Implementation of an On-chip Interconnect Using the i-SLIP Scheduling Algorithm

a semester project for

Dr. Adnan Aziz
EE382M – 7 VLSI 1

at

The University of Texas at Austin

Submitted by

John D. Pape

December 11, 2006

Table Of Contents

1. Project Overview 4
2. Specification 5
    2.1. Overview 5
        2.1.1. Packets 6
        2.1.2. Flow Control 6
        2.1.3. Technology 7
    2.2. Control Plane 7
        2.2.1. Algorithm 9
        2.2.2. Constraints 9
        2.2.3. Changes to i-SLIP 10
    2.3. Data Plane 10
        2.3.1. Receiving and buffering incoming packets 10
        2.3.2. Provide a data path from input buffers to output ports 10
3. Design 11
    3.1. Overview 11
    3.2. Input Blocks 13
        3.2.1. Incoming Packet 15
        3.2.2. Outgoing Packet 15
    3.3. Scheduler 16
        3.3.1. Algorithm 17
        3.3.2. Arbiters 18
        3.3.3. Programmable Priority Encoder 18
    3.4. Data Path Interconnect 19
    3.5. Output Blocks 20
    3.6. Optimization 20
        3.6.1. Microarchitectural Optimization 20
        3.6.2. Synthesis Optimization 20
    3.7. Design Process 20
        3.7.1. Tools 20
        3.7.2. Revision Control 21
        3.7.3. Issue Tracking 21
        3.7.4. Code Reviews 21
        3.7.5. Coding Style 21
4. Users Guide 22
    4.1. Interconnect Interface 22
        4.1.1. Input Devices 22
        4.1.2. Output Devices 23
        4.1.3. Timing 23
        4.1.4. Floorplanning 23
5. Testing 24
    5.1. Unit Verification 24
        5.1.1. Scheduler 24

5.1.2. Input Blocks ... 24
5.1.3. Output Blocks ... 25
5.2. System Verification ... 25
5.3. Results ... 25
6. Optimization ... 27
6.1. Timing ... 27
6.1.1. Programmable Priority Encoder ... 27
6.1.2. Two-phased packets ... 28
6.2. Area ... 28
6.2.1. Virtual Output Queues ... 28
6.2.2. Multi-cycle crossbar data transfer ... 29
6.2.3. Two-phased packets ... 29
6.3. Results ... 29
6.4. Future work ... 30
7. Conclusions ... 31
8. References ... 32
9. Appendix A – Design Source Code ... 33
10. Appendix B -- Design Layout ... 71
11. Appendix C – Testbench Source Code ... 72
12. Appendix D – Synthesis Reports ... 89
13. Appendix E – Simulation Results ... 98
14. Appendix F – Selected Design Schematics ... 103

1. PROJECT OVERVIEW

As fabrication technology continues to improve, smaller feature sizes allow increasingly more integration of system components onto a single die. Communication between these components can become the limiting factor for performance unless careful attention is given to designing high-performance interconnects. This project implements an on-chip SoC interconnect embodying the i-SLIP scheduling algorithm [1][2] for efficient communication between 8 SoC devices. The interconnect design is implemented in synthesizable Verilog RTL and synthesized using Synopsys Design Vision. It provides fast communication and full N-to-N routing capabilities.

The design was implemented in Verilog RTL and synthesized using Synopsys DesignVision and a 0.20 micron synthesis library to achieve a maximum frequency of 550MHz and area of 4774748 um^2. Several opportunities for further optimization remain, but are reserved for future work.

The remainder of this document is organized as follows: Section 2 provides the design specification. Section 3 describes the design and design methods. Section 4 provides a Users/Integration guide. Section 5 describes the testing strategy and results. Section 6 describes several of the optimization techniques used. Following the main document, the appendices provide the source code and layout and synthesis results.

2. SPECIFICATION

The following is the specification for the project. The specification begins with an overview of the interconnect functionality and continues with more detailed specification of the design's two main components: Control Plane and Data Plane.

2.1. Overview

The goal of this design is to provide a fast, efficient SoC interconnect between 8 on-chip devices. The eight devices are connected to one another through a single instance of the routing switch to be designed. Each device has three output ports, and three input ports which are described in Table 1.

Table 1 -- Device/Interconnect Pins

Name	Direction	Size	Description
PacketOut[35:0]	output	36-bits	Contains a communication packet being sent. Each packet is comprised of two phases for a total of 72 bits.
PacketOutValid	output	1-bit	Indicates that the PacketOut signals are valid
CreditOut	output	1-bit	Indicates that a credit is being sent
PacketIn[35:0]	input	36-bits	Contains a communication packet being received. Each packet is comprised of two phases for a total of 72 bits.
PacketInValid	input	1-bit	Indicates that the PacketIn signals are valid
CreditIn	input	1-bit	Indicates that a credit is being received

Figure 1 illustrates the 8-device SoC with the 8x8 routing switch to be designed in this project.

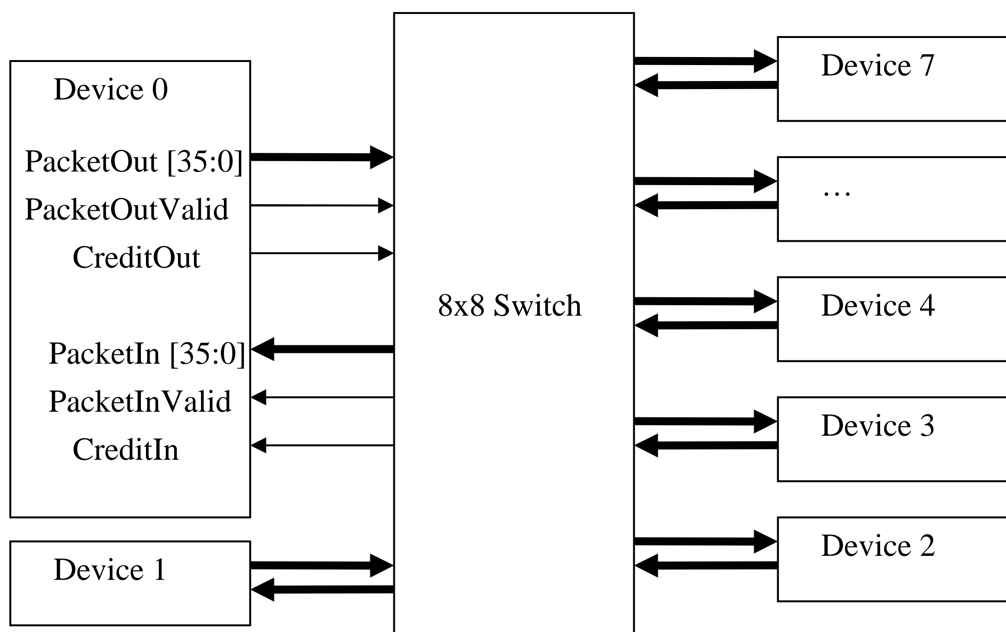


Figure 1 -- 8 device system with 8x8 interconnect

2.1.1. Packets

The devices communicate using a simple packet-based protocol. The packets are of fixed size, and include a 6-bit header and 66-bits of packet data for a total of 72 bits. The header is comprised of a 3-bit source identifier and 3-bit destination identifier (Table 2).

Table 2 -- Communication Packet

Bits	71 69	68 66	65 0
Field	Src	Dest	Packet Data

The *Packet Data* field is multipurpose and may contain commands, addresses, data, crc, or any other payload. The interconnect pays no attention to the contents of the *Packet Data* field, and simply passes it through as a payload. The Src field specifies the originating device, and the Dest field specifies the destination of the packet. Since the packets travel a relatively short distance on a well-characterizeable chip, it is assumed that the interconnect will be robust enough to not require additional parity, ecc, or crc.

2.1.2. Flow Control

Flow control is handled through a simple credit-debit system. Each device contains a counter for the number of packets it may have outstanding without receiving a new credit. At reset, each counter is initialized to the amount of buffer space in the device (or interconnect port) its output is directly connected to. A device decrements its counter when it sends a packet, and increments its counter when it receives a CreditIn. If the counter reaches 0, the device may not send any new packets until the counter becomes greater than 0 due to receiving a CreditIn.

The routing switch designed for this project will have buffer resources for 32 packets per input. While each switch input has dedicated buffer space for up to 32 packets, the buffers may be used for any combination of destinations. As an example, the buffers could store 32 packets from a source to a single destination, or 4 packets from a source to each of the 8 destinations (including the loopback destination).

2.1.3. Technology

The switch will be designed using the TSMC 0.20 micron process library available in the LRC, and used for VLSI-1 Labs 1-3.

2.2. Control Plane

The interconnect design consists of two main components: Data Plane and Control Plane. The Data Plane will handle the actual transmission of packets from a Source input to a Destination output, while the Control Plane will control the scheduling of the Data Plane to ensure the efficient and fair flow of data through the switch. This section focuses on the Control Plane.

The Control Plane acts as the central switch arbiter. It analyzes the buffered packets and their destinations at each input port and configures the Data Plane to connect inputs to outputs to allow data transfer across the switch. The Control Plane uses a scheduling algorithm to configure a large number of simultaneous connections, but also avoids conflicts of multiple inputs connecting to a single output or a single input connecting to multiple outputs. This switch will use a modified i-SLIP [1] scheduler.

The base i-SLIP specification comes from Chapter 3, Section 5 of [1]. The i-SLIP algorithm is derived from the SLIP algorithm[1], which is an improvement upon the Round-Robin Matching algorithm. Each of these algorithms assumes an N-input by N-output cell switch with input queuing. To alleviate head-of-line blocking at the input queues, each input maintains a separate queue for each possible output destination. The goals for each of these scheduling algorithms is to match input queues containing waiting Cells with output queues to achieve the maximum throughput while maintaining stability and eliminating starvation.

The SLIP algorithm matches inputs to outputs in a single iteration; however, after this iteration, several possible input and output ports may remain unutilized. The i-SLIP algorithm uses multiple iterations to find paths to utilize as many input and output ports as possible (pseudo-maxsize matching) until it converges to finding no more possible matches. The single iteration SLIP algorithm is a specialization of i-SLIP and may be characterized as i-SLIP with only a single iteration, or 1-SLIP.

Figure 3.12 [1] shows an example baseline implementation of the i-SLIP algorithm. This implementation requires an NxN bit memory to contain the states of the input queues (N inputs x N output queues per input), N Grant Arbiters, N Accept Arbiters, and decision register to send control information to the datapath. The Grant and Accept Arbiters each consist of programmable priority encoders and a state pointer to record which input

should have the highest priority on the next arbitration cycle. The N-bit feedback signal from the Decision registers to the Grant arbiters is an enable vector, which enables arbitration only for unmatched ports on each successive iterations. An example priority encoder block diagram is shown below as Figure 2.22 [1].

