A Universal, Dynamically Adaptable and Programmable Network Router for Parallel Computers

Taras I. Golota and **Sotirios G. Ziavras,** NJIT

Abstract - Existing message-passing parallel computers employ routers designed for a specific interconnection network and deal with fixed data channel width. There are disadvantages to this approach, because the system design and development times are significant and these routers do not permit run time network reconfiguration. Changes in the topology of the network may be required for better performance or fault-tolerance. In this paper, we introduce a class of high-performance universal (statically and dynamically adaptable) programmable routers (UPRs) for message-passing parallel computers. The universality of these routers is based on their capability to adapt at run and/or static times according to the characteristics of the systems and/or applications. More specifically, the number of bidirectional data channels, the channel size and the I/O port mappings (for the implementation of a particular topology) can change dynamically and statically. Our research focuses on system-level specification issues of the UPRs, their VLSI design and their simulation to estimate their performance. Our simulation of data transfers via UPR routers employs VHDL code in the Mentor Graphics environment.The results show that the

performance of the routers depends mostly on their current configuration. Details of the simulation and synthesis are presented.

Keywords: Parallel computer, message-passing computer, message routing, hardware router, universal router, adaptable router.

I. INTRODUCTION

There is significant interest in massively-parallel processing (MPP) systems with hundreds or thousands of processors. Scalable parallel computers are the major trend in current high-performance computer architecture. The ease of scaling makes message-passing (i.e., point-to-point) interconnection networks more feasible than purely bus-interconnected multiprocessors [2, 3] for MPP. Typically, a message-passing parallel computer is composed of 64 to 1024, or more, computational nodes connected in a network. Messages are sent from a processing element (PE) to its associated router. It is the router that facilitates the transfer of the message to its destination. The router can essentially become a performance bottleneck in scalable parallel computers. There are many factors which determine how the messages flow inside MPP systems. Most of the MPP systems are based on the distributed shared-memory (DSM) approach. The advantage of DSM computers is their scalability; every PE can access the memory of any other PE by issuing appropriate messages.

A typical PE (node) of the MPP system is a computational node with a high-performance CPU, local memory and a direct memory access (DMA) controller. Other nodes are accessible

via the interconnection network using the attached network router. The local bus conforms to the processor's protocol, and the communication coprocessor (router) converts the local bus protocol to the network protocol. The DMA controller performs direct memory access to the local memory for large messages and, if necessary, automatically does packet (de)composition (converts messages to and from the network protocol).

The routers in a message-passing parallel computer are interconnected to form a predefined topology. The channels are usually wired connections between adjacent routers and the topology of the wired connections could vary from system to system. The most widely used topologies are the mesh, hypercube and torus, each of them having its advantages and disadvantages. The (direct binary) hypercube was the most popular interconnection network in the 1980's because it can emulate other widely used topologies quite efficiently [2, 3, 5]. However, the hypercube is not scalable in practice. Therefore, several hypercube topological variations with reduced hardware complexity have been proposed [3]. The 3-D torus topology is very widely used nowadays [4]. An algorithm may assume several topologies during its execution in order to yield good performance. If the network cannot be reconfigured, then the required topologies are mapped onto the network topology using, most often, suboptimal solutions.

The two most popular techniques for routing messages are packet switching and circuit switching [5]. In packet switching, a message is broken into small packets that are transmitted through the network in the “store-and-forward” mode. Each time a packet traverses a link, the receiving node examines it to decide what to do. It may have to store the packet for a while before forwarding it toward its final destination. It is possible that packets of a single message will traverse different sets of links on their path from the source to the destination. Packets may experience delays at each switching point, depending on the traffic in the network. The circuit

switching technique first establishes a complete path between the source and the destination, and then starts transferring the message along this path. The circuit is kept open until the entire message has been transmitted. Without loss of generality, we implement wormhole routing [4], an improved version of pipelined circuit switching. We assume wormhole routing with short packets of four flits (flow-control digits). A 32-bit channel can accommodate a single flit in each communication cycle. Wormhole routing makes the message-transmission time almost indifferent to the length (number of hops) a message travels. It requires less hardware than other routing approaches [4]. Slight modifications in the design of our router could facilitate also the implementation of more complex communication operations (e.g., multicast [6], data reduction [11, 15]). Our router could support broadcasting [16] by connecting a single input to a contiguous range of outputs, as specified in a configuration table.

Minimal adaptive routing techniques allow some choice among several minimal-length paths, based on local or temporary conditions at the nodes. Usually the router determines the link (the output port) which belongs to a minimal path for an incoming packet. If the port is busy, then the packet waits for the transfer. If other ports are available, the router may choose from alternative links in a random, dimension-order manner. However, adaptive routing may produce deadlocks [4]. For dimension-order routing, a general problem is that the number of available paths decreases as packets come closer to their destinations. Non-minimal adaptive routing techniques allow packets to be routed on any, not necessarily minimal, path from the source to the destination. The major reasons to allow packets not to choose minimal paths are to avoid congestion nodes and to support fault-tolerance. If all minimal paths are congested but longer paths are available, then the packets are sent over longer non-congested paths. This technique has several problems related to deadlock and livelock prevention [5]. The Chaos router applies

randomized, non-minimal adaptive packet routing [8]; randomization eliminates the need for livelock protection based on probabilities. Our design supports deadlock-free routing by employing a static dimension-order technique (i.e., a distinct priority is assigned to each dimension in the topology). In contrast to other approaches that avoid deadlocks via complicated schemes (such as virtual queues) [4, 6], we apply an oblivious/deterministic routing approach to simplify the design (without any loss of generality). According to [18], current routers must be modified extensively to support fault-tolerant routing. Such modifications have been proposed in the latter paper for dimension-order routers. In fact, they are applicable to dimension-order routers without central crossbar which, in addition, can be partitioned into multiple modules (one module per dimension). Parallel computers could also satisfy the large computational demands of real-time applications. Real-time applications require a predictable communication network and good average performance. A router for such applications was presented in [19]. The router implements bandwidth regulation and deadline-based scheduling, while permitting best-effort traffic to capitalize on low-latency routing and switching schemes.

Three, primarily, design goals govern our project, namely universality with static and dynamic adaptability, programmability and high throughput with low latency. Presently, multicomputers do not employ adaptable routers. The decision to design a universal/adaptable router stems from the need to build it independently of any target message-passing parallel computer [7]. The success of this effort will facilitate faster design, test and development of parallel computers, and better computer performance (due to the support of dynamic reconfiguration of the topology and/or channel width). We assume that such routers are destined for DSM computers [7], which operate efficiently on a wide range of problems because of their capability to access memory with both global and local addresses. A relevant effort for the

design of Internet routers that can change configuration "on-the-fly," based on the applications' demands, has been initiated by the Active Networks DARPA project. However, a similar effort has yet to appear in the parallel computing arena.

The paper is organized as follows. Section II presents an overview of our proposed class of routers, including packet formats and the routing procedure. Section III presents the proposed architecture in detail. Section IV presents simulation results and addresses issues related to fault tolerance. Finally, conclusions are presented in Section V.

II. OVERVIEW OF THE UNIVERSAL AND PROGRAMMABLE (UPR) ROUTERS

The design of our UPR routers is consistent with the VLSI technology trend [8] of increased capabilities for single chips [17]. The programming model of the router at the macro level [9] is very efficient for statically and dynamically reconfigurable systems. Currently, custom-made network routers are commonly used in parallel computers and the reduction of communication latencies in DSM scaleable systems is one of the most critical problems that requires a gracious solution. However, only recently have researchers been designing hardware routers for good performance [8, 10-12]. Nevertheless, the main problem with these routers is that they are designed for specific systems and/or architectures [13]. Most often, these designs are for the mesh/torus architecture [4, 10, 14]. Also, despite the new trend in the design of scaleable parallel computers, that uses commercial_off_the_shelf (COTS) components to reduce the cost and increase the lifetime of a system, this has not been yet followed for router chips. We strongly believe that universal (statically and dynamically adaptable) programmable routers are absolutely

essential in the parallel computing field. The current adaptability of routers has to do only with nondeterministic routing and fault-tolerance issues [18]. In the former case, virtual channeling of messages for wormhole-routing systems is not even implemented in the most efficient manner due to time-sharing of channels [4]. Universal routers are absolutely essential mainly for two reasons. Firstly, they can be used to construct any parallel computer, independently of its topology and channel width, because of their static adaptability. Secondly, their dynamic adaptability can be taken advantage of to efficiently reconfigure the topology and channel widths of the parallel network, more than once, in order to match the requirements of the application throughout its implementation.

We propose a programmable and adaptable router that can be used in any design, independently of the chosen interconnection network. This router uses a programmable lookup table to map processor addresses to physical network routes. By maintaining configuration (lookup) tables within the PE memory or inside the router, it becomes easy to modify the network topology based on changing workloads, network failures or requests from the application algorithm. State-of-the-art routers are of simple design [1], but they are inflexible. With our current VLSI design, the UPR router can be adapted to become a full 32-bit 8×8, or 16-bit 16x16 or 8-bit 32x32 cross-point router. (That is , the number of channels as well as their width can be adapted at static or run time. Allowable channel widths are 8, 16 and 32 bits. Therefore, we can have thirty-two 8-bit bidirectional channels, sixteen 16-bit bidirectional channels or eight 32-bit bidirectional channels, respectively.)

Routing decisions are made locally by the UPR router based on the destination address stored in the packet header and the availability of outgoing channels. Packets are stored in the corresponding wait queue if that outgoing channel is full. We assume that each PE has four main

components, namely an UPR, a DMA controller, local memory (RAM) and a general-purpose processor (e.g., Pentium). Each one of the last three components can be placed on a separate VME board and all three boards can interface the same VME bus. The host processor supplies various commands and/or data for a message to be transmitted with handshaking. The responsibilities of the DMA controller that sits between the processor and the router are:

- configuration of the router with information provided by the host processor;
- composition of fixed-length packets in the format acceptable by the router;
- decomposition of messages received from other PEs;
- message passing between the host processor and the router;
- accessing the local memory without intervention of the host processor for long message transmission (the processor supplies to the DMA controller the starting address of the message in the RAM and the length of the message, and the DMA controller composes fixed-length packets for the router by fragmenting the long message).

Figure 1 shows the top-level architecture of the router with eight 32-bit bidirectional data channels. The 8-bit channels 28 through 31 are used to interface the local DMA controller/coprocessor as shown in Figure 2. There are two separate signal lines, PUSH_DATA_IN (input data ready) and FULL_OR_ACK_OUT (input buffer full or acknowledge for data read) for each 8-bit input channel. Generally, each 8-bit bidirectional physical channel has 20 lines (8-bit data bus input, PUSH_DATA_IN, FULL_OR_ACK_OUT, 8-bit data bus output, POP_DATA_OUT and FULL_OR_ACK_IN). PUSH_DATA_IN and FULL_OR_ACK_IN are input signals for a channel, whereas POP_DATA_OUT and FULL_OR_ACK_OUT are output signals from a channel. Therefore, the router has a total of 32x20 + 3 pins. The three extra pins are for the system clock, the system reset and the

configuration of the router. Therefore, the total number of functional pins is about 650. This number is easily implementable with state-of-the-art IC technology.

Figure 1 goes here.
Figure 2 goes here.

A. Packet Formats

We assume that a *message* is the basic form of "shipment" between the nodes. Any large message can be transmitted by dividing it into fixed-size *packets* that can be sent to the destination individually. With wormhole routing, a packet is further divided into 32-bit *flits*. Currently, our UPRs support two different types of packets, namely *normal packets* with fixed format for actual message passing and *initialization packets* for router configuration. Any kind of message could be built from such packets. Each normal packet contains four 32-bit flits, as shown in Table I. The first 32-bit flit of any packet is generally referred to as the *header*. The packet header contains a 14-bit destination address, a 14-bit source address and 4-bit control information (a 1-bit status field to handle the misrouting of messages blocked by faults and three bits reserved for future modifications). The 14-bit destination address field allows us to address a maximum of 2^{14} = 16,384 distinct PEs. This number is quite large and practically allows the implementation of any MPP machine. The second 32-bit flit of the packet contains a memory address for the destination PE. Therefore, the maximum size of a single PE's memory is 2^{32} = 4 Gwords (64-bit words). The last two 32-bit flits of the packet contain actual message data. One

could customize the packet format to meet specific system requirements. However, all normal packet formats must start with a 14-bit destination PE address.

Table I goes here.

The *initialization packet* is used to configure the router according to the chosen data channel width and the topology of the interconnection network. Initialization packets can be either 8 or 16 bits wide. The first two bits of any initialization packet contain control information. Four possible initialization packet formats follow. The *node address initialization packet* is used to configure the physical address of the router; this information is stored in a 14-bit node address register in the router. The physical address of the router depends on its position in the chosen interconnection network topology. The packet format is shown in Table I.

