algorithms (if they where implemented). The removal of links or entire nodes from the graphs is all that is required to simulate the faults.

An area of interest which has not been touched on in this thesis but does deserve further attention is that of broadcasting in Hypercycles. It is often desireable to send the same message to every node (or at least a grouping of nodes) in the network. Such messages include operating system messages, network configuration information, etc.. An efficient and unobtrusive method of broadcasting is desirable in any interconnection network, Hypercycles included.

The most obvious area of further work is in the development of the remainder of the Hypercycle router. The variable-range Random

Number Generator and the Port Counter in the Port Selector subsystem presents some interesting problems.

Bibliography

1. Berge C., *Principles of Combinatorics*, Academic Press, New York and London, 1971.

2. Bhuyan, L. N., and D. P. Agrawal, "Design and Performance of Generalized Interconnection Networks", *IEEE Trans. Comput.*,Vol. C-32, No. 12, pp. 1081-1090, Dec. 1983.

3. Bhuyan, L. N., and D. P. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network", *IEEE Trans. Computers.*, Vol. C-33, No. 4. pp. 323-333, Apr. 1984.

4. A. Broder, D. Dolev, M. Fischer, B. Simons "Efficient Fault Tolerant Routings in Networks" *Proceedings of the 16th Annual ACM Symposium on the Theory of Computing*, pp. 536-541, May 1984.

5. Carlsson, G. E. , J. E. Cruthirds, H. B. Sexton, and C. G. Wright, "Interconnection Networks Based on a Generalization of Cube-Connected Cycles", *IEEE Trans. Comput.*, Vol. C-34, No. 8, pp. 769-772, Aug. 1985.

6. E. Chow, H. Madan, J. Peterson, "A Real-Time Adaptive Message Routing Network for the Hypercube Computer", *Proceedings of the Real-Time Systems Symposium*, pp. 88-96, San Jose CA., Dec. 1987.

7. Dally, W.J., and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks", *IEEE Trans. Comput.*, Vol. C-36, No. 5, pp. 547-553, May 1987.

8. N. Dimopoulos, D. Radvan, K. F. Li, "Backtrack to the Origin and Retry Routing for Hypercycle-Based Interconnection Networks", *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 173-177, Victoria, B.C., Canada, June 1989.

9. Imase, M., T. Soneoka, and K. Okada, "Connectivity of Regular Directed Graphs with Small Diameters", *IEEE Trans. Comput.*, Vol. C-34, No. 3, pp. 267-273, Mar. 1985.

10. Peterson, J.C., J. O. Tuazon, D. Lieberman, M. Pniel, "The MARK III Hypercube -Ensemble Concurrent Computer", *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 71-73, Aug. 20-23 1985.

11. Rasmussen, R. D., G. S. Bolotin, N. J. Dimopoulos, B. F. Lewis, and R. M. Manning, "Advanced General Purpose Multicomputer for Space Applications", *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 54-57. Aug. 1987.

12. Rasmussen, R. D., N. J. Dimopoulos, G. S. Bolotin, B. F. Lewis, and R. M. Manning, "MAX: Advanced General Purpose Real-Time Multicomputer for Space Applications", *Proceedings of the IEEE Real Time Systems Symposium*, pp. 70-78, San Jose, CA., Dec. 1987.

13. G. Sabidussi, "Graph Multiplication", *Math. Zeitschr.*, Vol. 72, pp. 446-457, 1960.

14. Seitz, C. L., "The cosmic cube", *CACM*, Vol. 28, pp. 22-33, Jan. 1985.

15. Tuazon, J. O., J. C. Peterson, M. Pniel, and D. Lieberman, "Caltech/JPL MARK II Hypercube Concurrent Processor",*Proceedings of the 1985 International Conference on Parallel Processing*, pp. 666-673, Aug. 20-23, 1985.

16. Waltz, D. L., "Applications of the Connection Machine", *IEEE Computer*, pp. 85-97, January 1987.

17. L. D. Wittie, "Communication Structures for Large Networks of Microcomputers", *IEEE Trans. on Computers*, Vol. C-30, No. 4, pp.264-273, Apr. 1981.

18. Hussien G. Badr and Sunil Podar, "An Optimal Shortest-Path Routing Policy for Network Computers with Regular Mesh-Connected Topologies", *IEEE Trans. on Computers*, Vol. 38, No. 10, pp. 1362-1371, October 1989.

19. VLSI Technologies,*VSC100 Portable Library: VSC13533*, VLSI Technology, San Jose, CA., 1988.

20. Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, 1987.

21. Nikitas J. Dimopoulos, Don Radvan and Kin F. Li, "Backtrack-to-the-Origin-and-Retry Routing for Hypercycle-Based Interconnection Networks", *Canadian Conference on Electrical and Computer Engineering*, Montreal, P.Q., pp. 967-971, September 1989.

22. Motorola, *MC68020 User's Manual*, Prentice-Hall, London, 1984.

23. Mike Johnson, "System Considerations in the Design of the Am29000", *IEEE Micro*, August 1987, pp. 28-41.

24. Les Kohn and Neal Margulis, "Introducing the Intel i860 64-Bit Microprocessor", *IEEE Micro*, Vol. 9, No. 4, August 1989, pp. 15-30.

25. Special Issue on the IBM 3090, *IBM Journal*, Vol. 25, No. 1, 1986.

26. Charles Melear, "The Design of the 88000 RISC Family", *IEEE Micro*, Vol. 9, No. 2, April 1989, pp. 26-38.

27. Anujan Varma and C. S. Raghavenrda, "Fault-Tolerant Routing in Multistage Interconnection Networks", *IEEE Trans. on Comp.*, Vol. 38, No. 3, March 1989, pp. 385-393.

28. Seyed H. Hosseini, "On Fault-Tolerant Structure, Distributed Fault-Diagnosis, Reconfiguration, and Recovery of the Array Processor", *IEEE Trans. on Comp.*, Vol. 38, No. 7, July 1989, pp. 932-942.

29. Tom Diede, Carl F. Hagenmaier, Glen S. Miranker, Jonathan J. Rubenstein and Wiliam S. Worley, Jr., "The Titan Graphics Supercomputer Architecture", *IEEE Computer*, Vol. 21, No. 9, September 1988, pp. 13-30.

30. Kai Hwang and Faye A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

31. William C. Athas and Charles L. Seitz, "Multicomputers: Message-Passing Concurrent Computers", *IEEE Computer*, Vol. 21, No. 8, December 1981, pp. 12-26.

32. Richard M. Stein, "T800 and Counting", *Byte*, Vol. 13, No. 12, November 1988, pp. 287-296.

33. *ModL Language Reference*, Imagine That!, Inc., San Jose, 1988.

34. Ross Freeman, "User-Programmable Gate Arrays", *IEEE Spectrum*, Vol. 25, No. 13, December 1988, pp. 32-35.

35. *CASE User's Manual*, CASE Technology, Inc., Mountain View, Calif., 1988.

36. *SILOS II User's Manual*, Simucad, Inc., Palo Alto, 1988.

37. *The TTL Data Book: Volume 2*, Texas Instruments, 1985.

38. Daniel A. Reed and Dirk C. Grunwald, "The Performance of Multicomputer Interconnection Networks", *IEEE Computer*, Vol. 20, No. 6, July 1987, pp. 63-73.