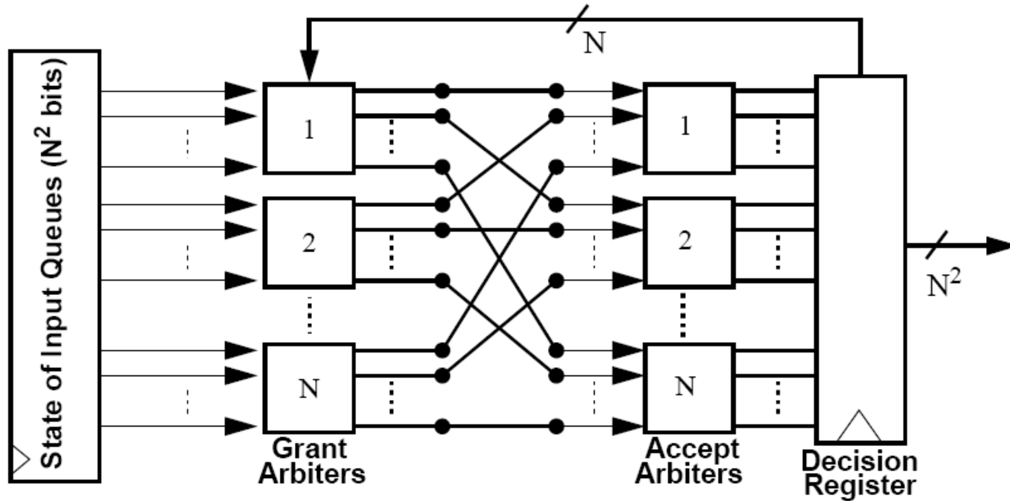


FIGURE 3.12 Interconnection of 2N arbiters to implement *i*-SLIP for an NxN switch.

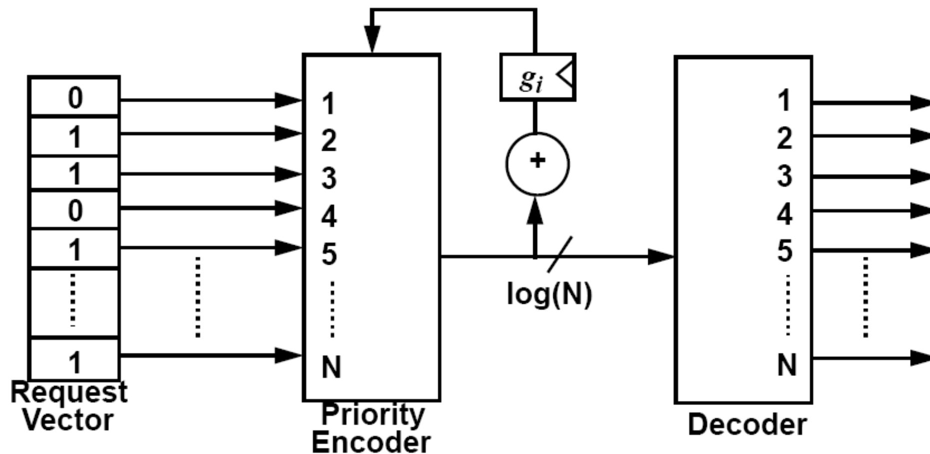


FIGURE 2.22 Round-robin *grant* arbiter for SLIP and RRM algorithms. The priority encoder has a programmed highest-priority, g_i. The *accept* arbiter at the input is identical.

Since the majority of the area is consumed by the programmable Priority encoders, the structure in Figure 3.13 [1] modifies the Figure 3.12 to reuse the priority encoding logic for both the Grant and Accept arbiters. This saves area, but introduces additional sequential logic overhead as the Grant result must be stored in the decision registers on every other cycle.

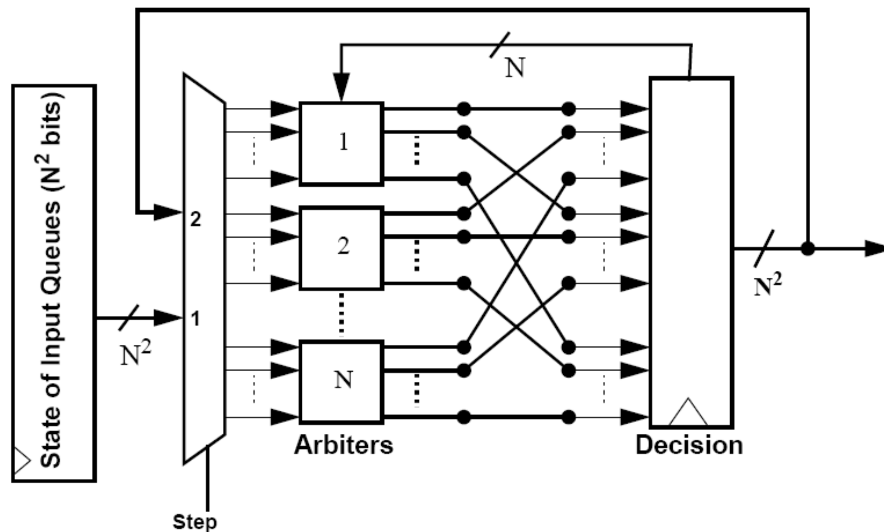


FIGURE 3.13 Interconnection of N arbiters to implement *i*-SLIP for an NxN switch. Each arbiter is used for both input and output arbitration. In this case, each arbiter contains *two* registers to hold pointers g_i and a_i.

2.2.1. Algorithm

The algorithm for i-SLIP, taken from [1], follows:

Step 1: *Request.* Each unmatched input sends a request to every output which for which it has a queued cell.

Step 2: *Grant.* In an unmatched output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the granted input iff the grant is accepted in Step 3 of the first iteration.

Step 3: *Accept.* If an unmatched input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the accepted output only if this input was matched in the first iteration.

2.2.2. Constraints

- My implementation of the i-SLIP protocol will use a fixed number of input and output ports. I will begin with an 8x8 design in synthesized static CMOS. Since the most complex portion of the design appears to be the programmable priority encoders, most of the design phase will be focused on optimizing their implementation for speed and area.

- The i-SLIP switch will be programmable to support a variable number of iterations from 1 thru N. [1] shows that the maximum number of iterations required to converge is N.

- Each input port will contain one virtual queue per output port. These virtual queues will share a buffer pool of up to 32-packets per input.

2.2.3. Changes to i-SLIP

Due to the flow control mechanisms described above, i-SLIP must be modified so it doesn't connect an input port to an output that does not have enough credit remaining to accept a data transfer. To handle this case, outputs with 0 credit will not participate in the arbitration iteration. If credit arrives between iterations of the same scheduling cycle, the port may become active again and participate in the remaining arbitration iterations.

2.3. Data Plane

The Data Plane handles the actual packet transfers in accordance with the schedule determined by the Control Plane. The Data Plane has two primary tasks:

1. Receive and buffer incoming packets
2. Provide a data path from input buffers to output ports

2.3.1. Receiving and buffering incoming packets

The Data Plane must buffer packets arriving on the input ports. It is responsible to finding available buffer space for the incoming packets and transmitting Credits when new buffer space becomes available. The Data Plane is also responsible to reporting occupancy of each Src/Dest virtual input queue to the Control plane for scheduling.

2.3.2. Provide a data path from input buffers to output ports

Once the Control Plane has reached a scheduling decision and programmed the interconnect in the Data Plane, the Data Plane must complete the transfer from virtual input queue to output. This design is most likely to use a crosspoint (crossbar) switch similar to those described in [3][4]. These crosspoint switch designs use switches to form electrical or logical connections from input to output. Once the connection is made, data is streamed across the connection.

Since large crosspoint switch designs tend to have large capacitive loads, several of these designs employ low-voltage-swing signaling to reduce the effect of the high capacitance. In addition, several crosspoint switches may be layered to form a parallel data path [1][2]. This design will seek a balance of number of crosspoint switch layers and Control Plane scheduling time. Since the Control Plane requires several iterations to determine a schedule, the Data Path can afford to take several cycles to actually transfer the data.

3. DESIGN

This section describes the microarchitecture of the 8x8 switch design. The section begins with an overview of the design, and then describes the three key components in more detail.

3.1. Overview

This design is an 8x8 crossbar for use as an on-chip SOC interconnect. The interconnect serves as a communication portal between 8 on-chip devices. Figure 2 shows an example interconnect configuration. It also shows that each device has an output and input port for a data packet and credits used for flow control. Section 2 (Specification) provides a more detailed specification for the interconnect protocol.

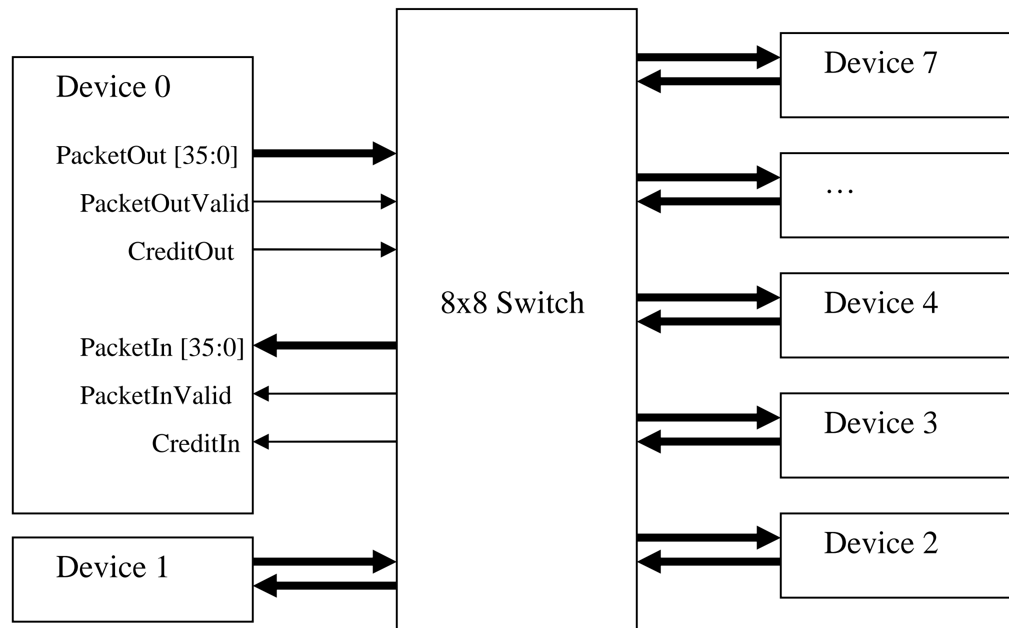


Figure 2 -- Interconnect Overview

This project is concerned with the design of the 8x8 switch. The switch is comprised of data buffering and connection logic, and an iterative scheduling algorithm. Figure 3 shows the block-diagram layout of the switch design. As shown in the figure, the design is comprised of four main component types: 8 input_blocks, a mux-based crossbar interconnect, 8 output_blocks, and a scheduler. While two sets of Devices are shown in the figure, the bottom row of Devices actually corresponds to the same Devices on the left-side column. They were duplicated at the bottom of the figure for convenience.