The *topology initialization packet* (Table I) is used to configure the operating topology and the data transfer mode for the router. Without loss of generality, three topology bits identify the current interconnection topology. They are stored in the 3-bit topology register. Configuration information for up to eight topologies can be loaded into the router during initialization. Of course, the set of eight "valid" topologies can be modified at run time as many times as needed by the application algorithm. Two transfer mode bits, stored in the transfer mode register XFER_MODE, specify the data channel width for ports. Tables II and III show the sets of topologies and transfer modes used throughout our presentation. The modular design of our routers supports increases in these populations. In the case of 16-bit data channel width, pairs of consecutive flits in the packet are supplied simultaneously to pairs of consecutive 8-bit channels. Similarly, in the case of 32-bit data channel width, all four flits of the packet are distributed

among four consecutive 8-bit channels. For the sake of simplicity, the current design assumes groupings of even numbers of channels.

Table II goes here.

Table III goes here.

The *dimension register initialization packet* (Table I) configures the dimension register with information that shows the separation of the different coordinates in the 14-bit address. Starting with the least significant bit, a zero bit in the dimension register indicates a change in dimension. For example, if the dimension register contains 01111111011111 for the 2-D topology, then the 1^{st} dimension of the network can have up to $2^6 = 64$ physical nodes and the second dimension can have up to $2^8 = 256$ physical nodes. So, the total number of nodes in the system can be up to 64x256 = 16,384.

The addressing of routers is done with respect to the current topology register and the dimension register. In our example, our topology register contains two and the dimension register contains 01111111011111. Assume the node N located at position 6 in the first dimension and position 3 in the second dimension. That is, its (x, y) coordinates are equal to (6,3). The information stored in the node address register will then be $00000011|000110_2 = 3|6 = 198$. (The first dimension starts from the right and the lowest coordinate in each dimension is zero.)

The *configuration table initialization packet* (Table I) writes a location of the configuration table with appropriate data. The router configuration table in our current implementation is a 256x5 RAM that actually stores connection information for seven different interconnect

topologies and channel widths. Therefore, the router configuration can change at run time to match the needs of the application, without wasting significant time to receive reconfiguration information from the host processor.

The table is logically divided into eight 32x5 sections. The first section is reserved for future use. The 2nd section is for the 1-D topology, the 3rd section is for the 2-D topology, and so on (see Table II). This way we could get up to a 7-D system. Each section of the table contains actual physical connection information that maps input ports to output ports for the respective topology. Extending our previous example, the configuration table section for the 2-D topology with 8-bit data channels and for 2-D with 16-bit data channels may contain the data shown in Tables IV and V, respectively:

Table IV goes here.

Table V goes here.

B. Routing Algorithm

Dimension-order routing, starting with the first dimension, is used. When the packet reaches its destination, it is routed to the CPU port. For example, consider 8-bit data channel width, a 2-D topology with dimension register 01111111011111 and local node address 3|6. The incoming packet to this node has destination address 00000100|000100 = 4|4. The address difference in the first dimension is 4-6=-2. So, the packet is routed to the port shown as -1 in the table. (Because the difference in the 1st dimension is negative.) According to the table, the packet will be

transferred to output port 1. In the 2-D mesh, this output port 1 is connected in the first dimension to the node 3|5. At 3|5, the difference in addresses is 4-5=-1, so the packet is transferred to output port 1. This output port connects to node 3|4. At 3|4, the difference of node addresses in the first dimension is zero. Therefore, the packet is routed in the second dimension. The difference of node addresses in the second dimension is 4-3=+1, so the packet is transferred to the port shown as +2 in the table. Again referring to the configuration table for 2-D, the +2 pointer points to output port 2. This output port 2 is connected in the second dimension to the node 4|4. The difference of node addresses in both dimensions is now zero. So, we have to route this packet to the CPU port. We have 8-bit data channel width, so the packet will be transferred to output port 28, which is connected to the DMA controller at the destination PE.

Fault-tolerant routing is also supported by UPRs. An output port may be blocked because of a faulty channel or its attached router. A watchdog timer in the router can identify faulty connections and force the router to reroute the packets. The watchdog timer initiates a process to set the least-significant bit in the Control field of the packet's first flit to 1. Without faults, this bit is 0. If this bit is 1, then the router skips dimension-order routing and chooses randomly any of the other dimensions that have to be traversed; if no other dimension has to be traversed, then it sends the packet randomly to any other connected port. The algorithm for fault-tolerant routing is presented in Section IV.

III. ROUTER ARCHITECTURE

All hardware blocks were designed using the VHDL language in the Mentor Graphics environment, so these blocks are realizable in hardware after synthesis is run. The entire design of the router is hierarchical and its main components are presented below.

A. Initializer

The initializer (Figure 3) is used to configure the router's behavior for specific network topology and data channel width on a system-wide basis. Initially, each host processor gets the configuration information from the I/O devices. At run time, new configurations may be obtained with normal packets. The second lower bit in the Control field of a normal packet is permanently set to 1 by a router if dimension-order routing is skipped because of a faulty link or node. The router may make a copy of this modified packet into its local processor, for the latter to initiate identification of the faulty resource. After identification, a new configuration may be set to avoid the faulty resource. Generally, overall system stability will not be penalized after the failure of a few components and the system can be reconfigured easily during normal operation.

Figure 3 goes here.

The initialization of the router begins with a high INIT signal from the DMA controller of the host processor (Figure 4). During initialization the 8-bit data transfer mode is used and port 28, that connects the DMA controller to its router, is active. Any ongoing routing of normal packets is put on hold. The host processor holds the INIT signal high until initialization is finished. The initializer block in the router reads data from the input buffer INBUF_DATA_28 and acknowledges it by making high READ_DONE_28. It generates signals to disable the

operation of all input and output buffers, except INBUF_DATA_28. This is required to avoid any routing during initialization.

Figure 4 goes here.

The initializer decodes the control bits of each incoming packet and activates the appropriate section responsible for that packet. It stores topology and channel width information, and local node address and dimension register value locally, and generates appropriate write signals for the main configuration table. After the host has initialized the main configuration table of its own router, it deactivates the INIT signal. With INIT low, the initializer proceeds with the initialization of the port tables. Each port has a 32x5 port table that contains +/- destination port addresses for the current topology; all ports have the same copy of the port table, and it is identical to part of the configuration table (see examples in Tables IV and V). The initializer loads a 1/8-th part of the main configuration table into these port tables according to the topology. During this upload, appropriate disable signals (DISABLE_ PORTS: signal to deactivate the operation of input and output buffers) are generated to deactivate all input and output ports, including INBUF_DATA_28. This avoids false initialization and undesirable routing. After the intializer is done with port table initialization, it enables all router ports for normal operation.

B. Input Port

The structure of the input ports is shown in Figure 5. Each 8-bit input port block contains the following components:

- an input buffer which can store up to four bytes of incoming packets (the router implements wormhole routing with 32-bit flits); its structure is shown in Figure 6;
- an output buffer which can store up to four bytes;
- a counter to count the number of bytes in incoming packets;
- a specially designed subtractor which generates the difference between the local PE's address and the destination address extracted from the packet header;
- a port mapping table for routing messages;
- a dimension decoder which generates the address of the output port to which the current packet has to be routed;
- a priority generator which assigns priority to each incoming packet in case of collision; and
- a 5x32 decoder which generates a control signal for the 32x32 crossbar switch that connects input buffers to output buffers.

Figure 5 goes here.

Figure 6 goes here.

B.1. Input Buffer

The input buffer (Figure 6) works in synchronization with the attached router and sends data to a circular queue. It also works in synchronization with output buffers and the crossbar for data

reads from the circular queue. The input buffer has the same modules for every 8-bit component channel. The input buffer entity has a demultiplexer, a circular queue, a multiplexer and a control block. For the three possible configurations of the input buffer width (8, 16, 32), there are also inputs from the synchronization block. The circular queue is used to buffer a maximum of four bytes. The one-to-four 8-bit bus demultiplexer is used to send a byte of information into an empty 8-bit queue register, according to the 2-bit "tail address" control signal from the control block. Four-to-one 8-bit bus multiplexers are used to send a byte of information from the first "filled" 8-bit queue register to the crossbar register, according to the 2-bit "head address" control signal. The control block keeps track of the changing tail/head addresses after getting data into the queue registers or sending them out of the queue to crossbar registers. At the same time, this block generates the correct select addresses for the de-multiplexer and multiplexer. It also maintains "buffer full" and "buffer empty" signals for the other blocks of the router. A circular queue of four 8-bit input registers is implemented by means of four registers in parallel, and "tail" and "head" pointers pointing to the next available register for writing and reading a byte of data, respectively. The control block keeps track of the "tail address" for the de-multiplexer according to empty queue registers. Once the data becomes available in the input of the de-multiplexer, it is written into the addressed 8-bit queue register. If all four queue registers are empty, then the "head address" and the "tail address" have the same value of 00; HEAD and TAIL point to the same location.

A write operation (Figure 7) is initiated by a high PUSH_DATA_IN (Data_ready) signal. Note that even though a write occurs after detection of high for PUSH_DATA_IN, the write operation is not level sensitive. This block will write data to the queue after detecting a high Data_ready signal and will wait for that signal to toggle again in order to perform another write

operation. Also, the write operation is not edge sensitive. Internally, the block generates a write clock that is synchronized with the system clock and this write clock is used for the actual write operation. The condition to initiate a write is FULL_OR_ACK=0. It is up to the external entity to drive valid data and activate Data_ready upon detecting a low FULL_OR_ACK. Also, Data_ready should stay high until the write operation is over.

Figure 7 goes here.

The TAIL address has a two-fold purpose. It is used to route input data to an available register for a write operation and to select an available register for it. The TAIL address is updated after each successful write operation in order to point to the next available register. If the buffer becomes full at any time, then FULL_OR_ACK is driven high until the buffer again has some available space to write into. The buffer full condition occurs when TAIL and HEAD point to the same register and the buffer is not empty.

The read operation (Figure 8) is simple relative to the write operation. Three signals are associated with the read operation. A high INBUF_VALID (data ready for the decoder/crossbar block) signal indicates valid data in the queue. The decoder/crossbar block read data from INBUF_DOUT (data byte output to decoder/crossbar block) upon detection of a high INBUF_VALID. The input buffer also expects an acknowledge signal INBUF_LOADED after the read operation is complete. The HEAD pointer always points to the first byte in the queue. It is updated after detection of a high INBUF_LOADED in order to point again to the start of the queue. The INBUF_VALID signal is zero if the queue is empty.

Figure 8 goes here.

B.2. Destination Address Loader

The destination address loader (Figure 9) is used to extract the destination address from the header of the packet. It also takes care of different data transfer modes in order to synchronize data transfers. This block handles four 8-bit channels at a time. The first two bytes of each packet received contain the destination address in the case of 8-bit data transfers. The first byte of a packet received by a channel contains part of a destination address in the case of 16-bit data transfers. The first byte of channel-0 and channel-1 contains part of a destination address for 32-bit data transfers.

Figure 9 goes here.

The destination address loader of a 32-bit input channel has four 14-bit registers to store temporary destination addresses. Each of these registers is loaded with part or all of the destination address, depending on the data channel width. In the case of 8-bit data transfers, all registers are loaded with the first two bytes of the header data received from each channel. That is, all registers will contain the destination address for the respective channels. In the case of 16/32-bit data transfers, the most-significant byte of even-numbered registers (D0 and D2) is loaded with the first data byte of even-numbered channels (DATA_0 and DATA_2). The least-significant byte of even-numbered registers (D0 and D2) is loaded with odd channel data (DATA_1 and DATA_3). Even-numbered registers contain destination addresses that are valid for the 16-bit data transfer mode. Also, in the case of 16/32-bit data transfers, odd-numbered registers (D1 and D3) are loaded with the first two bytes of odd channel data. Odd-numbered registers contain the destination address that is valid for the 8-bit data transfer mode. The actual

destination address is derived using these temporary registers and taking into account the current channel width.

B.3. Address Subtractor

The address subtractor (Figures 10 and 11) is used to make a routing dicision. It is a 2's complement subtractor. It makes use of the dimension register in order to avoid unnecessary carry propagation delay and decide about non-zero result accumulation for individual coordinates. The inputs to the subtractor are the 14-bit local node address, the 14-bit destination address and the 14-bit dimension register. The outputs of the subtractor are a 14-bit result and 14-bit status information. The subtractor is composed of 14 identical 1-bit "adder" slices and inverters. It converts the local address into the 2's complement form, by inverting it and using a carry-in of 1 (Figure 10 produces the 2' complement of the needed result whereas A in Figure 11 stands for A').

