RICE UNIVERSITY

Electrical & Computer Engineering Department

NETSIM

Reference Manual

Version 1.0

May 1993

J. Robert Jump

ECE Dept., Rice University
P.O. Box 1892
Houston, TX 77251-1892
email: jrj@rice.edu
Phone: (713) 527-8101 ext. 3576

This manual describes a simulator that has not been thoroughly tested and may contain bugs. Suggestions, criticisms, questions, or reports of any problems, errors, or bugs with the manual or the simulator are welcome and encouraged. Please send them to J. R. Jump at the above address.

Credits

The NETSIM code was written by J. R. Jump and Sridhar Lakshmanamurthy.

TABLE OF CONTENTS

1. INTRODUCTION

This manual describes a general-purpose interconnection network simulator called NETSIM. It can be used to construct and simulate a wide range of network models, including both direct and indirect networks. A distinctive characteristics of NETSIM is that it is modular, allowing the user to specify virtually any network structure by connecting together a number of network building blocks. It is designed to simulate large networks that use modern routing techniques, such as worm-hole and virtual-cut-through routing, but can also simulate networks that use store-and-forward routing. It can be used as a component of a simulation testbed capable of simulating a full parallel system that is driven by the execution of real programs. Alternatively, it can be used in a stand alone mode where packets are generated randomly and sent through the network. In this last case the goal is usually to characterize the network statistically.

1.2. INTERCONNECTION NETWORKS

This sections provides a brief review of interconnection networks. It describes the basic concepts of interconnection networks including switch structure, network structure, and routing techniques.

The purpose of an interconnection network is to provide communication paths between the modules of a parallel system. The simplest example is a bus, which provides a single path that all modules must share. It is usually the most inexpensive type of network, but also one of the slowest, since only one module can send data at a time. At the other extreme is the crossbar network that provides a direct connection between every pair of modules in the system. While it can provide more communication bandwidth than most other types of networks, it is also the most expensive, since it requires on the order of N^2 links to connect N modules. In between these two extremes are many other network structures that provide a wide range of tradeoffs between performance and cost. They provide multiple parallel communication paths but require some sharing of links, and paths in these networks may use more than one link.

Interconnection networks are usually constructed from switches and links. A *switch* is a device with one or more input terminals and one or more output terminals that can route data selectively from input terminals to output terminals. *Links* are communication channels used to connect switch output terminals to the input terminals of other switches.

A common choice for the internal organization of a switch is a crossbar. This type of switch can pass data through several terminals at the same time as long as none of them are sending to the same output terminal. The choice of which output terminal to use is made by a local controller inside the switch that uses information from the incoming data to make the selection. In the event that data at two input terminals are routed to the same output terminal at the same time, the switch must make a choice to transfer the data from one of the terminals and block the other. This is a source of contention or conflict in the network and can be a major restriction on its performance.

It is also common for networks to contain buffers for temporarily holding data that is blocked waiting for a free output terminal. These buffers can be inserted in various ways into the switches or in the links. Networks with buffers can hold blocked data at the switch where a conflict occurred. If buffers are not used and there is a conflict for an output terminal of a switch, the transfer from the input terminal that is not chosen is usually aborted, and the source module that initiated the transfer tries again.

There are two major categories of interconnection networks, *direct networks* and *indirect networks*. Direct networks have one switch associated with each module to be connected. Each switch has one terminal and link used to connect it to the module and one or more terminals with links to connect it directly to other switches in the network. The network used to interconnect the processors of a hypercube multicomputer is an example of a direct network. Fig. 1. illustrates an 8-node hypercube organization. Note that the number of links that must be used to pass data from one processor to another is different for different pairs of processors. Some pairs are adjacent in the network (i.e., share a link), and can communicate by passing data through only one link. Other pairs can only communicate through paths of several links.

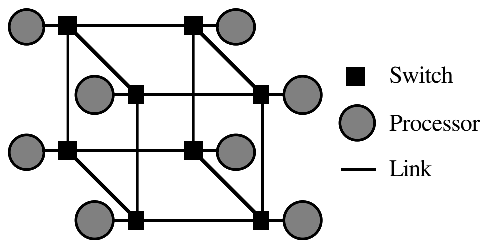

Fig. 1. 8-Node Direct Network (Hypercube).

In an indirect network, some switches are connected to modules, but some are not. That is, in addition to the switches that connect to both modules and other switches, there are also switches that only connect to other switches. Usually the switches in an indirect network are connected into stages as shown in Fig. 2. This type of network is called a *multistage interconnection network.* Data enters the network by moving from a module into a switch in stage 1. Then it proceeds to move from switches in one stage to switches in the next stage until it reaches the other side of the network. In networks with a regular structure, such as the one shown in Fig. 2, all data must pass through every stage, so that all packets pass through the same number of switches before reaching their destinations.

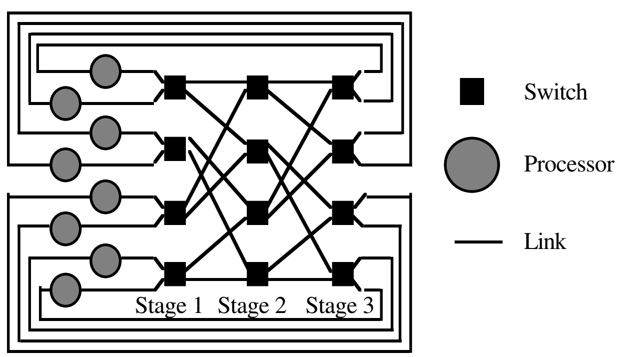

Fig. 2. 8-Node Indirect Network (Perfect Shuffle).