The 8 input_blocks are responsible for receiving, queuing, and transmitting packets from each of the design input ports. Incoming packets are queued in the data block, which indicates its pending packets to the scheduler. As packets are scheduled and transmitted

through the interconnect, the input_block returns credits back to the device connected to it, indicating that more buffer space is available for another packet.

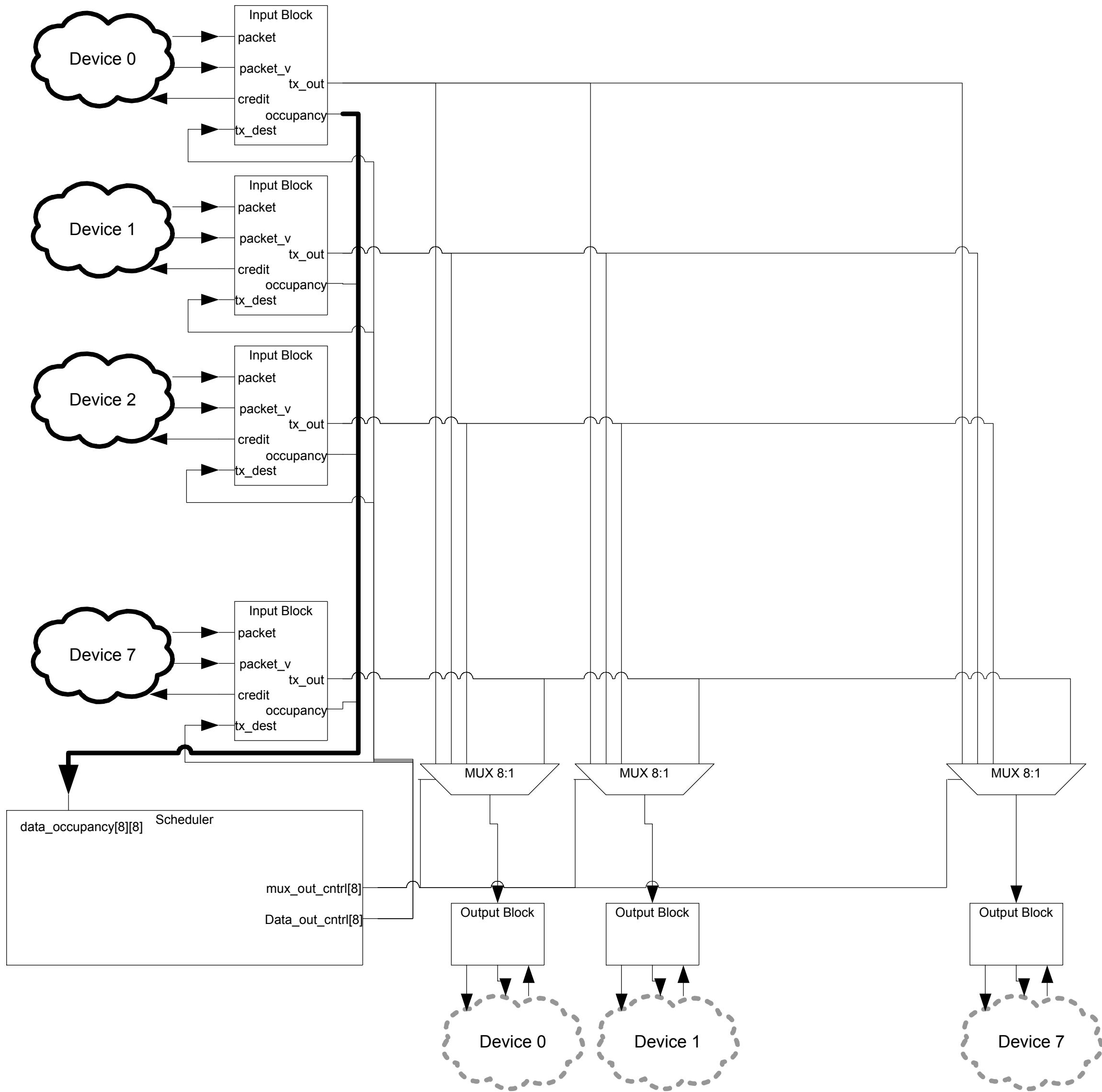


Figure 3 -- Interconnect Implementation Block Diagram

The Scheduler is responsible for configuring the input_blocks and the crossbar muxes to maximize utilization of the crossbar for each transmit cycle. Each data block indicates which destinations it has pending packets for, and the scheduler arbitrates to match inputs to destinations using the i-SLIP scheduling algorithm.

The crossbar interconnect uses a simple mux-based crossbar to enable a completely synthesized design. Once the scheduler configures the mux selects and data block destinations, the input_blocks simply pump the data through the interconnect to the output_blocks over multiple cycles. Once the output block receives the full packet, it transmits the packet to the connected output device.

3.2. Input Blocks

The input_blocks have three responsibilities: Receive incoming packets, Store the packets while waiting for scheduling, and transmitting the next packet to the selected destination once scheduling is complete. Figure 4 shows a detailed block diagram of the input_block design.

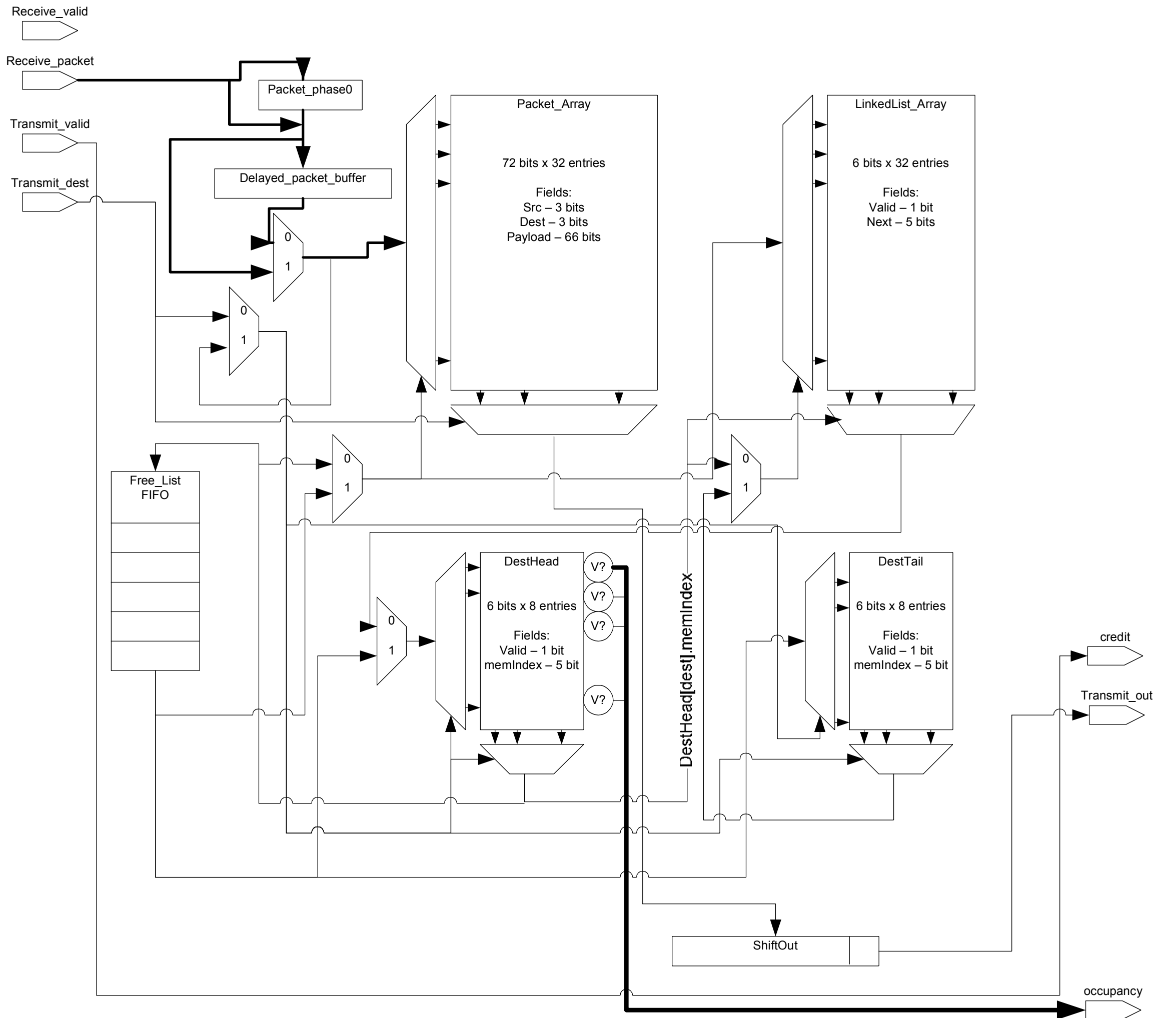


Figure 4 -- Input-Block Block Diagram

The input_block design is comprised of four memory arrays, a FIFO, and a shift register. The primary packet storage element is the Packet_Array, with the three other arrays (LinkedList_Array, DestHead, and DestTail) providing linked-list functionality.

The i-SLIP algorithm depends on each input port having a separate queue of pending packets for each output port in order to eliminate the problems caused by head-of-line blocking. Implementing separate physical queues for each output port can be very area inefficient because destination ports in high demand may have full output queues while

ports in low demand will be underutilized. This design solves this problem by using Virtual Output Queues (VOQ) [5].