Appendix A

Simulator Code

```
** HYPERCYCLE NODE
**          m0 = 3, m1 = 3, rho0 = 1, rho1 = 1
**
** NOTE: Global0 is delay
**       Global1 is number of packets sent
**       Global2 is number of packets received
**           Global3 is probability of packet generation (in %)
**           Global4 is the file number for the log file
integer packet[5], inpkt[], Port1Pkt[], Port2Pkt[];
integer Port3Pkt[], Port4Pkt[];
**
**-------------------------------------------------------
** The SEND_Q holds packets generated by the station itself.
** The LINK_Q holds packets that arrived through an IN port
** A packet is taken out from the SEND_Q only if the
** LINK_Q is empty. If no port is found, then it returns
** to the back of the SEND_Q to be rescheduled.
**
integer SEND_Q[][11], LINK_Q[][11], PROCESS_Q[][11];
real port_Q[];
**
**-------------------------------------------------------
** For each port, one defines a link and a state flag.
** The state flag is set if the port is state, i.e. if
** a packet has gone out of the Out part of the port.
** The link, is a tag that links to the previous port
** where the packet came from. e.g. if a packet came from
** port 2 and went out from port 4, link[4]=2. Thus if
** a responce came back (e.g. path not established)
** then one knows how to route the response so that it
** will reach the origin. A link being equal to zero
** indicates that the packet was taken from the
** SEND_Q (i.e. the packet originated at this station.
**
integer link[5];
integer state[5];
integer portNo;
integer A[2];                                  ** node address
integer RHO[2];                                ** connectivity vector
integer M[2];                                  ** degree vector

integer pkts_rec, pkts_sent, total_delay, average_delay, collides;

**
** CONSTANTS
**
constant processTime is 10;
constant nodes is 12;
**
** packet constants
**
constant TIMESENT is 0;
constant PORT is 1;
constant TIMELEFT is 2;
constant D0 is 3;
```

```
constant D1 is 4;

** types of queues
constant Q_LINK_Q is 1;
constant Q_PROCESS_Q is 3;
constant Q_SEND_Q is 4;

** types of port states
constant FREE is 0;
constant BUSY_NEW is 1;
constant BUSY_UP is 2;
constant BUSY_DOWN is 3;
constant BUSY_START is 4;
constant COLLISION is 5;
constant SYNC is 6;
constant COLLAPSE is 7;
**
**
**-------------------------------------------------

**************
** USER FUNCTIONS DEFINITIONS
**

**   Function CreatePacket produces a packet with the correct
distribution.
**    It returns zero if no packet is generated.
Integer CreatePacket()
{
If (RandomReal() > Global3)
       Return(FALSE);  ** It is not yet time to produce a packet
Else  ** Generate a packet
{
       do
       {
            packet[D0] = Random(3);
            packet[D1] = Random(4);
       }
       while ((packet[D0] == A[0]) && (packet[D1] == A[1]));

       packet[TIMESENT] = CURRENTSTEP;
       packet[TIMELEFT] = int(Global6);
     Global1++;
       pkts_sent++;
       return(TRUE);
      }
}

***** Route
** Routes the packet to the right port.
Integer Route()
{
Integer p, i, win_port, winner, el_port;
Integer aa, b, c, d, e, f, g;
```

```
For i = 0 to 1
{
      aa = A[i] - packet[D0+i];
      b = M[i] - integerabs(aa);
      d = (integerabs(aa) >= b);
      g = (aa < 0);
      if (d) c = b; else c = integerabs(aa);
      if (1 < c) e = 1; else e = c;
      if ((d && !g) || (!d && g)) f = e + 1; else f = e;
      if (f == 0)
            p = 0;
      else
            p = f + (i * 2); ** CHANGE THIS IF RHO CHANGES, OR MORE
DIMENSIONS

      if (p > 4)
      {
            Beep();
            UserError("Attempt to route to unknown port "+p+" . ");
            Abort;
      }
      else if (p > 0)
            if (state[p] == FREE)
                  PutFront(port_Q, p);
}

el_port = QueLength(port_Q);
if (el_port == 0)
      return(0);
winner = Random(el_port) + 1;
for i = 1 to el_port
{
      p = GetFront(port_Q);
      if (i == winner)
            win_port = p;
}
return(win_port);

}

***** QPkt
** Queues the local global dynamic array "packet" into the
** specified queue (actually an index to a q)
** packet is destroyed after it is queued.
PROCEDURE QPkt(Integer q)
{
      Integer found, slot, copy, size, temp;

      found = FALSE;
      slot = 0;
      switch(q)
      {
            case Q_LINK_Q:
                  size = GetDimension(LINK_Q);
                  while ((NOT found) AND (slot < size))
```

```
            {
                  if (LINK_Q[slot][10] == FALSE)     ** free slot
                  {
                        found = TRUE;
                        LINK_Q[slot][10] = TRUE;     ** occupy slot
                        for (copy = 0; copy <= 4; copy++)
                        {
                              temp = packet[copy];
                              LINK_Q[slot][copy] = temp;
                        }
                  }
                  slot++;
            }
            if (NOT found)        ** Allocate another slot
            {
                  MakeArray(LINK_Q, slot+1);
                  LINK_Q[slot][10] = TRUE;     ** occupy slot
                  for (copy = 0; copy <= 4; copy++)
                  {
                        temp = packet[copy];
                        LINK_Q[slot][copy] = temp;
                  }
            }
            BREAK;
      case Q_PROCESS_Q:
            size = GetDimension(PROCESS_Q);
            while ((NOT found) && (slot < size))
            {
                  if (PROCESS_Q[slot][10] == FALSE)     ** free
slot
                  {
                        found = TRUE;
                        PROCESS_Q[slot][10] = TRUE;     ** occupy
slot
                        for copy = 0 to 4
                        {
                              temp = packet[copy];
                              PROCESS_Q[slot][copy] = temp;
                        }
                  }
                  slot++;
            }
            if (NOT found)        ** Allocate another slot
            {
                  MakeArray(PROCESS_Q, slot+1);
                  PROCESS_Q[slot][10] = TRUE;     ** occupy slot
                  for copy = 0 to 4
                  {
                        temp = packet[copy];
                        PROCESS_Q[slot][copy] = temp;
                  }
            }
            BREAK;
      case Q_SEND_Q:
```

```
                packet[PORT] = 0;                          ** pkt comes
from me
                size = GetDimension(SEND_Q);
                while ((NOT found) AND (slot < size))
                {
                      if (SEND_Q[slot][10] == FALSE)    ** free slot
                      {
                            found = TRUE;
                            SEND_Q[slot][10] = TRUE;    ** occupy slot
                            for (copy = 0; copy <= 4; copy++)
                            {
                                  temp = packet[copy];
                                  SEND_Q[slot][copy] = temp;
                            }
                      }
                      slot++;
                }
                if (NOT found)     ** Allocate another slot
                {
                      MakeArray(SEND_Q, slot+1);
                      SEND_Q[slot][10] = TRUE;      ** occupy slot
                      for copy = 0 to 4
                      {
                            temp = packet[copy];
                            SEND_Q[slot][copy] = temp;
                      }
                }
                BREAK;
      } ** SWITCH
}

****** UnQPkt
Integer UnQPkt(Integer q, Integer slot)
{
      Integer copy, temp;

      switch(q)
      {
            case Q_LINK_Q:
                  if (LINK_Q[slot][10] == FALSE) return (FALSE);
                  LINK_Q[slot][10] = FALSE;
                  for copy = 0 to 4
                  {
                        temp = LINK_Q[slot][copy];
                        packet[copy] = temp;
                  }
                  BREAK;
            case Q_PROCESS_Q:
                  if (PROCESS_Q[slot][10] == FALSE) return (FALSE);
                  for copy = 0 to 4
                  {
                        temp = PROCESS_Q[slot][copy];
                        packet[copy] = temp;
                  }
                  PROCESS_Q[slot][10] = FALSE;   ** free up slot
```

```
                BREAK;
          case Q_SEND_Q:
                if (SEND_Q[slot][10] == FALSE) return (FALSE);
                for copy = 0 to 4
                {
                      temp = SEND_Q[slot][copy];
                      packet[copy] = temp;
                }
                SEND_Q[slot][10] = FALSE;     ** free up slot
                BREAK;
      } ** SWITCH
      return(TRUE);
}

***** ProcessReturnedPkt
** Processes packets returned to origin
Procedure ProcessReturnedPkt()
{
      Integer pdelay;

      if (packet[TIMELEFT] == 0)                ** if packet was
succesful
      {
            pdelay = CURRENTSTEP - packet[TIMESENT] - Global6;
            total_delay = total_delay + pdelay;
            pkts_rec++;
            Global2++;                                  ** One more packet
received
            average_delay = total_delay/pkts_rec;
            Global0 = (Global0*(Global2-1) + pdelay)/Global2;
      }
      else
            QPkt(Q_SEND_Q);
}

***** Arrived
** checks to see if packet has reached destination
Integer Arrived()
{
      If (packet[D0] != A[0]) return(FALSE);
      If (packet[D1] != A[1]) return(FALSE);

      return(TRUE);
}

Procedure CopyPkt(integer pkt)
{
      integer i, temp;

      switch (pkt)
      {
            case 1:
                  for i = 0 to 4
                  {
                        temp = packet[i];
```

```
                        Port1Pkt[i] = temp;
                }
                break;
        case 2:
                for i = 0 to 4
                {
                        temp = packet[i];
                        Port2Pkt[i] = temp;
                }
                break;
        case 3:
                for i = 0 to 4
                {
                        temp = packet[i];
                        Port3Pkt[i] = temp;
                }
                break;
        case 4:
                for i = 0 to 4
                {
                        temp = packet[i];
                        Port4Pkt[i] = temp;
                }
                break;
    }
}

***** SyncUpCollapse
** Acknowledges an upward collapse synchronisation
Procedure SyncUpCollapse(integer p)
{
    state[p] = COLLAPSE;

    switch (p)
    {
        case 1:
                Port1Out = Port1In;
                break;
        case 2:
                Port2Out = Port2In;
                break;
        case 3:
                Port3Out = Port3In;
                break;
        case 4:
                Port4Out = Port4In;
                break;
    }
}

***** SyncDownCollapse
** Begins a downward collapse synchronisation
Procedure SyncDownCollapse(integer p)
{
    state[p] = SYNC;
```

```
        switch (p)
        {
                case 0:
                        QPkt(Q_SEND_Q);
                        break;
                case 1:
                        CopyPkt(1);
                        Port1Out = PassArray(Port1Pkt);
                        break;
                case 2:
                        CopyPkt(2);
                        Port2Out = PassArray(Port2Pkt);
                        break;
                case 3:
                        CopyPkt(3);
                        Port3Out = PassArray(Port3Pkt);
                        break;
                case 4:
                        CopyPkt(4);
                        Port4Out = PassArray(Port4Pkt);
                        break;
        }
}

***** EstablishPath
** Connects two ports (including port 0) together
Procedure EstablishPath(Integer out_port)
{
        integer p;

        state[out_port] = BUSY_NEW;
        p = packet[PORT];
        link[out_port] = p;

        switch (out_port)
        {
                case 1:
                        CopyPkt(1);
                        Port1Out = PassArray(Port1Pkt);
                        break;
                case 2:
                        CopyPkt(2);
                        Port2Out = PassArray(Port2Pkt);
                        break;
                case 3:
                        CopyPkt(3);
                        Port3Out = PassArray(Port3Pkt);
                        break;
                case 4:
                        CopyPkt(4);
                        Port4Out = PassArray(Port4Pkt);
                        break;
        }
}
```

```
Procedure GetInPkt(integer p)
{
      integer i, temp;

      if (state[p] == FREE)
            state[p] = BUSY_DOWN;
      for i = 0 to 4
      {
            temp = inpkt[i];
            packet[i] = temp;
      }
      packet[PORT] = p;
      QPkt(Q_LINK_Q);
}


***************
** This message occurs for each step in the simulation.
on simulate
{
      Integer nextPort, slot, p, size, time, temp;

if (CreatePacket())       ** Produce a packet
      QPkt(Q_SEND_Q);

** Clear all output ports
Port1Out = 0;
Port2Out = 0;
Port3Out = 0;
Port4Out = 0;

** Check the input ports for newly arriving  packets
** and queue any into the LINK_Q

** Port #1
If (GetPassedArray(Port1In, inpkt)) ** If there is an incoming packet
      GetInPkt(1);

** Port #2
If (GetPassedArray(Port2In, inpkt)) ** If there is an incoming packet
      GetInPkt(2);

** Port #3
If (GetPassedArray(Port3In, inpkt)) ** If there is an incoming packet
      GetInPkt(3);

** Port #4
If (GetPassedArray(Port4In, inpkt)) ** If there is an incoming packet
      GetInPkt(4);

** Process incoming packets according to the previous port state
size = GetDimension(LINK_Q) - 1;
for slot = 0 to size
      if (UnQPkt(Q_LINK_Q, slot))
```

```
      {
            p = packet[PORT];
            switch (state[p])
            {
                  case BUSY_UP:
      ** collapse:      expected
                        SyncUpCollapse(p);
      ** sync collapse upward
                        if (link[p] == 0)                              **
collapse to me
                              ProcessReturnedPkt();
                        else
      ** collapse thru me
                        {
                              p = link[p];
                              packet[PORT] = p;                        **
link pkt backward
                              SyncDownCollapse(p);                     **
start collapse downward
                        }
                        break;
                  case SYNC:
      ** sync:           expected
                        state[p] = FREE;                               **
clear state
                        break;
                  case BUSY_START:                                     **
COLLISION!: unexpected
                        collides++;
                        SyncUpCollapse(p);                             **
collapse back
                  **
                  **  NOTE: We should be receiving the business end of a
SyncDownCollapse
                  **  from port p in the next step.
                  **
                        state[p] = COLLISION;                          **
don't expect sync back
                        break;
                  case BUSY_DOWN:
      ** new message:   unexpected
                        if (Arrived())
      ** route to me
                              QPkt(Q_PROCESS_Q);
                        else
      ** route thru me
                        {
                              nextPort = Route();
                              if (nextPort == 0)
      ** if no available port
                                    SyncDownCollapse(p);               **
start collapse downward
                              else
                                    EstablishPath(nextPort);           **
connect ports
```

```
                        }
                        break;
                  case COLLISION:
                        if ((((CURRENTSTEP MOD 2) == 1) && (p == 1)) ||
                           (((CURRENTSTEP MOD 2) == 0) && (p == 2)) ||
                           (((CURRENTSTEP MOD 2) == 1) && (p == 3)) ||
                           (((CURRENTSTEP MOD 2) == 0) && (p == 4)))
                        {
            ** force packet back
                              SyncUpCollapse(p);
                              state[p] = BUSY_NEW;
                        }
                        else
                        {
                              state[p] = COLLAPSE;
                              p = link[p];
                              if (p == 0)
                                    QPkt(Q_SEND_Q);
                              else
                                    SyncDownCollapse(p);
                        }
                        break;
                  default:
                        Beep();
                        UserError(" Bad state");
                        Abort;
                        break;
            }
      }

** Update states
for p = 1 to 4
      switch (state[p])
      {
            case BUSY_START:
                  state[p] = BUSY_UP;                    ** passed
collision period
                  break;
      }

** send out messages
size = GetDimension(SEND_Q) - 1;
for slot = 0 to size
      if (UnQPkt(Q_SEND_Q, slot))
      {
            nextPort = Route();
            if (nextPort == 0)                           ** if no available
port
                  QPkt(Q_SEND_Q);
            else
                  EstablishPath(nextPort);       ** connect ports
      }

** continue processing node packets
size = GetDimension(PROCESS_Q) - 1;
```

```
for slot = 0 to size
      if (UnQPkt(Q_PROCESS_Q, slot))
      {
            packet[TIMELEFT]--;
            IF (packet[TIMELEFT] == 0)
                  SyncDownCollapse(packet[PORT]);          ** send
packet back
            ELSE
                  QPkt(Q_PROCESS_Q);
      }  **End FOR .. IF

** Update states
for p = 1 to 4
      switch (state[p])
      {
            case BUSY_NEW:
                  state[p] = BUSY_START;        ** start collision
period
                  break;
            case COLLAPSE:
                  state[p] = FREE;              ** free up state for
next step
                  break;
      }

**  DONE!!!!!

}


** If the dialog data is inconsistent for simulation, abort.
on checkdata
{

}


** Initialize any simulation variables.
on initsim
{
      integer p, i;

      for p = 0 to 4
            state[p] = FREE;
      pkts_sent = 0;
      total_delay = 0;
      pkts_rec = 0;
      average_delay = 0;
      collides = 0;
      QueInit(port_Q);
      M[0] = 3;
      M[1] = 4;
      RHO[0] = 1;
      RHO[1] = 1;
```

```
      A[0] = int(a1);
      A[1] = int(a2);
      MakeArray(LINK_Q, 0);
      MakeArray(PROCESS_Q, 0);
      MakeArray(SEND_Q, 0);
      MakeArray(Port1Pkt, 5);
      MakeArray(Port2Pkt, 5);
      MakeArray(Port3Pkt, 5);
      MakeArray(Port4Pkt, 5);
}


** User clicked the dialog HELP button.
on help
{
showHelp();
}

on endsim
{
      FileWrite(Global4, a1, "", FALSE);
      FileWrite(Global4, a2, "", FALSE);
      FileWrite(Global4, "  | ", "", FALSE);
      FileWrite(Global4, pkts_sent, "", FALSE);
      FileWrite(Global4, pkts_rec, "", FALSE);
      FileWrite(Global4, average_delay, "", FALSE);
      FileWrite(Global4, collides, "", TRUE);
      global5++;
      if (global5 == nodes)
            FileClose(GLOBAL4);
}
```

```
** BINARY N-CUBE NODE (e-cube routing)
**
** NOTE: Global0 is delay
**       Global1 is number of packets sent
**       Global2 is number of packets received
**           Global3 is probability of packet generation (in %)
**           Global4 is the file number for the log file
integer packet[8], inpkt[], Port1Pkt[], Port2Pkt[];
integer Port3Pkt[], Port4Pkt[];
**
**-------------------------------------------------------
** The SEND_Q holds packets generated by the station itself.
** The LINK_Q holds packets that arrived through an IN port
** A packet is taken out from the SEND_Q only if the
** LINK_Q is empty. If no port is found, then it returns
** to the back of the SEND_Q to be rescheduled.
**
integer SEND_Q[][11], LINK_Q[4][11], PROCESS_Q[][11], HOLD_Q[][11];
integer IN_Q[4][11];
**
**-------------------------------------------------------
** For each port, one defines a link and a state flag.
** The state flag is set if the port is state, i.e. if
** a packet has gone out of the Out part of the port.
** The link, is a tag that links to the previous port
** where the packet came from. e.g. if a packet came from
** port 2 and went out from port 4, link[4]=2. Thus if
** a responce came back (e.g. path not established)
** then one knows how to route the response so that it
** will reach the origin. A link being equal to zero
** indicates that the packet was taken from the
** SEND_Q (i.e. the packet originated at this station.
**
integer link[5];
integer state[5];
integer portNo;
integer A[4];                                ** node address

integer  collides, SD[5], RD[5], DELAY_D[5];
real DLY[], REC[], SNT[];

**
** CONSTANTS
**
constant nodes is 16;
**
** packet constants
**
constant TIMESENT is 0;
constant PORT is 1;
constant TIMELEFT is 2;
constant D0 is 3;
constant D1 is 4;
constant D2 is 5;
constant D3 is 6;
```

```
constant BACKOFF is 7;

** types of queues
constant Q_LINK_Q is 1;
constant Q_PROCESS_Q is 3;
constant Q_HOLD_Q is 5;
constant Q_SEND_Q is 4;
constant Q_IN_Q is 2;

** types of port states
constant FREE is 0;
constant BUSY_NEW is 1;
constant BUSY_UP is 2;
constant BUSY_DOWN is 3;
constant BUSY_START is 4;
constant COLLISION is 5;
constant SYNC is 6;
constant COLLAPSE is 7;
**
**
**-------------------------------------------------

**************
** USER FUNCTIONS DEFINITIONS
**

***** Distance
** Returns the distance between me and packet destination
Integer Distance()
{
      integer i;
      integer dist;

      dist = 0;

      for (i=0;i<4;++i)
            dist += integerabs(A[i] - packet[D0+i]);

      return(dist);
}

**   Function CreatePacket produces a packet with the correct
distribution.
**    It returns zero if no packet is generated.
Integer CreatePacket()
{
      integer dist;

If (RandomReal() > Global3)
       Return(FALSE);  ** It is not yet time to produce a packet
Else  ** Generate a packet
{
       do
       {
            packet[D0] = Random(2);
```

```
            packet[D1] = Random(2);
            packet[D2] = Random(2);
            packet[D3] = Random(2);
        }
        while ((packet[D0] == A[0]) && (packet[D1] == A[1]) &&
                (packet[D2] == A[2]) && (packet[D3] == A[3]));

        dist = Distance();
        SD[dist]++;
        packet[TIMESENT] = CURRENTSTEP;
        packet[TIMELEFT] = int(Global6);
        return(TRUE);
       }
}

***** Route
** Routes the packet to the right port.
Integer Route()
{
Integer i, win_port;

win_port = 0;
i = 0;

while ((win_port == 0) && (i < 4))
{
      if ((packet[D0+i] - A[i]) != 0)
            if (state[i+1] == FREE)
                  win_port = i+1;
            else i = 4;
      i++;
}

return(win_port);
}

***** QPkt
** Queues the local global dynamic array "packet" into the
** specified queue (actually an index to a q)
** packet is destroyed after it is queued.
PROCEDURE QPkt(Integer q)
{
      Integer found, slot, copy, size, temp;

      found = FALSE;
      slot = 0;
      switch(q)
      {
            case Q_LINK_Q:
                  while (NOT found)
                  {
                        if (LINK_Q[slot][10] == FALSE)     ** free slot
                        {
                              found = TRUE;
                              LINK_Q[slot][10] = TRUE;     ** occupy slot
```

```
                        for (copy = 0; copy <= 7; copy++)
                        {
                              temp = packet[copy];
                              LINK_Q[slot][copy] = temp;
                        }
                  }
                  slot++;
            }
            BREAK;
      case Q_IN_Q:
            while (NOT found)
            {
                  if (IN_Q[slot][10] == FALSE)     ** free slot
                  {
                        found = TRUE;
                        IN_Q[slot][10] = TRUE;     ** occupy slot
                        for (copy = 0; copy <= 7; copy++)
                        {
                              temp = packet[copy];
                              IN_Q[slot][copy] = temp;
                        }
                  }
                  slot++;
            }
            BREAK;
      case Q_PROCESS_Q:
            size = GetDimension(PROCESS_Q);
            while (PROCESS_Q[slot][10] != FALSE)
                  slot++;
            PROCESS_Q[slot][10] = TRUE;     ** occupy slot
            for copy = 0 to 7
            {
                  temp = packet[copy];