Figure 10 goes here.

Figure 11 goes here.

By looking at Tables VI through VIII and the block diagram, it is clear that the result R is the same as that of a conventional adder. The output carry is propagated only if D is 1, otherwise the carry is killed. ACCin reflects a non-zero result in the previous adder slices for the current dimension. If D=1 and ACCin=1, then ACCout will be 1 irrespective of the current sum. If D=1 and ACCin=0 then ACCout=Sum. Also, from the block diagram the S bit reflects a non-zero

result in all previous slices in the current dimension, including the current slice. By looking at just one S bit, we know about a non-zero result in the current dimension.

Table VI goes here.

Table VII goes here.

Table VIII goes here.

The example in Table X for a 2D system illustrates the importance of the S bit. The local node address in the 2-D 128x128 system is 10100110000011_2 =83|3 and the destination address is 01010011011011_2=41|91. The 2^{nd} S bit is zero reflecting the fact that the result up to the 2^{nd} bit is zero. Also, look at the 5^{th} bit. It is 1, showing that the result up to the 5^{th} bit is non-zero. Recall that we are using dimension-order routing, starting with the lowest dimension. Since the result for the lower dimension is positive, the routing dimension is **+ve** according to Table IX (with i=6).

Table IX goes here.

Table X goes here.

The implemented subtractor is a carry-ripple subtractor (i.e., each bit slice introduces delay for carry propagation). But the carry from the previous bit slice is actually ANDed with a bit from the dimension register and is then given to the next bit slice. Therefore, wherever we have a zero bit in the dimension register we are actually dropping the previous carry and generating a

carry for the next bit slice equal to 0. Thus, the carry keeps propagating only for a continuous chain of 1's in the destination register. This chain of 1's is broken at the point where the dimension of the interconnection topology changes.

B.4. Dimension Decoder Block

This block generates the output port number to which a normal message should be routed. Without loss of generality, it can currently handle an interconnection network with up to four dimensions. Its PORT_TABLE (Figure 12) contains part of the main configuration table for the current topology and data channel width. It is composed of decoders, tri-state buffers and few AND gates.

Figure 12 goes here.

Dimension decoding is done without wasting any clock cycles for shifting. The dimension decoding logic is composed of four prioritized blocks, one for each dimension. The logic can be extended to support more operating dimensions. Each priority block contains two DRS blocks (Figure 13), a 4x16 decoder, two tri-state buffers and four AND gates. The DRS block reads particular bits of the dimension register, result register and status register according to a 14-bit control word supplied to it. Its structure has only tri-state buffers. It will output the i^{th} bit of all three registers if the i^{th} line of the control word is high. The dimension decoder block is shown in Figure 14.

Figure 13 goes here.

Figure 14 goes here.

Each priority block (Figure 15) receives PREVIOUS_NOT_SELECT_IN from the preceding dimension. A high signal means that dimension decoding has finished in the preceding dimension so that nothing will be decoded in the current dimension. If the dimension decoding in the previous block is not done, the block for the current dimension will perform decoding.

Figure 15 goes here.

Each priority block receives its DIM_CHANGE_TABLE data and DC_VALID bit. The former data is the relative coordinate (result of subtraction) in the current dimension. The latter bit is 0 if this data is zero, otherwise it is 1. Each priority block receives data from PORT_TABLE. The i^{th} dimension receives PORT_TABLE_DATA for the +i and –i output ports. The decoder decodes DIM_CHANGE_TABLE data to generate the control word for the DRS block in order to read the required bit of the dimension, result and status registers. If result(i)=0 and status(i)=1, then the tri-state buffer for the +i PORT_TABLE_DATA is activated. If result(i)=1 and status(i)=1, then the tri-state buffer for the -i PORT_TABLE_DATA is activated. If none of the above conditions is satisfied for the current dimension, then a low PREVIOUS _NOT_SELECT_OUT signal is generated so that the next dimension can proceed with the same kind of decoding process. If no dimension can satisfy any of these two conditions, then the tri-state buffer for the local (host) port will be activated at last. The outputs of all tri-state buffers are connected to the same bus. Only one tri-state buffer can drive the bus at a time. This bus is enabled by the DEST_ADDR_VALID signal (Figure 14) and the dimension decoder

generates a valid output port number only when the destination extractor has a valid destination address.

B.5. Priority Generator

The priority generator generates priority signals for all input ports to prevent collision when two input ports request the same destination port. The priority of each channel is the same as its ID, so that channel 0 is assigned the highest priority and channel 28 is assigned the lowest priority.

B.6. Port Decoder Block

It is based on a 5x32 decoder and generates control signals for the crossbar switch. It differs from a classical decoder because it latches the control word generated, and applies handshaking with the destination extractor and the crossbar switch. This block has a 32x1 input multiplexer, two registers and few logic gates (Figure 16). The multiplexer generates a valid control word if the "crossbar busy" SWITCH_ON bit is zero for the required output port (a busy signal from the crossbar indicates ongoing packet routing in the output port requested). When this 32-bit control word, namely CONTROL_I, is valid, only its i^{th} bit, where $0 \leq i \leq 31$, is 1 to indicate the need for a transfer to output port i; all other bits are 0. For the connection to be established with the output port, EFFECTIVE_PRIORITY should be 1 to avoid any collision (if the priority bit is 1 for the current input port, this input port has higher priority over other input ports in the case of competition for the same output port).

Figure 16 goes here.

The SWITCHED_ON bit is internally set to 1 initially. It is set to 1 after one clock cycle of a valid internal CONTROL_I signal. This mechanism ensures that CONTROL_I is latched only once after it is generated. The signal SWITCHED_ON will flag the output buffer to start a data transfer from the input buffer. It will be again pulled low after the last byte of the current packet has been transferred from the input buffer to the output buffer.

The acknowledgement signal from the destination extractor is actually the SWITCHED_ON signal delayed by one clock cycle. Generation of a pulse means that a valid CONTROL signal for the crossbar has been generated. The destination extractor pulls the DEST_ADDR_VALID signal low after it detects this pulse. Finally, OUTBUF_LAST_BYTE in Figure 16 is 1 if the last byte of the packet was loaded into the output buffer.

C. Crossbar

The router has one crossbar block (Figure 17). It switches an input channel to the appropriate output channel depending on the control word CONTROL_I received from the decoder of the input channel. The crossbar is composed of many tri-state buffers. Each tri-state buffer is controlled by a bit from a CONTROL_I word (from the 5x32 decoder) of an input channel. Input port data can be routed to any output port according to the current CONTROL_I words. The i^{th} output port is driven by the i^{th} bits of all CONTROL_I words. The BUSY signal (shown in Figure 16) is crossbar generated by bitwise ORing all 32 CONTROL_I words.

Figure 17 goes here.

D. Output Port

The structure of an output port (Figure 18) is very similar to that of an input port. This block works in synchronization with other routers for data reads from its circular queue and with the crossbar for data writes into its circular queue. The circular queue of four output registers is implemented with the “tail” and “head” pointers, as for the input ports. If all four queue registers are empty, then “head” and “tail” contain 00.

Figure 18 goes here.

If the input buffer has valid data and there is enough space in the output queue, then a write operation (Figure 19) into the output queue is controlled by (DEST)PORT_ADDR_READY. INBUF_READ_DONE is used to acknowledge the input buffer. The rising edge of INBUF_READ_DONE activates the write signal. The falling edge of INBUF_READ_DONE is used to update the "tail" pointer of the queue. During each write operation, the output buffer keeps track of empty space in the queue by updating OUTBUF_EMPTY. A high OUTBUF_EMPTY indicates empty space in the output queue. During each write operation, it also updates OUTBUF_VALID to flag the attached router about available data. As soon as the queue has valid data, the output buffer sets POP_DATA_OUT (data ready) to high and drives valid data on the DATA_OUT line (Figure 20). The attached router needs to complete the read operation and then generate an acknowledge signal. The output buffer will use this acknowledge signal to pull POP_DATA_OUT to low and update the "head" pointer of the queue. The output

buffer will not drive data out until FULL_OR_ACK is pulled low by the external entity. Therefore, FULL_OR_ACK (buffer full/acknowledge input data) serves the dual purpose of readiness and acknowledgement.

Figure 19 goes here.
Figure 20 goes here.

IV. FAULT-TOLERANCE AND SIMULATION RESULTS

Fault-tolerance in message passing is a critical problem [18]. As mentioned in Section II, messages blocked by faulty links (or their attached routers) are routed by our UPR via alternative paths. More specifically, if the lowest dimension to be traversed (according to dimension-order routing) is blocked, then the next higher dimension that must be traversed is chosen. The least significant bit in the Control field of the normal packet is then set to 1 to initiate a process of informing the host processor about the faulty channel. The ID of the latter channel is easily determined by the host processor based on the current configuration and the destination address. The host processor may then decide to write new data into the configuration table in order to bypass the faulty channel in future data transfers. If two or more dimensions correspond to faulty channels, then their total number is written into the Control field, so that the router can reroute the message by ignoring these dimensions. Relevant results are presented at the end of this section

We first simulated transfers of data packets for various combinations of binary hypercubes and data channel widths. The n-dimensional binary hypercube contains 2^n nodes [3, 5]. Two nodes are directly connected if and only if their n-bit addresses differ in a single bit. Each node has n neighbors and the diameter of the network is n. The simulations were performed using VHDL test benches. The simulations were carried out for two basic categories: initialization and normal operation. For the case of initialization, we estimated the time required to initialize the router for various test cases. The time required for router initialization is shown in Figure 21 for various instances of the binary hypercube topology. For fixed channel width, this time increases linearly with the number of dimensions.

Figure 21 goes here.

After thorough examination of the above results, we came up with an equation to estimate the initialization time for any case. The number of clock cycles for initialization is 15 + (4*d*w + 5)*3, where d is the number of dimensions in the system and w is the channel width in bytes.We assume that only channel 28 it used to load configuration information. 15 clock cycles are required to load information from the configuration table into all ports. The rest of the equation is for time required for the initialization of various configuration registers and the main configuration table. We need 3 clock cycles to load each byte into the router and that is why we multiply by 3. Now we will take a look at the bracket part of the equation. We have to load the 14-bit local node address, the 14-bit dimension register, the 2-bit data channel width register and the 3-bit topology register during initialization. Initialization of these registers needs 5 bytes of data. That is why we have a constant 5 inside the bracket part of the equation. We need two configuration table initialization packets (two bytes long each), one for the "+" port and another

for the "–" port, per dimension of each 8-bit data channel. For this reason, we have multiplied by 4 in the bracket part of the equation. The initialization time can be reduced significantly if a 16-bit or 32-bit channel is used to connect the router to the host processor during initialization.

In the simulation of normal packet transfers, we obtained the approximate time for different scenarios. The routing is considered done when all input packets come out of output ports. We calculated the routing time by considering packet collisions in all cases, without any fault on the links. In the following figure, the number of colliding input ports represents the number of input ports that are receiving simultaneously a packet with the same destination port address. Figure 22 shows the actual time consumed for routing. In general, the larger the number of colliding packets, the larger is the delay for routing – as predicted by theoretical analysis.

Figure 22 goes here.

Without collisions, every write into an input buffer takes one cycle, internal data transfer through the crossbar takes one cycle and a write into an output buffer takes one cycle. Without considering any type of routing, each packet (16 bytes long) takes 16*3 = 48 clock cycles to pass through the router for 8-bit channels. According to experimental results, it takes a few more cycles for actual data transfer. The router takes just two to four clock cycles per packet to make a routing decision depending on the data channel width. These are very impressive results for a universal, programmable router.

Let us now focus on simulation results for fault-tolerant routing. We carried out simulations for several systems comprising the binary hypercube and the torus interconnection networks. We simulated systems having from two to five dimensions. Therefore, the hypercube systems have 4, 8, 16, 32 and 64 nodes, respectively. For each torus instance, each dimension contains four

nodes. More specifically, we simulated symmetric 2-D, 3-D, 4-D and 5-D tori with 16, 64, 256 and 1024 nodes, respectively. Two bits are used for each coordinate. We chose for the simulations channel width equal to 16 bits. Our router can handle this width since it has 16 bidirectional 16-bit channels and we have assumed up to five dimensions for the simulations.

We used three random number generators with uniform distribution to determine: (a) the nodes that are to transmit packets, (b) the faulty channels and (c) the destinations for one-to-one data transfers. We present here results for cases where 10%, 20% or 40% of the nodes transmit packets. We also assume that 1%, 2%, 5% or 10% of the channels are faulty. Figures 23 through 28 show the slowdown resulting from faulty connections. These results show that our router performs very well even with large numbers of faulty links. Of course, the binary hypercube performs better than the torus with the same number of dimensions because it has fewer nodes.

