Design and Implementation of a Router using a Xilinx FPGA

by

Stephen L. Chamberlin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

Author .....
Department of Electrical Engineering and Computer Science
May 17, 1993

Certified by .....
Gregory M. Papadopoulos
Assistant Professor
Thesis Supervisor

Accepted by .....
Leonard A. Gould
Chairman, Departmental Committee on Graduate Students

Design and Implementation of a Router using a Xilinx FPGA

by

Stephen L. Chamberlin

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis examines the possibility of implementing a router chip in programmable logic to allow quick and easy modifications of the hardware design. A design was adapted from PaRC, a packet switched routing chip designed for the Monsoon parallel processing dataflow computer. A functional description of the router chip was written in the Verilog hardware description language, which can be compiled into a gate-level model using hardware synthesis tools. In this first attempt, the design used nearly eight times more space than is available on a Xilinx 4010 FPGA. A more intelligent synthesis tool could have reduced the required size roughly 50%. With further modifications of the design, it appears promising that a simple router could be fit on a single 4010 FPGA.

Thesis Supervisor: Gregory M. Papadopoulos
Title: Assistant Professor

Acknowledgments

I would like to thank everyone who helped me with this thesis, both directly and indirectly. Special thanks go to Greg Papadopoulos, Andy Boughton, James Hoe, Tom Durgavich, Richard Davis, Yev Gurevich, and Eric Heit. Thanks to Len Granowetter for teaching me LaTeX in a day. Also thanks to Jeff Morrow, Danielle Russell, and Ken Wu for their help proofreading, and to Dave LeCompte for just being himself. Thanks, Mom, for providing me with many years of love and support. Lastly, I would like to thank Waldo, but I can't find him anywhere.

To everyone who helped, named and unnamed— Thanks. I couldn't have done it without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel processing as a technique for improving the performance of computers is becoming increasingly widespread. Massively parallel designs that make use of thousands of processing nodes are considered the most promising technology for achieving teraflops performance and beyond in the near future. The work involved in a computation is divided into separate parts. These parts can be computed in parallel, allowing the overall computation to complete in a short time[8].

Each node may consist of a processor, local memory, and other devices. Because they do not share memory, the nodes must be interconnected in a way that allows them to communicate. A simple machine with few nodes might connect them using a shared bus, but this scheme only provides enough bandwidth for a few nodes at best. Typically the nodes in a parallel computer are connected by a network, a collection of switches and communication channels that allows the nodes to send messages to one another. The switches, or routers, control the path a message takes through the channels as it travels from the source node to the destination node. The use of a network increases the communication bandwidth of the parallel computer since many messages can be sent simultaneously using different channels.

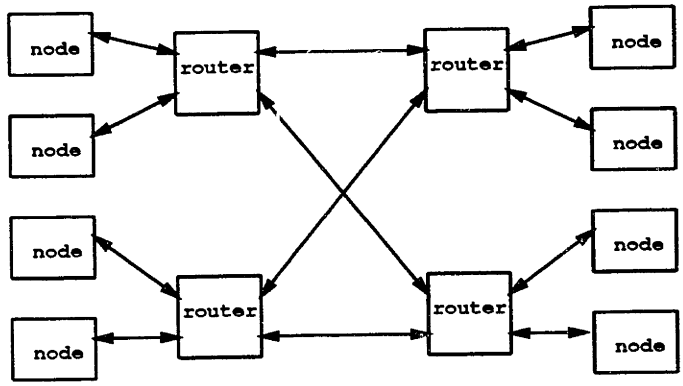

Figure 1-1: Data Flow in a Typical Network

1.1 The Role of Routers

The design of the network and the routers that direct the flow of messages through it is an essential part of the design of a parallel computer since it controls how the nodes send messages to one another. A typical network is shown in figure 1-1. A collection of nodes is connected by a network of communication channels. Routers at the junction of channels control the paths packets take through the network, allowing any node to communicate with any other node.

Messages are typically broken up into packets before they are transmitted. Each packet contains routing and sequencing information in its header, and may be a fixed or variable length depending on the design. The packets make their way from the source to destination nodes through stages of routers. A router receives a packet on one of its input ports and sends it out of one of its output ports. The choice of output port is based upon the packet's destination, contained in the header. In some cases, the choice may also be influenced by the network load. If there is more than one path from the router to the destination node, the router may attempt to send the packet along the path with the least traffic.

The total latency of the message is the sum of the start-up overhead, communication latency, and blocking time. The start-up overhead is the time needed to handle the packet at the source and destination nodes. It is mainly a factor of the system

software design and the node/network interface. The communication latency is the time between when the head of the packet enters the network and the tail of the packet leaves the network. It depends on the router switching technique (discussed in section 2.1.3), latency of the routers, and to a lesser extent on the latency of the communication channels.[1] The blocking time is the sum of all delays the packet encounters due to contention for shared resources. It may be high if there is a lot of network traffic.

1.1.1 Router Requirements

The latency of the network affects the performance of the computer, as the time needed to move data from one node to another must be kept small in order to achieve fine grain parallelism. If a node sends a message and has no other work to do while it awaits a reply, the latency of the message directly effects the length of time it takes to perform the computation. A two-phase protocol can be used to prefetch data before it is needed, masking reasonable latencies. However, as the latency increases the processor will eventually run out of useful work to do while waiting. The network latency is a function of the network configuration, the amount of traffic in the network, and the latency of the router. Thus a good router must facilitate low-latency communication through the network.

Additionally, the network needs to provide high bandwidth communication between the nodes. Since most of the time the nodes will be working cooperatively on a complex computation, a large amount of messages will usually need to be exchanged to coordinate their efforts. The router needs to provide sufficient bandwidth through the network to assure that typical computations are not limited by the available communication bandwidth.

[1] Because the router latency is typically much greater than the channel latency, the latter is usually ignored when computing overall communication latency.

1.2 Router Design

Routers are usually manufactured using a lengthy and expensive ASIC process similar to that used for CPUs and other VLSI designs. However, in order to perform network research it would be useful to be able to put together a "quick and dirty" router for testing purposes without having to pay the time and cost penalties involved in making a new custom ASIC design. For small research computers it may be difficult to justify the costs involved in a new router design, yet existing routers may not provide some necessary functionality for the new computer design.

It may also be useful to be able to modify a router's design after it has been in use for a period of time. A change in the computer's network architecture, system software, typical network load, the discovery of some new routing algorithm, or the appearance of a bug may make it desirable to make minor modifications to the router design and then replace the modified routers in the computer. With a traditional ASIC design this is difficult and expensive.

1.2.1 Implementing a Router on an FPGA

Field-Programmable Gate Arrays (FPGAs) are designed to provide exactly these sorts of capabilities[15]. They are intended to provide the benefits of custom CMOS VLSI design while avoiding the associated design overhead time, initial engineering cost, and inherent risk. Because they can be reprogrammed an unlimited number of times, they can be used in designs where the hardware is changed dynamically or where it must adapt to different applications.

The low initial cost and high adaptability of an FPGA make it a natural candidate for implementing a router for small research computers, computers that are likely to undergo large amount of modification, and the other cases mentioned above. However, until recently it wasn't practical to attempt implementing a router on an FPGA. The best FPGAs had only the equivalent of a few thousand gates, were limited in their maximum possible clock rates, and were relatively expensive. However, given recent improvements in FPGA technology, designers are now exploring the possibility of

implementing a simple router on a field-programmable gate array, allowing them to benefit from all the advantages of FPGAs. For example, Autonet [11, 9] uses an FPGA to implement part of its router switch mechanism.

1.3 Overview

The Xilinx XC4000 family of FPGAs[14] introduced in 1992 is a significant improvement in speed and density over previous FPGA designs. It supports programmable designs using up to approximately 20,000 transistors, running at speeds of up to 40 to 50 MHz. The remainder of this document explores the design and implementation of a simple router intended to be programmed onto a Xilinx XC4000 family FPGA. A router design was adapted from PaRC[5], a packet switched routing chip designed for the Monsoon parallel processing dataflow computer[10]. Chapter 2 provides an overview of the router design, including an explanation of some design decisions and a description the chip's basic behavior. Chapter 3 describes the implementation of each of the router's subunits in Verilog, a hardware description language which can be compiled into a gate-level model using synthesis tools. The synthesis process is described in chapter 4. The router design was compiled using the Synopsys hardware synthesis tool[12], resulting in a gate-level model which used nearly eight times more space than is available on a Xilinx 4010 FPGA. Finally, chapter 5 assesses the success of the design and makes some suggestions for future work which shows the possibility of allowing a simple router to fit on a single FPGA.

Chapter 2

Router Design

2.1 Design Decisions

Before work could begin on the implementation of the router, several basic design decisions about the router and its desired behavior had to be made. These decisions included whether to make the router switch blocking or non-blocking, how many inputs and outputs the router should have, and what switching technique to use. Table 2.1 compares the choices that were possible in terms of their cost in area (gate usage), and effect on latency and performance. The router design that was chosen was a 3x3 non-blocking switch using virtual cut-through switching. The reasoning behind these choices is explained below.

2.1.1 Blocking vs. Non-Blocking Switches

An N input, N output switch can be conceptualized as the junction of 2N unidirectional communication channels. If we connect the input and output channels with some constant number of connection channels K, any input channel can be connected to any output channel by closing the appropriate switches from among the 2KN total switches (figure 2-1). Up to K pairs of input and output channels may be connected simultaneously with this arrangement. However, with this design it is possible that an incoming packet on one of the input channels might not be able to be transmitted

Table 2.1: Design Decision Tradeoffs

parameter	gate count	network latency	performance
blocking switch	low	normal	poor
non-blocking switch	high	normal	good
2x2 switch	low	high	normal
3x3 switch	medium	medium	normal
4x4 switch	high	low	normal
circuit switching	low	normal	poor
packet switching	high	high	normal
virtual cut-through	high	low	good
wormhole routing	low	low	normal

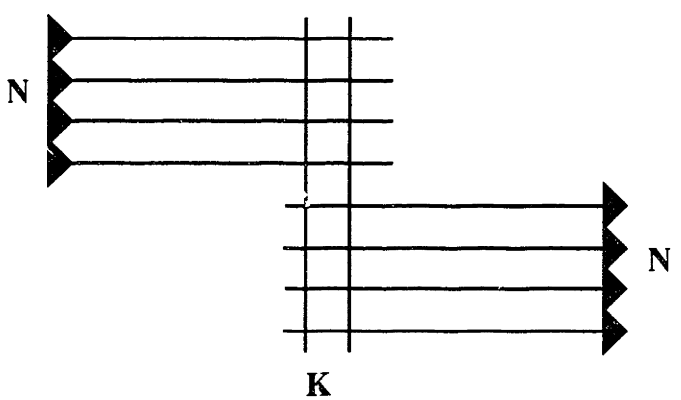

Figure 2-1: A Blocking Switch

to its output channel even if the output were available, if all of the K available connection channels were already in use. Therefore this type of switch design is called a blocking switch. The size of a blocking switch grows linearly with N.

In contrast, a non-blocking switch (figure 2-2) connects the input and output channels in such a way that any input can always be connected to any output, so that a packet is never blocked if its output is available. However, in this arrangement N^2 switches are needed to connect the inputs and outputs, hence the size of a non-blocking switch grows with the square of N.

Because the size advantage of blocking switches is significant only for larger values

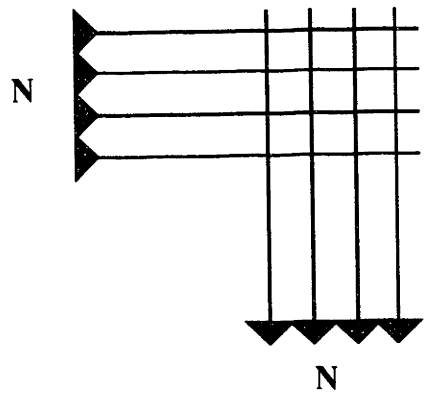

Figure 2-2: A Non-Blocking Switch

of N and the router being designed was to have a small N, a non-blocking switch design was used to prevent the potential addition of unnecessary delays to the message latency due to blocking inside the switch.

2.1.2 Router Size

The number of inputs and outputs the router would have was another significant design decision. Because chip space was very limited the number could not be large. A 4x4 switch was the biggest that could reasonably be considered. Since a 1x1 switch has no useful functionality and a symmetric switch was desired, the choices were narrowed to 2x2, 3x3, and 4x4. Each input and output port added to the pin count and gate count of the router. Finally, a rather unconventional 3x3 design was chosen as a compromise between space and performance. This choice does not limit the potential functionality, as it is always possible to build a larger switch out of several smaller ones, although this composite switch will exhibit poorer performance. Figure 2-3 shows how a 9x9 switch can be built from 2 stages of the 3x3 switches described in this document.