                  PROCESS_Q[slot][copy] = temp;
            }

            if (slot == (size - 1))
            {
                  MakeArray(PROCESS_Q, size+1);
                  PROCESS_Q[size][10] = FALSE;
            }
            BREAK;
      case Q_SEND_Q:
            packet[PORT] = 0;                              ** pkt comes
from me
            size = GetDimension(SEND_Q);
            while (SEND_Q[slot][10] != FALSE)
                  slot++;
            SEND_Q[slot][10] = TRUE;     ** occupy slot
            for copy = 0 to 7
            {
                  temp = packet[copy];
                  SEND_Q[slot][copy] = temp;
            }
```

```
                if (slot == (size - 1))
                {
                        MakeArray(SEND_Q, size+1);
                        SEND_Q[size][10] = FALSE;
                }
                BREAK;
        case Q_HOLD_Q:
                size = GetDimension(HOLD_Q);
                while (HOLD_Q[slot][10] != FALSE)
                        slot++;
                HOLD_Q[slot][10] = TRUE;     ** occupy slot
                for copy = 0 to 7
                {
                        temp = packet[copy];
                        HOLD_Q[slot][copy] = temp;
                }

                if (slot == (size - 1))
                {
                        MakeArray(HOLD_Q, size+1);
                        HOLD_Q[size][10] = FALSE;
                }
                BREAK;
    } ** SWITCH
}

****** UnQPkt
Integer UnQPkt(Integer q, Integer slot)
{
    Integer copy, temp;

    switch(q)
    {
        case Q_LINK_Q:
                if (LINK_Q[slot][10] == FALSE) return (FALSE);
                LINK_Q[slot][10] = FALSE;
                for copy = 0 to 7
                {
                        temp = LINK_Q[slot][copy];
                        packet[copy] = temp;
                }
                BREAK;
        case Q_IN_Q:
                if (IN_Q[slot][10] == FALSE) return (FALSE);
                IN_Q[slot][10] = FALSE;
                for copy = 0 to 7
                {
                        temp = IN_Q[slot][copy];
                        packet[copy] = temp;
                }
                BREAK;
        case Q_PROCESS_Q:
                if (PROCESS_Q[slot][10] == FALSE) return (FALSE);
                for copy = 0 to 7
                {
```

```
                        temp = PROCESS_Q[slot][copy];
                        packet[copy] = temp;
                    }
                    PROCESS_Q[slot][10] = FALSE;    ** free up slot
                    BREAK;
              case Q_SEND_Q:
                    if (SEND_Q[slot][10] == FALSE) return (FALSE);
                    for copy = 0 to 7
                    {
                        temp = SEND_Q[slot][copy];
                        packet[copy] = temp;
                    }
                    SEND_Q[slot][10] = FALSE;     ** free up slot
                    BREAK;
              case Q_HOLD_Q:
                    if (HOLD_Q[slot][10] == FALSE) return (FALSE);
                    for copy = 0 to 7
                    {
                        temp = HOLD_Q[slot][copy];
                        packet[copy] = temp;
                    }
                    HOLD_Q[slot][10] = FALSE;     ** free up slot
                    BREAK;
        } ** SWITCH
        return(TRUE);
}

***** ProcessReturnedPkt
** Processes packets returned to origin
Procedure ProcessReturnedPkt()
{
        Integer pdelay, dist;

        pdelay = CURRENTSTEP - packet[TIMESENT] - Global6;
        dist = Distance();
        RD[dist]++;
        DELAY_D[dist] += pdelay;
}

***** Arrived
** checks to see if packet has reached destination
Integer Arrived()
{
        If ((packet[D0] != int(a1)) || (packet[D1] != int(a2)) ||
           (packet[D2] != int(a3)) || (packet[D3] != int(a4)))
              return(FALSE);

        else
              return(TRUE);
}

Procedure CopyPkt(integer pkt)
{
        integer i, temp;
```

```
        switch (pkt)
        {
                case 1:
                        for i = 0 to 7
                        {
                                temp = packet[i];
                                Port1Pkt[i] = temp;
                        }
                        break;
                case 2:
                        for i = 0 to 7
                        {
                                temp = packet[i];
                                Port2Pkt[i] = temp;
                        }
                        break;
                case 3:
                        for i = 0 to 7
                        {
                                temp = packet[i];
                                Port3Pkt[i] = temp;
                        }
                        break;
                case 4:
                        for i = 0 to 7
                        {
                                temp = packet[i];
                                Port4Pkt[i] = temp;
                        }
                        break;
        }
}

***** SyncUpCollapse
** Acknowledges an upward collapse synchronisation
Procedure SyncUpCollapse(integer p)
{
        state[p] = COLLAPSE;

        switch (p)
        {
                case 1:
                        Port1Out = Port1In;
                        break;
                case 2:
                        Port2Out = Port2In;
                        break;
                case 3:
                        Port3Out = Port3In;
                        break;
                case 4:
                        Port4Out = Port4In;
                        break;
        }
}
```

```
***** SyncDownCollapse
** Begins a downward collapse synchronisation
Procedure SyncDownCollapse(integer p)
{
      state[p] = SYNC;

      switch (p)
      {
            case 1:
                  CopyPkt(1);
                  Port1Out = PassArray(Port1Pkt);
                  break;
            case 2:
                  CopyPkt(2);
                  Port2Out = PassArray(Port2Pkt);
                  break;
            case 3:
                  CopyPkt(3);
                  Port3Out = PassArray(Port3Pkt);
                  break;
            case 4:
                  CopyPkt(4);
                  Port4Out = PassArray(Port4Pkt);
                  break;
      }
}

***** EstablishPath
** Connects two ports (including port 0) together
Procedure EstablishPath(Integer out_port)
{
      integer p;

      state[out_port] = BUSY_NEW;
      p = packet[PORT];
      link[out_port] = p;

      switch (out_port)
      {
            case 1:
                  CopyPkt(1);
                  Port1Out = PassArray(Port1Pkt);
                  break;
            case 2:
                  CopyPkt(2);
                  Port2Out = PassArray(Port2Pkt);
                  break;
            case 3:
                  CopyPkt(3);
                  Port3Out = PassArray(Port3Pkt);
                  break;
            case 4:
                  CopyPkt(4);
                  Port4Out = PassArray(Port4Pkt);
```

```
                    break;
        }
}

Procedure GetInPkt(integer p)
{
        integer i, temp;

        if (state[p] == FREE)
                state[p] = BUSY_DOWN;
        for i = 0 to 7
        {
                temp = inpkt[i];
                packet[i] = temp;
        }
        packet[PORT] = p;
        QPkt(Q_LINK_Q);
}


***************
** This message occurs for each step in the simulation.
on simulate
{
        Integer nextPort, slot, p, size, time, temp, count;

if (CreatePacket())         ** Produce a packet
        QPkt(Q_SEND_Q);

** Clear all output ports
Port1Out = 0;
Port2Out = 0;
Port3Out = 0;
Port4Out = 0;

** Check the input ports for newly arriving  packets
** and queue any into the LINK_Q

count = 0;
** Port #1
If (GetPassedArray(Port1In, inpkt)) ** If there is an incoming packet
{
        count++;
        GetInPkt(1);
}

** Port #2
If (GetPassedArray(Port2In, inpkt)) ** If there is an incoming packet
{
        count++;
        GetInPkt(2);
}

** Port #3
If (GetPassedArray(Port3In, inpkt)) ** If there is an incoming packet
```

```
{
      count++;
      GetInPkt(3);
}

** Port #4
If (GetPassedArray(Port4In, inpkt)) ** If there is an incoming packet
{
      count++;
      GetInPkt(4);
}

** Randomly reorder LINK_Q
while (count > 0)
{
      if (UnQPkt(Q_LINK_Q,Random(4)))
      {
            QPkt(Q_IN_Q);
            count--;
      }
}

** Process incoming packets according to the previous port state
for slot = 0 to 3
      if (UnQPkt(Q_IN_Q, slot))
      {
            p = packet[PORT];
            switch (state[p])
            {
                  case BUSY_UP:
      ** collapse:       expected
                        SyncUpCollapse(p);
      ** sync collapse upward
                        if (link[p] == 0)                          **
collapse to me
                              ProcessReturnedPkt();
                        else
      ** collapse thru me
                        {
                              p = link[p];
                              SyncDownCollapse(p);                 **
start collapse downward
                        }
                        break;
                  case SYNC:
      ** sync:           expected
                        state[p] = FREE;                           **
clear state
                        break;
                  case BUSY_START:                                 **
COLLISION!: unexpected
                        collides++;
                        SyncUpCollapse(p);
      ** Send packet back
```

```
                        state[p] = COLLISION;                    **
Don't use port for new packets
                        break;
                  case BUSY_DOWN:
      ** new message:   unexpected
                        if (Arrived())
      ** route to me
                              QPkt(Q_PROCESS_Q);
      ** devour packet
                        else
      ** route thru me
                              QPkt(Q_HOLD_Q);
      ** wait
                        break;
                  case COLLISION:
      ** fix a collision
                        if (((CURRENTSTEP MOD 2) && (A[p-1] == 1)) ||
                           (!(CURRENTSTEP MOD 2) && (A[p-1] == 0)))
                        {
                              SyncUpCollapse(p);
                              state[p] = BUSY_NEW;
                        }
                        else
                        {
                              state[p] = COLLAPSE;
                              p = link[p];
      ** get old port link
                              packet[PORT] = p;
                              if (p == 0)
                                    QPkt(Q_SEND_Q);
                              else
                                    QPkt(Q_HOLD_Q);
      ** requeue packet
                        }
                        break;
                  default:
                        Beep();
                        UserError(" Bad state");
                        Abort;
                        break;
            }
      }

** Update states
for p = 1 to 4
      switch (state[p])
      {
            case BUSY_START:
                  state[p] = BUSY_UP;                  ** passed
collision period
                  break;
      }

** send out messages
slot = 0;
```

```
while (UnQPkt(Q_HOLD_Q, slot))
{
            nextPort = Route();
            if (nextPort == 0)                          ** if no available
port
                  QPkt(Q_HOLD_Q);
            else
                  EstablishPath(nextPort);        ** connect ports
      slot++;
}


** send out messages
slot = 0;
while (UnQPkt(Q_SEND_Q, slot))
{
      nextPort = Route();
      if (nextPort == 0)                          ** if no available port
            QPkt(Q_SEND_Q);
      else
            EstablishPath(nextPort);        ** connect ports
      slot++;
}

** continue processing node packets
slot = 0;
while (UnQPkt(Q_PROCESS_Q, slot))
{
      packet[TIMELEFT]--;
      IF (packet[TIMELEFT] == 0)
            SyncDownCollapse(packet[PORT]);           ** send packet
back
      ELSE
            QPkt(Q_PROCESS_Q);
       slot++;
}

** Update states
for p = 1 to 4
      switch (state[p])
      {
            case BUSY_NEW:
                  state[p] = BUSY_START;        ** start collision
period
                  break;
            case COLLAPSE:
                  state[p] = FREE;              ** free up state for
next step
                  break;
      }

**  DONE!!!!!

}
```