A *routing protocol* is a mechanism for controlling the movement of data through the switches and links of a network In most routing protocols, data is organized into packets consisting of several smaller units of data called *flits*. A flit is the smallest unit of data that can be transferred between two switches in a single cycle. It has a fixed size determined by the number of wires in the link. The first flit, called the *head flit,* of a packet contains routing information used by the switches to select output terminals. When a packet moves through a network, its head flit goes first and establishes a path by selecting output terminals of the switches it passes through. The flits that follow the head flit do not contain routing information and must follow the path established by the head. The last flit of a packet is called the *tail* of the packet, and it tears down the path by freeing each switch output terminal it passes through.

Since only one flit can pass through a link at a time, the flits of a packet can become distributed over several buffers. There are two common routing protocols distinguished by the way the flits of a packet can be spread over buffers in the network. The first routing protocol is called *store-and-forward routing*, because it moves all of the flits into a buffer before it moves any flits of the packet out of the buffer. In store-and-forward routing the flits of a packet are spread over at most two buffers. The second routing protocol is characterized by the property that the head and following flits can leave a buffer as soon as they have a free output terminal and space available in the next buffer; the head does not need to wait until its tail flit catches up before it moves to the next buffer. The flits of a packet can become spread out over several buffers in the network. This type of routing has been characterized in two ways. If each switch has enough buffer space to hold an entire packet and a head flit does not start to move into a new buffer until there is enough free space in that buffer to hold an entire packet, the routing protocol is called *virtual-cut-through routing*. If the head flit can start moving as soon as there is at least one free flit position in the next buffer, the protocol is called *worm-hole routing*. In some definitions of worm-hole routing the amount of buffer space is assumed to be small, frequently one or two flits.

1.2. NETSIM, YACSIM, AND PARCSIM

NETSIM is an extension of the discrete event simulator YACSIM. Therefore, the user must be familiar with YACSIM in order to use NETSIM. This manual only discusses the NETSIM extensions. The user should consult the YACSIM Reference Manual for a description of YACSIM objects and operations.

NETSIM and YACSIM are implemented by providing a set of data structures and subroutines in the C programming language. Therefore, to write a NETSIM simulation, one writes a C program that declares NETSIM and YACSIM objects and calls the subroutines provided to manipulate these objects. The instructions for running a NETSIM simulation are exactly the same as for a YACSIM simulation and are given in the YACSIM Reference Manual.

NETSIM is an integral part of a parallel architecture simulator called PARCSIM, although the full structure of PARCSIM need not be used to simulate networks. PARCSIM adds processor and memory modules, as well as the NETSIM capability, to YACSIM. This allows the user to specify a complete parallel system with processors, memory modules, and an interconnection network. PARCSIM along with a number of profilers constitute the Rice Parallel Processing Testbed (RPPT). The profilers are used to instrument a parallel program whose execution on a parallel system can then be simulated with PARCSIM. In this way, interconnection networks can be studied with packets generated by the execution of real programs. Alternatively, NETSIM alone can simulate interconnection networks with the packets generated by a random process as has been done in many previous studies.

1.3. NETSIM MODULES

While an m×n switch was seen as a basic building block in Section 1.1, there are several ways to implement switches, and different implementation choices can have a significant effect on the performance of a network. Therefore, the choice of switch implementation is an important design parameter and one we might want to investigate using NETSIM. To avoid providing a large number switch types as basic NETSIM modules, and because we can not predict all types of switches that might be useful, NETSIM provides modules that are more primitive than switches and from which a large variety of switches can be constructed. NETSIM provides two such modules, multiplexers and demultiplexers, that along with buffers are sufficient to construct most switch and networks of interest.

The basic NETSIM modules are described below:

Multiplexers: These are multi-input, single-output modules used to merge data. A multiplexer is the module that resolves conflict when two of its input terminals have data to transfer to its output terminal at the same time. This arbitration is currently implemented as a semaphore with a FIFO queuing discipline.

Demultiplexers: These are single-input, multi-output modules used to route data along one of several paths. This module is the one that implements the routing mechanism of a network. It uses data in the head flit of a packet at its input terminal to select which one of its output terminals the packet will pass through. The user can specify the routing algorithm used to make this choice.

Buffers: These are modules used to provide temporary storage for flits as they move through a network. They are currently implemented as finite FIFO queues. The user can specify the maximum number of flits a buffer can hold.

Network Ports: These are single-input, single-output modules used as interface units between a network and its external environment. There are two types of network ports. **Input ports** provide an interface through which packets are passed into the network, and packets are removed from the network through **output ports**.

Symbols that can be used to represent these modules in a network diagram are shown in Fig. 3.

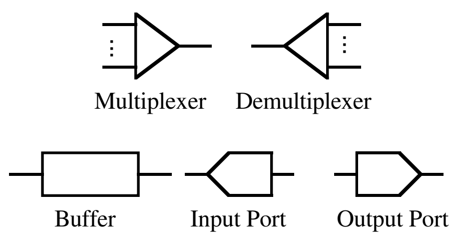

Fig. 3. Basic Network Module Symbols.

To build a simulation model for a network, the user creates the modules by calling NETSIM routines that allocate descriptors for the modules and return pointers to those descriptors. To connect two modules, the user calls a connection routine with pointers to the modules' descriptors as arguments. An output terminal of any of the modules except an output port can be connected to an input terminal of any module except an input port.

1.4. NETSIM PACKETS

Data is sent through a network in the form of packets consisting of several flits. Each packet contains routing information used by the network's demultiplexers to route it through the network. There are NETSIM operations for creating packets, putting them into a network through an input port, and removing them from the network through an output port. The simulator automatically measures the time each packet spends in the network.