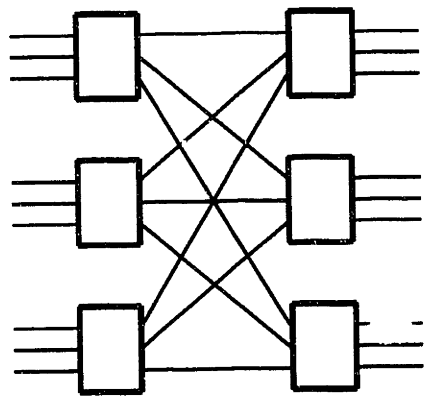

Figure 2-3: A 9x9 Switch Built from 3x3 Switches

2.1.3 Switching Techniques

The most important and fundamental design issue that had to be resolved was what switching technique to use. The switching technique is the mechanism that controls how data is moved from an input channel to an output channel. For example, data might be buffered at the input and later transmitted from the output, or they might be passed directly from input to output. Depending on what switching technique is used by the routers, the network latency can vary greatly. Circuit switching, packet switching, virtual cut-through, and wormhole routing were considered as possible switching techniques for the router.

Circuit switching works by setting up a complete, unblocked path from the source to the destination and then transmitting the packet over this path all at once, much like the way a circuit is set up to connect two parties through a telephone network. Once the packet has been transmitted, the path from source to destination is torn down. If a second packet attempts to set up a path that uses one of the communication channels already in use by the first packet's path, it will be blocked and no progress can be made sending the second packet until the first packet has been sent and its path is torn down. For this reason, circuit switching does not generally make effective use of the communication channel bandwidth when the amount of traffic is high, as

many channels will be unused. However, circuit switching can be effective even under these conditions if the packet size is very large.

In contrast to circuit switching, packet switching buffers packets in the intermediate routers as they make their way from the source to the destination node. Each router contains buffering space for holding packets. Incoming packets are stored in these packet buffers upon their arrival at the input port. A flow control mechanism is used to ensure that packets do not arrive when no buffering space is available. Since the packet is buffered in the router, the channel the packet arrived on is now free to be used by other packets. Once the entire packet is received and buffered, it can be transmitted to the appropriate output channel as soon as it is available. In this way the packet hops from one router in the path to the next until it arrives at its destination. Packet switching makes better use of the available bandwidth because if a packet's path is blocked, it sends it as far along the path as it can go, utilizing bandwidth that circuit switching wastes.

Virtual cut-through[6] works identically to packet switching, with one important exception. Whereas packets are always buffered in packet switching, they are only buffered in virtual cut-through if their output ports are blocked. Otherwise the router can begin transmitting the packet as soon as it determines which output port it should go to. The router does not need to wait for the entire packet to be received; it may actually be transmitting the packet on an output port and receiving it on an input port simultaneously. The packet may "cut through" several stages of routers if none of the needed outputs are blocked, so that it is possible for the head of a packet to reach the destination before the tail has left the source.

Wormhole routing[3, 8] works in a fashion similar to virtual cut-through. Packets are further subdivided into units called flits. When there is no network contention, flits pass through from input to output without being buffered, similar to virtual cut-through. As the header flit makes its way from source to destination, the remaining flits in the packet follow the same route in pipeline fashion. When the header flit is blocked, it and the flits behind it are buffered in flit buffers in the routers comprising the path from source to destination. However, these flit buffers need only be very

small.

Virtual cut-through routing was chosen for the router design because of its high bandwidth, lower latency, and relative simplicity compared to the other switching techniques. In addition, a virtual cut-through router design in [5] was already available as a starting point for the new design, reducing the amount of initial engineering work needed. The main disadvantage of virtual cut-through is that since a packet may be sent out before it is fully received, it is difficult to do error checking in the router. If an error is detected in the tail of a packet, its head may have already been sent out, making it too late to throw the packet away and signal an error condition. Instead, the packet with the error propagates through the network. To compensate for this, end to end error checking of all packets must be done. This can be accomplished in software, reducing the amount of hardware necessary in the router.

2.2 Overview of Functionality

The design of the router to be programmed into the FPGA was adapted from the design of the PaRC packet switched routing chip[5]. The few changes that were made were mostly simplifications of the PaRC design. Since it was anticipated that space on the Xilinx FPGA would be extremely limited, a number of PaRC's features were omitted in an attempt to make the router as small as possible. Specifically, the control port, statistics gathering features, and most of the error-checking capabilities were omitted. The ability to route packets using several different algorithms was scrapped in favor of a fixed routing algorithm. Lastly, while PaRC has 4 input channels and 4 output channels, the new router design has only 3 of each to conserve chip space and pin count.

A top level diagram of the router is shown if figure 2-4. Packets enter the router through one of the three input ports and are stored in the packet buffers. The input port determines which of the output ports the packet should be sent to and makes a request to the output port's scheduler, giving the location of the packet. When the packet buffers in the input port are full, it must signal the sender not to send any

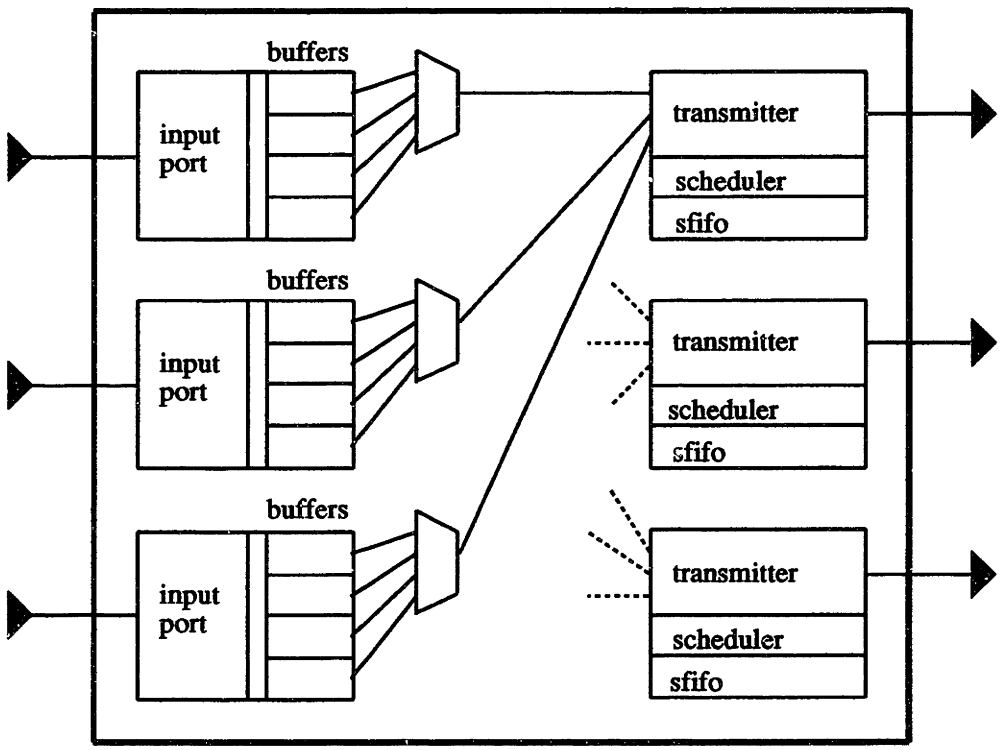

Figure 2-4: Router Top Level View

more packets.

The scheduler in each output port keeps track of the buffered packets waiting to be transmitted from the port. When the output port becomes free, the scheduler chooses which packet should be sent next. The transmitter gets the location of the packet from the scheduler, reads the packet out of its buffer, and transmits it to the next stage of the network.

2.3 External Interface

Each communication channel consists of 16 bits of data, an accompanying clock, and a wire for flow control signals (figure 2-5). The clock is aligned with the data such that its rising edge occurs while the data is stable. This clock is necessary because each router in the network operates asynchronously, so the router needs a means of

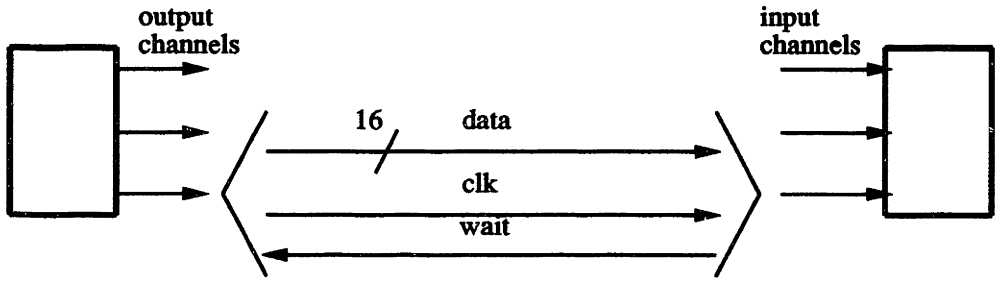

Figure 2-5: A Communication Channel

synchronizing to the incoming data from the routers in the previous stage. The flow control line is used to signal the sender to stop sending packets when all of the packet buffers are full.

The packets are fixed length, twelve words each, for a total packet size of 192 bits. The first word of the packet is the header word. It contains the information that determines how the packet should be routed. This information is placed at the beginning of the packet so that the packet can be transmitted as soon as possible after it arrives. The remaining eleven words of the packet can be used for any purpose. They may all be used to carry data, or some of them may be reserved for carrying a CRC. If some words are used as a CRC, they will not be treated differently by the router, but the receiving node can use the CRC to check the packet for errors when it arrives.

To indicate the start of a new packet, the most significant bit (bit 15) of the header word is used as a start-of-packet (SOP) bit. The remaining 15 bits of the header word, bits [14..0], determine how the packet will be routed (see section 2.7). The SOP bit is a 1 during the first word of a new packet. When no packet is being transmitted on a channel, the data bits may take on any values as long as bit 15 is a 0[1]. When the input port sees that bit 15 has changed to a 1, it reads and stores

[1]The use of specific idle patterns (all of which have a 0 in the SOP bit) can allow channel transmission errors to be detected even while no packets are being sent.

that word and the following eleven words. Bit 15 only has a special meaning in the header word; in the data words it is treated the same as any other data bit. Once the entire packet has been read, the input port returns to looking at the SOP bit to find the start of the next packet. Idle words between packets are not necessary; the next packet may begin on the word after the last word of the previous packet.

An alternative to this scheme would be to have a separate frame bit for each channel that is 1 during the words of the packet and 0 at all other times. Although this would make it easier to support variable length packets, it adds an extra wire to each channel and wastes twelve bits of bandwidth per packet, whereas the SOP bit wastes only one bit per packet.

2.4 Buffering

The organization of the packet buffers in the input port has a large effect on the router's performance. A simple scheme would have been to organize the packet buffers as a fifo. However, this is inefficient because if the packet at the head of the fifo is blocked, all of the packets behind it become blocked, even if the outputs they need are available. Hence an output channel may sit idle even while there is a packet waiting to be sent out it.

The buffer organization adapted from PaRC uses four buffers in each input port, each one big enough to hold one twelve-word packet. Instead of organizing the buffers as a fifo, they are arranged so that a packet can be read out of any of the buffers at any time. Thus a packet will never be blocked solely because another packet in a different buffer is blocked. In addition, each packet buffer has its own output circuitry, so that different packet buffers within the same input port can be read simultaneously. This allows packets destined to different output ports but buffered at the same input port to be transmitted at the same time.

The organization of the buffers results in behavior which is similar to virtual channel flow control[2]. Its effect is to make effective use of the available bandwidth by utilizing the communication channels to the maximum possible extent. However,

not using a fifo to buffer the packets makes the job of the scheduler more difficult, since it must still ensure that packets traveling between the same input and output are sent in the order that they are received.

2.5 Scheduling

Each output port has its own local scheduler which keeps track of the packets which need to go to the output. Each input port has a request line to each scheduler. When a new packet arrives at the input port, the request line to the output port the packet should go to is raised. The location of the buffer within the input port that the packet is stored in is also sent to the scheduler. The scheduler stores the request information in a fifo. Each time the output port is ready to transmit a new packet, the next request record is read from the fifo and the corresponding packet is read from the packet buffer and transmitted. Since the request records are written to the fifo in the order in which the packets are received, this ensures that packets traveling between the same input and output will not be reordered. It also provides first-come, first-served scheduling for packets arriving on different input channels.

One drawback to this scheduling scheme is that every output port must have enough fifo space to hold a request record from every packet buffer in every input port. This much space is needed in each output port in case the router is completely filled with packets that all need to be transmitted from that output. However, since there are three output ports on the router, this means that at most one-third of the router's fifo buffering space for request records will be in use at any time. Fortunately this is not a great problem, as the size of these fifos is small compared to the other components.

2.6 Flow Control

Flow control between each input port and the corresponding output port on the previous router ensures that the input port does not receive more packets than it has

space to buffer. The mechanism used is a simple start/stop signaling mechanism. When the last of the free buffers in an input port begins to be filled by an incoming packet, the input port raises the WAIT line, signaling the previous router not to send any further packets. When buffering space is freed by transmitting stored packets through one of the output ports, the input port lowers the WAIT signal, allowing the previous router to begin sending packets again. This scheme assures that an input port will never receive more packets than it can buffer as long as the time it takes the WAIT signal to reach the previous router is less than the time it takes that router to finish transmitting the packet it is currently sending.

2.7 Routing Algorithm

The router must decide which of the three output ports an incoming packet should be directed to based upon 15 bits of routing data from the header word of the packet. The way in which this routing is done determines what network topologies the router can be used in. The routing algorithm that was implemented is suitable for use in a butterfly or fat tree network.