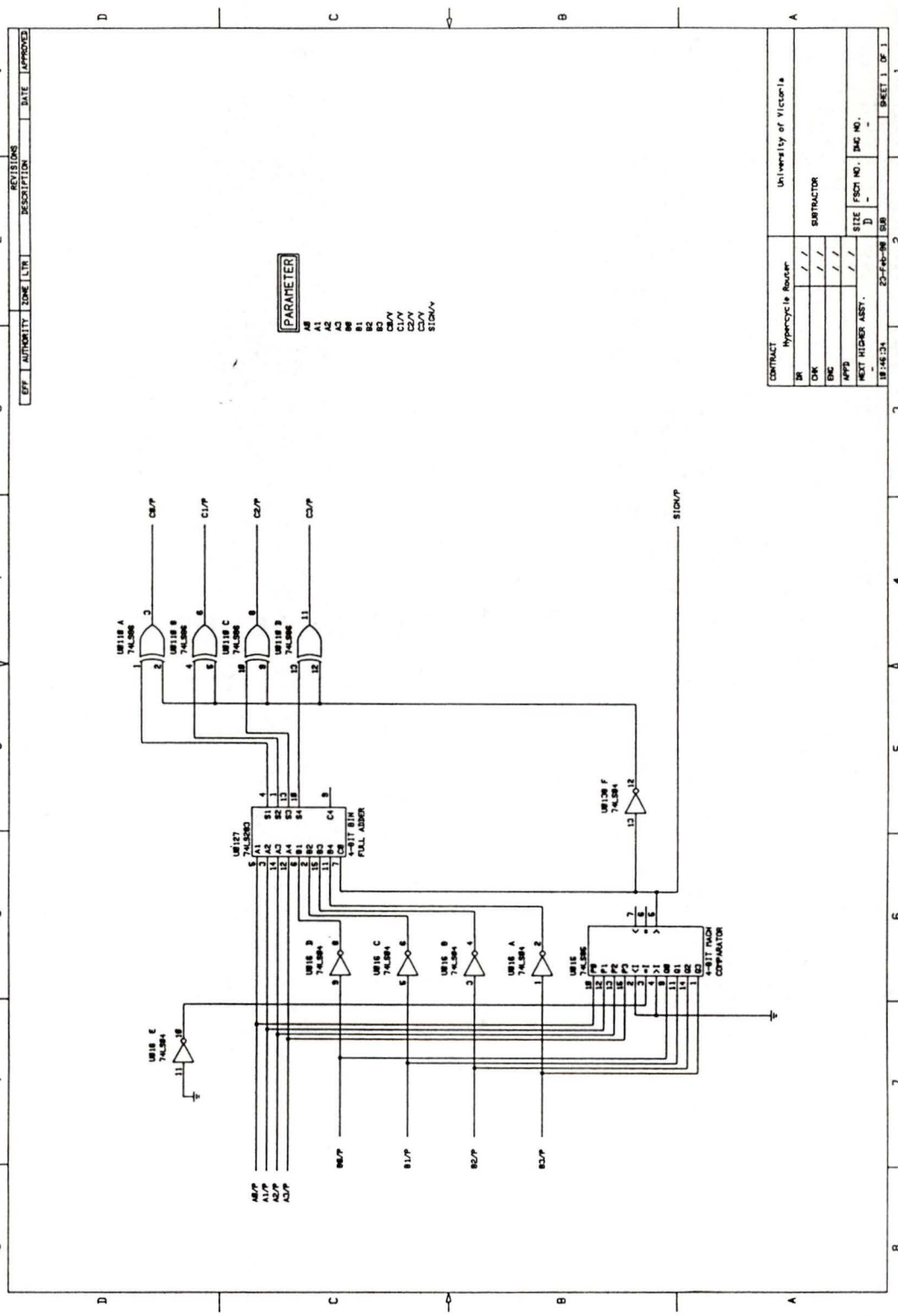
```
** If the dialog data is inconsistent for simulation, abort.
on checkdata
{

}


** Initialize any simulation variables.
on initsim
{
      integer p, i;

      for p = 0 to 4
      {
            state[p] = FREE;
            DELAY_D[p] = 0;
            SD[p] = 0;
            RD[p] = 0;
      }

      collides = 0;
      A[0] = int(a1);
      A[1] = int(a2);
      A[2] = int(a3);
      A[3] = int(a4);
      for (i = 0; i < 4; ++i)
      {
            IN_Q[i][10] = FALSE;
            LINK_Q[i][10] = FALSE;
      }
      MakeArray(SNT, 5);
      MakeArray(REC, 5);
      MakeArray(DLY, 5);
      MakeArray(PROCESS_Q, 512);
      MakeArray(PROCESS_Q, 1);
      PROCESS_Q[0][10] = FALSE;
      MakeArray(SEND_Q, 512);
      MakeArray(SEND_Q, 1);
      SEND_Q[0][10] = FALSE;
      MakeArray(HOLD_Q, 512);
      MakeArray(HOLD_Q, 1);
      HOLD_Q[0][10] = FALSE;
      MakeArray(Port1Pkt, 8);
      MakeArray(Port2Pkt, 8);
      MakeArray(Port3Pkt, 8);
      MakeArray(Port4Pkt, 8);
}


** User clicked the dialog HELP button.
on help
{
showHelp();
```