In PARCSIM simulations, packets may be part of a larger data unit called a message. To facilitate the simulation of messages, each packet provides a link back to the message to which it belongs. This, along with sequence numbers that can be assigned to packets, can be used by routines external to NETSIM to decompose messages into packets before transmission through a network and to reassemble received packets into messages.

Each packet is implemented by two YACSIM events, one for the front end or head of the packet and one for the rear end or tail of the packet. The event for the head of the packet establishes a path through the network using the routing functions of the demultiplexers and routing information in the packet. The event for the tail of the packet manages the distribution of flits between a packet's head and tail, and follows the path established by the head event. The creation and scheduling of these events is done automatically when a packet is sent into the network. They terminate when the packet's flits reach their destination port.

1.5. COMPILING NETSIM SIMULATIONS

To compile a simulation program under the UNIX operating system, you need access to two files, *sim.h* and *netsim.o*. The file *sim.h* should be included in all the files that make up your program. It contains useful pre-defined symbols, declarations of all the YACSIM and NETSIM operations available to the user, and definitions of the simulation object types. The file *netsim.o* is the library of all YACSIM and NETSIM operations.

To compile a simulation program, use a command line of the form:

cc "your options for the compiler" "your files" netsim.o

You may include any options you want for the C compiler such as -g, -o, etc. To use this command, you must put *sim.h* and *netsim.o* where the compiler can find them, for example in the same directory as the source code for the simulation program, or use full path names for them.

An alternative way to compile a simulation program is to use a command line of the form:

netsim "your options for the compiler" "your files"

To use this form, both sim.h and netsim.o, along with the command file netsim, must all be in the same directory, and that directory must have been passed to the command program netsim when it was compiled. The full name of this directory must be on your search path.

2. NETSIM OPERATIONS

This section describes of NETSIM's user operations. Since NETSIM is an extension of YACSIM all of these operations must be called from within YACSIM programs.

2.1. NETWORK CONSTRUCTION OPERATIONS

These are the operations used to create and interconnect network modules. For each of the five types of modules, there is an operation for constructing an instance of that module. There is also a general operation for connecting an output terminal of one module to an input terminal of another.

*BUFFER *NewBuffer(id, size)*
int id;
int size;

> This operation creates and returns a pointer to a new network buffer. It assigns the buffer the number *id* used to identify the buffer in trace statements. *Size* designates the maximum number of flits that can be stored in the buffer.

*MUX *NewMux(id, fanin)*
int id;
int fanin;

> This operation creates and returns a pointer to a new multiplexer with *fanin* input terminals and identifier *id.*

*DEMUX *NewDemux(id, fanout, routingfcn)*
int id;
int fanout;
rtfunc routingfcn;

> This operation creates and returns a pointer to a new demultiplexer with *fanout* output terminals and identifier *id*. It also assigns the function *routingfcn* to be used by the demultiplexer in routing packets to one of its output terminals. The type *rtfunc* is a pointer to a function with three arguments that returns an integer. The routing function is called with the following arguments each time a packet enters a demultiplexer:
>
> src - a pointer to an integer whose value is the id of the source processor sending the packet
>
> dest - a pointer to an integer whose value is the id of the destination processor receiving the packet
>
> demuxid - an integer whose value is the id of the demultiplexer.
>
> The user must provide a function that can compute the demultiplexer output terminal from these parameters. The first two arguments point to fields in a packet descriptor. They are passed as pointers so that the routing function can modify the values as the packet moves from node to node in the network. For example, one way to implement the classical bit-directed routing algorithm for perfect shuffle networks is use the least significant bit of the destination at each stage to determine which switch output to use. That bit is then discarded and the destination bits shifted so that the next stage will find the next most significant bit in the least significant bit position.

*IPORT *NewIPort(id, size)*
int id;
int size;

This operation creates and returns a pointer to a new network input port with identifier *id* and the capacity to buffer up to *size* packets.

*OPORT *NewOPort(id, size)*
int id;
int size;

This operation creates and returns a pointer to a new network output port with identifier *id* and the capacity to buffer up to *size* packets.

void NetworkConnect(src, dest, src_index, dest_index)
*MODULE *src;*
*MODULE *dest;*
int src_index;
int dest_index;

This operation is used to construct an interconnection network by interconnecting the basic modules (i.e., buffers, multiplexers, demultiplexers, and ports). It connects an output terminal of the module pointed to by *src* to an input terminal of the module pointed to by *dest*. If the source is a demultiplexer, then *src_index* specifies which output terminal is used. Since all other types of modules have only one output terminal, this parameter is ignored for them. The parameter *dest_index* designates an input terminal for the destination module if it is a multiplexer. It is ignored for all other types of modules.

2.2. NETWORK DELAYS

The operations described here are used to specify the various delays that affect the movement of flits through the network. All delays are an integral multiple of a basic cycle time. This cycle time can be viewed as the period of a system clock used to synchronize the operation of the network.