Each router has an input that is used to select two bits from among the 15 bits of routing data in the header. These two bits are used to select one of the three output ports to send the packet two. By providing the proper input to each router in the various stages of the network, the destination address can be used as the routing data (figure 2-6). Each pair of bits allows the packet to choose between three outputs, and the last (fifteenth) bit allows a choice between two. Therefore this router can be used in networks having up to $(2)3^7 = 4374$ nodes.

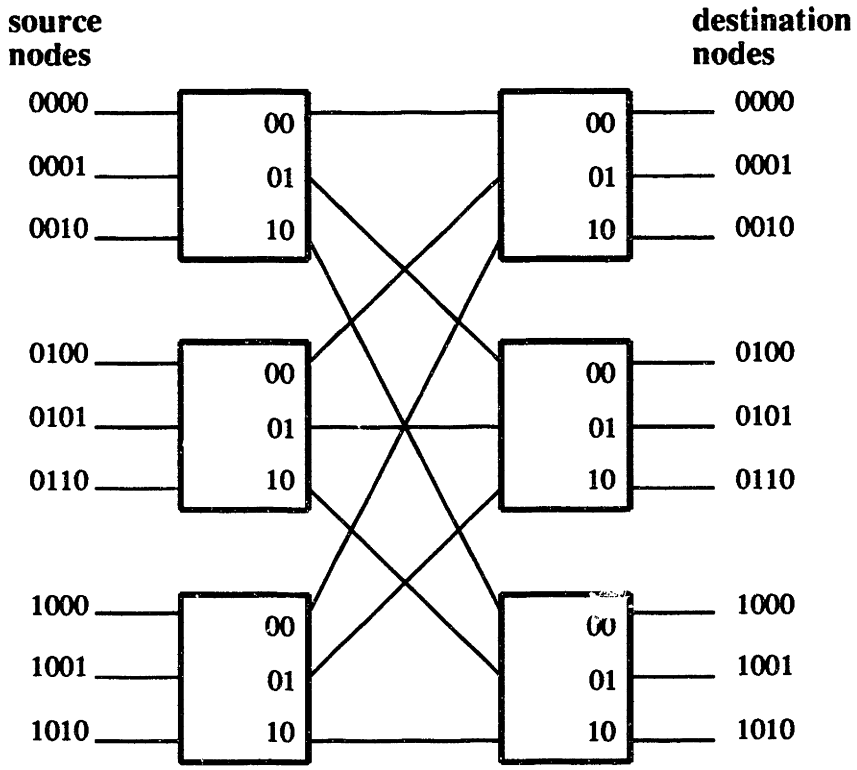

Figure 2-6: Using Destination Address as Routing Data

Chapter 3

Router Implementation

3.1 Verilog

The router design was written and simulated using the Verilog hardware description language[13]. The use of a hardware description language (HDL) allowed the design to be implemented at a functional level, with the actual gate level implementation done by logic synthesis tools. It is analogous to writing a computer program in a high level language and then using a compiler to convert it into machine code versus writing the program in assembly code. With an HDL it is also much easier to make modifications to the design that might require changes to be made to thousands of gates.

Hardware is defined in Verilog in terms of modules. Each module has a specified set of inputs and outputs. Verilog code within the module determines how the outputs behave for given input values. Modules may be instantiated within other modules, so that complex hierarchical designs can be made. The top-level module in the router design is called EZROUTER. It instantiates 3 input ports and three output ports, and specifies how they are connected. In addition, the EZROUTER module specifies the input and output connections of the entire router and how they are connected to the input and output ports.

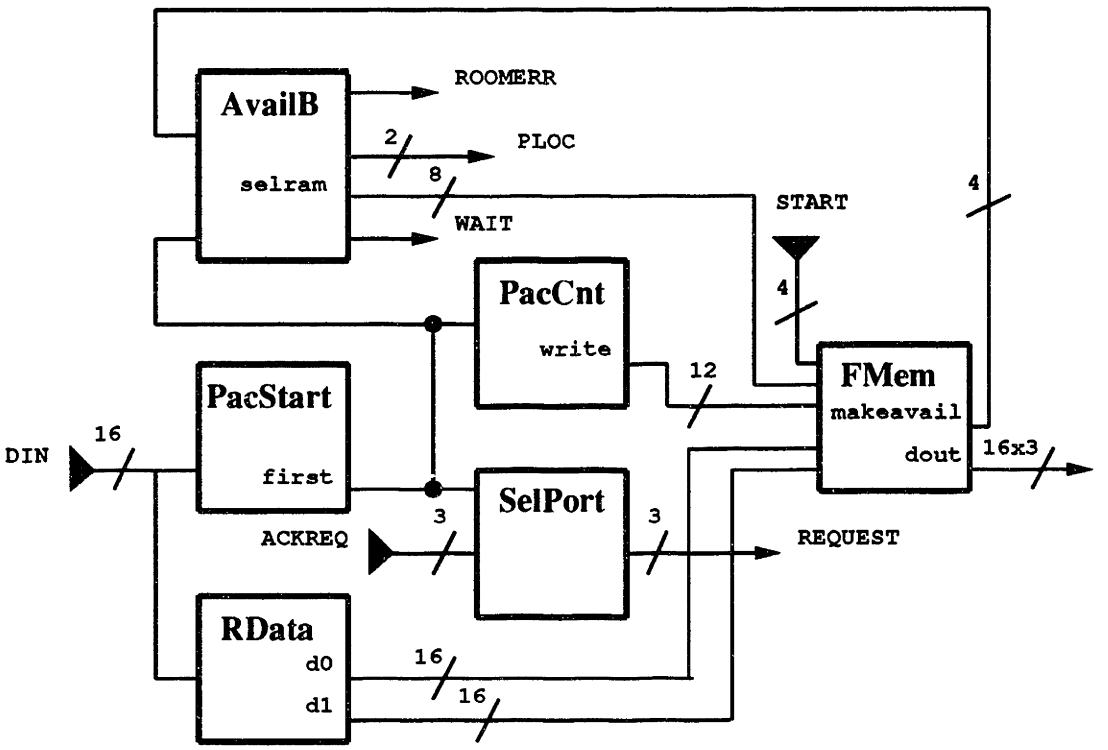

Figure 3-1: Input Port Overview

3.2 Input Port Components

An overview of the input port is shown in figure 3-1. The input port design and the descriptions of its components are based upon those found in [5], which describes the PaRC routing chip. For the most part, the design described here is a subset of the design presented in that report.

Each of the components in the input port is described in turn below. All of the components in the input port are clocked using the external clock provided with the incoming data (XCLK).

3.2.1 Pacstart

The PACSTART component detects the start of a new packet. It raises the FIRST signal high during the header word of a new packet. This signal is used by other modules to control their operation. PACSTART watches the SOP bit, waiting for it to turn to 1. When it does, it raises FIRST for a single clock cycle, then counts out the remaining words of the packet. After the last word of the packet is received, PACSTART resumes checking the SOP bit to find the start of the next packet.

3.2.2 RData

Temporary buffering of the incoming data words is provided by The RDATA component. While a packet is being received, the two most recently received words are made available to the other modules at the d0 and d1 outputs. Whenever an even-numbered word is received it replaces the value at d0, and odd-numbered words similarly replace the value at d1. Thus each data word is available for two cycles, one right after it is received and one after the following word is received. This is necessary to allow proper operation of the FMEM component described in section 3.2.5.

3.2.3 SelPort

The SELPORT component makes the decision where to send the incoming packet. It uses the eight-bit RINF (routing information) signal supplied from a source external to the router to choose two bits from the packet header. Bits [3..0] of RINF are used to choose one of the bits from the header to use as the LSB of the output port to send the packet to, and bits [7..4] are used to chose the MSB. Once SELPORT determines what output port to send the packet to, the request line to that port's scheduler is raised. SELPORT holds the request line high until it receives an acknowledgment signal from the scheduler, indicating that the request record has been stored. SELPORT will not make a request to the scheduler if there is no buffering space to store the incoming packet, although the flow-control mechanism should prevent this from ever happening.

3.2.4 AvailB

The AVAILB component keeps track of which of the packet buffers are available, and determines which of the four buffers in the input port an incoming packet will be stored in. It generates the WAIT flow control signal for the input port whenever all four of the buffers are in use. It also generates a ROOMERR error if a packet arrives when all the buffers are full.

When a new packet arrives, AVAILB chooses an empty packet buffer to store it in and puts that buffer's number on the PLOC output. This information is used by the output port scheduler to keep track of the packet so that it may later be read out of the buffer and transmitted. AVAILB also raises the select lines of the proper registers within the FMEM component (described in 3.2.5) so that the incoming packet will be stored in the chosen buffer. The words of the incoming packet are counted, and the WRITE lines to the packet buffer are raised in order as each word arrives. The select lines are held high until the last word of the packet is written into the buffer. AVAILB then waits for another packet to arrive.

3.2.5 FMem

A diagram of the FMEM component is shown in figure 3-2. It consists of the four packet buffers and three 4-to-1 multiplexors, one for each output port. The output ports directly control the muxes so that they can chose which of the four packet buffers they want to draw data from.

The internal structure of a packet buffer is shown in figure 3-3. Each packet buffer is implemented as a single PACKBUFF module in the Verilog code. Two banks of six 16-bit latches serve as the buffer's storage. These latches are implemented in the WORDREG module. Data is written to a latch whenever its select and write lines are both high. Each latch also has a READ input. The latch drives its output when READ is high, otherwise the output is left floating. When the PACKBUFF receives a START signal from one of the output ports, output counting circuitry asserts the READs one at a time and controls the mux so that the words of the packet appear