Figure 23 goes here.

Figure 24 goes here.

Figure 25 goes here.

Figure 26 goes here.

Figure 27 goes here.

Figure 28 goes here.

V. CONCLUSIONS

We have presented the design and evaluation of a universal (statically and dynamically adaptable) hardware router for message-passing parallel computers. This router introduces outstanding flexibility. It is topology and channel width independent. In addition, the topology and channel width can change at run time, as many times as required by the application algorithm and the appearance of faulty connections. Not only does this router introduce "unlimited" robustness that can reduce dramatically the design and development times for message-passing parallel computers, but its performance also is outstanding.

Acknowledgments

This research was supported in part jointly by NSF and DARPA under the New Millennium Computing Point Design grant ASC-9634775. The authors would like to thank J. Patel and P. Papathanasiou for debugging the router's VHDL code, running some simulations and editing some files.

References

[1] C. L. Seitz and W.-K. Su, "A Family of Routing and Communication Chips Based on Mosaic, "*Symp. Integr. Syst.*, Wash., 1993.

[2] S.G. Ziavras, H. Grebel, and A.T. Chronopoulos, ``A Scalable/Feasible Parallel Computer Implementing Electronic and Optical Interconnections for 156 TeraOPS Minimum Performance,'' PetaFlops *Arch. Worksh*., April 22-25, 1996, 235-266.

[3] S.G. Ziavras, "RH: A Versatile Family of Reduced Hypercube Interconnection Networks,'' *IEEE* Trans. Paral. Distr. Syst. 5(11), 1994, 1210-1220.

[4] W.J. Dally and C.L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Comput*. 36 (3), 1987, 547-553.

[5] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability,Programmability*, McGraw Hill 1993.

[6] K. Bolding, S.-C. Cheung, S.-E. Choi, C. Ebeling, S. Hassoun, T.A. Ngo and R. Wille, "The Chaos Router Chip: Design and Implementation of an Adaptive Router, " Proc. VLSI, IFIP, 1993, 311-320.

[7] T. Golota, Y. Cai, T. Fukaya, K. Linga, S. Ziavras, and D. Misra, "VHDL Modeling of the BLITZEN Massively Parallel Processor (MPP)," MARLUG Spring Conf., May 1996.

[8] S. Konstantinidou and L. Snyder, ``The Chaos Router," *IEEE Trans. Comput*. 43(12), 1994, 1386-1397.

[9] J. Bhasker, *A VHDL Prime*, Prentice Hall, 1996.

[10] A.A. Chien, "A Cost and Speed Model for k-ary n-Cube Wormhole Routers," *IEEE Trans. Paral. Distr. Syst*. 9(2), 1998, 150-162.

[11] A.A. Chien, M.D. Hill, and S.S. Mukherjee, "Design and Challenges for High-Performance Network Interfaces," *IEEE Computer*, Nov. 1998, 42-44.

[12] S.W. Daniel, K.G. Shin, and S.K. Yun, ``A Router Architecture for Flexible Routing and Switching in Multihop Point-to-Point Networks,'' *IEEE Trans. Paral. Distr. Syst.* 10(1), Jan. 1999, 62-75.

[13] S.G. Ziavras and S. Krishnamurthy, ``Evaluating the Communications Capabilities of the Generalized Hypercube Interconnection Network,'' *Conc.: Pract. Exper*. 11(6), 1999, 281-300.

[14] S.G. Ziavras, ``Investigation of Various Mesh Architectures with Broadcast Buses for High Performance Computing,'' *VLSI Design* 9(1), Jan. 1999, 29-54.

[15] S.G. Ziavras and A. Mukherjee, ``Data Broadcasting and Reduction, Prefix Computation, and Sorting on Reduced Hypercube Parallel Computers,'' *Paral. Comput*. 22, June 1996, 595-606.

[16] A. Bar-Noy and C.-T. Ho, ``Broadcasting Multiple Messages in the Multiport Model,'' *IEEE Trans. Paral. Distr. Syst*. 10(5), May 1999, 500-508.

[17] M. Birnbaum and H. Sachs, "How VSIA Answers the SOC Dilemma, "*IEEE Computer*, June 1999, 42-50.

[18] R. V. Bopanna and S. Chalasani, "Fault-Tolerant Communication with Partitioned Dimension-Order Routers, *"IEEE Trans. Paral. Distr. Syst* 10(10), Oct. 1999, 1028-1039.

[19] J. Rexford , J. Hall and K.G. Shin, "A Router Architecture for Real-Time Communication in Multicomputer Networks," *IEEE Trans. Comput.* 47(10), 1998.

Table I. Packet formats.

flit 1			**flit 2**	**flit 3**	**flit 4**
14 bits	14 bits	4 bits	32 bits	32 bits	32 bits
Destination address	Source address	Control	Address in RAM for destination	Data	Data

Normal packet.

2-bit Control	14 bits
00	Local node address

Node address initialization packet.

bit #:	7 6	5	4 2	1 0
	2-bit Control	1 bit	3 bits	2 bits
	01	Reserved	Topology	Transfer mode

Topology initialization packet.

2-bit Control	14 bits
10	Dimension register

Dimension register initialization packet.

2-bit Control	1 bit	8 bits	5 bits
11	X	Configuration table address	Configuration table data

Configuration table initialization packet.

Table II. Topology table.

Bit-0	Bit-1	Bit-2	Topology
0	0	0	Reserved
0	0	1	1-D
0	1	0	2-D
0	1	1	3-D
1	0	0	4-D
1	0	1	5-D
1	1	0	6-D
1	1	1	7-D

Table III. Transfer mode table.

Bit-0	Bit-1	Transfer mode
0	0	8 bits
0	1	16 bits
1	0	32 bits
1	1	Reserved

TABLE IV. Configuration table for the 2-D topology with 8-bit channel width.

Main address	Dimension	Output port
01000000	+1	00000
01000001	-1	00001
01000010	+2	00010
01000011	-2	00011
01000100	….. Unused …..	
01011100	CPU port	11100
01011101		
01011110		
01011111		

TABLE V. Configuration table for the 2-D topology with 16-bit channel width.

Main address	Dimension	Output port
01000000	+1	00000
01000001	+1	00001
01000010	-1	00010
01000011	-1	00011
01000100	+2	00100
01000101	+2	00101
01000110	-2	00110
01000111	-2	00111
01001000		
……	Unused	
01011100	CPU port	11100
01011101	CPU port	11101
01011110		
01011111		

TABLE VI. Truth table for the conventional 1-bit full adder.

A	B	Cin	Sum	Cout_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

TABLE VII. Truth table for the actual adder slice.

D	Cout
0	0
1	Cout_I

TABLE VIII. Truth table for the implemented adder slice (D:dimension-register bit).

D	R=Sum	ACCin	S	ACCout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

TABLE IX. Table for the routing decision at a bit location i, where D(i)=0 and i=0,1,2,....

R(i)	S(i)	Routing Dimension
0	0	Go to next dimension or zero port, if current dimension is the last one
0	1	+ve (increase coordinate)
1	0	Not possible
1	1	-ve (decrease coordinate)

TABLE X. Example of routing.

D	01111110111111
A	10100110000011 = 83|3
A’	01011001111100 (1's complem)
B	01010011011011 = 41|91
Cin	00000000000001
R	10101101011000= -42|+88
Status	11111101111000

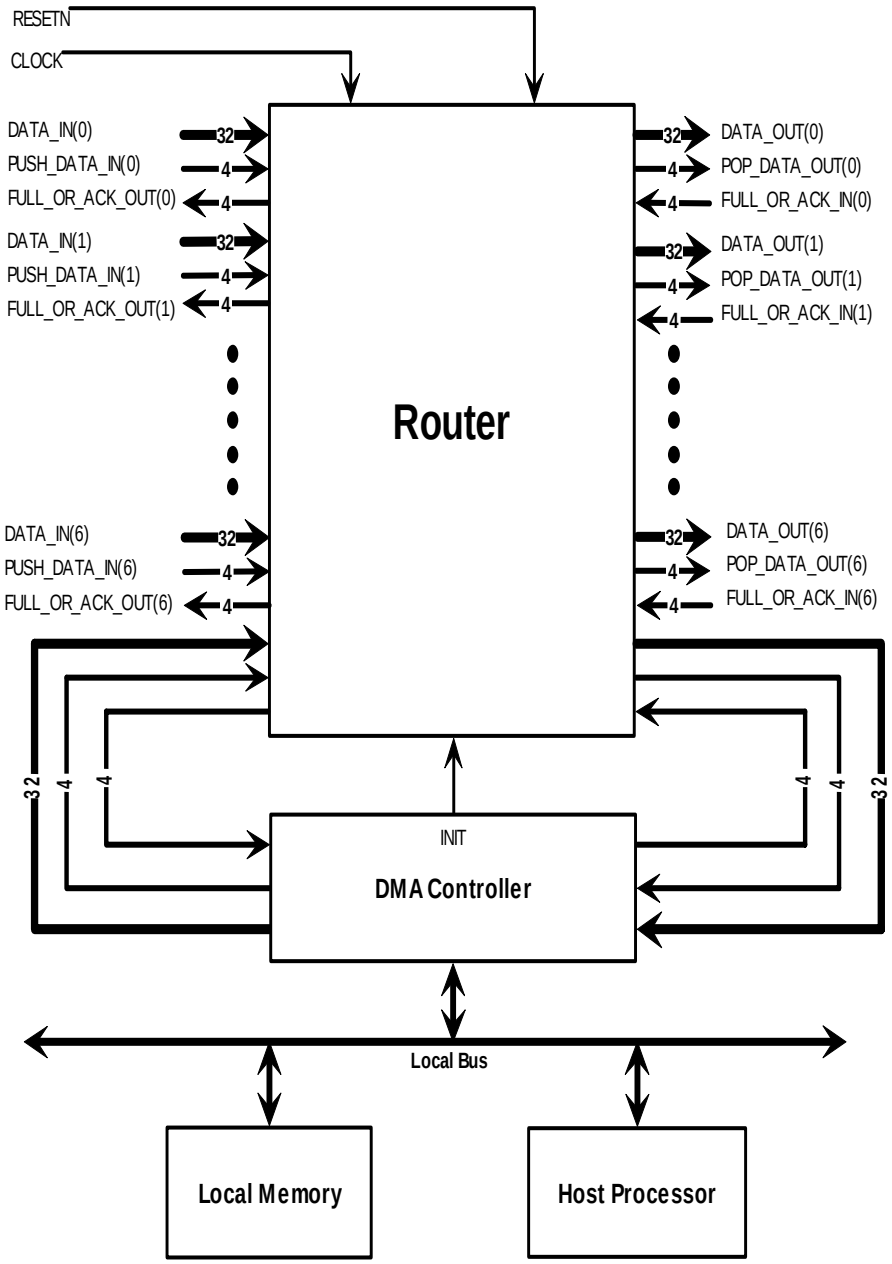


Figure 1. Top-level architecture of the router and its interface with the local PE.