The VOQs are implemented using a shared pool of 32-entries in the Packet_Array. The DestHead and DestTail arrays keep track of the Packet_Array location of the oldest and newest pending packets respectively for each output port. The LinkedList_Array maintains a Linked-List style chain of packets beginning with the DestHead of each port and ending with the DestTail of each port. Since elements may be deleted from Packet_Array in any order, The Free_List FIFO keeps a circular queue of all of the free indexes in Packet_Array. The next value in the Free_List is used to allocate new packet entries. When packet entries are removed from the Packet_Array, the index being removed is pushed back onto the Free_List.

Packets are sent to the Input Block in two phases. The first phase contains the 36 most significant bits of the packet, including the source and destination header information. The second phase contains the 36 least significant bits. The input_block registers the first phase in the packet_phase0 buffer until the second phase arrives. Once the second phase arrives, the full packet is either stored in the Packet Array if there is not a scheduled send request in the same cycle, or temporarily stored in the delayed_packet_buffer otherwise.

In order to limit the number of read and write ports on each memory array, the design shown in Figure 4 does not support simultaneous linked-list updates due to incoming packets and scheduled outgoing packets. To accommodate this, scheduled send requests take precedence over incoming packets, and incoming packets must wait on cycle in the delayed_packet_buffer before being queued in the Packet Array. Since the scheduler cannot send requests on consecutive cycles, and full packets can only arrive every other cycle, the packets are only delayed in the delayed_packet_buffer for a maximum of one cycle..

The scheduler requires occupancy information from each of the input_blocks in order to perform its scheduling task. For this input_block design, the occupancy is determined by looking at the valid bit of each of the DestHead entries. A valid bit in a DestHead entry indicates that this input port has at least one packet pending for the output tracked by the DestHead entry. The input_block module aggregates these bits and outputs them on the occupancy output port.

When the scheduler sends a transmit request to the input_block. The input_block loads the oldest pending packet for the chosen destination into the ShiftOut register to be shifted across the interconnect. The ShiftOut register shifts the data out 9-bits per cycle, so the entire 72-bit packet takes 8 cycles to shift. This is the same number of cycles as required by the scheduler to determine the next schedule.

The following two sections use pseudo-code to describe the operation of the input_block unit.

3.2.1. Incoming Packet

For incoming packets, the receive_valid signal will be high, and receive_packet will have half of the 2-phase packet. Once both phases of the packet are received, the input_block must perform the following steps:

1. Find an empty entry in the Packet_Array
2. If there were no packets pending for this destination, initialize the DestHead array with this entry
3. If there were packets pending for this destination, update the LinkedList and DestTail arrays to insert the new packet in the linked list for this destination

Pseudocode:

```
// Get the next free entry in the packet array
writeIndex <= pop(FreeListFIFO);

// Load the packet into the packet array
PacketArray[writeIndex] <= receive_packet;

// Update the linked list for this destination
If(!DestHead[receive_packet.dest].valid)
        DestHead[receive_packet.dest].memIndex <= writeIndex;
        DestHead[receive_packet.dest].valid <= 1;

If(DestTail[receive_packet.dest].valid)
        LinkedListArray[DestTail[receive_packet.dest].memIndex].next <= writeIndex;
        LinkedLIstArray[DestTail[receive_packet.dest].memIndex].valid <= 1;

DestTail[receive_packet.dest].valid <= 1;
DestTail[receive_packet.dest].memIndex <= writeIndex;
```

3.2.2. Outgoing Packet

For scheduler transmit requests, the transmit_valid signal will be high, and the transmit_dest will indicate which destination this input is scheduled to transmit to. The input_block module must perform the following steps:

1. Load the output shift register with the oldest packet (DestHead) for the selected destination
2. Update the LinkedList and DestHead to remove the packet from the linked list for the selected destination
3. Deallocate the PacketArray entry that was just moved to the output shift register, and return its index to the FreeList FIFO
4. Return a credit to the device attached to this input port, indicating there is room for another packet in the PacketArray

Pseudocode:

```
// Load the shift register
ShiftOut <= PacketArray[DestHead[transmit_dest].memIndex];

// Update the linked list
DestHead[dest] <= LinkedListArray[DestHead[dest].memIndex];
LinkedListArray[DestHead[dest].memIndex].valid <= 0;
```

```
// If this is the last packet for the selected dest, invalidate the tail
If(DestHead[dest].memIndex == DestTail[dest].index)
        DestTail[dest].valid <= 0;

// Put the PacketArray entry number back on the free list
FreeListFIFO.push(DestHead[dest].memIndex);

// Return a credit to the device sending packets to this port
Credit <= 1;
```

3.3. Scheduler

The Scheduler acts as the central switch arbiter. It analyzes the occupied Virtual Output Queues of each input_block and configures the input_blocks and interconnect muxes to connect inputs to outputs and allow serial data transfer across the switch. The scheduling algorithm attempts to achieve a large number of simultaneous connections, but also avoids conflicts of multiple inputs connecting to a single output or a single input connecting to multiple outputs. The scheduling algorithm chosen is a modified i-SLIP [1] scheduler.

The base i-SLIP specification comes from Chapter 3, Section 5 of [1]. The i-SLIP algorithm is derived from the SLIP algorithm[1], which is an improvement upon the Round-Robin Matching algorithm. Each of these algorithms assumes an N-input by N-output cell switch with input queuing. To alleviate head-of-line blocking at the input queues, each input maintains a separate queue for each possible output destination. As shown in Section 3.2, these queues are implemented as Virtual Output Queues utilizing a shared memory bank. The goals for the scheduling algorithm is to match input queues containing waiting packets with output queues to achieve the maximum throughput while maintaining stability and eliminating starvation.

The SLIP algorithm matches inputs to outputs in a single iteration; however, after this iteration, several possible input and output ports may remain unutilized. The i-SLIP algorithm uses multiple iterations to find paths to utilize as many input and output ports as possible (pseudo-maxsize matching) until it converges to finding no more possible matches. The single iteration SLIP algorithm is a specialization of i-SLIP and may be characterized as i-SLIP with only a single iteration, or 1-SLIP.

Figure 5 shows the i-SLIP scheduler implementation chosen for this design. The input to the scheduler is the occupancy vectors from each of the input_blocks with packets waiting to be scheduled. There are such 8 vectors (1 per crossbar input port), each with 8 bits (1 per destination per input). This 8x8 state is used as the request vector into the iterative arbitration logic.

The scheduler also contains 8 Grant Arbiters and 8 Accept Arbiters. The Grant and Accept Arbiters each consist of programmable priority encoders and a state pointer to record which input should have the highest priority on the next arbitration cycle. The 8-bit feedback signal from the Decision registers to the Grant arbiters is an enable vector, which enables arbitration only for unmatched ports on each successive iterations.

Finally, after a number of iterations, the scheduler arrives at a final scheduling solution which it outputs to each of the input_blocks (indicating which destination the data block has been scheduled to transmit a packet for) and each interconnect mux (indicating which input_block it is receiving data from).

As an enhancement to the original i-SLIP algorithm proposed in [1], this scheduler also includes an 8-bit busy input from each of the switch outputs. These busy signals are asserted if that output does not have enough downstream credit to send another transfer. When the busy signal is asserted, that output port is disabled from the Grant Arbitration.

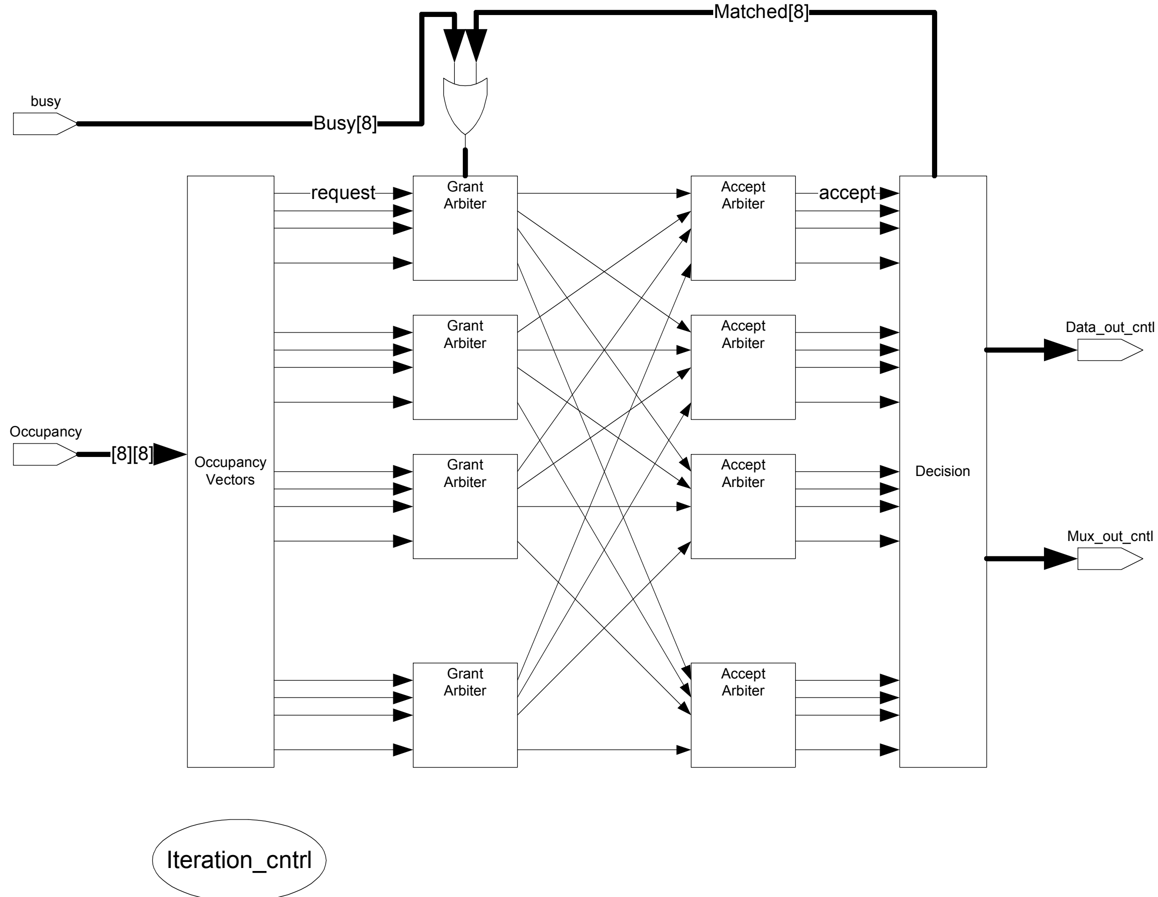


Figure 5 -- Scheduler Block Diagram

3.3.1. Algorithm

The algorithm for i-SLIP, taken from [1], follows:

Step 1: *Request.* Each unmatched input sends a request to every output which for which it has a queued cell.