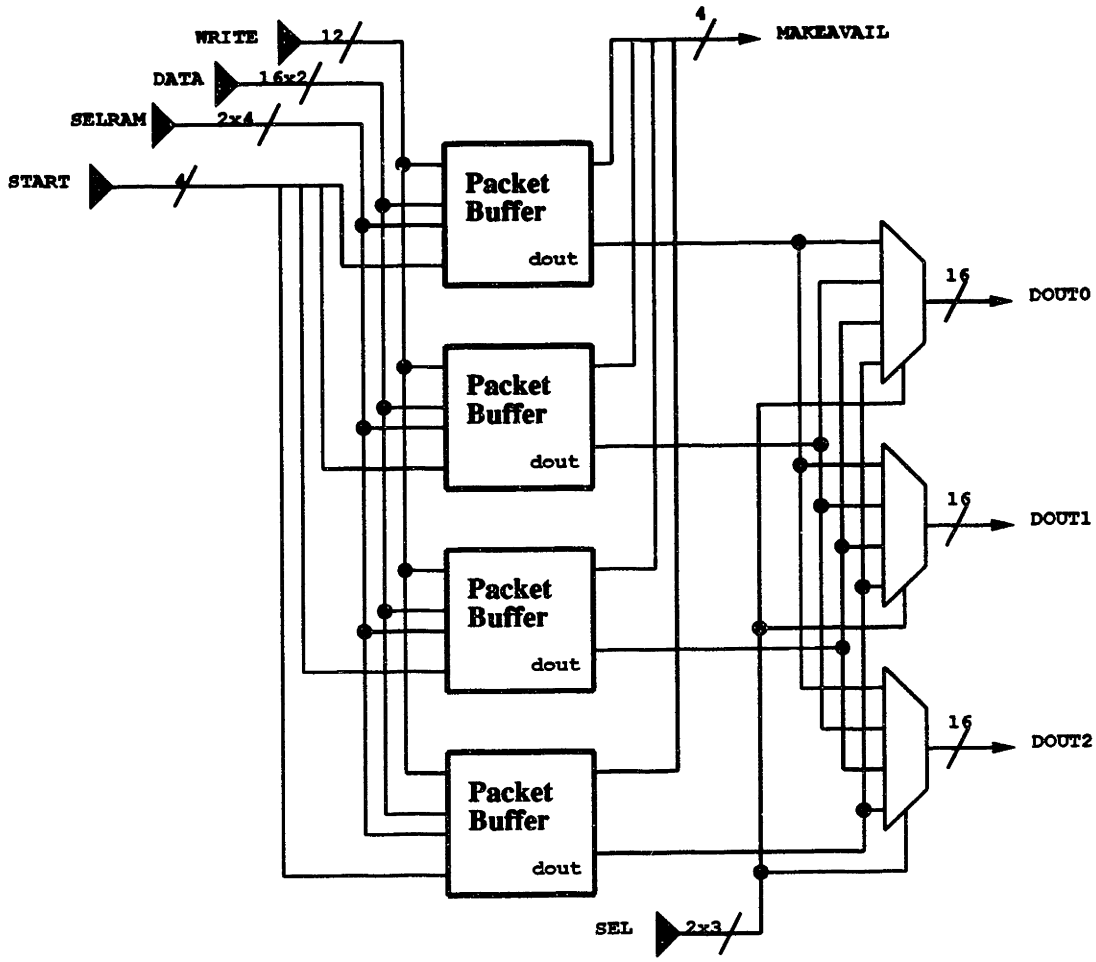

Figure 3-2: FMem Structure

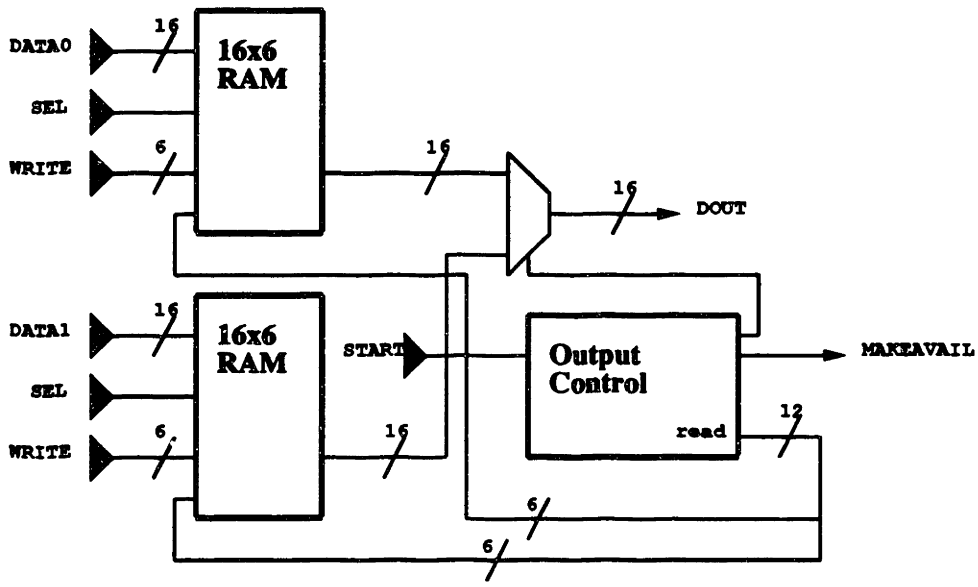

Figure 3-3: Packet Buffer Structure

sequentially at DOUT. The process of reading data out of the PACKBUFF is done using the router's internal clock (ICLK) since that is the clock that the output ports use.

The output counting/control circuitry also keeps track of the availability status of the buffer. Whenever SELECT and WRITE are both high, indicating that the buffer is being written to, MAKEAVAIL is cleared to show that the buffer is no longer available. When the data is read out of the packet buffer in response to a START signal, MAKEAVAIL is set as the last word is read out to show that the buffer is once again empty.

The buffering space in the PACKBUFF is interleaved using the two six-word banks of latches. D0 and SEL0 are the inputs to the first bank, while D1 and SEL1 serve the same purpose in the second. The memory was interleaved in this fashion to make it easier to write the packet data into the buffers. Because there was no simple way to generate control signals that change at times other than the positive clock edge, the WRITE signal to a latch must last a full clock cycle. Since the data must be held

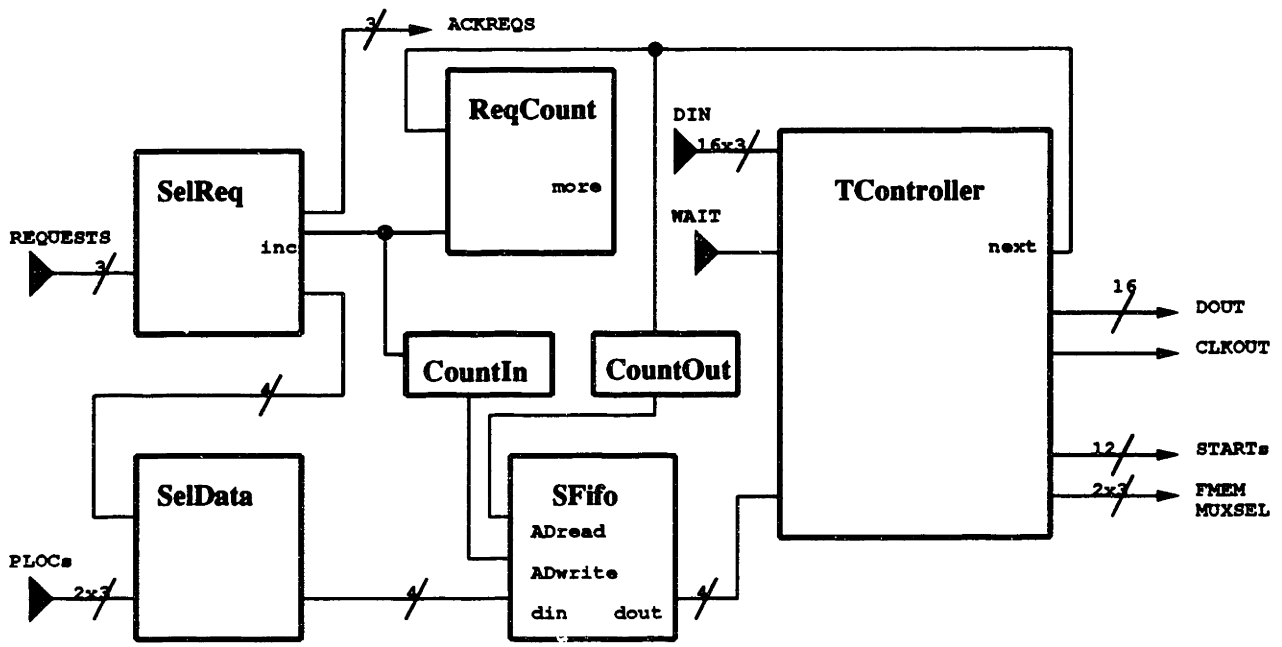

Figure 3-4: Output Port Overview

valid for a time after WRITE is deasserted to assure that the correct data is written to the latch, a write to a latch must take two clock cycles. If the memory were not interleaved, it would not be able to keep up with incoming data as a new word arrives every clock cycle. Interleaving the memory allows the beginning of a two-cycle write to one bank to be overlapped with the end of a two-cycle write to the other bank. In this way the PACKBUFF is able to store a new word every clock cycle.

3.3 Output Port Components

A diagram of the output port is shown in figure 3-4. The output ports are all clocked using the router's internal clock, ICLK. Each of the components of the output port are described below. Again, these descriptions are based upon the design presented in [5].

3.3.1 SelReq

The SELREQ component receives the requests from the input ports. There is one request line input to SELREQ for each of the input ports. If more than one input port makes a request at the same time, SELREQ chooses among them using a fixed priority scheme. SELREQ acknowledges the request, sets the mux inputs in the SELDATA component to read in the PLOC info from the chosen input port, and increments the counter in COUNTIN so that the next request record will be written to the next location in the SFIFO.

3.3.2 SelData

The PLOC data from the input port is combined with the input port number to form the request record in the SELDATA component. The input port number is provided from SELREQ and is also used to select the PLOC data from the proper input port. The request record is then provided as input to the SFIFO.

3.3.3 SFifo

The request records for packets that need to be transmitted from the output port are stored in the SFIFO. Every clock cycle, the output of SELDATA is written into the the SFIFO location pointed to by COUNTIN. Most of the time this will not be a meaningful request record, but the data will be overwritten on the next cycle so it does no harm to write it into the SFIFO. When a meaningful request record is written, SELREQ will also increment COUNTIN so that the request record will not be overwritten. The output of the SFIFO, corresponding to the request record for the packet that is to be transmitted next, is controlled by the COUNTOUT counter.

Because the SFIFO is written to on every clock cycle, an extra location is needed to avoid overwriting needed data when the SFIFO is full. The SFIFO is therefore implemented as 13 latches, enough to hold a request for every packet buffer on the router plus the extra location. Each four-bit latch is implemented as an SREG module. The SREG is identical to the WORDREG in all aspects but its size.

3.3.4 ReqCount

The REQCOUNT component keeps a count of the number of packets that are waiting to be transmitted from the output port. It is incremented when COUNTIN is incremented, and decremented when COUNTOUT is incremented. If the count is greater than zero, REQCOUNT asserts MORE. This signal is used by the TCONTROLLER to determine if it has work to do.

3.3.5 TController

The TCONTROLLER component is responsible for actually reading the packets out of the buffers and transmitting them off the router into the network. If no packet is currently being sent, MORE is high, and the next router in the network is not asserting WAIT, a new packet can be transmitted. The TCONTROLLER gets the request record for the packet to be sent from the SFIFO and asserts NEXT to increment COUNTOUT. The FMEM mux in the input port that holds the packet is set to select the data from the right packet buffer, and a START signal is sent to that buffer. Another mux within the TCONTROLLER is set to send the data coming from the appropriate input port to the output channel. The TCONTROLLER counts the words in the packet as they are sent out, and when the packet is finished being transmitted it waits until it can send out a new packet. When no packet is being sent out, the TCONTROLLER drives the data bits of the output channel with zeros.

Chapter 4

Putting the Router on an FPGA

4.1 Xilinx

The router design was intended to be programmed onto a Xilinx XC4010 FPGA, a member of the XC4000 family[15]. The FPGA is electrically programmable by loading configuration data into its internal memory cells. It can be reprogrammed an unlimited number of times, allowing the design in the FPGA to be modified or replaced with an entirely new design.

The XC4010 consists of a regular matrix of configurable logic blocks (CLBs). These CLBs are the functional elements that are used to construct the user's logic. They are connected by programmable interconnect paths. A customized configuration can be created by programming the internal static memory cells that determine the logic functions and interconnections implemented in the FPGA.

Each CLB consists of a pair of independent four-input function generators and a pair of edge-triggered D-type flip-flops. There is also a third three-input function generator that can implement any Boolean function of the outputs of the first two function generators and a third input which comes from outside the CLB. The outputs of the three function generators can be connected to the outputs of the CLB, or they can be clocked into the flip-flops, whose outputs are then available outside the CLB. A simplified diagram of a XC4000 family CLB is shown in figure 4-1. The XC4010 FPGA contains a 20 by 20 matrix of 400 CLBs, so the router design must be capable of being

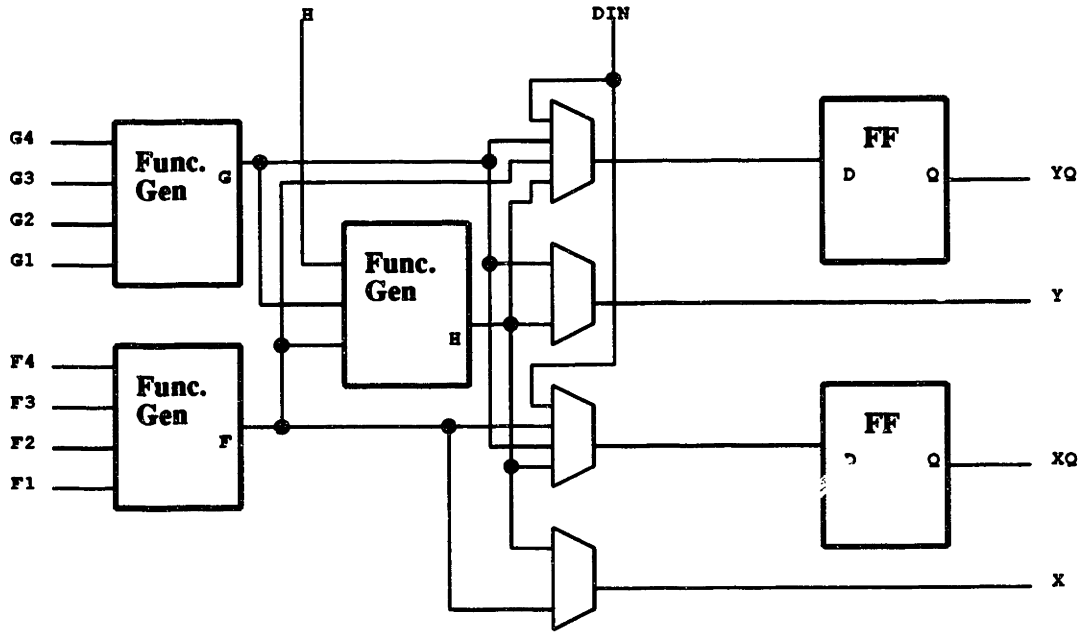

Figure 4-1: XC4000 Configurable Logic Block

implemented using at most this number of CLBs (neglecting routing constraints) if it is to fit on the chip.

4.2 Synopsys Compilation

Once the entire router design was written in Verilog and simulated to make sure there were no obvious design errors, it was synthesized into a gate-level model using the Synopsys hardware synthesis tool[12]. Each module was synthesized in turn. Synopsys works by reading a library file that defines the gates and other macros that are available in the hardware technology being used, then reading the Verilog code that defines the module and compiling it into a collection of these gates and macros. It also provides technical information about the compiled design such as the amount of area it consumes and timing information.

Table 4.1: Sizes of Modules Compiled By Synopsys

module	CLBs
wordreg	8.5
availb	52.0
packbuff	141.5
fmem	644.0
paccnt	23.5
pacstart	10.5
rdata	41.5
sync	1.0
selport	29.0
iport	805.0
sreg	2.0
sfifo	26.0
countin	28.5
countout	28.5
reqcount	11.5
seldata	4.0
selreq	14.5
tcontroller	87.0
oport	197.5
ezrouter	3007.5

4.2.1 Area

The modules of the router design were compiled in Synopsys with the optimizer set to produce hardware designs with minimal area. The sizes of the compiled modules are shown in table 4.1. The entire router uses slightly more than 3000 CLBs, seven and a half times the number of CLBs available on the XC4010. However, this size is misleading because it does not make efficient use of the Xilinx library's macros[15].