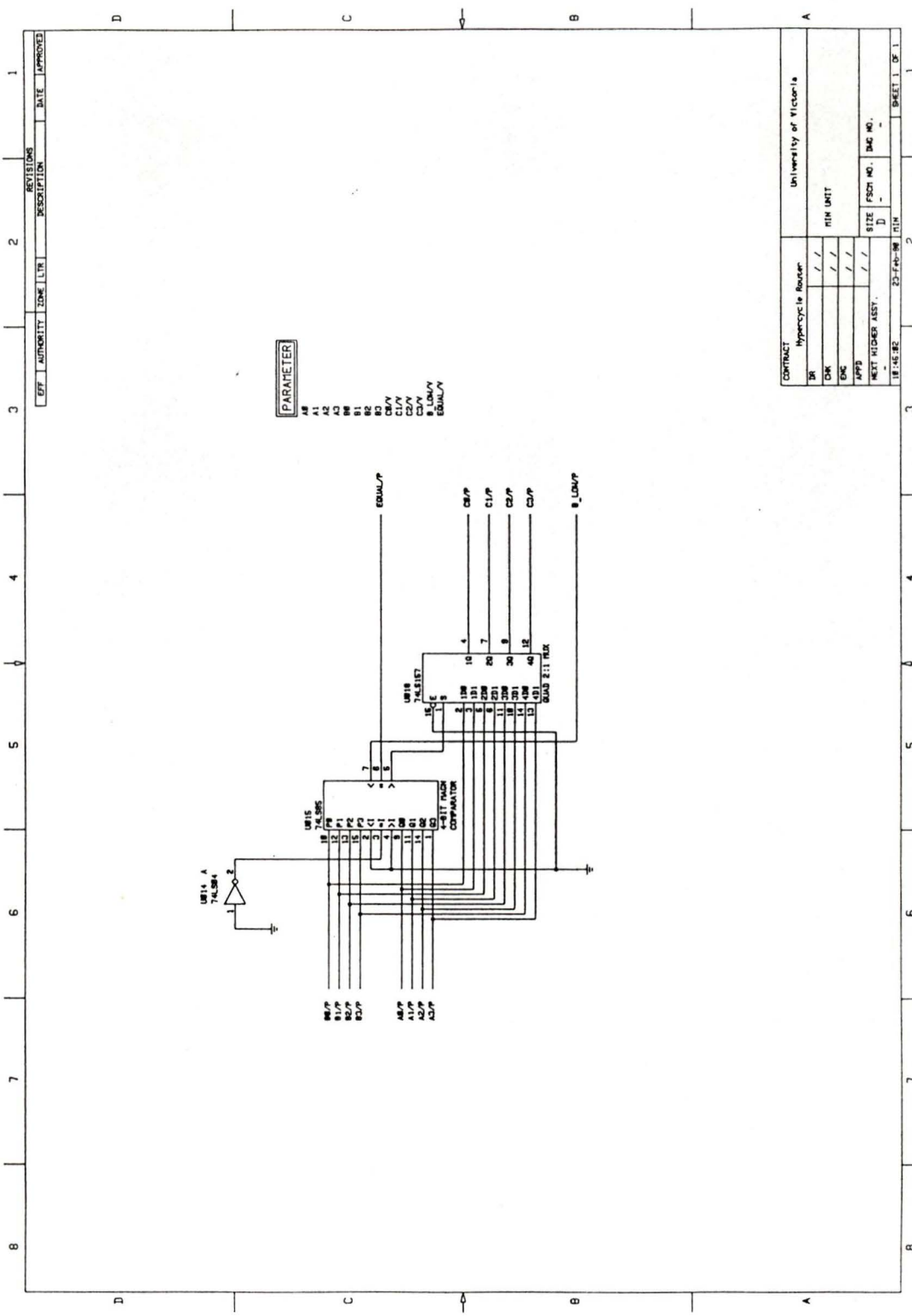
```
}

on endsim
{
      integer i;
      Integer pdelay, dist;
      Integer slot;

      Global5 += collides;

      GetPassedArray(Global0,DLY);
      GetPassedArray(Global1,SNT);
      GetPassedArray(Global2,REC);

slot = 0;
while (UnQPkt(Q_PROCESS_Q, slot))
{
      pdelay = CURRENTSTEP - packet[TIMESENT] - Global6 +
                  packet[TIMELEFT];
      dist = Distance();
      RD[dist]++;
      DELAY_D[dist] += pdelay;
      slot++;
}
slot = 0;
while (UnQPkt(Q_LINK_Q, slot))
{
      pdelay = CURRENTSTEP - packet[TIMESENT] - Global6;
      dist = Distance();
      DELAY_D[dist] += pdelay;
      slot++;
}
slot = 0;
while (UnQPkt(Q_SEND_Q, slot))
{
      pdelay = CURRENTSTEP - packet[TIMESENT] - Global6;
      dist = Distance();
      DELAY_D[dist] += pdelay;
      slot++;
}
slot = 0;
while (UnQPkt(Q_HOLD_Q, slot))
{
      pdelay = CURRENTSTEP - packet[TIMESENT] - Global6;
      dist = Distance();
      DELAY_D[dist] += pdelay;
      slot++;
}
slot = 0;
while (UnQPkt(Q_IN_Q, slot))
{
      pdelay = CURRENTSTEP - packet[TIMESENT] - Global6;
      dist = Distance();
      DELAY_D[dist] += pdelay;
      slot++;
```

```
}

        for (i=1;i<=4;++i)
        {
                DLY[i] += DELAY_D[i];
                REC[i] += RD[i];
                SNT[i] += SD[i];
        }

}
```