The head flit experiences a delay at every module it passes through. The flits that follow the head only experience a delay, called a *flit delay*, in moving from one buffer to another. It is assumed that the flit delay parameter is long enough to allow a flit to move between two buffers and any multiplexers or demultiplexers between them. The extra delay experienced by the head flit as it moves through multiplexers and demultiplexers is overhead due to the work required to perform routing and arbitration.

void NetworkSetCycleTime(x)
double x;

Execution of this operation sets the network cycle time to *x*. All other delays are integer multiples of the cycle time. The default cycle time is 1.0. This value is used if NetworkSetCycleTime() is never called. It must only be called once and before the first call to DriverRun() that starts the simulation.

void NetworkSetFlitDelay(d)
int d;

Execution of this operation sets the time required to move a flit from one buffer or port to another buffer or port to *d* times the network cycle time. This value is used for all buffers and ports in the simulated network. The default value is 1, and it will be used if the user does not explicitly change it. If the user does set the flit delay, this must be done before the first call to DriverRun().

void NetworkSetMuxDelay(d)
int d;

Execution of this operation sets the time required to move the head flit through a multiplexer to *d* times the cycle time. The default value is 0. This operation must be called before the first call to DriverRun().

void NetworkSetArbDelay(d)
int d;

Execution of this operation sets the time for a multiplexer to arbitrate between competing flits to *d* times the flit delay. The default value is 0. This operation must be called before the first call to DriverRun().

void NetworkSetDemuxDelay(d)
int d;

Execution of this operation sets the time required to execute the routing function and then move a flit through a demultiplexer to *d* times the flit delay. The default value is 0. This operation must be called before the first call to DriverRun().

void NetworkSetPktDelay(d)
int d;

This operations determines the time to move a packet into an input port or out of an output port. That time is dependent on the number of flits in the packet and is given by

$$d \times (\text{number of flits in the packet}) \times (\text{network cycle time}).$$

The default value is 0. This operation must be called before the first call to DriverRun().

2.3. NETWORK THRESHOLDS

There are two parameters that determine the flow control technique that will be used to move flits through a simulated network. The first of these is the number of flit positions in a buffer that must be free before a new packet can start to enter that buffer. The second is a flag that specifies whether or not a packet's tail flit must be in the same buffer as its head flit before the head flit can move out of the buffer. These parameters can be used to specify the routing mode used by the network. For example, the three most common routing modes are specified in the following way:

Store-and-Forward: Threshold = Packet Size; Head waits for its tail

Virtual Cut Through: Threshold = Packet Size; Head doesn't wait for its tail

Worm Hole: Threshold < Packet Size (usually 1); Head doesn't wait for its tail

void NetworkSetThresh(t)
int t;

The threshold of a buffer is the number of flit positions that must be free before another packet can start to enter the buffer. This operation sets the threshold for all buffers to *t*. The default value is 1. This operation must be called before the first call to DriverRun().

void NetworkSetWFT(i)
int i;

This operations determines whether or not the head flit of a packet will wait in a buffer until its tail flit is also in that buffer before attempting to leave the buffer. The argument *i* has two possible values: WAIT if heads are to wait for their tails, and NOWAIT if not. The default value is NOWAIT. This operation must be called before the first call to DriverRun().

2.4. PACKET OPERATIONS

These operations are used to create and initialize packets, to send them into a network through an input port, and to remove them from a network through an output port.

*PACKET *NewPacket(seqno, msgptr, size, src, dest)*
int seqno;
*MESSAGE *msgptr;*
int size;
int src;
int dest;

This operation creates and returns a pointer to a new packet and assigns it values given by the arguments. S*eqno* can be used to order the packets of a message. *Msgptr* is a pointer to the packet's message, and *size* is the number of flits in the packet. *Src* and *dest* are identifiers of the sending and receiving processors for the packet. They are used by the demultiplexer routing functions to move the packet through the network.

double PacketSend(pkt, port)
*PACKET *pkt;*
*IPORT *port;*

This operation attempts to send the packet pointed to by *pkt* into a network through the network input port pointed to by *port*. If the operation fails (because the port's queue is full), it returns -1.0; otherwise, it returns the time the network needed to move the packet into the port. Note that this operation returns at the same simulation time it was called and before the packet is actually moved into the network. The activity that called PacketSend() can use its return value to delay for the send time, or it can continue and overlap its future operation with the movement of the packet into the port (for example, to simulate DMA).

*PACKET *PacketReceive(port)*
*OPORT *port;*

This operation attempts to remove a packet from a network through the port pointed to by *port*. If there are no packets available at this port, it returns the NULL pointer.

*PKTDATA*PacketGetData(pkt)*
*PACKET *pkt;*

This operation is used to access the data in a packet. It returns a pointer to a structure of type PKTDATA that contains the packet's data. This structures is defined as follows:

```
struct PKTDATA {
    int        seqno;       /* User supplied ID for sequencing a message's packets  */
    MESSAGE    *mesgptr;    /* A pointer to the message that this packet belongs to */
    int        pktsize;     /* Number of flits in the packet                        */
    int        srccpu;      /* The ID of the CPU sending the packet                 */
    int        destcpu;     /* The ID of the CPU receiving the packet               */
    double     recvtime;    /* Time to remove the packet from an output port        */
    double     createtime;  /* Time the packet was created                          */
    double     nettime;     /* Time the packet spent in the network                 */
    double     blktime;     /* Time the packet was blocked in the network           */
    double     oporttime;   /* Time the packet spent waiting in an output port      */
};
```

The first five of these fields are set when the packet is created with NewPacket(). The value of the *recvtime* field is the time required to move the packet out of the output port. PacketReceive() returns a packet pointer at the same simulation time that it is called and before the packet is actually removed from the port. *Recvtime* can be added to the current simulation time to find the earliest time at which the data is really available. The activity that called PacketReceive() can use this value to delay for the receive time, or it can continue and overlap its future operation with the movement of the packet out of the port (for example, to simulate DMA).

The last four fields are variables used to measure the time a packet spends in various states as it moves through the network. They are defined in the section on network statistics below.

void PacketFree(pkt)
*PACKET *pkt;*

This operation returns the packet pointed to by *pkt* to a pool of free packets. It also updates those network statistics records that have been activated (Section 2.6).

2.5. PACKET SYNCHRONIZATION OPERATIONS

The operations described here provide the user with a means to synchronize the movement of packets into and out of a network. They can be used to test the status of a port, and they provide access to semaphores within the ports that can be used to synchronize sending and receiving processes.