When Synopsys compiles the Verilog code and converts it into gates, it uses algorithms designed to convert the code into fast, small blocks of hardware. Unfortunately, just as a C compiler may fail to produce the most efficient machine code, the Synopsys compiler does not always do an efficient job of converting the Verilog code to hardware. For example, a section of Verilog code might be written that implements a four-bit counter, but Synopsys does not interpret it as such. Instead it sees it as

a register whose value changes according to some set of rules. It then designs the counter as a register with some added combinational logic that causes it to be loaded with the right values at the right times. This can be considerably more wasteful than using a predefined counter macro which might have special support in the CLBs.

It is possible to specify the use of macros like the counter macro directly in the Verilog code, assuring that they are used in the hardware generated by Synopsys. However, since these macros are not part of the Verilog language, designs that reference them cannot be simulated in Verilog. This makes it very inconvenient to use them in Verilog designs. Still, it is important to know how much space the design would consume if efficient use was made of the Xilinx macros.

The module that makes the most inefficient use of space is the PACKBUFF module. Rather than using twelve 16-bit latches to provide the memory for packet storage, two 16x8 RAMs could be used with some performance loss. An optional mode for each CLB allows the memory look-up tables in the function generators to be used as a 16x2 or 32x1 bit array of memory cells. A 16x8 RAM can be implemented in four CLBs using the Xilinx macro, so two 16x8 RAMs, enough to store an entire packet, would use only eight CLBs. The remaining circuitry in PACKBUFF uses roughly 40 CLBs[1], so a PACKBUFF could be implemented in 48 CLBs. A 16x4 RAM consuming only two CLBs could also be used to replace the SFIFO.

The COUNTIN and COUNTOUT modules can be replaced by eight bit counter macros. With the addition of the extra logic needed in these modules beyond the counters, each module can be implemented in twelve CLBs. Similarly, the other modules which contain counters can have them replaced by appropriately-sized counter macros, saving further CLBs. Table 4.2 shows the approximate sizes of the modules after considering the addition of these macros. This reduces the area that the router design requires to approximately 1572 CLBs, a savings of roughly 50%.

[1]The other circuitry in PACKBUFF will actually use fewer than 40 CLBs if RAMs are used instead of latches, as the control logic will be simplified.

Table 4.2: Estimated Sizes of Modules Using Xilinx Macros

module	CLBs
wordreg	-
availb	48.5
packbuff	47.5
fmem	268.0
paccnt	5.0
pacstart	7.0
rdata	25.0
sync	1.0
selport	29.0
iport	387.0
sreg	-
sfifo	2.0
countin	12.0
countout	12.0
reqcount	11.5
seldata	4.0
selreq	14.5
tcontroller	83.5
oport	137.0
ezrouter	1572.0

Table 4.3: Maximum Delay Paths

module	max delay path	delay
wordreg	datareg[15] to dout[15]	7.4 ns
availb	avail3 to roomerr	12.0 ns
packbuff	readred[6] to fout[15]	19.8 ns
fmem	packbuff2:readreg[0] to dout0[15]	31.8 ns
paccnt	write_reg[11] to write[11]	12.0 ns
pacstart	wordreg_reg[1] to first	24.0 ns
rdata	dout0_reg[15] to dout0[15]	5.0 ns
sync	dout_reg to dout	5.0 ns
selport	sync2:dout_reg to request[2]	5.0 ns
iport	fmem:packbuff2:readreg[0] to dout0[15]	31.8 ns
sreg	the_data_reg[3] to dout[3]	7.4 ns
sfifo	sreg6:the_data_reg[3] to dout[3]	7.4 ns
countin	fwrite_reg[12] to fwrite[12]	12.0 ns
countout	fread_reg[12] to fread[12]	12.0 ns
reqcount	mycount_reg[1] to more	24.0 ns
seldata	ldout_reg[3] to ldout[3]	6.0ns
selreq	ackreqs_reg[2] to ackreqs[2]	12.0 ns
tcontroller	wordreg_reg[1] to dout[15]	36.0 ns
oport	tcontroller:wordreg_reg[0] to dout[15]	36.0 ns

4.2.2 Technical Details

Synopsys also provides approximate timing data about the compiled design. It identifies the path with the maximum delay in each module. This data is limited in its usefulness, however, since it does not take in to account any information about the placement of the CLBs on the FPGA, the times are merely based on the number of gates a signal must propagate through. Table 4.3 shows the path with the maximum delay for each module.

Unfortunately, it is difficult to get an estimate of the maximum possible clock speed using this data. For example, if the clock period were 20 ns, it would still be all right to have a delay of 24 ns between the counter and MORE in the REQCOUNT module. The only effect would be that when the request count went from zero to one, it would take an extra clock cycle for the TCONTROLLER to recognize that a new packet was waiting to be transmitted, due to the long delay of MORE. Similarly, the

36 ns delay between wordreg and dout in TCONTROLLER is not a limiting factor because it only means that the data will enter the network slightly later than it might have otherwise.

A much better way of determining the maximum clock speed would be to incorporate the timing information provided by Synopsys into the Verilog code. The design could then be simulated in Verilog using different clock speeds to find the maximum speed at which the router still works properly. Software exists to perform exactly this task, but unfortunately it was not available at the time this design was done. Thus it is difficult to make any guess as to the router's maximum clock speed, beyond noting that it is bounded by the 50MHz limit imposed by the underlying Xilinx XC4000 technology.

Chapter 5

Conclusions

Even with optimistic estimates of the space savings that could be achieved by making better use of the macros in the Xilinx library, the router design does not even come close to fitting on a Xilinx XC4010 FPGA. At best it still uses more than 4 times the number of available CLBs. Thus the router design as presented fails in its attempt to provide the functionality of a router that can fit on an FPGA. However, this does not mean that the work done here is useless and that the idea of using an FPGA in a router design is unsound.

Continuing improvements in FPGA technology will soon make it possible to fit this router design on an FPGA. The most powerful member of the XC4000 family, the XC4020, has a matrix of 900 CLBs, more than twice the number in the XC4010 and roughly half the number needed for the router design. Undoubtedly, future models of FPGAs will have even greater numbers of CLBs, allowing them to be configured for more complex designs. Improvements in the synthesis tools would also help to fit the design onto an FPGA. Future versions of Synopsys and other synthesis tools will likely be more intelligent and better able to make efficient use of the gates and macros available in the technology being used, allowing the same design to be fit in a smaller space.

There are a number of possible ways that FPGAs could be used for this router design or other designs without waiting for technology improvements, however. If waiting for the FPGA and synthesis tools is not practical, the design can be changed

instead. An obvious modification to save space would be to reduce the amount of buffering space in each input port. However, this would also degrade the performance of the router. Packets that otherwise might have been able to pass through the router will instead be blocked if the number of buffers is reduced. For example, if the number of buffers is reduced from four to three, and the three buffers all contain packets that want to go to an output port that is blocked, a fourth packet in a router in a previous network stage that wants to go to one of the unblocked outputs will not be able to make any progress. It will be prevented from being sent to the router by the WAIT flow control signal. If instead there were four packet buffers, the fourth packet could use that buffer and be sent out the unblocked output port, passing the three blocked packets.

Another alternative is to eliminate some of the input and output ports. Instead of using a 3x3 router design, a 2x2 design could be used, reducing the size of the router by one-third. This would also add to the typical latency of a packet through the network, however. Since smaller switches would be used, more of them would be needed to connect the same number of nodes, hence a typical packet would have to travel through more routers on its trip from the source to the destination node.

Another option would be to use a different routing technique that required less buffering space, such as circuit switching or wormhole routing. Again, however, the performance of the router using the new switching technique would need to be carefully analyzed to make sure it was adequate for the conditions in which it would typically be used.

The design could also potentially be fit on several FPGAs if it could be partitioned so that part of the design fit on each FPGA. Unfortunately, breaking the design up across many FPGAs is likely to be difficult. The more FPGAs that are used and the more complicated the partitioning of the design across them becomes, the more cumbersome and time consuming it will be to change the design if modifications are needed. Thus, having many FPGAs to deal with instead of just one also negates much of the original advantage of using an FPGA.

One alternative used by the creators of Autonet[11] is to use an FPGA to imple-

ment part of the router design, and implement the rest in conventional hardware. The Autonet router has its scheduler implemented in a Xilinx FPGA, but the rest of the design is done in CMOS. By placing only part of the router on the FPGA it can easily be made to fit. In addition, if the designer knows ahead of time which portions of the design are likely to be modified in the future and which are not, those components that are likely to be modified can be placed on an FPGA. This still allows changes to easily be made to the components that are most likely to need them.

Appendix A

Verilog Code

A.1 availb.v

```
// Available buffer tracker
// Revised:   March 9, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module availb (
  avail0,    // (1 bit) buffer 0 is available
  avail1,    // (1 bit) buffer 1 is available
  avail2,    // (1 bit) buffer 2 is available                    10
  avail3,    // (1 bit) buffer 3 is available
  first,     // (1 bit) high during first word of a new packet
  xclk,      // (1 bit) clock for this input port
  reset,     // (1 bit) resets when high
  roomerr,   // (1 bit) high if no room for a new packet
  ploc,      // (2 bits) buffer packet is being placed into
  selram,    // (8 bits) select lines for buffers
  mywait     // (1 bit) flow control bit, senders should wait when high
);                                                              20

// port inputs and outputs
input avail0;
input avail1;
input avail2;
input avail3;
input first;
input xclk;
input reset;
output roomerr;
output [1:0] ploc; reg [1:0] ploc;                              30
output [7:0] selram; reg [7:0] selram;
output mywait;
```

```
// internal variables
reg [3:0] wordreg;
reg writing;
wire startwrite;

assign startwrite = first & ~mywait;
assign mywait = ~avail0 & ~avail1 & ~avail2 & ~avail3;
assign roomerr = mywait & first;

always @(reset) begin
  if (reset) begin
    wordreg = 4'h0;
      writing = 0;
      ploc = 2'h0;
    selram = 8'h00;
   end
end

always @(posedge xclk) begin
   if (startwrite) begin
      #1 writing = 1;
      wordreg = 4'h0;
  end
end

always @(posedge xclk) begin
  if (writing && wordreg != PACKSIZE-1)
    #1 wordreg = wordreg + 1;
  else if (wordreg == PACKSIZE-1) begin
    #1 wordreg = 4'h0;
      writing = 0;
      selram = 8'h00;
  end
end

always @(posedge xclk) begin
  if (avail0 && startwrite) begin
    #1 ploc = 2'b00;
      selram = 3;
   end
   else if (avail1 && startwrite) begin
      #1 ploc = 2'b01;
    selram = 12;
  end
  else if (avail2 && startwrite) begin
    #1 ploc = 2'b10;
      selram = 48;
   end
   else if (avail3 && startwrite) begin
      #1 ploc = 2'b11;
    selram = 192;
  end
end
```

```
endmodule
```

A.2 countin.v

```
// Pointer for writing to FIFO
// Revised:    April 8, 1993
// Author:     Steve Chamberlin

#include "thesis.h"

module countin (
  iclk,      // (1 bit) internal chip clock
  reset,     // (1 bit) resets when high
  inc,       // (1 bit) increments counter when high                    10
  fwrite     // (BUFFERS*INPORTS+1 bits) writes for FIFO
);

// port inputs and outputs
input iclk;
input reset;
input inc;
output [INPORTS*BUFFERS:0] fwrite; reg[INPORTS*BUFFERS:0] fwrite;

//internal variables                                                    20

always @(reset) begin
  if (reset)
    fwrite = 1;
end

always @(posedge iclk) begin
  if (inc)
    if (fwrite != 13'b1000000000000)
      //4x4 change to:  if (fwrite != 17'b10000000000000000)           30
        #1 fwrite = fwrite << 1;
    else
        #1 fwrite = 1;
end

endmodule
```

A.3 countout.v

```
// Pointer for reading from FIFO
// Revised:    April 8, 1993
// Author:     Steve Chamberlin

#include "thesis.h"
```

```
module countout (
  iclk,      // (1 bit) internal chip clock
  reset,     // (1 bit) resets when high
  next,      // (1 bit) increments counter when high                    10
  fread      // (BUFFERS*INPORTS+1 bits) writes for FIFO
);

// port inputs and outputs
input iclk;
input reset;
input next;
output [INPORTS*BUFFERS:0] fread; reg[INPORTS*BUFFERS:0] fread;

//internal variables                                                    20

always @(reset) begin
  if (reset)
    fread = 1;
end

always @(posedge iclk) begin
  if (next)
    if (fread != 13'b1000000000000)
      //4x4 change to:  if (fread != 17'b10000000000000000)            30
        #1 fread = fread << 1;
    else
        #1 fread = 1;
end

endmodule
```