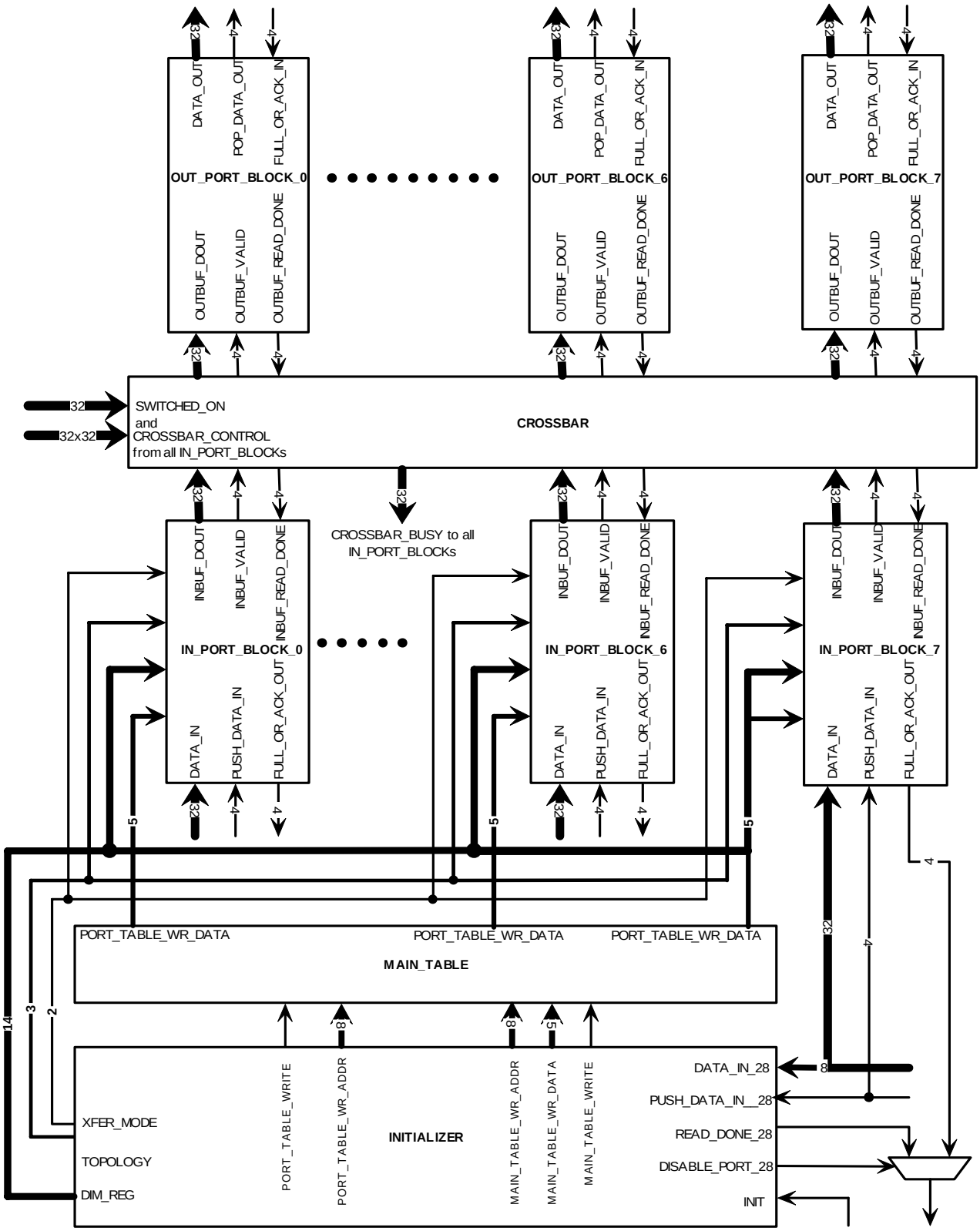


Figure 2. Top-level structure of the router.

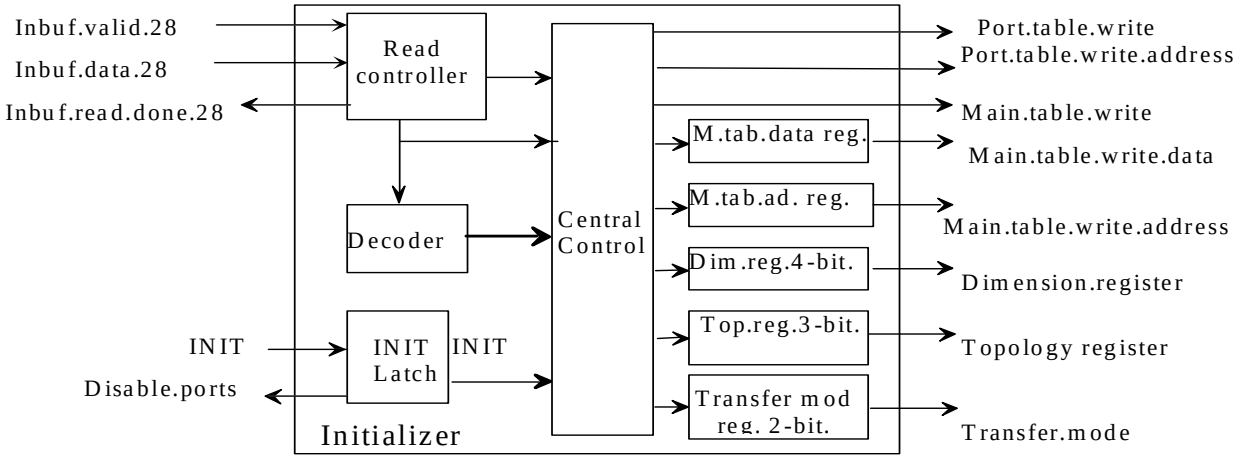


Figure 3. Block diagram of the initializer.

```
IF (INIT = 1)                                   // Detect Initialization Mode
THEN
    READ [8 or 16 bits from port #28]           // 8-bit bus transfers
    HEADER = 2 most significant bits of the packet;
        IF (HEADER = 00)
          MODE = Node Address Initialization;
          Local_node_address=14-bit Incoming Address;
        ENDIF
        IF (HEADER = 01)
          MODE = Topology Initialization;
          Topology [3-bit] = 2,3,4-th bits; Transfer_mode [2-bit] = 0,1-st bits;
        ENDIF
        IF (HEADER = 10)
          MODE = Dimension Register Initialization;
          Dimension Register = lower 14 bits;
        ENDIF
        IF (HEADER = 11)
          MODE = Configuration Table Initialization;
          Conf_table_address[8-bit]=[5-12]-th bits;
          Conf_table_data[5-bit]=[0-4]-th bits;
          Write_Main_Confuguration_Table;           // Total of 256 records
          IF (Last_record_reached = 1)
                INIT = 0; Flag_end_of_init = 1;
          ENDIF
        ENDIF
    Initializer Outputs Activated; // Based on command from initialization packets
ELSIF (INIT = 0)
    IF (Flag_end_of_init = 1)         // Detect the last step of initialization
        // Initializer uploads the port tables from config. table. Each port has a 32-record
        // table containing +/- dest. port addresses according to the current topology
        DISABLE_ PORTS = 1; Port_Tables_Load [32-records];
        // During the upload, DISABLE_ PORTS signals are high to deactivate the
        // operation of input (output) buffers and all input (output) ports, including
        // INBUF_DATA_28. This avoids false initialization and undesirable routing.
        IF (Port_Tables_Load [32-records] = FINISHED)
                DISABLE_ PORTS = 0;
                // enable all router ports for normal operation
        ENDIF
        Flag_end_of_init = 0;
    ELSIF (Flag_end_of_init = 0)
        Initializer does nothing // Router's Usual Transfer Mode
    ENDIF
ENDIF
```

Figure 4. Pseudocode for router initialization.

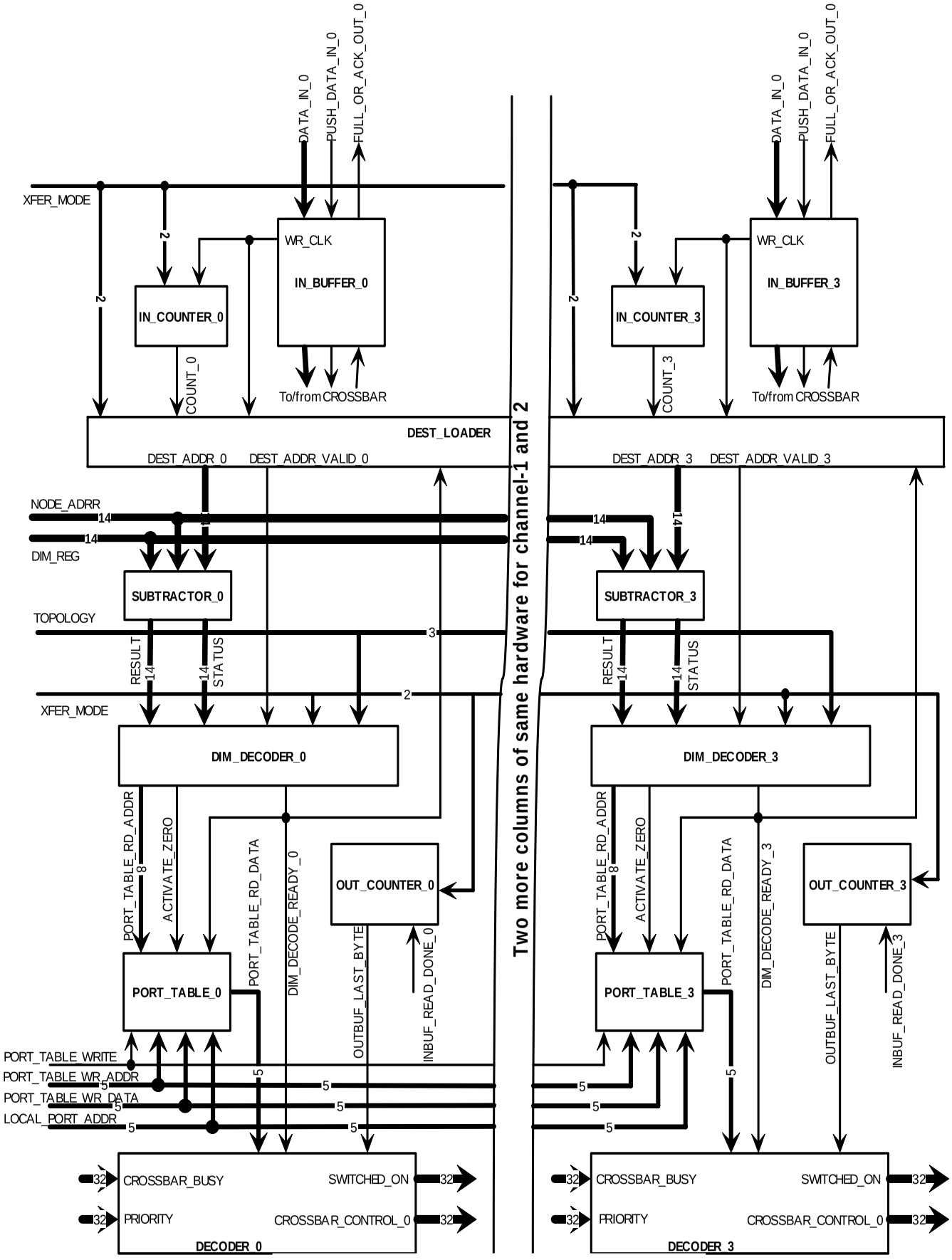


Figure 5. 32-bit input port.

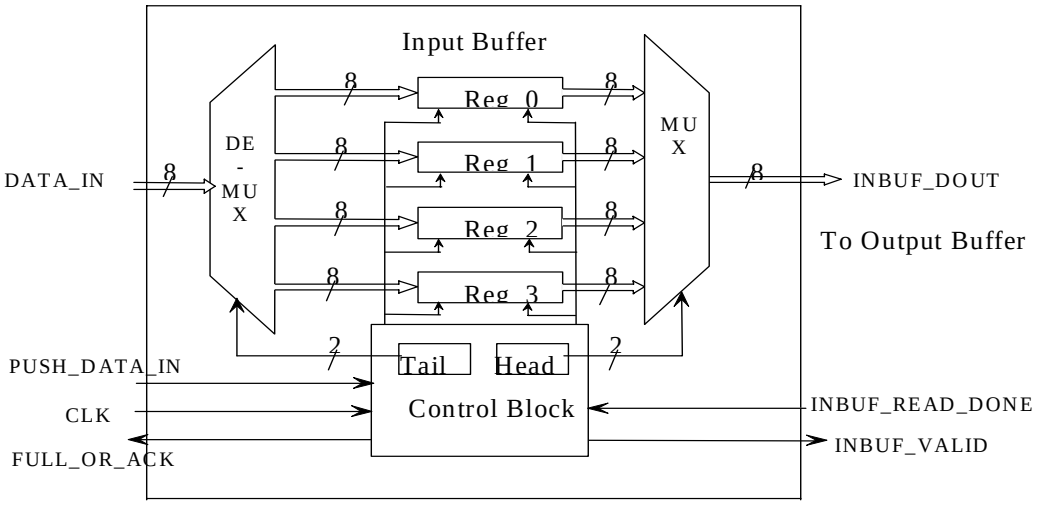


Figure 6. Block diagram of the 8-bit input buffer.