*SEMAPHORE *IPortSemaphore(port)*
*IPORT *port;*

This operation returns a pointer to a semaphore within the input port pointed to by *port*. If the port is full and can not accept new packets, a sending activity can wait on this semaphore. When space becomes available, the port will signal the semaphore and release the activity.

int IportSpace(port)
*IPORT *port;*

This operation returns the number of packets that can be sent to the input port pointed to by *port* before its queue becomes full.

int IportGetId(port)
*IPORT *port;*

This operation returns the user assigned id of the port pointed to by *port*.

*SEMAPHORE *OPortSemaphore(port)*
*OPORT *port;*

This operation returns a pointer to a semaphore within the port pointed to by *port*. If the port is empty, a sending activity can wait on this semaphore. When a packet arrives in the port, it will signal the semaphore and release the waiting activity.

int OportPackets(port)
*OPORT *port;*

This operation returns the number of packets that are currently queued at the network output port pointed to by *port*.

int OportGetIdport)
*OPORT *port;*

This operation returns the user assigned id of the port pointed to by *port*.

2.6. NETWORK STATISTICS

NETSIM can generate several useful statistics during a simulation. They are implemented with YACSIM statistics records, so they are capable of automatically calculating means and histograms for each of the statistics that are collected. Statistics can be collected on the following variables:

NETTIME: The time a packet spends in the network, defined as the difference between the time the head flit of the packet enters an input port and the time the tail flit of that packet leaves an output port. The variable is also called the *packet latency*.

BLKTIME: The total time a packet is blocked while in the network. This includes time the head flit of the packet waits for a multiplexer, for free buffer space, and for packets ahead of it in a buffer to move out of that buffer. It is a measure of the congestion in the network.

OPORTTIME: The time a packet spends in an output port waiting for an activity to take it out by executing a PacketReceive() operation. It is measured from the time the packet's tail enters the port until its head starts to leave the port.

MOVETIME: The time a packet is making progress through the network. It is defined as

NETTIME - BLKTIME - OPORTTIME

This variable represents an ideal latency. That is, it is a measure of how much time the packet would spend in the network if it were never blocked, and if it were removed from the network as soon as it arrived at an output port.

CREATETIME: This is the time at which it the packet was created by NewPacket(). Subtracting this from the current simulation time just before freeing the packet with PacketFree() will give the life time of the packet.

Each of the first four variables above is said to be *active* if the packet is currently accumulating time for that variable. Otherwise the variable is said to be *idle*. These variables are updated for every packet as it switches from active and idle. The variable CREATETIME is active the whole time the packet exists.

All five variables can be accessed after a packet has been received. They are available as fields in a structure of type PKTDATA associated with each packet. This structure is obtained with the operation PacketGetData() as described in Section 2.4.

There are five YACSIM statistics records built into NETSIM and used to collect statistics on the variables defined above. They are updated automatically with the total time the associated variable was active whenever the packet is freed by calling PacketFree(). **Note that these statistics records are not updated until the packet is freed, and are not updated at all if it is never freed**.

The following operations are used to activate and access the automatic statistics collection features of NETSIM. Statistics records and their operations are defined in the YACSIM Reference Manual.

void NetworkCollectStats(type,histflg,nbin,lowbin,highbin)
int type;
int histflg
int nbin;
double lowbin;
double highbin;

This operation activates the automatic collection of network statistics. Five different statistics records exist and are specified by the first parameter *type*, which can have any one of the following five values:

NETTIME	Activate collection of "total time in the network" statistics
BLKTIME	Activate collection of "time blocked in the network" statistics
OPORTTIME	Activate collection of "time waiting in an output port" statistics
MOVETIME	Activate collection of "time packet moving" statistics
LIFETIME	Activate collection of "time packet is alive" statistics

All of the five are point statistics records with the calculation of means enabled. Histograms can be enabled or not by setting the parameter *histflg* to HIST or NOHIST. If histograms are enabled, the number of bins will be *nbins*+2 with *nbins* equal-sized bins between *lowbin* and *highbin* and an overflow bin on each end.

void NetworkResetStats()

This operation resets all five statistics records.

void NetworkStatPtr(type)
int type;

This operation returns a pointer to the network statistics record specified by the *type* parameter. The five possible values for *type* are the same as for the NetworkCollectStats() operation. The chapter on statistics records in the YACSIM Reference Manual describes the various operations that can be performed on a statistics record once you have a pointer to it.

void NetworkStatRept()

Although standard or customized reports can be generated for any statistics record, this operation generates a simplified presentation of the statistics automatically collected by NETSIM during a simulation. It prints the following values:

Network throughput: The total number of packet passed through the network divided by the total time of the simulation. This measure is also available as the rate of the "total time in the network" statistics record.

Average packet latency: The mean of the NETIME variable.

Ideal latency: The mean of the MOVETIME variable.

Normalized latency: The ratio of ideal latency to average latency.

Average time packets blocked in the network: The mean of the BLKTIME variable.

Average time packets wait to be received: The mean of the OPORTTIME variable.

Average packet lifetime: The mean of the lifetimes of all packets generated during a simulation.

3. EXAMPLE NETWORK SIMULATIONS

This section provides two examples of NETSIM simulations. Although they are for very simple networks containing only a few modules, they illustrate the use of most of the NETSIM modules and operations.

3.1. SWITCH #1

This network is a 2 by 2 switch with buffers on the output terminals. It is realized by connecting a 2-input multiplexer to a 2-output demultiplexer as shown in Fig. 4. This is essentially a bus where the connection between the multiplexer and demultiplexer must be used in all transfers. Therefore only one transfer can take place at a time.