Step 2: *Grant.* In an unmatched output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i to the highest priority element of the round-robin schedule is incremented (modulo N) to

one location beyond the granted input iff the grant is accepted in Step 3 of the first iteration.

Step 3: *Accept.* If an unmatched input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the accepted output only if this input was matched in the first iteration.

3.3.2. Arbiters

The arbiters are an important piece of the scheduler design. The Grant Arbiters and Accept Arbiters are identically designed with the exception of the rules determining when the priority state may be updated.

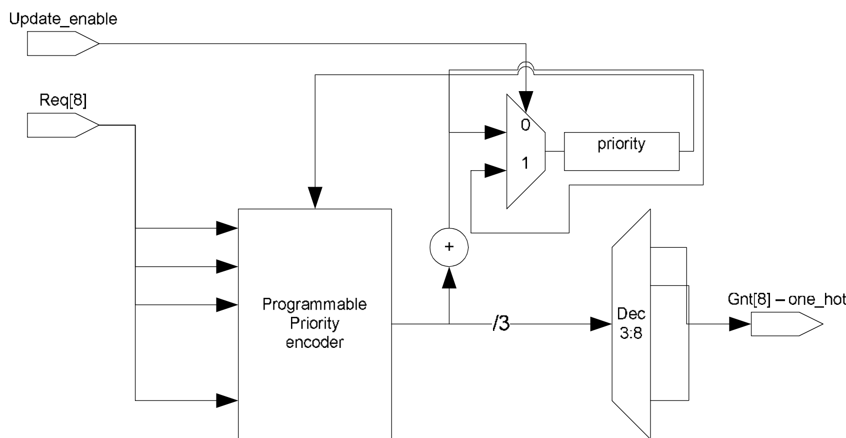


Figure 6 -- Arbiter Block Diagram

Figure 6 illustrates the arbiter design chosen for this implementation. The arbiter is based on a simple round-robin arbiter, with the exception that it also includes an update_enable signal to allow the i-SLIP algorithm to only update the priority under certain circumstances (as described in Section 3.3.1). This limited updating produces de-synchronizing behavior between the Grant and Accept arbiters, producing improved traffic fairness and decreasing undesirable bursting characteristics.

3.3.3. Programmable Priority Encoder

The most timing-critical component of the scheduler design is the Programmable Priority Encoder (PPE) utilized by each Arbiter. An 8-bit priority encoder takes 8 bits on inputs 0 thru 7, and outputs a number indicating the highest priority input that was active. A simple priority encoder always has a fixed priority (0 – highest, 7 – lowest), but a programmable priority encoder has an additional input indicating which input has the highest priority. The speed of the programmable priority encoder will determine how fast the arbitration logic can run, and is likely to be the critical path of the overall interconnect design.

The PPE chosen for this design is a hybrid design combining two simple PEs. The input to one of the simple PEs is masked by a thermometer encoding based on the programmed priority level [5]. If that PE provides any grant, it takes precedence over the non-masked simple PE. The non-masked simple PE does not take a priority, and determines the output when the masked PE does not find a 1-input between the programmed priority and input 7. [5] showed that this design produces minimized delay, and has a smaller area than more classic fast PPE designs.

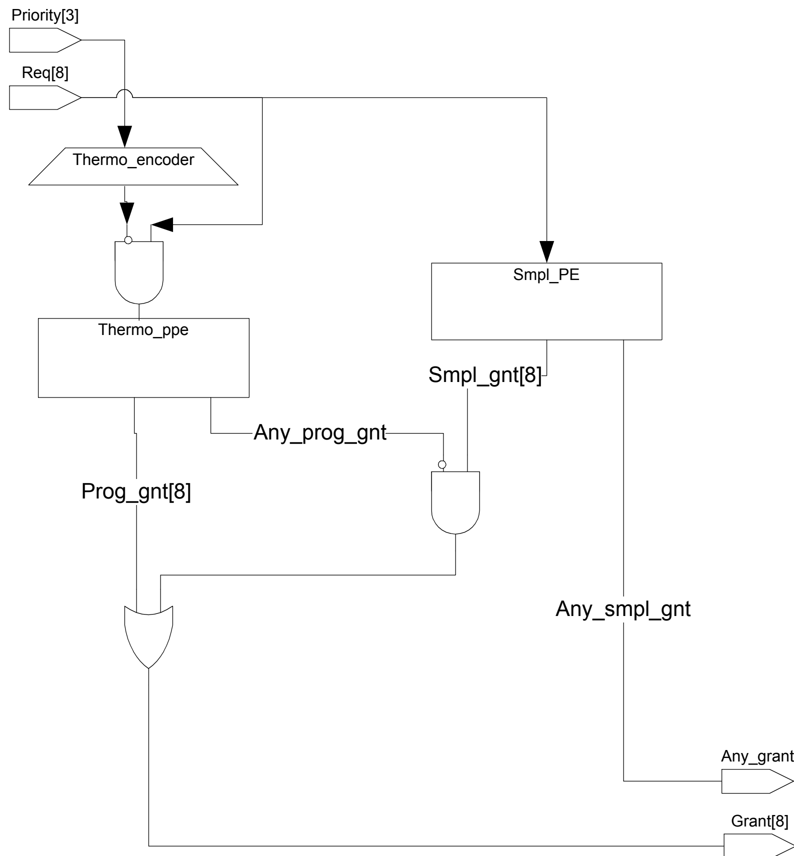


Figure 7 -- Programmable Priority Encoder

[7] also describes parallel prefix equations for a simple priority encoder. However, the synthesis tool generated a reasonable tree-style priority encoder given a simple casez-style Verilog encoding.

3.4. Data Path Interconnect

The interconnect from input port to output port will use a mux-based crossbar (Figure 3). A crosspoint switch may be more efficient, but is impossible to design using simple synthesis techniques. Since this design will be fully synthesized, the mux-based crossbar is a simple and straightforward design. Each path through the interconnect is 9-bits wide to accommodate a 72-bit packet transfer in 8 cycles.

3.5. Output Blocks

The output_blocks receive packet data 9-bits at a time from the input_blocks through the interconnect and reconstruct the full 72-bit packets. Once the packet is reconstructed, the output_block transfers the packet to the end device in two packet phases. If the output_block runs out of credits, it signals that it is busy to the scheduler so the scheduler will not schedule packets to it until it receives more credit from the connected device.

3.6. Optimization

Once the initial implementation is complete, several techniques will be used to optimize the design. Since this on-chip interconnect requires multiple scheduling iterations to find a scheduling solution, the primary design concern is minimizing data and control delay. Since the design will be fully synthesized, the primary optimization techniques will be microarchitectural changes and synthesis tool constraints. Circuit-level optimization techniques will not be explored.

3.6.1. Microarchitectural Optimization

There will be three primary opportunities for microarchitectural optimization. The first opportunity is in the programmable priority encoders, which are expected to be on the critical path determining the scheduler arbitration delay. I have chosen one of the two optimized designs proposed in [5], but it may not necessarily be the best choice for this design. [5] explores three other PPE designs that I may try in order to minimize the PPE delay.

The second microarchitectural opportunity is to match the interconnect data-shift delay to the delay of the iterative scheduler. The design described so far will requires 8 9-bit cycles to transfer a 72-bit packet from the input_block to the output_block. If scheduling takes less than 8 cycles, then multiple bits may be shifted through using a wider interconnect in order to match the data transfer delay to the scheduling delay.

The third opportunity may be to pipeline the scheduler to perform Grant and Accept arbitration simultaneously. [5] describes one technique to accomplish this, which effectively halves the scheduling delay.

3.6.2. Synthesis Optimization

The synthesis optimization will be achieved by constraining the synthesis tool timing requirements, and letting the synthesis tool attempt to minimize critical path length to achieve the timing goals.

3.7. Design Process

3.7.1. Tools

Since this design will be fully synthesizable, it can use many of the same tools as used in the course lab assignments. This design will use the following tools:

Table 3 -- Design Tools

Tool	Purpose
Vim	Verilog Editing
Synopsys VCS	Verilog Functional Simulation
Synopsys Design_vision	Design Synthesis, Area, and Timing Optimization
Synopsys Primetime	Timing Analysis
Silicon Ensemble	Auto Place and Route
Novas Debussy	Waveform Viewer
CVS	Revision Control
Perl	Scripting
Microsoft Visio	Block Diagrams
Microsoft Word	Documentation and Reports

3.7.2. Revision Control

Revision control is a crucial component for any complex code-based design. This design will utilize the Concurrent Versioning System (CVS) for revision control. A repository has been setup in my user directory to checkin files. Although this project only has a single developer, a revision control system such as CVS is still critical in order to archive incremental design changes along with revisioning information.

3.7.3. Issue Tracking

If this were a large design project with multiple developers and verification engineers, a free off-the-shelf issue tracking solution like BugZilla would be perfect for tracking design tasks and issues. However, since there is only one developer on this project, a private, web-based Wiki will be used for issue tracking.

The Wiki tracking will include three types of issues, and three states. The three issue types are Enhancement_Request, Bug, and Task. The three states are OPEN, FIXED, WILL_NOT_FIX.

3.7.4. Code Reviews

Periodic code reviews help can help identify design issues by bringing "fresh eyes" to the code. With only one developer, no fresh eyes are available for the code review. However, I will do simplified reviews by periodically reviewing and updating the source code comments, running VCS simulations and checking for Linting errors, and running synthesis and checking for errors and warnings.

3.7.5. Coding Style

The verilog RTL will conform to generally-accepted naming conventions and coding style. All module and variable names will be lowercase with letters, numbers, and underscores. All instances will include an instance name. Comments will be generously used to disambiguate code.

4. USERS GUIDE

The 8x8 interconnect design implemented for this project is very straightforward to integrate into an on-chip system. The communication protocol between devices is straightforward and simple, involving only fixed sized packets with source and destination information in the packet headers. A simple debit-credit system maintains flow control into and out of the interconnect, eliminating buffer overrun.

Section 2 (Specification) describes the detailed communication protocol. Integrators and device designers should refer to the Specification protocol to communicate with the 8x8 interconnect design.