Appendix B

Sub-Unit Schematics


PARAMETER
A0
A1
A2
A3
B0
B1
B2
B3
C0/V
C1/V
C2/V
C3/V
SIGN/V
C0/P
C1/P
C2/P
C3/P
SIGN/P
A0/P
A1/P
A2/P
A3/P
B0/P
B1/P
B2/P
B3/P
U8110 A 74LS86
U8110 B 74LS86
U8110 C 74LS86
U8110 D 74LS86
U8127 74LS283 4-BIT BIN FULL ADDER
U8130 F 74LS04
U8116 E 74LS04
U8116 D 74LS04
U8116 C 74LS04
U8116 B 74LS04
U8116 A 74LS04
U8115 74LS85 4-BIT MAGN COMPARATOR
CONTRACT
Hypercycle Router
University of Victoria
SUBTRACTOR
DR
CHK
ENG
APPD
MEXT HIGHER ASSY.
SIZE D
FSCM NO.
DWG NO.
SHEET 1 OF 1
23-Feb-88
SUB
EFF
AUTHORITY
ZONE
LTR
DESCRIPTION
DATE
APPROVED
REVISIONS


REVISIONS
EFF
AUTHORITY
ZONE
LTR
DESCRIPTION
DATE
APPROVED
PARAMETER
A0
A1
A2
A3
B0
B1
B2
B3
CB/V
C1/V
C2/V
C3/V
B_LOW/V
EQUAL/V
UB15
74LS85
4-BIT MAGN
COMPARATOR
UB18
74LS157
QUAD 2:1 MUX
UB14 : A
74LS04
EQUAL/P
CB/P
C1/P
C2/P
C3/P
B_LOW/P
B0/P
B1/P
B2/P
B3/P
A0/P
A1/P
A2/P
A3/P
CONTRACT
Hypercycle Router
University of Victoria
MIN UNIT
DR
CHK
ENG
APPD
MFGR HIGHER ASSY.
SIZE
D
FSCM NO.
DWG NO.
SHEET 1 OF 1
23-Feb-98
MIN