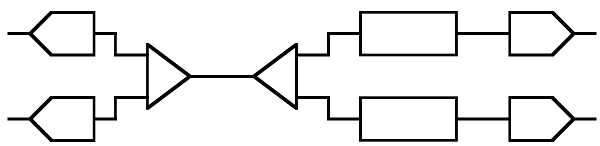

Fig. 4. 2×2 Switch Realized as a Bus.

Program Listing:

```
#include "sim.h"

IPORT *inport0;
IPORT *inport1;
OPORT *outport0;
OPORT *outport1;

int npkt = 6;
int pktsz = 2;
int bufsz = 2;
double send_delay = 1.25;
double recv_delay = 1.75;

int router(src,dest,id)
int *src;
int *dest;
int id;
{
   return *dest;   /* Trivial router, uses destination directly */
}

void UserProcessS()
/*   There is one of these processes at each input port of the network.  It simply    */
/*   generates packets and sends them into the network.  If the input port is         */
/*   full, the process waits in the port's semaphore until space is available.        */
/*   After sending a packet, the process delays for the amount of time it would       */
/*   take to move the packet into the input port plus a user specified delay.         */
{
   PACKET *pkt;
   IPORT *inport;
   int i, s, k=0;
   double senddelay;

   if ((i = ActivityArgSize(ME)) == 1) {          /* If attached to port 1            */
      s = i;                                      /* Remember the source module id    */
      inport = inport1;                           /* Remember the port id             */
```

```
      i = 100;                                /* Number port 1 packets from 100   */
   }
   else {                                     /* Else attached to port 0          */
      s = 0;                                  /* Remember the source module id    */
      inport = inport0;                       /* Remember the port id             */
      i = 200;                                /* Number port 0 packets from 200   */
   }
   while (npkt > 0) {                         /* Send npkt packets into the net   */
      pkt = NewPacket(i+k,NULL,pktsz);        /* Get a new packet                 */
      senddelay = PacketSend(pkt,inport,s,k%2);   /* Try to send it into the net  */
      if (senddelay >= 0.0) {                 /* If the send was succussful       */
         npkt--;                              /* One fewer packet to send         */
         ProcessDelay(senddelay);             /* Delay for the packet move time   */
         printf("Packet %d delivered to net at time %g\n",i+k,GetSimTime());
         k++;                                 /* Used to generate packet numbers  */
      }
      else SemaphoreWait(IPortSemaphore(inport)); /* Otherwise wait in the semaphore */
      ProcessDelay(send_delay);               /* Delay for extra time             */
   }
}

void UserProcessR()
/*   There is one of these processes at each output port of the network.  It waits   */
/*   until a packet arrives at the port and then removes it from the network.        */
{
   PACKET *pkt;
   OPORT *outport;
   PKTDATA *pktdata;
   int i,j,k=0;

   if ((i = ActivityArgSize(ME)) == 1) {      /* If attached to port 1            */
      outport = outport1;                     /* Remember the port id             */
   }
   else {                                     /* Else attached to port 0          */
      outport = outport0;                     /* Remember the port id             */
   }
   while (1) {
      if (OPortPackets(outport)) {            /* If there is a packet in the port */
         pkt = PacketReceive(outport);        /* Take it out of the network       */
         pktdata = PacketGetData(pkt);        /* Get a pointer to its data        */
         printf("Receiving packet %d at time %g\n",pktdata->seqno,GetSimTime());
         printf("    Life time = %g; Total time in the net = %g\n",
            GetSimTime()-pktdata->createtime, pktdata->nettime);
         printf("    Time blocked in the net = %g, Time in an output port = %g\n",
            pktdata->blktime, pktdata->oporttime);
         PacketFree(pkt);                     /* Release the packet               */
      }
      else SemaphoreWait(OPortSemaphore(outport));/* Otherwise wait in th semaphore  */
      ProcessDelay(recv_delay);               /* Delay for extra time             */
   }
}

UserMain(argc, argv)
int argc;
char** argv;
{
  PROCESS* process;
  BUFFER   *buf0, *buf1;
  MUX      *mux;
  DEMUX    *demux;

  if (argc > 1)                     /* 1st argument is the number of packets         */
     npkt = atoi(argv[1]);
  if (argc > 2)                     /* 2nd argument is the packet size               */
     pktsz = atoi(argv[2]);
```

```
    if (argc > 3)                   /* 3rd argument is the buffer size           */
       bufsz = atoi(argv[3]);
    if (argc > 4)                   /* 4th argument is the extra send delay      */
       send_delay = (double)atoi(argv[4]);
    if (argc > 5)                   /* 5th argument is the extra receive delay   */
       recv_delay = (double)atoi(argv[5]);

    NetworkSetFlitDelay(1);         /* All delays set to 1 times cycle time      */
    NetworkSetMuxDelay(1);
    NetworkSetArbDelay(1);
    NetworkSetDemuxDelay(1);
    NetworkSetPktDelay(1);

    inport0 = NewIPort(10, 2);        /* Create input ports                      */
    inport1 = NewIPort(11, 2);
    mux   = NewMux(20, 2);            /* Create multipolexer                     */
    demux = NewDemux(30, 2, router);  /* Create demultiplexer                    */
    buf0 = NewBuffer(40, bufsz);      /* Create buffers                          */
    buf1 = NewBuffer(41, bufsz);
    outport0 = NewOPort(50, 2);       /* Create ouput ports                      */
    outport1 = NewOPort(51, 2);

    /* Connect modules to make the network:                                      */
    /*   This network is a 2 x 2 switch realized as a bus with buffers on its outputs   */
    /*   It consists of a 2-input MUX followed by a 2-ouput demux               */

    NetworkConnect(inport0,mux,0,0);   /* Input port 0 connected to terminal 0 of mux   */
    NetworkConnect(inport1,mux,0,1);   /* Input port 1 connected to terminal 1 of mux   */

    NetworkConnect(mux,demux,0,0);     /* Mux output connected to demux input     */

    NetworkConnect(demux,buf0,0,0);    /* Demux terminal 0 connected to buffer 0  */
    NetworkConnect(demux,buf1,1,0);    /* Demux terminal 1 connected to buffer 1  */

    NetworkConnect(buf0,outport0,0,0); /* Buffer 0 connected to output port 0     */
    NetworkConnect(buf1,outport1,0,0); /* Buffer 1 connected to output port 1     */

    process = NewProcess("UserSend0",UserProcessS,0);  /* Create sender process 0    */
    ActivitySetArg(process,NULL,0);                    /* Pass process its id        */
    ActivitySchedTime(process, 0.0, INDEPENDENT);      /* Schedule process           */

    process = NewProcess("UserSend1",UserProcessS,0);  /* Create sender process 1    */
    ActivitySetArg(process,NULL,1);                    /* Pass process its id        */
    ActivitySchedTime(process, 0.0, INDEPENDENT);      /* Schedule process           */

    process = NewProcess("UserRecv0",UserProcessR,0);  /* Create receiver process 0  */
    ActivitySetArg(process,NULL,0);                    /* Pass process its id        */
    ActivitySchedTime(process, 0.0, INDEPENDENT);      /* Schedule process           */

    process = NewProcess("UserRecv1",UserProcessR,0);  /* Create receiver process 1  */
    ActivitySetArg(process,NULL,1);                    /* Pass process its id        */
    ActivitySchedTime(process, 0.0, INDEPENDENT);      /* Schedule process           */

    NetworkCollectStats(NETTIME,NOHIST,0.0,0.0);       /* Collect Statistics         */
    NetworkCollectStats(BLKTIME,NOHIST,0.0,0.0);
    NetworkCollectStats(OPORTTIME,NOHIST,0.0,0.0);
    NetworkCollectStats(MOVETIME,NOHIST,0.0,0.0);
    NetworkCollectStats(LIFETIME,NOHIST,0.0,0.0);

    DriverRun(0.0);                                    /* Start the simulation       */

    NetworkStatRept();                                 /* Print statistics           */
  }
```