4.1. Interconnect Interface

The interconnect interface is comprised of 8 input device interfaces, 8 output device interfaces, and a clock and reset signal (Figure 2). The most common use case for the interconnect is for each device connecting to an input interface to also connect to an output interface. This would allow up to 8 system devices to connect to 7 other system devices with both an input and output port. In such a configuration, the interconnect provides a loopback port if the packet header's source and destination fields are equal. While this is expected to be the most common use case, there is no requirement for an input device to also connect to an output port. One could potentially connect 8 sources and 8 separate sinks to the interconnect. For instance, the devices on the left side of (Figure 4) could potentially be different devices then those along the bottom of the figure.

4.1.1. Input Devices

Each input device interface has three ports: *packet_in*, *packet_valid*, and *credit*. Input devices communicate with output devices through the interconnect by constructing and sending a 72-bit packet to the input interface. The 72-bit packet is sent to the input device's respective *packet_in* port in two phases. The first phase sends packet[71:36], while the second phase sends packet[35:0]. The input device must assert the *packet_valid* signal for each phase, indicating that valid packet information is being transmitted on the *packet_in* port. The phases are not required to be on consecutive clock cycles.

Flow control is maintained through a simple credit/debit system. Each interconnect input interface contains sufficient buffering for up to 32 packets from each input device. After reset, each input device may send up to 32 packets before it must wait for more buffer space to become available. The interconnect design signals when more buffer space becomes available via the input interface *credit* output. The *credit* output is asserted for one clock cycle each time a buffer entry is freed.

Packets sent from a source to a destination are guaranteed to arrive at that destination in the same order as sent. However, no ordering is guaranteed from a source to different destinations, or from different sources to the same destination.

4.1.2. Output Devices

Each output device interface has three ports similar to the input interface, but with the directions reversed: *packet_out*, *packet_valid*, and *credit*. Output devices receive packets on the *packet_out* port in two phases. The first phase is packet[71:36], and the second phase is packet[35:0]. Valid packet data is indicated by the *packet_valid* output of the interconnect. After reset, each output device must have enough buffer space to receive 32 packets. Output devices can signal when more buffer space is available via the *credit* input to the interconnect.

4.1.3. Timing

The 8x8 interconnect was synthesized in DesignVision from Verilog RTL using a 0.20 micron library. The synthesized design achieves a maximum frequency of 560 MHz in this technology. Inputs to the interconnect require a setup time of 59.1ps, and outputs from the interconnect will meet a setup time of up to 1661ps, less wire delay from output port to device.

4.1.4. Floorplanning

The interconnect design is quite large due to the large Virtual Output Queue buffer capacity at each input interface. The timing-optimized interconnect design requires 4774748 um^2 of area. This is approximately 5 mm^2, or about $1/20^{th}$ of a 10mm x 10mm die. This area may be reduced using SRAM arrays rather than the register files produced by the synthesis tool, or by using a shared packet pool for all of the input blocks (See Section 6.4 – Future work).

5. TESTING

The interconnect design underwent a large testing effort to verify functional correctness. The testing involved both unit and system-level testing using structured and random stimulus and assertions. The testing occurred in two phases: unit-level verification and system-level verification.

5.1. Unit Verification

The first phase of testing was unit-level verification of the design's three most complex components—input_block, output_block, and scheduler. Each of these blocks was tested using a unit-level testbench comprised of the block and random or directed stimulus generators. A combination of post-processing checking of the simulation output and runtime assertions checked the correctness of the units.

5.1.1. Scheduler

The scheduler unit verification was accomplished using a combination of directed input stimulus and assertions. The inputs to the scheduler are the occupancy vectors of each of the input_blocks. For unit-level testing, short testcases that exercised the occupancy vectors were used to stimulate the scheduler inputs. The scheduling decisions were partially checked using assertion blocks to verify that the decisions conformed to all-0 or 1-hot encoding. For the unit-level scheduler testbench, no checking was done to verify that a correct schedule was determined given the current occupancy vectors. That checking was deferred to the system-level simulations.

5.1.2. Input Blocks

The input_block unit verification was accomplished using random behavioral functional models (BFM) of an input device and an output_block. The input device BFM generates random input packets and drives them into the input_block according to the interconnect specification. The input device BFM inserts randomly delays between packets and follows the appropriate flow-control protocol to prevent overrunning the interconnect packet buffers.

On the output of the input_block, an output_block BFM was used to model the block that receives packets on the outbound side of the internal crossbar switch. Since the unit-level testbench only tests a single input_block, the output_block BFM does not have to mux data from several different inputs.

During simulation, the input device BFM generates random packets with random destination IDs and drives them into the input_block under test. When it does so, it prints the contents of the packet to the simulation log. On the other side of the input_block, the output_block BFM mimics the scheduler block by observing the input_block's occupancy vector and randomly picking one of the pending destinations to request. The

output_block BFM requests a packet and then receives the requested packet and prints its contents to the simulation log. Following simulation, a perl script post-processes the simulation log to ensure that the output_block BFM received the packets in the correct order.

5.1.3. Output Blocks

Only very limited unit-level testing was done on the output_block module due to its relative simplicity compared to the scheduler and input_block designs. A small testbench was used and select vectors were driven into the ouput_block design. The design was verified by looking at simulation waveforms.

5.2. System Verification

Once the three main design components were verified via unit-level testing, the 8x8 interconnect design was constructed and tested in a system-level verification environment. The primary means for this system-level verification was random input and output stimulus and post-processed result checking using the Perl script from the input_block unit-verification, with some slight modifications to handle multiple source and destination IDs.

The random input device BFMs used in the input_block unit-verification were directly reused for the system-verification. Eight of these BFM were instantiated at the interconnect inputs and parameterized with their own unique source IDs. A simple output device BFM was written to receive 2-phase packets on the output of the 8x8 interconnect and randomly return credits. Eight of these BFMs were instantiated at the output ports and parameterized with unique destination IDs.

During simulation, the input device BFMs printed the packets they sent to the simulation log, while the output device BFMs printed the packets they received. After simulation, a perl script checked the following three items for correctness:

1. Each source sent packets using its source id
2. Each destination only received packets with its destination id
3. Each destination received packets from each source in the same order that the source sent packets to that destination.

In addition, the zero-1hot assertion checkers remained connected to the scheduler, adding an additional level of checking at the system level.

5.3. Results

The split unit-level and system-level testing strategy worked well for the 8x8 interconnect verification. By starting with the unit-level testing, low-level design bugs were flushed out without having to deal with the additional complexities of the other complex design components. Bugs in the unit-level verification were easy to diagnose and fix. Unit-level testing uncovered problems in the complex arbiter wiring of the scheduler, a bit-order

problem in the priority encoders, and several bugs in the input_block linked list update logic. In addition, unit-level testing uncovered a microarchitectural limitation in the input_block when an incoming packet arrives in the same clock as a scheduler send request. The microarchitecture was changed to allow this case.

Once the unit-level bugs were worked out, the system-level verification tested the interaction between the three main design units as well as the wiring of the 8x8 interconnect. Several problems were found in the wiring of the input_blocks to the scheduler and output_blocks, and yet more linked-list update problems were found, causing several dropped packets.

The post-processing perl script worked well for discovering and diagnosing problems. A perl script was much easier to implement and modify than trying to implement the checking in behavioral verilog.

I also spent quite a bit of time debugging waves in the Novas Debussy debugging tools. Debussy supports viewing memory contents and allows easy tracing of signals throughout the design. Using Debussy saved many hours of frustration that would have been caused by the more limited Synopsys Virsim tools.

6. OPTIMIZATION

Optimization is a key component of any design cycle. First cut designs are often implemented using inefficient micro-architectural or circuit techniques. Synthesis and analysis tools can uncover inefficient design components and timing paths, which can be redesigned to improve timing and area. Often, area and timing optimization go hand in hand—removing gates from a long path can reduce the delay through the path.

Each step of the design process presents opportunity for optimization—from architecture to circuit design to layout. Since this project is designed to be fully synthesizable, the opportunities for optimization are limited to the microarchitecture and the synthesis tools compile options and effort. However, since synthesis-tool optimizations are bounded by the microarchitectural decisions, microarchitecture presents the greatest opportunity for improvement.

For this design, the most critical design aspect is timing. On-chip interconnects must run a very high speeds to keep up with the blocks for which they facilitate communication. To maximize speed, the DesignVision synthesis tool was configured to minimize timing paths between clocked elements. Options were also enabled to allow the tool to rearrange Boolean logic on a block-by-block basis to improve this timing.

In addition, several microarchitecture design techniques were used to minimize timing and reduce area. Using these techniques along with the synthesis options, the interconnect design was able to achieve a 550MHz clock frequency. Several of microarchitectural optimizations are enumerated below.

6.1. Timing

The following are some of the microarchitectural optimizations used to improve timing.

6.1.1. Programmable Priority Encoder

The slowest timing path through the scheduler passes through the programmable priority encoder logic of the grant/accept arbiters. There are many different ways to implement a priority encoder that supports a programmable priority [5]. Some require rotating the request vector and then rotating the resulting priority vector, while others resemble a ripple-carry-adder design. However, I chose a design that utilizes a request masking mechanism and two simple non-programmable priority encoders for this implementation. The design has very similar delay to a non-programmable priority encoder, with only three times the complexity.

The simple non-programmable priority encoder used in the design also has opportunity for optimization. [7] provides parallel prefix equations for finding the first 1 in a vector that can be used in one of the tree-adder configurations. I chose to implement the simple priority encoder using a simple verilog casez statement. When Boolean optimizations were enabled in the DesignVision synthesis tool, it synthesized the casez into a tree-like structure, achieving very low delay through the priority encoder.

6.1.2. Two-phased packets

Another microarchitectural change used to improve timing is the use of a 2-phase packet transfer on the interconnect input and output interfaces. The 2-phase transfer allows two timing enhancements. First, the first packet phase can be registered in the input_block for quick access in the next clock cycle. This is important because the first packet phase contains the packet header and destination information, which is used in the input_block logic to update the linked-list pointers. Since the input_block registers this information and doesn't use it for addressing until the second phase arrives, this allows for maximum delay from input device to the 8x8 interconnect since only setup time of the phase0 register must be met. The second phase does not contain any addressing information, so it also does not need to pass through any logic before entering the PacketArray.

Second, the two-phased packet design allows the input_block to temporarily store incoming packets in an elasticity buffer in the case where it is handling a scheduler packet send request. Since a packet is only ready for processing on every other cycle, this gives the elasticity buffer time to empty in the next clock after handling the scheduler's request. This allows the input_block to use single ported memories instead of dual ported memories, reducing the timing and area overhead of handling multiple ports.

6.2. Area

The following microarchitectural features are designed to optimize design area.