A.4 ezrouter.v

```
// Simple 3-in 3-out router
// Revised:   April 8, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module ezrouter (
  iclk,      // (1 bit) internal chip clock
  reset,     // (1 bit) resets when high
  din0,      // (16 bits) data input for input port 0                   10
  xclk0,     // (1 bit) clock synced with data on din0
  waitout0,  // (1 bit) wait output for input port 0
  din1,      // (16 bits) data input for input port 1
  xclk1,     // (1 bit) clock synced with data on din1
  waitout1,  // (1 bit) wait output for input port 1
  din2,      // (16 bits) data input for input port 2
  xclk2,     // (1 bit) clock synced with data on din2
  waitout2,  // (1 bit) wait output for input port 2
  //4x4 add these lines:
```

```
  //din3,                                                              20
  //xclk3,
  //waitout3,
  rinf,      // (8 bits) routing control information
  clkout0,   // (1 bit) output clock to be used with dout0
  clkout1,   // (1 bit) output clock to be used with dout1
  clkout2,   // (1 bit) output clock to be used with dout2
  //4x4 add these lines:
  //clkout3,
  dout0,     // (16 bits) data output for output port 0
  waitin0,   // (1 bit) wait input for output port 0                    30
  dout1,     // (16 bits) data output for output port 1
  waitin1,   // (1 bit) wait input for output port 1
  dout2,     // (16 bits) data output for output port 2
  waitin2,   // (1 bit) wait input for output port 2
  //4x4 add these lines:
  //dout3,
  //waitin3,
  roomerr    // (1 bit) pulses high when there is no room for a new packet
);
                                                                        40
// uses 119 i/o pins for 3x3, 155 i/o pins for 4x4.
// if clkout pins are shared, uses 117 pins for 3x3, 152 pins for 4x4.

// port inputs and outputs
input iclk;
input reset;
input [15:0] din0;
input xclk0;
output waitout0;
input [15:0] din1;                                                      50
input xclk1;
output waitout1;
input [15:0] din2;
input xclk2;
output waitout2;
//4x4 add these lines:
//input [15:0] din3;
//input xclk3;
//output waitout3;
input [7:0] rinf;                                                       60
output [15:0] dout0;
input waitin0;
output [15:0] dout1;
input waitin1;
output [15:0] dout2;
input waitin2;
//4x4 add these lines:
//output [15:0] dout3;
//input waitin3;
output roomerr;                                                         70
output clkout0;
output clkout1;
output clkout2;
```

```
//4x4 add these lines:
//output clkout3;

// internal variables
wire rme0;
wire rme1;
wire rme2;                                                    80
//4x4 add these lines:
//wire rme3;
wire [INPORTS*4-1:0] bstart0;
wire [INPORTS*4-1:0] bstart1;
wire [INPORTS*4-1:0] bstart2;
//4x4 add these lines:
//wire [INPORTS*4-1:0] bstart3;
wire [3:0] start0;
wire [3:0] start1;
wire [3:0] start2;                                            90
//4x4 add these lines:
//wire [3:0] start3;
wire [1:0] fmsel0;
wire [1:0] fmsel1;
wire [1:0] fmsel2;
//4x4 add these lines:
//wire [1:0] fmsel3;
wire areq00;
wire areq01;
wire areq02;                                                  100
//4x4 add these lines:
//wire areq03;
wire areq10;
wire areq11;
wire areq12;
//4x4 add these lines:
//wire areq13;
wire areq20;
wire areq21;
wire areq22;                                                  110
//4x4 add these lines:
//wire areq23;
//wire areq30;
//wire areq31;
//wire areq32;
//wire areq33;
wire req00;
wire req01;
wire req02;
//4x4 add these lines:                                        120
//wire req03;
wire req10;
wire req11;
wire req12;
//4x4 add these lines:
//wire req13;
wire req20;
```

```
wire req21;
wire req22;
//4x4 add these lines:                                                    130
//wire req23;
//wire req30;
//wire req31;
//wire req32;
//wire req33;
wire [1:0] ploc0;
wire [1:0] ploc1;
wire [1:0] ploc2;
//4x4 add these lines:
//wire [1:0] ploc3;                                                      140
wire [15:0] dout00;
wire [15:0] dout01;
wire [15:0] dout02;
//4x4 add these lines:
//wire [15:0] dout03;
wire [15:0] dout10;
wire [15:0] dout11;
wire [15:0] dout12;
//4x4 add these lines:
//wire [15:0] dout13;                                                    150
wire [15:0] dout20;
wire [15:0] dout21;
wire [15:0] dout22;
//4x4 add these lines:
//wire [15:0] dout23;
//wire [15:0] dout30;
//wire [15:0] dout31;
//wire [15:0] dout32;
//wire [15:0] dout33;
                                                                         160
assign roomerr = rme0 | rme1 | rme2;
//4x4 change to: assign roomerr = rme0 | rme1 | rme2 | rme3;
assign start0 = bstart0[3:0] | bstart1[3:0] | bstart2[3:0];
//4x4 change to: assign start0 = bstart0[3:0] | bstart1[3:0] | bstart2[3:0] | bstart3[3:0];
assign start1 = bstart0[7:4] | bstart1[7:4] | bstart2[7:4];
//4x4 change to: assign start1 = bstart0[7:4] | bstart1[7:4] | bstart2[7:4] | bstart3[7:4];
assign start2 = bstart0[11:8] | bstart1[11:8] | bstart2[11:8];
//4x4 change to: assign start2 = bstart0[11:8] | bstart1[11:8] | bstart2[11:8] | bstart3[11:8];
//4x4 add these lines:
//assign start3 = bstart0[15:12] | bstart1[15:12] | bstart2[15:12] | bstart3[15:12];   170

iport u0 (din0,start0,{fmsel2,fmsel1,fmsel0},rinf,areq00,areq10,areq20,xclk0,iclk,reset,rme0,waitout0,req00,req01,rec
//4x4 change to:
//iport u0 (din0,start0,{fmsel3,fmsel2,fmsel1,fmsel0},rinf,areq00,areq10,areq20,areq30,xclk0,iclk,reset,rme0,waitout

iport u1 (din1,start1,{fmsel2,fmsel1,fmsel0},rinf,areq01,areq11,areq21,xclk1,iclk,reset,rme1,waitout1,req10,req11,rec
//4x4 change to:
//iport u1 (din1,start1,{fmsel3,fmsel2,fmsel1,fmsel0},rinf,areq01,areq11,areq21,areq31,xclk1,iclk,reset,rme1,waitout

iport u2 (din2,start2,{fmsel2,fmsel1,fmsel0},rinf,areq02,areq12,areq22,xclk2,iclk,reset,rme2,waitout2,req20,req21,rec
//4x4 change to:
```

```
//iport u2 (din2,start2,{fmsel3,fmsel2,fmsel1,fmsel0},rinf,areq02,areq12,areq22,areq32,xclk2,iclk,reset,rme2,waitout

//4x4 add these lines:
//iport u10 (din3,start3,{fmsel3,fmsel2,fmsel1,fmsel0},rinf,areq03,areq13,areq23,areq33,xclk3,iclk,reset,rme3,waitou

oport u3 (iclk,reset,waitin0,req00,req10,req20,areq00,areq01,areq02,ploc0,ploc1,ploc2,dout00,dout10,dout20,fmsel0,
//4x4 change to:
//oport u3 (iclk,reset,waitin0,req00,req10,req20,req30,areq00,areq01,areq02,areq03,ploc0,ploc1,ploc2,ploc3,dout00,d
                                                                                                          190
oport u4 (iclk,reset,waitin1,req01,req11,req21,areq10,areq11,areq12,ploc0,ploc1,ploc2,dout01,dout11,dout21,fmsel1,
//4x4 change to:
//oport u4 (iclk,reset,waitin1,req01,req11,req21,req31,areq10,areq11,areq12,areq13,ploc0,ploc1,ploc2,ploc3,dout01,d

oport u5 (iclk,reset,waitin2,req02,req12,req22,areq20,areq21,areq22,ploc0,ploc1,ploc2,dout02,dout12,dout22,fmsel2,
//4x4 change to:
//oport u5 (iclk,reset,waitin2,req02,req12,req22,req32,areq20,areq21,areq22,areq23,ploc0,ploc1,ploc2,ploc3,dout02,d

//4x4 add these lines:
//oport u11 (iclk,reset,waitin3,req03,req13,req23,req33,areq30,areq31,areq32,areq33,ploc0,ploc1,ploc2,ploc3,dout03,
endmodule
```

A.5 fmem.v

```
// FMEM block
// Revised:   March 8, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module fmem (
  din0,      // (16 bit) data input for bank 0
  din1,      // (16 bit) data input for bank 1
  selectRam, // (2*BUFFERS bits) select bit for banks 0 and 1                    10
  write,     // (PACKSIZE bits) write strobe for each wordreg
  start,     // (BUFFERS bits) buffer should begin outputting contents
  reset,     // (1 bit) resets buffer logic when high
  iclk,      // (1 bit) internal clock for the router
  selectBuf, // (2*OUTPORTS bits) select for multiplexor to output ports
  dout0,     // (16 bits) data out for output port0
  dout1,     // (16 bits) data out for output port1
  dout2,     // (16 bits) data out for output port2
  //4x4 add these lines:
  //dout3,                                                                     20
  makeavail  // (4 bit) signal that a buffer is ready for a new packet
);

// port inputs and outputs
input [15:0] din0;
input [15:0] din1;
input [2*BUFFERS-1:0] selectRam;
input [PACKSIZE-1:0] write;
input [BUFFERS-1:0] start;
```

```
input reset;
input iclk;
input [2*OUTPORTS-1:0] selectBuf;
output [15:0] dout0; reg [15:0] dout0;
output [15:0] dout1; reg [15:0] dout1;
output [15:0] dout2; reg [15:0] dout2;
//4x4 add these lines:
//output [15:0] dout3; reg [15:0] dout3;
output [3:0] makeavail;

//internal variables
wire  [15:0] douta;
wire  [15:0] doutb;
wire  [15:0] doutc;
wire  [15:0] doutd;

packbuff u0 (din0,din1,selectRam[0],selectRam[1],write,start[0],reset,iclk,douta,makeavail[0]);
packbuff u1 (din0,din1,selectRam[2],selectRam[3],write,start[1],reset,iclk,doutb,makeavail[1]);
packbuff u2 (din0,din1,selectRam[4],selectRam[5],write,start[2],reset,iclk,doutc,makeavail[2]);
packbuff u3 (din0,din1,selectRam[6],selectRam[7],write,start[3],reset,iclk,doutd,makeavail[3]);

always @(selectBuf or douta or doutb or doutc or doutd) begin
  #1
  case (selectBuf[1:0])
    0: dout0 = douta;
    1: dout0 = doutb;
    2: dout0 = doutc;
    3: dout0 = doutd;
  endcase
  case (selectBuf[3:2])
    0: dout1 = douta;
    1: dout1 = doutb;
    2: dout1 = doutc;
    3: dout1 = doutd;
  endcase
  case (selectBuf[5:4])
    0: dout2 = douta;
    1: dout2 = doutb;
    2: dout2 = doutc;
    3: dout2 = doutd;
  endcase
  //4x4 add these lines:
  //case (selectBuf[7:6])
  //   0: dout3 = douta;
  //   1: dout3 = doutb;
  //   2: dout3 = doutc;
  //   3: dout3 = doutd;
  //endcase
end

endmodule
```

A.6 iport.v

```
// Input Port
// Revised:   March 9, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module iport (
  din,      // (16 bit) data input
  start,    // (4 bits) start signals for each buffer
  selbuf,   // (OUTPORTS*2 bits) 2 bits for each output multiplexor                10
  rinf,     // (8 bits) routing information control
  ackreq0,  // (1 bit) acknowledgement of port 0 request
  ackreq1,  // (1 bit) acknowledgement of port 1 request
  ackreq2,  // (1 bit) acknowledgement of port 2 request
  //4x4 add these lines:
  //ackreq3,
  xclk,     // (1 bit) clock for input port
  iclk,     // (1 bit) internal clock for the router
  reset,    // (1 bit) resets buffer logic when high
  roomerr,  // (1 bit) no room for an incoming packet error                        20
  mywait,   // (1 bit) flow control tells sender to wait
  req0,     // (1 bit) request line for output port 0
  req1,     // (1 bit) request line for output port 1
  req2,     // (1 bit) request line for output port 2
  //4x4 add these lines:
  //req3,
  ploc,     // (2 bits) buffer number of packet making request
  dout0,    // (16 bits) data out to output port 0
  dout1,    // (16 bits) data out to output port 1
  dout2     // (16 bits) data out to output port 2                                30
  //4x4 add these lines:
  //dout3
);

// port inputs and outputs
input [15:0] din;
input [3:0] start;
input [OUTPORTS*2-1:0] selbuf;
input [7:0] rinf;
input xclk;                                                                      40
input iclk;
input reset;
input ackreq0;
input ackreq1;
input ackreq2;
//4x4 add these lines:
//input ackreq3;
output roomerr;
output mywait;
output req0;                                                                     50
output req1;
```

```
output req2;
//4x4 add these lines:
//output req3;
output [1:0] ploc;
output [15:0] dout0;
output [15:0] dout1;
output [15:0] dout2;
//4x4 add these lines:
//output [15:0] dout3;                                                    60

//internal variables
wire [15:0] d0;
wire [15:0] d1;
wire first;
wire avail0;
wire avail1;
wire avail2;
wire avail3;
wire savail0;                                                             70
wire savail1;
wire savail2;
wire savail3;
wire [7:0] sel;
wire [PACKSIZE-1:0] write;


fmem u0 (d0,d1,sel,write,start,reset,iclk,selbuf,dout0,dout1,dout2,{avail3,avail2,avail1,avail0});
//4x4 change to:
//fmem u0 (d0,d1,sel,write,start,reset,iclk,selbuf,dout0,dout1,dout2,dout3,{avail3,avail2,avail1,avail0});
pacstart u1 (din,reset,xclk,first);
rdata u2 (din,xclk,reset,d0,d1);
paccnt u3 (first,xclk,reset,write);
sync u4 (avail0,xclk,savail0);
sync u5 (avail1,xclk,savail1);
sync u6 (avail2,xclk,savail2);
sync u7 (avail3,xclk,savail3);
availb u8 (savail0,savail1,savail2,savail3,first,xclk,reset,roomerr,ploc,sel,mywait);
selport u9 (first,xclk,iclk,avail0,avail1,avail2,avail3,reset,{ackreq2,ackreq1,ackreq0},din,rinf,{req2,req1,req0});
//4x4 change to:                                                          90
//selport u9 (first,xclk,iclk,avail0,avail1,avail2,avail3,reset,{ackreq3,ackreq2,ackreq1,ackreq0},din,rinf,{req3,req2,req
endmodule
```