3.2. SWITCH #2

This network is also a 2 by 2 switch, but it is realized as a crossbar with buffers on its input terminals as show in Fig. 5. It uses two multiplexers with their outputs connected to the inputs of two demultiplexers providing four possible paths from input ports to output ports. Therefore, two packets could be transmitted through the switch at the same time if they are directed to different output terminals.

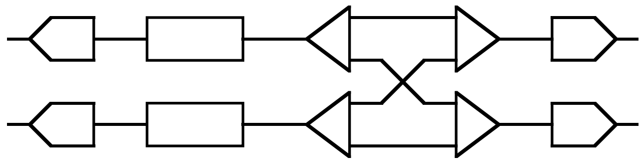

Fig. 5. 2×2 Switch Realized as a Crossbar.

Program Listing:

The only differences between this program and the one for Switch #1 are in the routine UserMain listed below.

```
UserMain(argc, argv)
int argc;
char** argv;
{
  PROCESS* process;
  BUFFER   *buf0, *buf1;
  MUX      *mux0, *mux1;
  DEMUX    *demux0, *demux1;

  if (argc > 1)                   /* 1st argument is the number of packets         */
     npkt = atoi(argv[1]);
  if (argc > 2)                   /* 2nd argument is the packet size               */
     pktsz = atoi(argv[2]);
  if (argc > 3)                   /* 3rd argument is the buffer size               */
     bufsz = atoi(argv[3]);
  if (argc > 4)                   /* 4th argument is the extra send delay          */
     send_delay = (double)atoi(argv[4]);
  if (argc > 5)                   /* 5th argument is the extra receive delay       */
     recv_delay = (double)atoi(argv[5]);

  NetworkSetFlitDelay(1);         /* All delays set to 1 times cycle time          */
  NetworkSetMuxDelay(1);
  NetworkSetArbDelay(1);
  NetworkSetDemuxDelay(1);
  NetworkSetPktDelay(1);
  NetworkSetWFT(NOWAIT);

  inport0 = NewIPort(10, 2);      /* Create input ports                            */
  inport1 = NewIPort(11, 2);
  buf0 = NewBuffer(20, bufsz,1);  /* Create buffers                                */
  buf1 = NewBuffer(21, bufsz,1);
  demux0 = NewDemux(30, 2, router);  /* Create demultiplexers                      */
  demux1 = NewDemux(31, 2, router);
  mux0   = NewMux(40, 2);         /* Create multiplexers                           */
  mux1   = NewMux(41, 2);
  outport0 = NewOPort(50, 2);     /* Create ouput ports                            */
  outport1 = NewOPort(51, 2);

  /* Connect modules to make the network:                                          */
```

```
  /*   This network is a 2 x 2 switch realized as a crossbar with buffers on its      */
  /*   outputs.  It consists of two 2-output demultiplexers followed by two 2-input   */
  /*   multiplexers                                                                     */

  NetworkConnect(inport0,buf0,0,0);   /* Input port 0 connected to buffer 0           */
  NetworkConnect(inport1,buf1,0,0);   /* Input port 1 connected to buffer 1           */

  NetworkConnect(buf0,demux0,0,0);    /* Buffer 0 connected to demux 0                */
  NetworkConnect(buf1,demux1,0,0);    /* Buffer 1 connected to demux 1                */

  NetworkConnect(demux0,mux0,0,0);    /* Demux 0, term 0 connected to mux 0, term 0   */
  NetworkConnect(demux0,mux1,1,0);    /* Demux 0, term 1 connected to mux 1, term 0   */
  NetworkConnect(demux1,mux0,0,1);    /* Demux 1, term 0 connected to mux 0, term 1   */
  NetworkConnect(demux1,mux1,1,1);    /* Demux 1, term 1 connected to mux 1, term 1   */

  NetworkConnect(mux0,outport0,0,0);  /* Mux 0 connected to output port 0             */
  NetworkConnect(mux1,outport1,0,0);  /* Mux 1 connected to output port 1             */

  process = NewProcess("UserSend0",UserProcessS,0);  /* Create sender process 0       */
  ActivitySetArg(process,NULL,0);                    /* Pass process its id           */
  ActivitySchedTime(process, 0.0, INDEPENDENT);      /* Schedule process              */

  process = NewProcess("UserSend1",UserProcessS,0);  /* Create sender process 1       */
  ActivitySetArg(process,NULL,1);                    /* Pass process its id           */
  ActivitySchedTime(process, 0.0, INDEPENDENT);      /* Schedule process              */

  process = NewProcess("UserRecv0",UserProcessR,0);  /* Create receiver process 0     */
  ActivitySetArg(process,NULL,0);                    /* Pass process its id           */
  ActivitySchedTime(process, 0.0, INDEPENDENT);      /* Schedule process              */

  process = NewProcess("UserRecv1",UserProcessR,0);  /* Create receiver process 1     */
  ActivitySetArg(process,NULL,1);                    /* Pass process its id           */
  ActivitySchedTime(process, 0.0, INDEPENDENT);      /* Schedule process              */

  NetworkCollectStats(NETTIME,NOHIST,0.0,0.0);       /* Collect Statistics            */
  NetworkCollectStats(BLKTIME,NOHIST,0.0,0.0);
  NetworkCollectStats(OPORTTIME,NOHIST,0.0,0.0);
  NetworkCollectStats(MOVETIME,NOHIST,0.0,0.0);
  NetworkCollectStats(LIFETIME,NOHIST,0.0,0.0);

  DriverRun(0.0);                                    /* Start the simulation          */

  NetworkStatRept();                                 /* Print statistics              */
}
```