6.2.1. Virtual Output Queues

The iSLIP scheduling algorithm requires each input_block to report pending transactions to the scheduler in an occupancy vector. The simplest way for the input blocks to do this is by maintaining a separate FIFO queue for each destination port. However, such an implementation is area inefficient if packet requests are heavily weighted toward only a few destinations. The output FIFOs that receive the most requests will fill up quickly while the other FIFOs will remain underutilized. Not only is this an inefficient utilization of resources, but it also puts back-pressure on the input device; if the input device's next packet is destined for a destination with a full FIFO, its following packets will be stalled by head-of-line blocking. This design resolves this problem by using Virtual Output Queues.

Virtual Output Queues utilize a shared buffer pool to emulate separate output FIFOs. Incoming packets are placed in the next available buffer slot, and a linked list structure is used to maintain FIFO ordering for each destination. The shared buffer pool maximizes FIFO utilization since any packet can go in any buffer slot. The Virtual Output Queues can handle any distribution of packet destinations from 32 packets all to the same destination to 4 packets to each of the 8 destinations. However, this ability comes at an extra cost due to the extra logic and memory overhead to maintain the linked list structure. For this design, 2304 bits of shared buffer pool requires 448 bits (19.4%) to maintain the linked-list state. It also requires additional decode and mux logic to control the linked list updates and index into the additional memory arrays.

6.2.2. Multi-cycle crossbar data transfer

One of the most wiring-intensive components of the 8x8 interconnect is the crossbar switch connecting input_blocks to output_blocks. Custom-designed crossbars utilize simple transmission gate switches to connect input wires to output wires placed in a grid. However, since this design is synthesizable, muxes are used to select the input to output connections.

To reduce the area required to connect input_blocks to output_blocks, the data from input_block to output_block is sent in 8 9-bit data transfers rather than a single 72-bit data transfer. This was done to maximize the utilization of the interconnect and scheduler. The scheduler requires 8 clocks to arrive at an interconnect scheduling decision, allowing up to 8 clocks to shift data across the interconnect from input to output. Since the packets are 72-bits wide, the design can shift 9-bits at a time and still be available to be reprogrammed at the schedulers next scheduling decision. Shifting all 72-bits would decrease latency through the interconnect, but greatly increase the wiring and muxing overhead of the crossbar switch.

6.2.3. Two-phased packets

The final area optimization is the use of two-phased packet transfers into and out of the 8x8 interconnect. Using two phases reduces the number of input and output ports by nearly 50% from 1184 to 608 pins.

6.3. Results

Using these optimization techniques, the design is able to achieve a worst-case timing path of 1800ps and an area of 4774748 um^2. This timing translates to a maximum overall operating frequency of 550MHz and approximately 5 mm^2 of die space (or 1/20th of a 10mm x 10mm die).

The largest portions of the timing path and area come from the input_block module due to the large Virtual Output Queues and their linked-list overhead. The area of the three main design blocks is shown in Table 4.

Table 4 - Area by design block

Block	Area/instance (um^2)	Total Area (um^2)
input_block	561836	4494688
scheduler	75879	75879
output_block	25155	201240
Total		**4771807**

While the overall interconnect timing path is 1800ps, the scheduler itself has a maximum timing path of 502ps. With this delay, the scheduler could potentially run near 2GHz.

6.4. Future work

The design still has room for further optimization; however, these will have to be reserved for future work. The following three optimizations could further improve the design:

1. Further improve input port packet buffering utilization by sharing a single packet buffer pool between all of the input_blocks rather than each having its own shared buffer pool.

2. Synthesize the shared buffer pool arrays and linked-list management into SRAMS rather than register files. These arrays account for most of the design area, and the area would be greatly reduced it dense SRAMs were used for the data storage. The synthesis tool and library did not support SRAMs, and used less dense register files instead.

3. Increase the frequency of the scheduler to take advantage of its faster timing path. The scheduler could potentially run 4x faster than the input_block components, meaning it could arrive at a schedule every 2 input_block clocks.

4. Use a more efficient, custom-designed crossbar interconnect. Much more efficient designs are available for connecting input ports to output ports according to a schedule. Other designs are both more area and timing efficient through their use of pass gates and sense amps, but they are not synthesizable.

5. Synthesize using a newer technology (e.g. 65 nm).

7. CONCLUSIONS

This project designed an 8x8 on chip interconnect module utilizing the iSLIP scheduling algorithm. The design was implemented in Verilog RTL and synthesized using Synopsys DesignVision and a 0.20 micron synthesis library to achieve a maximum frequency of 550MHz and area of 4774748 um^2. The design was tested using unit-level and system-level testbenches utilizing random stimulus and strategically-placed assertions. Several microarchitectural optimizations were used to reduce area and critical timing paths including Virtual Output Queues, a thermometer programmable priority encoder, and multi-phase packet and data transfers.

The following Appendices provide the design and testbench source code as well as synthesis and layout results.

8. REFERENCES

[1] McKeown, N. W. 1995 *Scheduling Algorithms for Input-Queued Cell Switches*. Doctoral Thesis. UMI Order Number: UMI Order No. GAX96-02658., University of California at Berkeley.

[2] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, and M.Horowitz, "*The **Tiny Tera**: A Packet Switch Core*," IEEE Micro, vol. 17, pp.26-33, Jan.-Feb. 1997.

[3] Wijetunga, P., "High-performance crossbar design for system-on-chip," *System-on-Chip for Real-Time Applications, 2003. Proceedings. The 3rd IEEE International Workshop on* , vol., no.pp. 138- 143, 30 June-2 July 2003

[4] Shin, H.J.; Hodges, D.A., "A 250-Mbit/s CMOS crosspoint switch," *Solid-State Circuits, IEEE Journal of* , vol.24, no.2pp.478-486, Apr 1989

[5] Gupta, P.; McKeown, N., "Designing and implementing a fast crossbar scheduler," *Micro, IEEE* , vol.19, no.1pp.20-28, Jan/Feb 1999

[6] Pape, J; "Implementation of an On-chip Interconnect Using the i-SLIP Scheduling Algorithm: Intermediate Report – Specification and Timeline," Nov 2006.

[7] N. Weste and D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective. Addison-Wesley, 2004.

[8] Mhamdi, L., Kachris, C., and Vassiliadis, S. 2006. A reconfigurable hardware based embedded scheduler for buffered crossbar switches. In *Proceedings of the 2006 ACM/SIGDA 14th international Symposium on Field Programmable Gate Arrays* (Monterey, California, USA, February 22 - 24, 2006). FPGA '06. ACM Press, New York, NY, 143-149.

[9] Kun-Yung Ken Chang; Shang-Tse Chuang; McKeown, N.; Horowitz, M., "A 50 Gb/s 32×32 CMOS crossbar chip using asymmetric serial links," *VLSI Circuits, 1999. Digest of Technical Papers. 1999 Symposium on* , vol., no.pp.19-22, 1999

[10] Brinkmann, A.; Niemann, J.-C.; Hehemann, I.; Langen, D.; Porrmann, M.; Ruckert, U., "On-chip interconnects for next generation system-on-chips," *ASIC/SOC Conference, 2002. 15th Annual IEEE International* , vol., no.pp. 211- 215, 25-28 Sept. 2002

[11] Partridge, C., Carvey, P. P., Burgess, E., Castineyra, I., Clarke, T., Graham, L., Hathaway, M., Herman, P., King, A., Kohalmi, S., Ma, T., Mcallen, J., Mendez, T., Milliken, W. C., Pettyjohn, R., Rokosz, J., Seeger, J., Sollins, M., Storch, S., Tober, B., and Troxel, G. D. 1998. A 50-Gb/s IP router. *IEEE/ACM Trans. Netw.* 6, 3 (Jun. 1998), 237-248.

[12] P. Guerrier and A. Greiner, "A Generic Architecture for On-Chip PacketSwitched Interconnections", Proc. Design Automation and Test in Europe, 2000.

[13] Lines, A., "Nexus: an asynchronous crossbar interconnect for synchronous system-on-chip designs," *High Performance Interconnects, 2003. Proceedings. 11th Symposium on* , vol., no.pp. 2- 9, 20-22 Aug. 2003

[14] S.Q. Zheng, M. Yang, and F. Masetti-Placci, "Constructing schedulers for high-speed, high-capacity switches/routers," Int. J. Comput. Appl., vol.25, no.4, pp.264–271, 2003.

[15] Si-Qing Zheng, Mei Yang, Francesco Masetti: Hardware Scheduling in High-speed, High-capacity IP Routers. IASTED PDCS 2002: 631-636