A.7 oport.v

```
// Output Port
// Revised:   April 9, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module oport (
  iclk,      // (1 bit) internal clock for the router
```

```
 reset,     // (1 bit) resets buffer logic when high
 mywait,    // (1 bit) inhibits sending of packets when high
 req0,      // (1 bit) request line from input port 0
 req1,      // (1 bit) request line from input port 1
 req2,      // (1 bit) request line from input port 2
 //4x4 add these lines:
 //req3,
 areq0,     // (1 bit) acknowledgement of iport 0's request
  areq1,       // (1 bit) acknowledgement of iport 1's request
 areq2,     // (1 bit) acknowledgement of iport 2's request
  //4x4 add these lines:
  //areq3,
  ploc0,      // (2 bits) buffer number of packet from port 0
  ploc1,      // (2 bits) buffer number of packet from port 1
  ploc2,      // (2 bits) buffer number of packet from port 2
  //4x4 add these lines:
  //ploc3,
  din0,       // (16 bits) data from input port 0
  din1,       // (16 bits) data from input port 1
  din2,       // (16 bits) data from input port 2
  //4x4 add these lines:
  //din3,
  fmuxsel,    // (2 bits) mux select for fmem block
  start,      // (BUFFERS*INPORTS bits) start lines for input buffers
  dout,       // (16 bits) data out to the network
  clkout      // (1 bit) clock out to the network
);

// port inputs and outputs
input iclk;
input reset;
input req0;
input req1;
input req2;
//4x4 add these lines:
//input req3;
input [1:0] ploc0;
input [1:0] ploc1;
input [1:0] ploc2;
//4x4 add these lines:
//input [1:0] ploc3;
input [15:0] din0;
input [15:0] din1;
input [15:0] din2;
//4x4 add these lines:
//input [15:0] din3;
input mywait;
output [11:0] start;
output [1:0] fmuxsel;
output areq0;
output areq1;
output areq2;
//4x4 add these lines:
//output areq3;
```

```
output [15:0] dout;
output clkout;

//internal variables
wire more;
wire next;
wire inc;
wire [1:0] portnum;                                                    70
wire [3:0] sfin;
wire [3:0] sfout;
wire [INPORTS*BUFFERS:0] write;
wire [INPORTS*BUFFERS:0] read;

sfifo u0 (write,read,sfin,sfout);
selreq u1 (iclk,{req2,req1,req0},reset,{areq2,areq1,areq0},inc,portnum);
//4x4 change to:
//selreq u1 (iclk,{req3,req2,req1,req0},reset,{areq3,areq2,areq1,areq0},inc,portnum);
seldata u2 (iclk,ploc0,ploc1,ploc2,portnum,sfin);                      80
//4x4 change to:
//seldata u2 (iclk,ploc0,ploc1,ploc2,ploc3,portnum,sfin);
reqcount u3 (iclk,reset,inc,next,more);
countin u4 (iclk,reset,inc,write);
countout u5 (iclk,reset,next,read);
tcontroller u6 (iclk,reset,more,mywait,sfout,din0,din1,din2,next,start,fmuxsel,dout,clkout);
//4x4 change to:
//tcontroller u6 (iclk,reset,more,mywait,sfout,din0,din1,din2,din3,next,start,fmuxsel,dout,clkou
endmodule
                                                                       90
```

A.8 paccnt.v

```
// Packet word counter and write strobe generator
// Revised:   March 9, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module paccnt(
  first,     // (1 bit) high during first word of a new packet
  xclk,      // (1 bit) clock for this input port
  reset,     // (1 bit) resets when high                              10
  write      // (PACKSIZE bits) write strobes for packet buffers
);

// port inputs and outputs
input first;
input xclk;
input reset;
output [PACKSIZE-1:0] write; reg [PACKSIZE-1:0] write;

always @(reset) begin                                                  20
  if (reset)
    #1 write = 0;
```

```
end

always @(posedge xclk) begin
  if (first) begin
    #1 write = 1;
  end
  else if (write != 0 && write != 1<<(PACKSIZE-1))
    #1 write = write << 1;
  else if (write == 1<<(PACKSIZE-1))
    #1 write = 0;
end

endmodule
```

A.9 packbuff.v

```
// Packet buffer consisting of 2 interleaved RAM banks
// Revised:   March 8, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module packbuff (
  din0,      // (16 bit) data input for bank 0
  din1,      // (16 bit) data input for bank 1
  sel0,      // (1 bit) select for bank 0
  sel1,      // (1 bit) select for bank 1
  write,     // (PACKSIZE bits) write strobe for each wordreg
  start,     // (1 bit) signal for this buffer to begin outputting contents
  reset,     // (1 bit) resets buffer logic when high
  iclk,      // (1 bit) internal clock for the router
  dout,      // (16 bits) data out
  makeavail  // (1 bit) signal that this buffer is ready for a new packet
);

// port inputs and outputs
input [15:0] din0;
input [15:0] din1;
input sel0;
input sel1;
input [PACKSIZE-1:0] write;
input start;
input reset;
input iclk;
output [15:0] dout; reg [15:0] dout;
output makeavail; reg makeavail;

//internal variables
wire [15:0] dout0;
wire [15:0] dout1;
reg [PACKSIZE-1:0] read;
reg muxselect;
```

```
//depending on PACKSIZE, instantiate as many wordregs as needed, interleaving
//them so din0 and dout0 serve wordregs with even write and read numbers.
wordreg u0 (din0,dout0,sel0,write[0],read[0]);
wordreg u1 (din0,dout0,sel0,write[2],read[2]);
wordreg u2 (din0,dout0,sel0,write[4],read[4]);
wordreg u3 (din0,dout0,sel0,write[6],read[6]);
wordreg u4 (din0,dout0,sel0,write[8],read[8]);
wordreg u5 (din0,dout0,sel0,write[10],read[10]);

wordreg u6 (din1,dout1,sel1,write[1],read[1]);
wordreg u7 (din1,dout1,sel1,write[3],read[3]);
wordreg u8 (din1,dout1,sel1,write[5],read[5]);
wordreg u9 (din1,dout1,sel1,write[7],read[7]);
wordreg u10 (din1,dout1,sel1,write[9],read[9]);
wordreg u11 (din1,dout1,sel1,write[11],read[11]);

always @(muxselect or dout0 or dout1) begin
  if (muxselect) #1 dout = dout1;
  else #1 dout = dout0;
end

always @(reset) begin
  if (reset) begin
    read = 3;
    makeavail = 1;
    muxselect = 0;
   end
end

always @(write or sel0 or sel1) begin
  if ((write != 0) && (sel0 || sel1))
    #1 makeavail = 0;
end

always @(posedge iclk) begin
  if (start && read == 3) begin
    #1 muxselect = 1;
    read = read << 1;
  end
  else if (read != 3 && read != 1<<(PACKSIZE-1)) begin
    #1 muxselect = ~muxselect;
    read = read << 1;
  end
  else if (read == 1<<(PACKSIZE-1)) begin
    #1 muxselect = 0;
    read = 3;
  end
end

always @(posedge iclk) begin
  if (read == 3<<(PACKSIZE-2))
    #1 makeavail = 1;
end
```

```
endmodule
```

A.10 pacstart.v

```
// Packet Start detector
// Revised:   March 9, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module pacstart(
  din,          // (16 bits) input port data in
  reset,      // (1 bit) resets when high
  xclk,       // (1 bit) clock for this input port                    10
  first       // (1 bit) high for first word of a new packet
);

// port inputs and outputs
input [15:0] din;
input reset;
input xclk;
output first; reg first;

// internal variables                                                  20
reg [3:0] wordreg;

always @(wordreg or din) begin
  if (wordreg != 0) #1 first = 0;
  else #1 first = din[SOP];
end

always @(reset) begin
  if (reset)
    wordreg = 0;                                                      30
end

always @(posedge xclk) begin
  if (din[SOP] && wordreg == 0)
    #1 wordreg = 1;
  else if (wordreg != 0 && wordreg != PACKSIZE - 1)
    #1 wordreg = wordreg + 1;
  else if (wordreg == PACKSIZE - 1)
    #1 wordreg = 0;
end                                                                   40

endmodule
```

A.11 rdata.v

```
// Received data generator
// Revised:    March 9, 1993
// Author:     Steve Chamberlin

#include "thesis.h"

module rdata(
  din,          // (16 bits) input port data in
  xclk,       // (1 bit) clock for this input port
  reset,      // (1 bit) resets when high
  dout0,      // (16 bits) even data words
  dout1       // (16 bits) odd data words
);

// port inputs and outputs
input [15:0] din;
input reset;
input xclk;
output [15:0] dout0; reg [15:0] dout0;
output [15:0] dout1; reg [15:0] dout1;

// internal variables
reg [3:0] wordreg;

always @(reset) begin
  if (reset)
    #1 wordreg = 0;
end

always @(posedge xclk) begin
  if (~wordreg[0])
    #1 dout0 = din;
  else
    #1 dout1 = din;
end

always @(posedge xclk) begin
  if (din[SOP] && wordreg == 0)
    #1 wordreg = 1;
  else if (wordreg != 0 && wordreg != PACKSIZE-1)
    #1 wordreg = wordreg + 1;
  else if (wordreg == PACKSIZE-1)
    #1 wordreg = 0;
end

endmodule
```

A.12 reqcount.v

```
// Request Counter
// Revised:    April 8, 1993
// Author:     Steve Chamberlin
```

```
#include "thesis.h"

module reqcount (
  iclk,      // (1 bit) internal chip clock
  reset,     // (1 bit) resets when high
  inc,       // (1 bit) increments counter when high
  next,      // (1 bit) decrements counter when high
  more       // (1 bit) high if there are queued requests
);

// port inputs and outputs
input iclk;
input reset;
input inc;
input next;
output more;

//internal variables
reg [3:0] mycount;

assign more = mycount != 0;

always @(reset) begin
  if (reset)
    mycount = 0;
end

always @(posedge iclk) begin
  if (inc && ~next)
    mycount = mycount + 1;
  else if (~inc && next)
    mycount = mycount - 1;
end

endmodule
```

A.13 seldata.v

```
// Select Packet Info Block
// Revised:   April 8, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module seldata (
  iclk,      // (1 bit) internal chip clock
  ploc0,     // (2 bits) ploc from input port 0
  ploc1,     // (2 bits) ploc from input port 1
  ploc2,     // (2 bits) ploc from input port 2
  //4x4 add these lines:
  //ploc3,
```

```
  muxsel,    // (2 bits) mux selector
  ldout      // (4 bits) packet description info
);

// port inputs and outputs
input iclk;
input [1:0] ploc0;                                                    20
input [1:0] ploc1;
input [1:0] ploc2;
//4x4 add these lines:
//input [1:0] ploc3;
input [1:0] muxsel;
output [3:0] ldout; reg[3:0] ldout;

//internal variables
wire [3:0] dout;
                                                                      30
assign dout[3:2] = muxsel;
assign dout[1] = (ploc0[1] & ~muxsel[0] & ~muxsel[1]) |
                 (ploc1[1] & muxsel[0] & ~muxsel[1]) |
                 (ploc2[1] & ~muxsel[0] & muxsel[1]);
//4x4 change last term to:
// (ploc3[1] & muxsel[0] & muxsel[1])

assign dout[0] = (ploc0[0] & ~muxsel[0] & ~muxsel[1]) |
                 (ploc1[0] & muxsel[0] & ~muxsel[1]) |
                 (ploc2[0] & ~muxsel[0] & muxsel[1]);                 40
//4x4 change last term to:
// (ploc3[0] & muxsel[0] & muxsel[1])

always @(dout or iclk) begin
  if (~iclk)
    #1 ldout = dout;
end

endmodule
                                                                      50
```