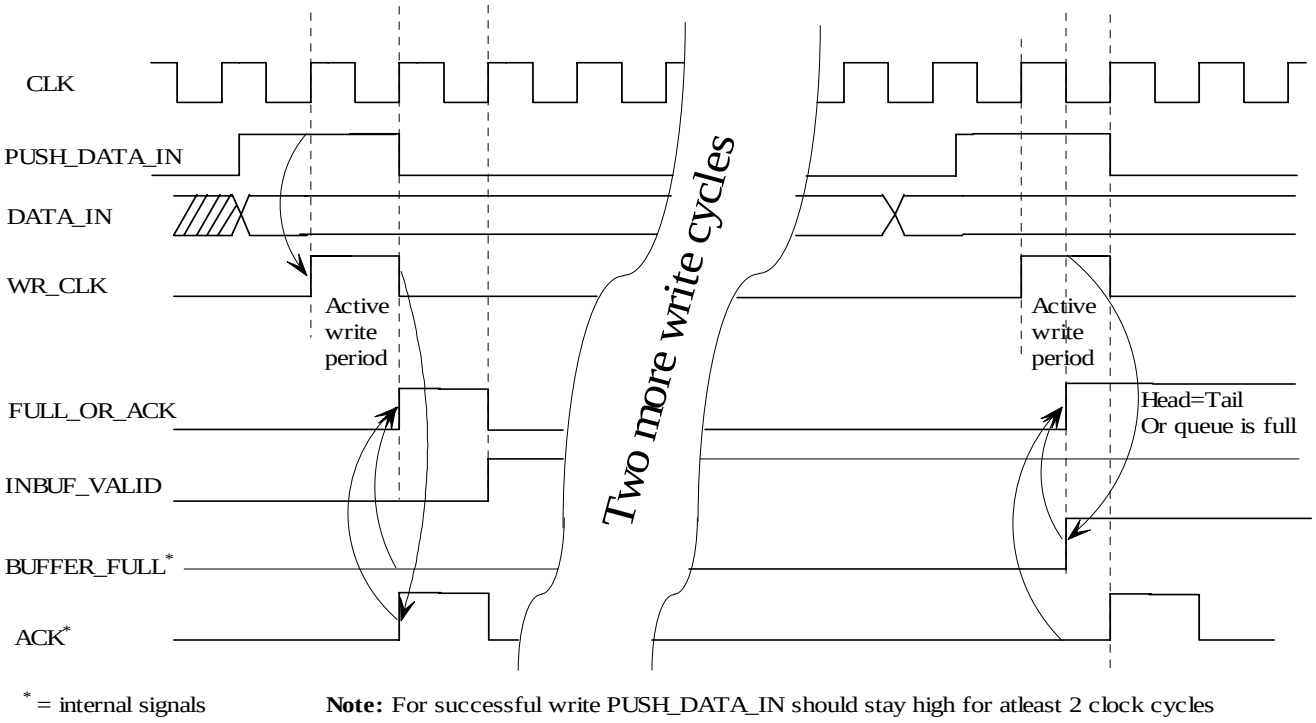


Figure 7. Timing diagram for the write operation (input buffer).

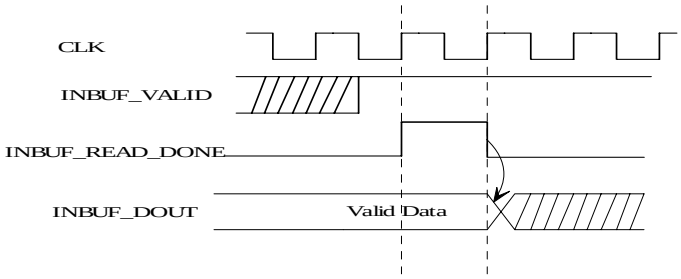


Figure 8. Timing diagram for the read operation (input buffer).

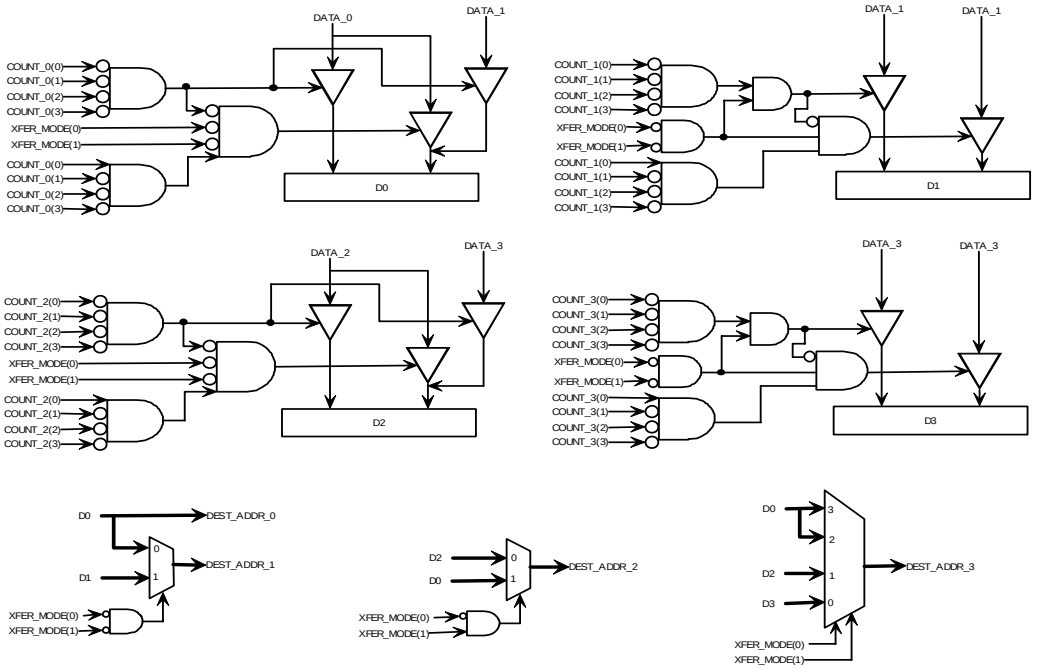
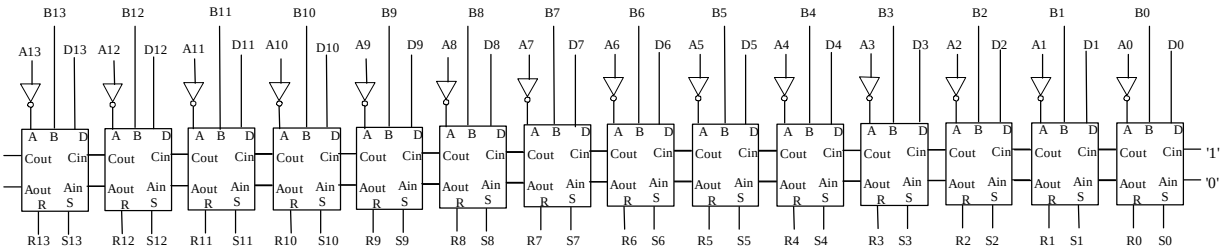


Figure 9. Destination address loader (it extracts the destination address).



B = Destination Address R = Result Word S = Status Word D = Dimension Reg.

A = Local Node Address

Figure 10. 14-bit subtractor.

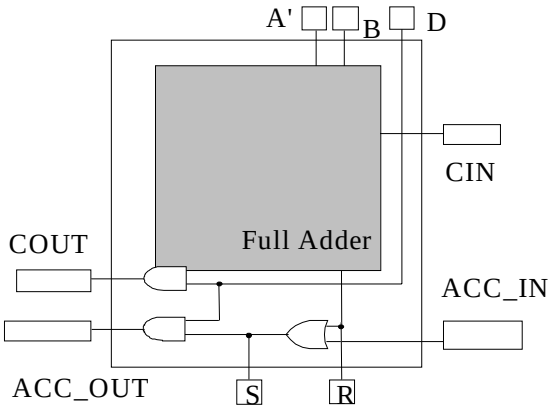


Figure 11. 1-bit subtractor slice.

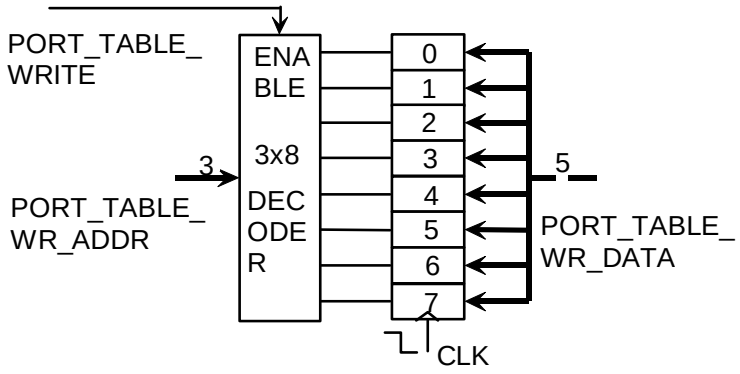


Figure 12. Port table PORT_TABLE for the current topology.

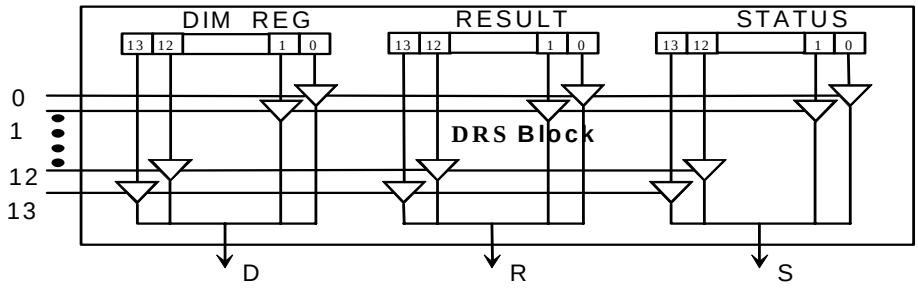


Figure 13. DRS block (reads a bit from each register).

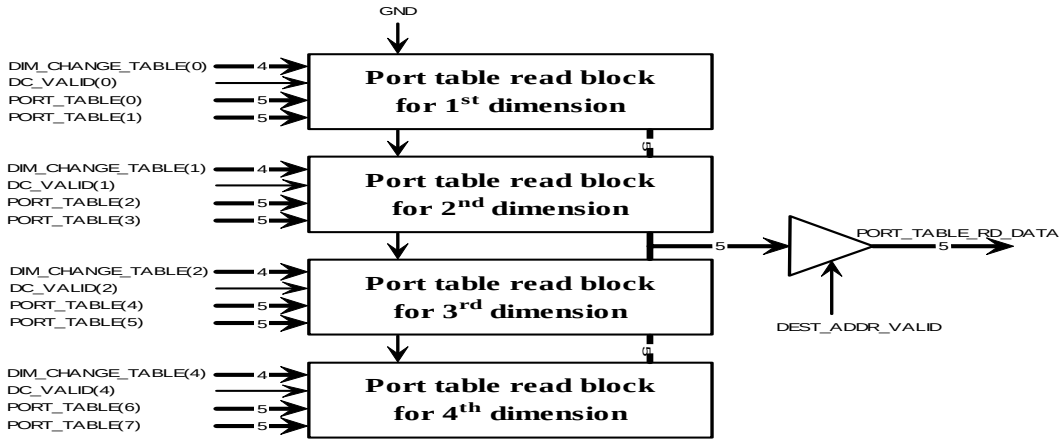


Figure 14. Dimension decoder.

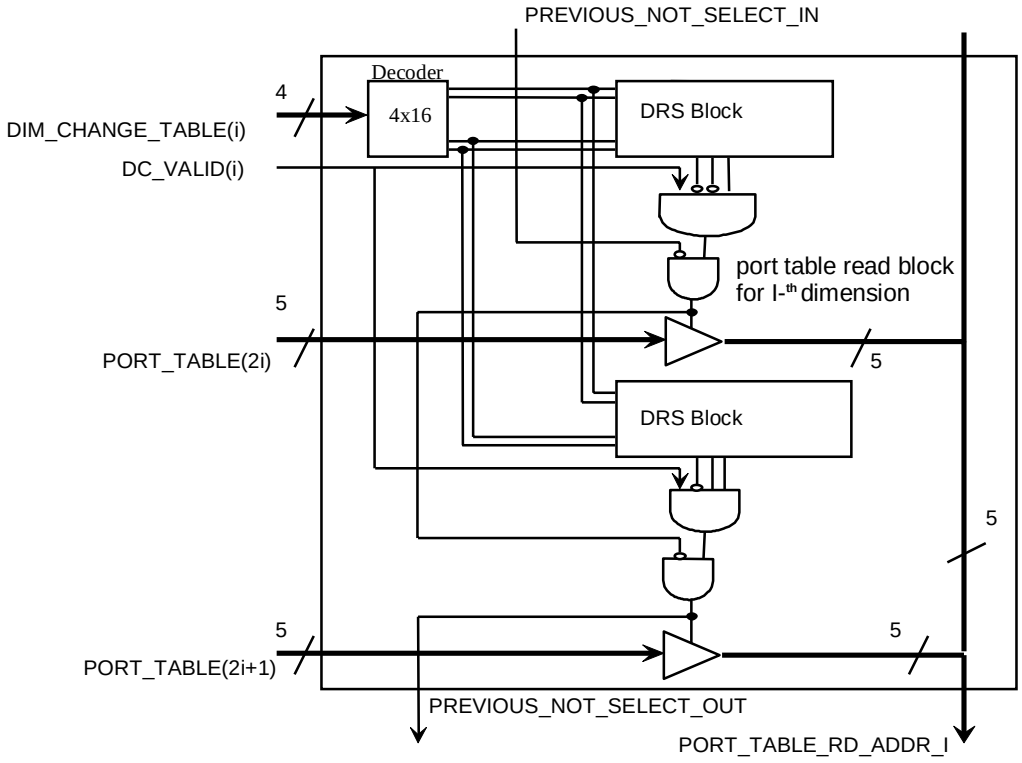


Figure 15. Port table read block for the i^{th} dimension.