APPENDIX 1: DEFINED SYMBOLS

This appendix list symbols defined in the file sim.h that are used as arguments to NETSIM operations. They are described in the discussions of the operations that use them and are only listed here.

Routing Modes:

NOWAIT	0	
WAIT	1	

Statistics Constants:

NETTIME	1	
BLKTIME	2	
OPORTTIME	3	
MOVETIME	4	
LIFETIME	5	
HIST	2	(Also used in YACSIM)
NOHIST	3	(Also used in YACSIM)

APPENDIX 2: SUMMARY OF OPERATIONS

Network Construction Operations

BUFFER *NewBuffer(id,sz)	/* Creates & returns a pointer to a new network buffer */
MUX *NewMux(id,fanin)	/* Creates & returns a pointer to a new multiplexer */
DEMUX *NewDemux(id,fanout,routingfcn)	/* Creates & returns a pointer to a new demux */
IPORT *NewIport(id,sz)	/* Creates and returns a pointer to a new iport */
OPORT *NewOPort(id,sz)	/* Creates and returns a pointer to a new oport */
void NetworkConnect(source, dest, src_index, dest_index)	/* Connects two modules */

Network Delay Operations

void NetworkSetCycleTime(x)	/* Sets the cycle time. All other times a multiple of it */
void NetworkSetFlitDelay(d)	/* Sets the flit delay */
void NetworkSetMuxDelay(d)	/* Sets the MUX delay */
void NetworkSetArbDelay(d)	/* Sets the multiplexer arbitration time */
void NetworkSetDemuxDelay(d)	/* Sets the demultplexer routing delay */
void NetworkSetPktDelay(d)	/* Sets the packet delay */

Network Threshold Operations

void NetworkSetThresh(i)	/* Sets buffer threshold */
void NetworkSetWFT(t)	/* Sets wait-for-tail flag: t = WAIT or NOWAIT */

Packet Operations

PACKET *NewPacket(seqno,msgptr,sz,src,dest)	/* Creates & returns a ptr to a packet */
double PacketSend(pkt,port)	/* Sends a packet through a network input port */
PACKET *PacketReceive(port)	/* Receives a packet from a network output port */
PKTDATA *PacketGetData(pkt)	/* Returns a pointer to a packet's user accessible data */
void PacketFree(pkt)	/* Returns a packet to the pool of free packets */

Packet Synchronization Operations

SEMAPHORE *IPortSemaphore(iptr)	/* Returns a pointer to an iport's PortReady sema */
int IPortSpace(port)	/* Returns # of free packet spaces in an iport */
int IPortGetId(port)	/* Returns user assigned id of an iport */
SEMAPHORE *OPortSemaphore(optr)	/* Returns a pointer to an oport's semaphore */
int OPortPackets(port)	/* Returns # of packets available in an oport */
int OPortGetId(port)	/* Returns the user assigned ID of an oport */

Network Statistics Operations

void NetworkCollectStats(type,histflg,nbin,low,high)	/* Activates automatic statistics collection for the network */
void NetworkResetStats()	/* Resets all network statistics records */
STATREC *NetworkStatPtr(type)	/* Returns a pointer to a network statistics record */
void NetworkStatRept()	/* Prints a report of network statistics */

APPENDIX 3: ALPHABETICAL OPERATION LIST