Appendix C

1.5 Micron Simulation Macros

```
.MACRO F74LS00
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
P3 .NAND 3 2 P1 P2
P6 .NAND 3 2 P4 P5
P8 .NAND 3 2 P9 P10
P11 .NAND 3 2 P12 P13
.EOM F74LS00
.MACRO F74LS00A
+ P1 P2 P3
P3 .NAND 3 2 P1 P2
.EOM F74LS00A
.MACRO F74LS00B
+ P4 P5 P6
P6 .NAND 3 2 P4 P5
.EOM F74LS00B
.MACRO F74LS00C
+ P11 P12 P13
P11 .NAND 3 2 P12 P13
.EOM F74LS00C
.MACRO F74LS00D
+ P8 P9 P10
P8 .NAND 3 2 P9 P10
.EOM F74LS00D

.MACRO F74LS02
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
P1 .NOR 10 10 P2 P3
P4 .NOR 10 10 P5 P6
P10 .NOR 10 10 P8 P9
P13 .NOR 10 10 P11 P12
.EOM F74LS02
.MACRO F74LS02A
+ P1 P2 P3
P1 .NOR 10 10 P2 P3
.EOM F74LS02A
.MACRO F74LS02B
+ P4 P5 P6
P4 .NOR 10 10 P5 P6
.EOM F74LS02B
.MACRO F74LS02C
+  P11 P12 P13
P13 .NOR 10 10 P11 P12
.EOM F74LS02C
.MACRO F74LS02D
+  P8 P9 P10
P10 .NOR 10 10 P8 P9
.EOM F74LS02D

.MACRO F74LS03
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
N1   .AND 0 0 P1 P2
N2   .AND 0 0 P4 P5
N3   .AND 0 0 P9 P10
N4   .AND 0 0 P12 P13
P3   .INV/DDZ 22 18 N1
P6   .INV/DDZ 22 18 N2
P8   .INV/DDZ 22 18 N3
```

```
P11 .INV/DDZ 22 18 N4
.EOM F74LS03

.MACRO F74LS04
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
P2 .INV 2 2 P1
P4 .INV 2 2 P3
P6 .INV 2 2 P5
P8 .INV 2 2 P9
P10 .INV 2 2 P11
P12 .INV 2 2 P13
.EOM F74LS04
.MACRO F74LS04A
+ P1 P2
P2 .INV 2 2 P1
.EOM F74LS04A
.MACRO F74LS04B
+ P3 P4
P4 .INV 2 2 P3
.EOM F74LS04B
.MACRO F74LS04C
+ P5 P6
P6 .INV 2 2 P5
.EOM F74LS04C
.MACRO F74LS04D
+ P12 P13
P12 .INV 2 2 P13
.EOM F74LS04D
.MACRO F74LS04E
+ P10 P11
P10 .INV 2 2 P11
.EOM F74LS04E
.MACRO F74LS04F
+  P8 P9
P8 .INV 2 2 P9
.EOM F74LS04F

.MACRO F74LS05
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
P2  .INV/DDZ 22 18 P1
P4  .INV/DDZ 22 18 P3
P6  .INV/DDZ 22 18 P5
P8  .INV/DDZ 22 18 P9
P10 .INV/DDZ 22 18 P11
P12 .INV/DDZ 22 18 P13
.EOM F74LS05

.MACRO F74LS08
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
P3 .AND 3 4 P1 P2
P6 .AND 3 4 P4 P5
P8 .AND 3 4 P9 P10
P11 .AND 3 4 P12 P13
.EOM F74LS08
.MACRO F74LS08A
+ P1 P2 P3
P3 .AND 3 4 P1 P2
.EOM F74LS08A
```

```
.MACRO F74LS08B
+ P4 P5 P6
P6 .AND 3 4 P4 P5
.EOM F74LS08B
.MACRO F74LS08C
+ P11 P12 P13
P11 .AND 3 4 P12 P13
.EOM F74LS08C
.MACRO F74LS08D
+ P8 P9 P10
P8 .AND 3 4 P9 P10
.EOM F74LS08D

.MACRO F74LS09
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
N1   .NAND 0 0 P1 P2
N2   .NAND 0 0 P4 P5
N3   .NAND 0 0 P9 P10
N4   .NAND 0 0 P12 P13
P3   .INV/DDZ 20 15 N1
P6   .INV/DDZ 20 15 N2
P8   .INV/DDZ 20 15 N3
P11 .INV/DDZ 20 15 N4
.EOM F74LS09

.MACRO F74LS10
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
P6   .NAND 15 15 P3 P4 P5
P8   .NAND 15 15 P9 P10 P11
P12 .NAND 15 15 P1 P2 P13
.EOM F74LS10
.MACRO F74LS10A
+ P1 P2 P12 P13
P12 .NAND 15 15 P1 P2 P13
.EOM F74LS10A
.MACRO F74LS10B
+ P3 P4 P5 P6
P6   .NAND 15 15 P3 P4 P5
.EOM F74LS10B
.MACRO F74LS10C
+ P8 P9 P10 P11
P8   .NAND 15 15 P9 P10 P11
.EOM F74LS10C

.MACRO F74LS11
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
P6   .AND 13 11 P3 P4 P5
P8   .AND 13 11 P9 P10 P11
P12 .AND 13 11 P1 P2 P13
.EOM F74LS11
.MACRO F74LS11A
+ P1 P2 P12 P13
P12 .AND 13 11 P1 P2 P13
.EOM F74LS11A
.MACRO F74LS11B
+ P3 P4 P5 P6
P6   .AND 13 11 P3 P4 P5
.EOM F74LS11B
```

```
.MACRO F74LS11C
+  P8 P9 P10 P11
P8  .AND 13 11 P9 P10 P11
.EOM F74LS11C

.MACRO F74LS157
+ P1 P2 P3 P4 P5 P6 P7 P9 P10 P11 P12 P13 P14 P15
P4  .OR 0 0 N11 N12
P7  .OR 0 0 N13 N14
P12 .OR 0 0 N15 N16
P9  .OR 0 0 N17 N18
N11 .AND 5 5 P2 N19
N12 .AND 5 5 P3 N20
N13 .AND 5 5 P5 N19
N14 .AND 5 5 P6 N20
N15 .AND 5 5 P14 N19
N16 .AND 5 5 P13 N20
N17 .AND 5 5 P11 N19
N18 .AND 5 5 P10 N20
N19 .NOR 0 0 P15 P1D
N20 .NOR 0 0 P15 SN
SN .INV 5 5 P1
P1D .AND 0 0 P1
.EOM F74LS157

.MACRO F74LS283
+ P1 P2 P3 P4 P5 P6 P7 P9 P10 P11 P12 P13 P14 P15
N10 .NOR  5 5 P5  P6
N11 .NAND 3 4 P5  P6
N12 .NOR  5 5 P3  P2
N13 .NAND 3 4 P3  P2
N14 .NOR  5 5 P14 P15
N15 .NAND 3 4 P14 P15
N16 .NOR  5 5 P12 P11
N17 .NAND 3 4 P12 P11
N20 .AND 3 4 -N10 N11
N21 .AND 3 4 -N12 N13
N22 .AND 3 4 -N14 N15
N23 .AND 3 4 -N16 N17
N25 .AOI 5 4 -P7 N11 / N10
N26 .AOI 5 4 -P7 N11 N13 / N13 N10 / N12
N27 .AOI 5 6 -P7 N11 N13 N15 / N10 N13 N15 / N12 N15 /  N14
P9  .AOI 5 6 -P7 N11 N13 N15 N17 / N13 N15 N17 N10 /
+              N15 N17 N12 / N14 N17 /  N16
P4  .XOR 4 5 N20 P7
P1  .XOR 4 5 N25 N21
P13 .XOR 4 5 N22 N26
P10 .XOR 4 5 N23 N27
.EOM F74LS283

.MACRO F74LS83A
+ P1 P2 P3 P4 P5 P6 P7 P9 P10 P11 P12 P13 P14 P15
N10 .NOR  5 5 P10  P11
N11 .NAND 3 2 P10  P11
N12 .NOR  5 5 P8  P7
N13 .NAND 3 2 P8  P7
N14 .NOR  5 5 P3  P4
N15 .NAND 3 2 P3  P4
```

```
N16 .NOR  5 5 P1  P16
N17 .NAND 3 2 P1  P16
N20 .AND 3 4 -N10 N11
N21 .AND 3 4 -N12 N13
N22 .AND 3 4 -N14 N15
N23 .AND 3 4 -N16 N17
N25 .AOI 5 4 -P13 N11 / N10
N26 .AOI 5 4 -P13 N11 N13 / N13 N10 / N12
N27 .AOI 5 6 -P13 N11 N13 N15 / N10 N13 N15 / N12 N15 /  N14
P14  .AOI 5 6 -P13 N11 N13 N15 N17 / N13 N15 N17 N10 /
+              N15 N17 N12 / N14 N17 /  N16
P9  .XOR 4 5 N20 P13
P6  .XOR 4 5 N25 N21
P2  .XOR 4 5 N22 N26
P15 .XOR 4 5 N23 N27
.EOM F74LS83A

.MACRO F74LS85
+ P1 P2 P3 P4 P5 P6 P7 P9 P10 P11 P12 P13 P14 P15
P7 .AOI 5 6 P15 A4 / P13 A3 N4 / P12 A2 N4 N3 / P10 A1 N4 N3 N2
+ / N4 N2 N3 P4 N1 / N4 N3 N2 N1 P3
P6 .AND 3 4 P3 A5
P5 .AOI 5 6 P3 N1 N2 N3 N4 / P2 N1 N2 N3 N4 / P9 A1 N4 N3 N2
+ / P11 A2 N4 N3 / P14 A3 N4 / P1 A4
A1 .NAND 3 2 P9 P10
A2 .NAND 3 2 P11 P12
A3 .NAND 3 2 P13 P14
A4 .NAND 3 2 P1 P15
A5 .AND  4 4 N1 N2 N3 N4
N1 .XNOR 6 5 P9  P10
N2 .XNOR 6 5 P11 P12
N3 .XNOR 6 5 P13 P14
N4 .XNOR 6 5 P1  P15
.EOM F74LS85

.MACRO F74LS86
+ P1 P2 P3 P4 P5 P6 P8 P9 P10 P11 P12 P13
P3  .XOR 4 5 P1  P2
P6  .XOR 4 5 P4  P5
P8  .XOR 4 5 P9  P10
P11 .XOR 4 5 P12 P13
.EOM F74LS86
.MACRO F74LS86A
+ P1 P2 P3
P3  .XOR 4 5 P1  P2
.EOM F74LS86A
.MACRO F74LS86B
+ P4 P5 P6
P6  .XOR 4 5 P4  P5
.EOM F74LS86B
.MACRO F74LS86C
+ P11 P12 P13
P11 .XOR 4 5 P12 P13
.EOM F74LS86C
.MACRO F74LS86D
+ P8 P9 P10
P8  .XOR 4 5 P9  P10
.EOM F74LS86D
```

VITA

Surname: Radvan Given Names: Don

Place of Birth: Cartierville, Quebec Date of Birth: April 12, 1964

Educational Institutions Attended:

University of Victoria 1982 to 1990

Degrees Awarded:

B. Eng. University of Victoria 1988

Publications:

N. J. Dimopoulos, D. Radvan, K. F. Li, "Backtrack to the Origin and Retry Routing for Hypercycle-Based Interconnection Networks", *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, B.C., Canada, June 1989, pp. 173-177.

Nikitas J. Dimopoulos, Don Radvan and Kin F. Li, "Backtrack-to-the-Origin-and-Retry Routing for Hypercycle-Based Interconnection Networks", *Canadian Conference on Electrical and Computer Engineering*, Montreal, P.Q., Canada, September 1989, pp. 967-971.

N. J. Dimopoulos, D. Radvan, K. F. Li, "Performance Evaluation of the Backtrack-to-the-Origin-and-Retry Routing for Hypercycle-Based Interconnection Networks", *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990, pp. 278-284.

N. J. Dimopoulos, D. Radvan, W. A. Keddy, "Learning in Asymptotically Behaving Neural Networks", *Proceedings of the International Joint Conference on Neural Networks*, San Diego, Calif., June 1990, pp. 233-238.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to the users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Performance Evaluation and Router Design for Backtrack-to-the-Origin-and-Retry Routing in Hypercycle-Based Interconnection Networks

Author [redacted]
(Signature)

Don Radvan
(Name)

September 27, 1990
(Date)