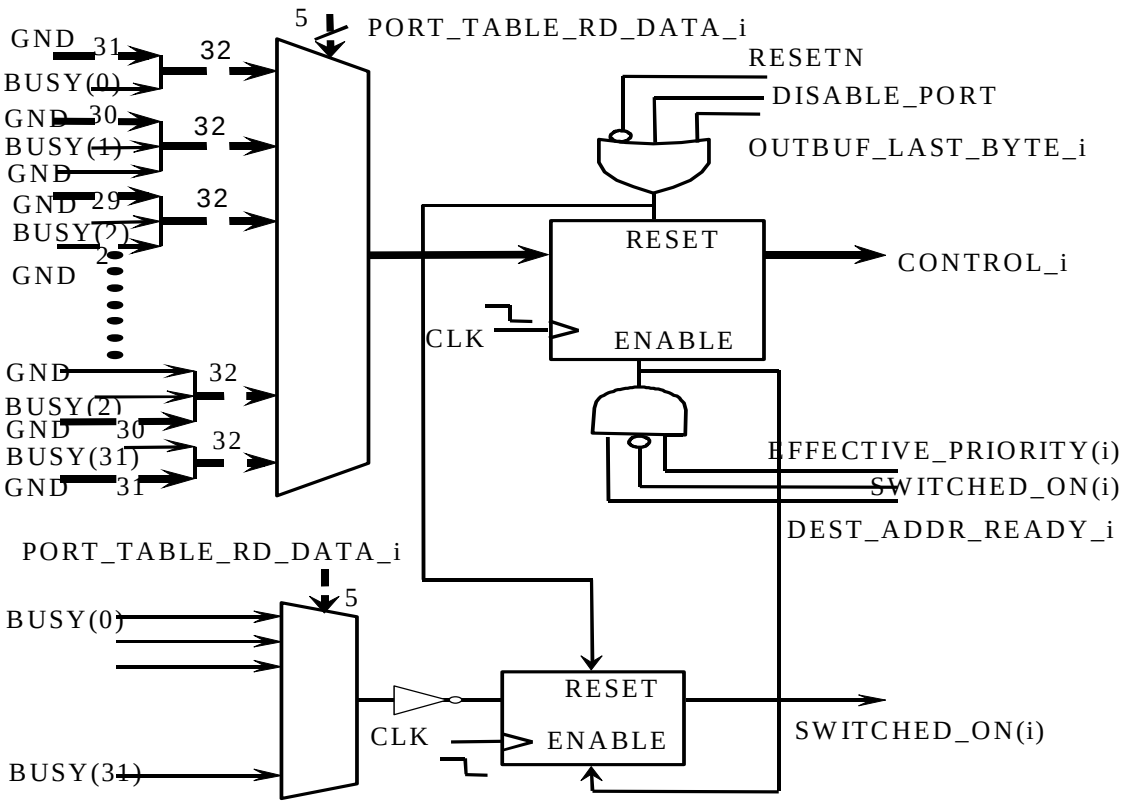


Figure 16. A 5x32 decoder with latching mechanism.

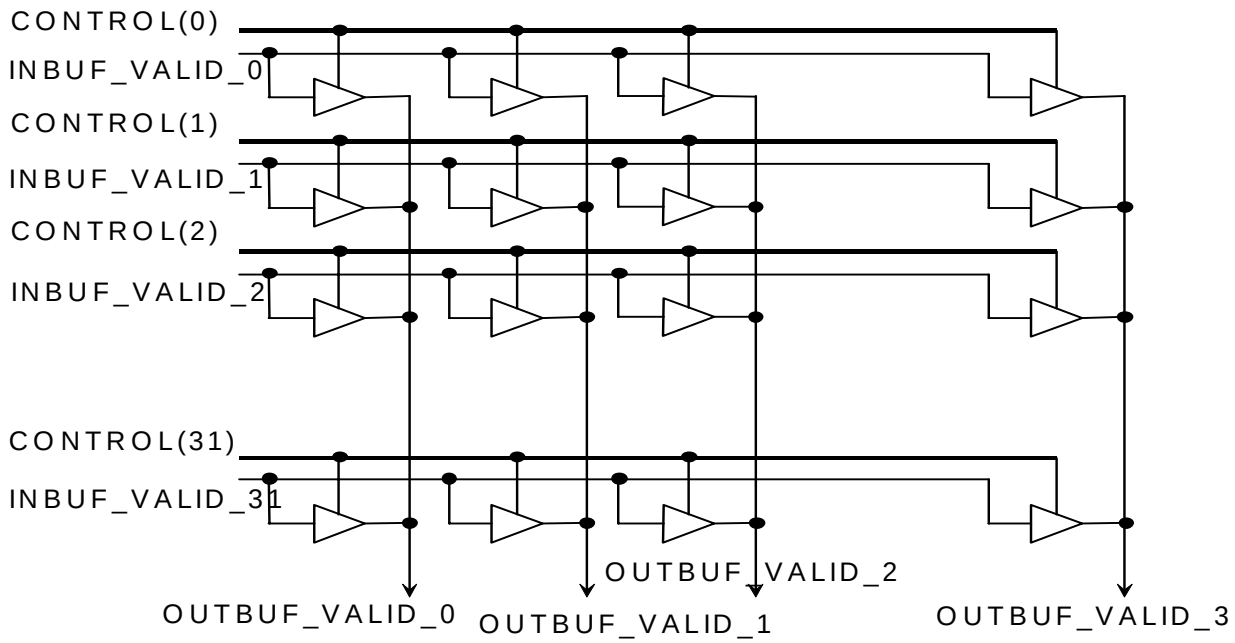


Figure 17. 32x32 crossbar switch.

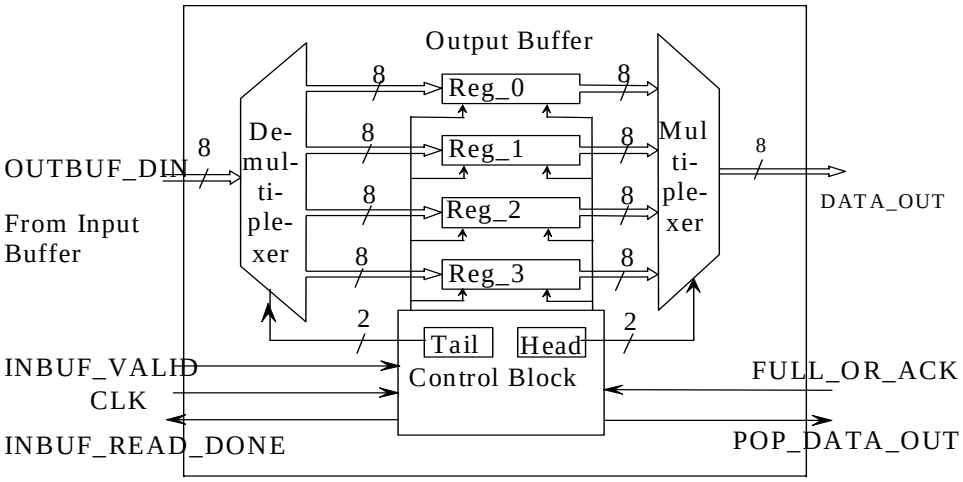


Figure 18. Block diagram of the output buffer.

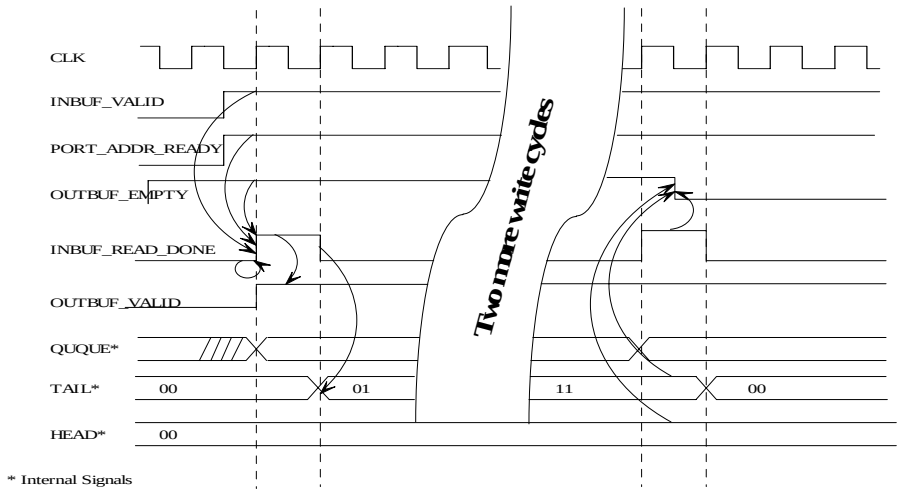


Figure 19. Timing diagram of a write operation into the output queue.

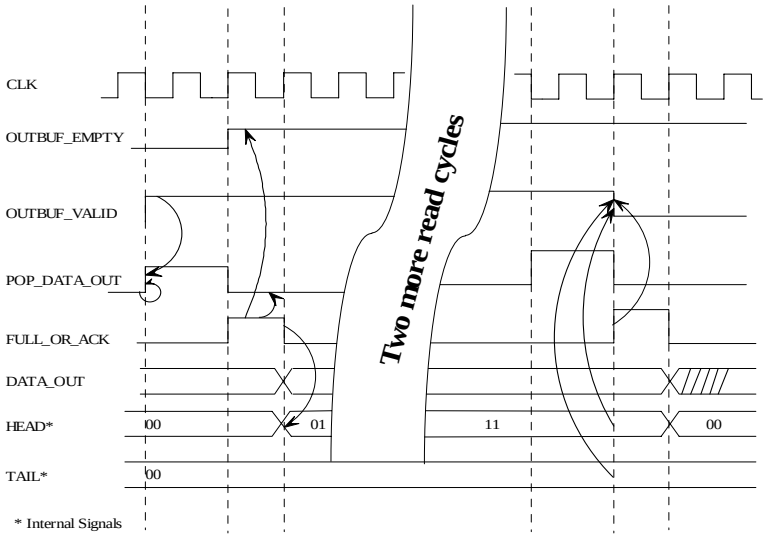


Figure 20. Timing diagram of a read operation from the output queue.

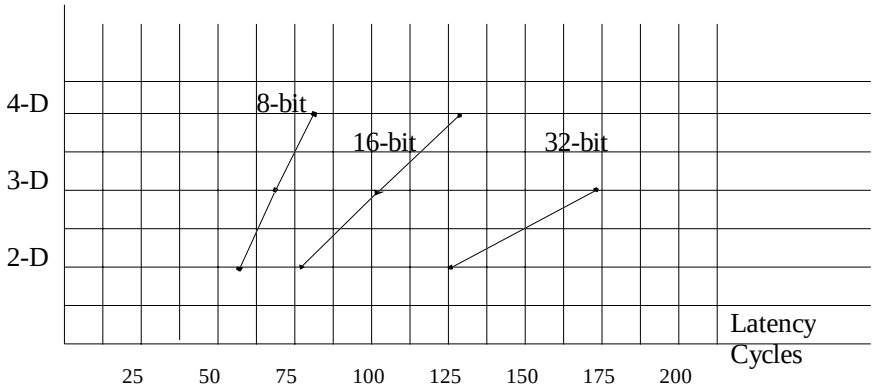


Figure 21. Router initialization time for 2-D, 3-D and 4-D hypercubes.

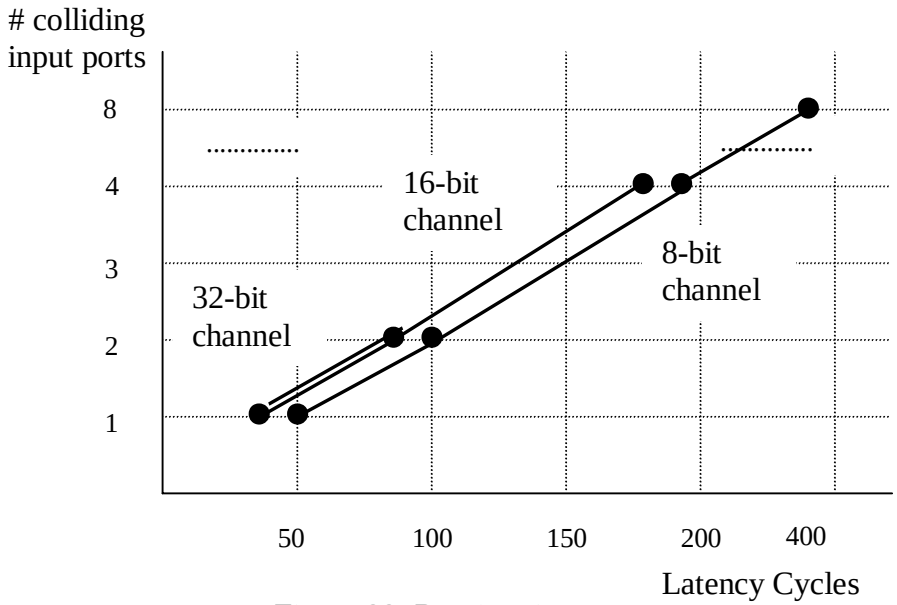


Figure 22. Routing time.