A.14 selport.v

```
// Selects which port packet should be sent to
// Revised:   April 7, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module selport(
  first,     // (1 bit) high during first word of a new packet
  xclk,      // (1 bit) clock for this input port
  iclk,      // (1 bit) internal clock                                 10
  avail0,    // (1 bit) buffer 0 available
  avail1,    // (1 bit) buffer 1 available
  avail2,    // (1 bit) buffer 2 available
```

```
  avail3,     // (1 bit) buffer 3 available
  reset,      // (1 bit) resets when high
  ackreq,     // (OUTPORTS bits) ack of request from each output port
  din,        // (16 bits) incoming packet data
  rinf,       // (8 bits) routing information control
  request     // (OUTPORTS bits) requests to each output port, synced to iclk
);

// port inputs and outputs
input first;
input xclk;
input iclk;
input avail0;
input avail1;
input avail2;
input avail3;
input reset;
input [OUTPORTS-1:0] ackreq;
input [15:0] din;
input [7:0] rinf;
output [OUTPORTS-1:0] request;

reg [1:0] portreg;
reg [OUTPORTS-1:0] xreq;

sync u0 (xreq[0],iclk,request[0]);
sync u1 (xreq[1],iclk,request[1]);
sync u2 (xreq[2],iclk,request[2]);
//4x4 add these lines:
//sync u3 (xreq[3],iclk,request[3]);

always @(reset) begin
  if (reset) begin
    #1 xreq = 0;
    portreg = 0;
  end
end

always @(posedge xclk) begin
  if (first && (avail0 | avail1 | avail2 | avail3)) begin
    // don't make a request if packet can't be stored in a buffer
    #1 portreg[0] = din[rinf[3:0]];
    portreg[1] = din[rinf[7:4]];
    //may need to wait a clock cycle here?
    xreq[portreg] = 1;  // this will die if portreg == 3 when using 3x3
  end
end

always @(ackreq) begin
  if (ackreq != 0)
    #1 xreq = 0;
end

endmodule
```

A.15 selreq.v

```
// Select Request Block
// Revised:   April 8, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module selreq (
  iclk,      // (1 bit) internal chip clock
  requests,  // (INPORTS bits) request lines fr      ports
  reset,     // (1 bit) resets when high
  ackreqs,   // (INPORTS bits) acknowledgements of requests
  inc,       // (1 bit) increments count of queued requests
  muxsel     // (2 bits) selects ploc info from proper input port
);

// port inputs and outputs
input iclk;
input [INPORTS-1:0] requests;
input reset;
output [INPORTS-1:0] ackreqs; reg [INPORTS-1:0] ackreqs;
output inc; reg inc;
output [1:0] muxsel; reg [1:0] muxsel;

//internal variables

always @(reset) begin
  if (reset) begin
    ackreqs = 0;
    inc = 0;
  end
end

always @(posedge iclk) begin
  if (requests[0] && ~ackreqs[0]) begin
    #1 ackreqs[0] = 1;
    inc = 1;
    muxsel = 0;
  end
  else if (requests[1] && ~ackreqs[1]) begin
    #1 ackreqs[1] = 1;
    inc = 1;
    muxsel = 1;
  end
  else if (requests[2] && ~ackreqs[2]) begin
    #1 ackreqs[2] = 1;
    inc = 1;
    muxsel = 2;
  end
  //4x4 add these lines:
```

```
  //else if (requests[3] && ~ackreqs[3]) begin
  //   #1 ackreqs[3] = 1;
  //   inc = 1;
  //   muxsel = 3;
  //end
  else #1 inc = 0;
end

always @(posedge iclk) begin
  if (ackreqs[0] && ~requests[0])
    #1 ackreqs[0] = 0;
  if (ackreqs[1] && ~requests[1])
    #1 ackreqs[1] = 0;
  if (ackreqs[2] && ~requests[2])
    #1 ackreqs[2] = 0;
  //4x4 add these lines:
  //if (ackreqs[3] && ~requests[3])
  //   #1 ackreqs[3] = 0;

end

endmodule
```

A.16 sfifo.v

```
// SFIFO
// Revised:   April 8, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module sfifo (
  write,    // (BUFFERS*INPORTS+1 bits) write lines for fifo locations
  read,     // (BUFFERS*INPORTS+1 bits) read lines for fifo locations
  din,       // (4 bits) data input
  dout      // (4 bits) data output
);

//port inputs and outputs
input [3:0] din;
input [BUFFERS*INPORTS:0] write;
input [BUFFERS*INPORTS:0] read;
output [3:0] dout;

sreg u0 (din,dout,write[0],read[0]);
sreg u1 (din,dout,write[1],read[1]);
sreg u2 (din,dout,write[2],read[2]);
sreg u3 (din,dout,write[3],read[3]);
sreg u4 (din,dout,write[4],read[4]);
sreg u5 (din,dout,write[5],read[5]);
sreg u6 (din,dout,write[6],read[6]);
sreg u7 (din,dout,write[7],read[7]);
```

```
sreg u8 (din,dout,write[8],read[8]);
sreg u9 (din,dout,write[9],read[9]);
sreg u10 (din,dout,write[10],read[10]);
sreg u11 (din,dout,write[11],read[11]);
sreg u12 (din,dout,write[12],read[12]);
//4x4 add these lines:
//sreg u13 (din,dout,write[13],read[13]);
//sreg u14 (din,dout,write[14],read[14]);
//sreg u15 (din,dout,write[15],read[15]);
//sreg u16 (din,dout,write[16],read[16]);


endmodule
```

A.17 sreg.v

```
// 4-bit custom word register
// Revised:    April 8, 1993
// Author:     Steve Chamberlin

#include "thesis.h"

module sreg (
  din,      // (4 bit) data input
  dout,     // (4 bit) data out
  write,    // (1 bit) write strobe, active high
  read      // (1 bit) wordreg drives dout when read is high, z otherwise
);

//port inputs and outputs
input [3:0] din;
input write;
input read;
output [3:0] dout; reg [3:0] dout;

//internal registers and variables
reg [3:0] the_data;

always @(read or the_data) begin
  if (read) #1 dout = the_data;
  else #1 dout = 4'bzzzz;
end

always @(write or din) begin
   if (write)
      the_data = din;
end

endmodule
```

A.18 sync.v

```
// synchronizer
// Revised:   March 9, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module sync(
  din,      // (1 bit) data in
  newclk,   // (1 bit) clock to synchronize to
  dout      // (1 bit) data out
);

// port inputs and outputs
input din;
input newclk;
output dout; reg dout;

// internal
reg a;

always @(posedge newclk) begin
  dout = a;
  #1 a = din;
end

endmodule
```

A.19 tcontroller.v

```
// Transmitter Controller
// Revised:   April 8, 1993
// Author:    Steve Chamberlin

#include "thesis.h"

module tcontroller (
  iclk,      // (1 bit) internal chip clock
  reset,     // (1 bit) resets when high
  more,      // (1 bit) high when there are packets mywaiting to be sent
  mywait,    // (1 bit) low if receiver can accept more packets
  pacinfo,   // (4 bits) port and buffer number of packet
  din0,      // (16 bits) data input from port0
  din1,      // (16 bits) data input from port1
  din2,      // (16 bits) data input from port2
  //4x4 add these lines:
  //din3,
  next,      // (1 bit) high when transmitter wants info on next packet
  start,     // (BUFFERS*INPORTS bits) start lines for input buffers
  fmuxsel,   // (2 bits) mux select for fmem block
```

```
  dout,      // (16 bits) data out to network
  clkout     // (1 bit) clk out to network
);

// port inputs and outputs
input iclk;
input reset;
input more;
input mywait;
input [3:0] pacinfo;                                                    30
input [15:0] din0;
input [15:0] din1;
input [15:0] din2;
//4x4 add these lines:
//input [15:0] din3;
output next; reg next;
output [BUFFERS*INPORTS-1:0] start; reg [BUFFERS*INPORTS-1:0] start;
output [1:0] fmuxsel; reg [1:0] fmuxsel;
output [15:0] dout; reg [15:0] dout;
output clkout;                                                          40

// internal variables
reg [3:0] wordreg;
reg sending;
reg [1:0] portmux;
reg [15:0] mout;

assign clkout = ~iclk;

always @(reset) begin                                                   50
  if (reset) begin
    wordreg = 0;
    start = 0;
    next = 0;
    fmuxsel = 0;
    dout = 0;
    sending = 0;
  end
end
                                                                        60
always @(mout or wordreg) begin
  if (wordreg != 0)
    #1 dout = mout;
  else
    #1 dout = 0;
end

always @(din0 or din1 or din2 or portmux) begin
//4x4 change to:
//always @(din0 or din1 or din2 or din3 or portmux) begin               70
  case (portmux)
    0: mout = din0;
    1: mout = din1;
    2: mout = din2;
```

```
      3: mout = din2;
      //4x4 change to:
      //3: mout = din3;
    endcase
end
                                                                    80

always @(posedge iclk) begin
  if (more && ~mywait && ~sending) begin
    #1 sending = 1;
    fmuxsel = pacinfo[1:0];
    portmux = pacinfo[3:2];
    wordreg = 1;
    start[(pacinfo[3:2]*4)+pacinfo[1:0]] = 1;
    next = 1;
  end
  else if (sending && wordreg != PACKSIZE-1) begin                  90
    next = 0;
    start[(pacinfo[3:2]*4)+pacinfo[1:0]] = 0;
    wordreg = wordreg + 1;
  end
  else if (wordreg == PACKSIZE-1) begin
    wordreg = wordreg + 1;
    sending = 0;
  end
  else if (wordreg == PACKSIZE)
    wordreg = 0;                                                    100
end

endmodule
```

A.20 thesis.h

```
#define INPORTS 3
#define OUTPORTS 3

/* INPORTS and OUTPORTS cannot be changed without also modifying the code.
   There are comments in the code explaining what changes need to be made to
   use 4x4 rather than 3x3. Other sizes can also be made, making changes in
   the code analagous to those for 4x4. */

#define BUFFERS 4      /* number of packet buffers per input port. currently,
                          changing this parameter will cause everything to      10
                          break. To get things working with BUFFERS != 4,
                          significant changes to the code are necessary. */
#define PACKSIZE 12    /* number of 16-bit words in a packet. some regs will
                          have to be resized if PACKSIZE is made > 15 */
#define SOP 15         /* start of packet bit */
```

A.21 wordreg.v

```
// 16-bit custom word register
// Revised:    March 8, 1993
// Author:     Steve Chamberlin

#include "thesis.h"

module wordreg (
  din,      // (16 bit) data input
  dout,     // (16 bit) data out
  select,   // (1 bit) selects this buffer for writing if high                    10
  write,    // (1 bit) write strobe, active high
  read      // (1 bit) wordreg drives dout when read is high, z otherwise
);

//port inputs and outputs
input [15:0] din;
input select;
input write;
input read;
output [15:0] dout; reg [15:0] dout;                                            20

//internal registers and variables
reg [15:0] the_data;

always @(read or the_data) begin
  if (read) #1 dout = the_data;
  else #1 dout = 16'bzzzzzzzzzzzzzzzz;
end

always @(select or write or din) begin                                         30
   if (select && write) begin

      the_data = din;
   end
end

endmodule
```

Bibliography

[1] H. Ahmadi and W. G. Denzel. A survey of modern high-performance switching techniques. *IEEE Journal on Selected Areas in Communications*, 7(7), September 1989.

[2] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), March 1992.

[3] W.J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39, June 1990.

[4] G. L. Frazier and Y. Tamir. Hardware support for high priority traffic in VLSI communication switches. Technical report, UCLA Computer Science Department, December 1990.

[5] C. F. Joerg. Design and implementation of a packet switched routing chip. Technical report, Massachusetts Institute of Technology, December 1990.

[6] P. Kermani and L. Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks*, 3, September 1979.

[7] P. Kermani and L. Kleinrock. A tradeoff study of switching systems in computer communication networks. *IEEE Transactions on Computers*, C-29, December 1980.

[8] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2), February 1993.

[9] S. Owicki and A. Karlin. Factors in the performance of the AN1 computer network. In *1992 ACM SIGMETRICS and PERFORMANCE '92 Conference on Measurement and Modeling of Computer Systems*, June 1992.

[10] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token store architecture. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[11] M. Schroeder, A. Birrel, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: A high speed, self-configuring local area network using point to point links. *IEEE Journal on Selected Areas in Communications*, 9(8), October 1992.

[12] *Synopsys Design Compiler Reference Manual.* Synopsys, Inc., Mountain View, CA, 1992.

[13] *Verilog-XL Release Notes.* Cadence Design Systems, Lowell, MA, 1989.

[14] *Xilinx: The XC4000 Data Book.* Xilinx, San Jose, CA, 1992.

[15] *Xilinx Synopsys Interface User Guide.* Xilinx, San Jose, CA, 1992.