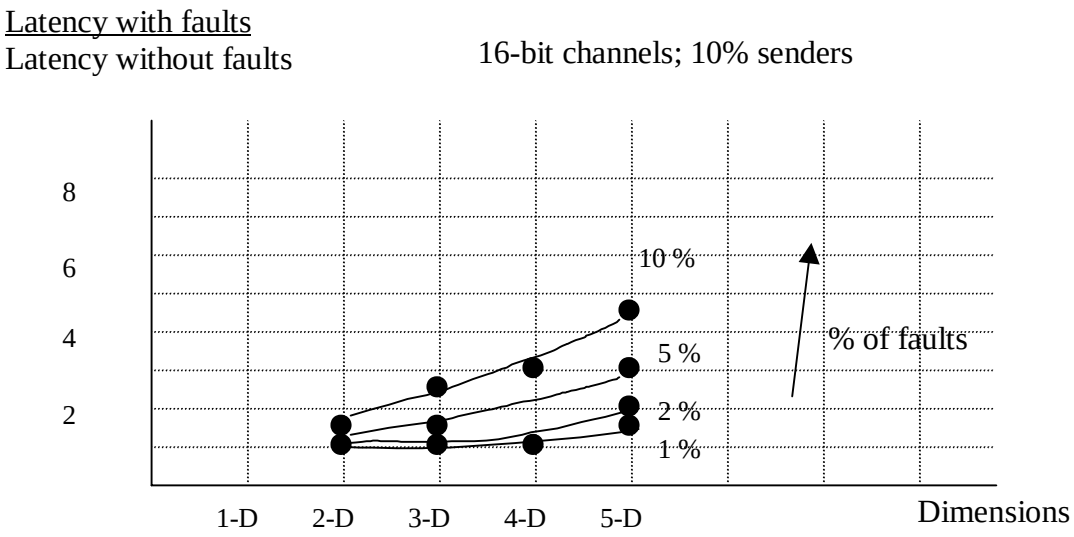


Figure 23. Simulation results for binary hypercubes with 10% senders.

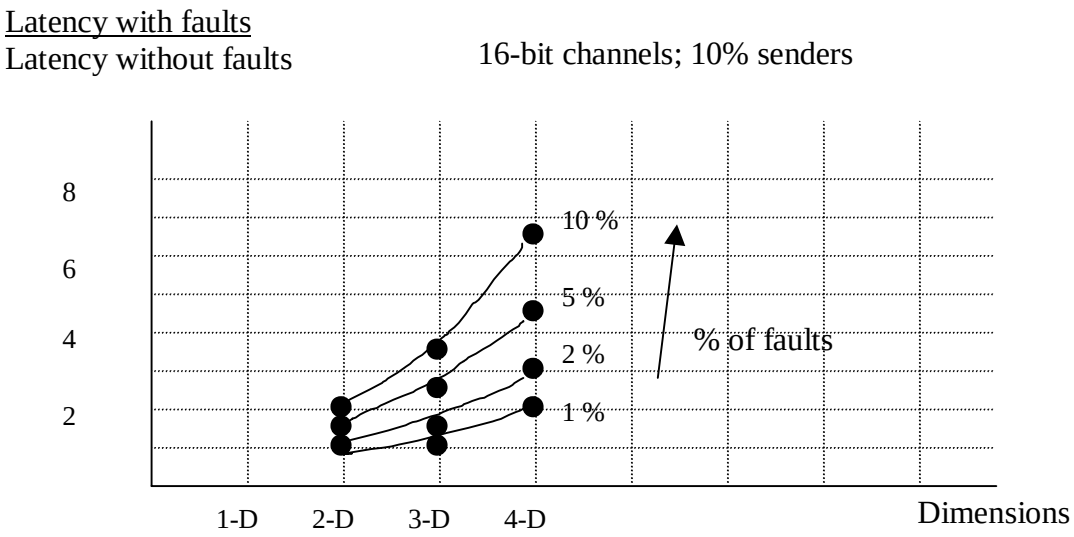


Figure 24. Simulation results for tori with 10% senders.

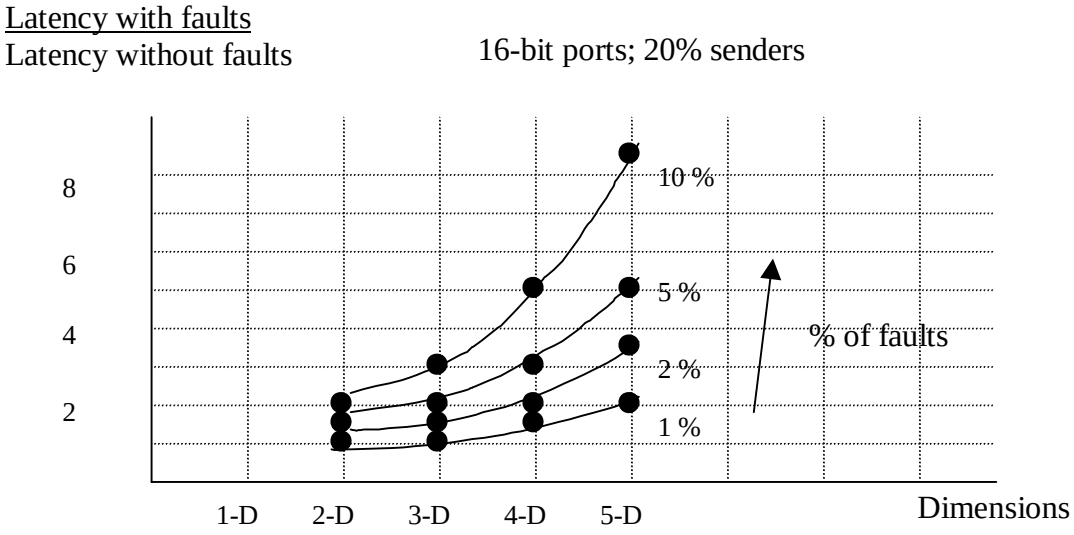


Figure 25. Simulation results for binary hypercubes with 20% senders.

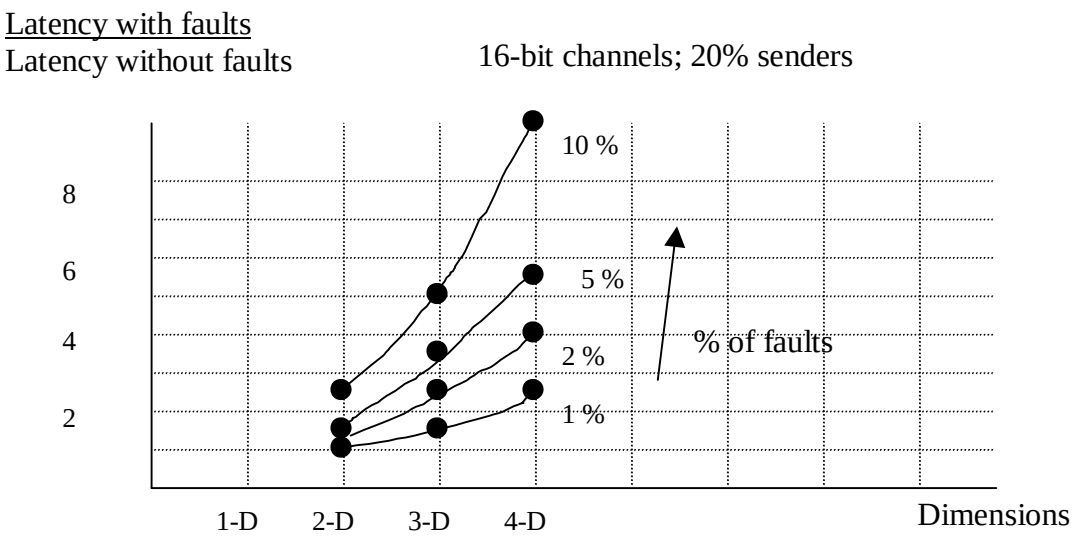


Figure 26. Simulation results for tori with 20% senders.

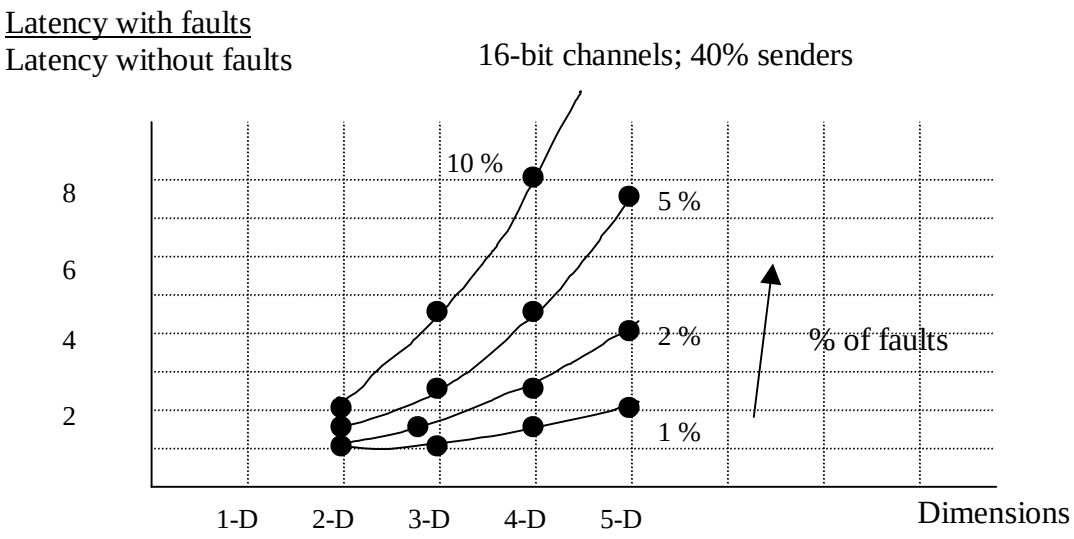


Figure 27. Simulation results for binary hypercubes with 40% senders.

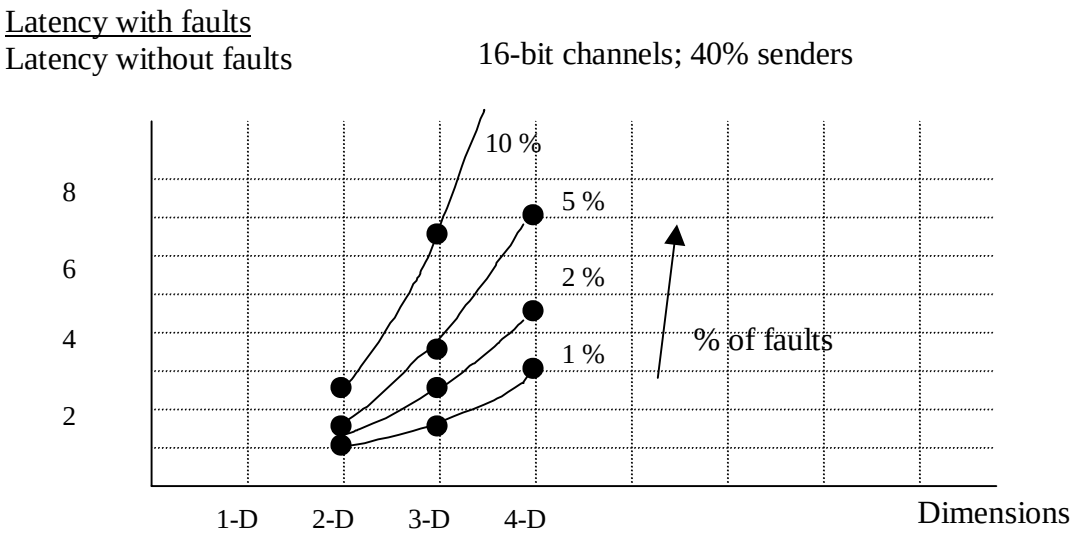


Figure 28. Simulation results for tori with 40% senders.