9. APPENDIX A – DESIGN SOURCE CODE

The following pages include the source code for the 8x8 interconnect design. The first file (interconnect.v) is the top-level file.

```
//---------------------------------------------------------------------------
// Module: interconnect
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//  This module is the top-level module for an 8x8 crossbar interconnect.  The
//  module is comprised of 8 input blocks that receive and queue requests from
//  input devices, 8 output blocks that send packets to the output devices, and
//  a crossbar scheduler that implements the iSLIP scheduling protocol.
//---------------------------------------------------------------------------
module interconnect( clk, reset_,
    input0_packet, input0_packet_v, input0_credit, output0_packet, output0_packet_v,
output0_credit,
    input1_packet, input1_packet_v, input1_credit, output1_packet, output1_packet_v,
output1_credit,
    input2_packet, input2_packet_v, input2_credit, output2_packet, output2_packet_v,
output2_credit,
    input3_packet, input3_packet_v, input3_credit, output3_packet, output3_packet_v,
output3_credit,
    input4_packet, input4_packet_v, input4_credit, output4_packet, output4_packet_v,
output4_credit,
    input5_packet, input5_packet_v, input5_credit, output5_packet, output5_packet_v,
output5_credit,
    input6_packet, input6_packet_v, input6_credit, output6_packet, output6_packet_v,
output6_credit,
    input7_packet, input7_packet_v, input7_credit, output7_packet, output7_packet_v,
output7_credit
    );

  //--------------------------
  // Parameters
  //--------------------------
  parameter PACKET_WIDTH = 72;
  parameter SHIFT_WIDTH = 9;
  parameter NUM_PORTS = 8;

  //--------------------------
  // Ports
  //--------------------------
  input clk;
  input reset_;

  input  [PACKET_WIDTH/2-1:0] input0_packet    ;
  input                       input0_packet_v  ;
  output                      input0_credit    ;
  output [PACKET_WIDTH/2-1:0] output0_packet   ;
  output                      output0_packet_v ;
  input                       output0_credit   ;

  input  [PACKET_WIDTH/2-1:0] input1_packet    ;
  input                       input1_packet_v  ;
  output                      input1_credit    ;
  output [PACKET_WIDTH/2-1:0] output1_packet   ;
  output                      output1_packet_v ;
  input                       output1_credit   ;

  input  [PACKET_WIDTH/2-1:0] input2_packet    ;
  input                       input2_packet_v  ;
  output                      input2_credit    ;
  output [PACKET_WIDTH/2-1:0] output2_packet   ;
  output                      output2_packet_v ;
  input                       output2_credit   ;

  input  [PACKET_WIDTH/2-1:0] input3_packet    ;
  input                       input3_packet_v  ;
  output                      input3_credit    ;
  output [PACKET_WIDTH/2-1:0] output3_packet   ;
  output                      output3_packet_v ;
  input                       output3_credit   ;
```

```
input  [PACKET_WIDTH/2-1:0] input4_packet    ;
input                       input4_packet_v  ;
output                      input4_credit    ;
output [PACKET_WIDTH/2-1:0] output4_packet   ;
output                      output4_packet_v ;
input                       output4_credit   ;

input  [PACKET_WIDTH/2-1:0] input5_packet    ;
input                       input5_packet_v  ;
output                      input5_credit    ;
output [PACKET_WIDTH/2-1:0] output5_packet   ;
output                      output5_packet_v ;
input                       output5_credit   ;

input  [PACKET_WIDTH/2-1:0] input6_packet    ;
input                       input6_packet_v  ;
output                      input6_credit    ;
output [PACKET_WIDTH/2-1:0] output6_packet   ;
output                      output6_packet_v ;
input                       output6_credit   ;

input  [PACKET_WIDTH/2-1:0] input7_packet    ;
input                       input7_packet_v  ;
output                      input7_credit    ;
output [PACKET_WIDTH/2-1:0] output7_packet   ;
output                      output7_packet_v ;
input                       output7_credit   ;

//--------------------------
// Internal wires between inputs, scheduler, and outputs
//--------------------------
wire [ NUM_PORTS-1   : 0 ] input_decision_v; // input scheduling decision val
wire [ NUM_PORTS-1   : 0 ] input0_decision;  // input scheduling decision
wire [ SHIFT_WIDTH-1 : 0 ] input0_data_out;  // input->output data
wire [ NUM_PORTS-1   : 0 ] input0_request;   // input->output pending vector
wire [ NUM_PORTS-1   : 0 ] input1_decision;
wire [ SHIFT_WIDTH-1 : 0 ] input1_data_out;
wire [ NUM_PORTS-1   : 0 ] input1_request;
wire [ NUM_PORTS-1   : 0 ] input2_decision;
wire [ SHIFT_WIDTH-1 : 0 ] input2_data_out;
wire [ NUM_PORTS-1   : 0 ] input2_request;
wire [ NUM_PORTS-1   : 0 ] input3_decision;
wire [ SHIFT_WIDTH-1 : 0 ] input3_data_out;
wire [ NUM_PORTS-1   : 0 ] input3_request;
wire [ NUM_PORTS-1   : 0 ] input4_decision;
wire [ SHIFT_WIDTH-1 : 0 ] input4_data_out;
wire [ NUM_PORTS-1   : 0 ] input4_request;
wire [ NUM_PORTS-1   : 0 ] input5_decision;
wire [ SHIFT_WIDTH-1 : 0 ] input5_data_out;
wire [ NUM_PORTS-1   : 0 ] input5_request;
wire [ NUM_PORTS-1   : 0 ] input6_decision;
wire [ SHIFT_WIDTH-1 : 0 ] input6_data_out;
wire [ NUM_PORTS-1   : 0 ] input6_request;
wire [ NUM_PORTS-1   : 0 ] input7_decision;
wire [ SHIFT_WIDTH-1 : 0 ] input7_data_out;
wire [ NUM_PORTS-1   : 0 ] input7_request;

wire [NUM_PORTS-1:0] output_available;     // output has credit available
wire [NUM_PORTS-1:0] output0_decision;     // output scheduling decision
wire [NUM_PORTS-1:0] output1_decision;
wire [NUM_PORTS-1:0] output2_decision;
wire [NUM_PORTS-1:0] output3_decision;
wire [NUM_PORTS-1:0] output4_decision;
wire [NUM_PORTS-1:0] output5_decision;
wire [NUM_PORTS-1:0] output6_decision;
wire [NUM_PORTS-1:0] output7_decision;
wire [NUM_PORTS-1:0] output_decision_v;    // output schedule valid

// Scheduler control and status
reg sched_start;
```

```
wire sched_done;

//--------------------------
// Input blocks
//--------------------------
input_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) input0_block(
    .clk         ( clk                 ),
    .reset_      ( reset_              ),
    .packet_in   ( input0_packet       ),
    .packet_v    ( input0_packet_v     ),
    .credit      ( input0_credit       ),
    .tx_dest     ( input0_decision     ),
    .tx_dest_v   ( input_decision_v[0] ),
    .tx_out      ( input0_data_out     ),
    .request_out ( input0_request      )
    );

input_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) input1_block(
    .clk         ( clk                 ),
    .reset_      ( reset_              ),
    .packet_in   ( input1_packet       ),
    .packet_v    ( input1_packet_v     ),
    .credit      ( input1_credit       ),
    .tx_dest     ( input1_decision     ),
    .tx_dest_v   ( input_decision_v[1] ),
    .tx_out      ( input1_data_out     ),
    .request_out ( input1_request      )
    );

input_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) input2_block(
    .clk         ( clk                 ),
    .reset_      ( reset_              ),
    .packet_in   ( input2_packet       ),
    .packet_v    ( input2_packet_v     ),
    .credit      ( input2_credit       ),
    .tx_dest     ( input2_decision     ),
    .tx_dest_v   ( input_decision_v[2] ),
    .tx_out      ( input2_data_out     ),
    .request_out ( input2_request      )
    );

input_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) input3_block(
    .clk         ( clk                 ),
    .reset_      ( reset_              ),
    .packet_in   ( input3_packet       ),
    .packet_v    ( input3_packet_v     ),
    .credit      ( input3_credit       ),
    .tx_dest     ( input3_decision     ),
    .tx_dest_v   ( input_decision_v[3] ),
    .tx_out      ( input3_data_out     ),
    .request_out ( input3_request      )
    );

input_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) input4_block(
    .clk         ( clk                 ),
    .reset_      ( reset_              ),
    .packet_in   ( input4_packet       ),
    .packet_v    ( input4_packet_v     ),
    .credit      ( input4_credit       ),
    .tx_dest     ( input4_decision     ),
    .tx_dest_v   ( input_decision_v[4] ),
    .tx_out      ( input4_data_out     ),
    .request_out ( input4_request      )
    );

input_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) input5_block(
    .clk         ( clk                 ),
    .reset_      ( reset_              ),
    .packet_in   ( input5_packet       ),
    .packet_v    ( input5_packet_v     ),
    .credit      ( input5_credit       ),
```

```
    .tx_dest     ( input5_decision     ),
    .tx_dest_v   ( input_decision_v[5] ),
    .tx_out      ( input5_data_out     ),
    .request_out ( input5_request      )
    );

input_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) input6_block(
    .clk         ( clk                 ),
    .reset_      ( reset_              ),
    .packet_in   ( input6_packet       ),
    .packet_v    ( input6_packet_v     ),
    .credit      ( input6_credit       ),
    .tx_dest     ( input6_decision     ),
    .tx_dest_v   ( input_decision_v[6] ),
    .tx_out      ( input6_data_out     ),
    .request_out ( input6_request      )
    );

input_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) input7_block(
    .clk         ( clk                 ),
    .reset_      ( reset_              ),
    .packet_in   ( input7_packet       ),
    .packet_v    ( input7_packet_v     ),
    .credit      ( input7_credit       ),
    .tx_dest     ( input7_decision     ),
    .tx_dest_v   ( input_decision_v[7] ),
    .tx_out      ( input7_data_out     ),
    .request_out ( input7_request      )
    );

//-------------------------
// Output blocks
//-------------------------
output_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) output0_block(
    .clk          ( clk                  ),
    .reset_       ( reset_               ),
    .input_sel    ( output0_decision     ),
    .sel_v        ( output_decision_v[0] ),
    .data0_in     ( input0_data_out      ),
    .data1_in     ( input1_data_out      ),
    .data2_in     ( input2_data_out      ),
    .data3_in     ( input3_data_out      ),
    .data4_in     ( input4_data_out      ),
    .data5_in     ( input5_data_out      ),
    .data6_in     ( input6_data_out      ),
    .data7_in     ( input7_data_out      ),
    .credit_in    ( output0_credit       ),
    .credit_avail ( output_available[0]  ),
    .data_out     ( output0_packet       ),
    .data_out_v   ( output0_packet_v     )
    );

output_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) output1_block(
    .clk          ( clk                  ),
    .reset_       ( reset_               ),
    .input_sel    ( output1_decision     ),
    .sel_v        ( output_decision_v[1] ),
    .data0_in     ( input0_data_out      ),
    .data1_in     ( input1_data_out      ),
    .data2_in     ( input2_data_out      ),
    .data3_in     ( input3_data_out      ),
    .data4_in     ( input4_data_out      ),
    .data5_in     ( input5_data_out      ),
    .data6_in     ( input6_data_out      ),
    .data7_in     ( input7_data_out      ),
    .credit_in    ( output1_credit       ),
    .credit_avail ( output_available[1]  ),
    .data_out     ( output1_packet       ),
    .data_out_v   ( output1_packet_v     )
    );
```

```
output_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) output2_block(
    .clk          ( clk                  ),
    .reset_       ( reset_               ),
    .input_sel    ( output2_decision     ),
    .sel_v        ( output_decision_v[2] ),
    .data0_in     ( input0_data_out      ),
    .data1_in     ( input1_data_out      ),
    .data2_in     ( input2_data_out      ),
    .data3_in     ( input3_data_out      ),
    .data4_in     ( input4_data_out      ),
    .data5_in     ( input5_data_out      ),
    .data6_in     ( input6_data_out      ),
    .data7_in     ( input7_data_out      ),
    .credit_in    ( output2_credit       ),
    .credit_avail ( output_available[2]  ),
    .data_out     ( output2_packet       ),
    .data_out_v   ( output2_packet_v     )
    );

output_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) output3_block(
    .clk          ( clk                  ),
    .reset_       ( reset_               ),
    .input_sel    ( output3_decision     ),
    .sel_v        ( output_decision_v[3] ),
    .data0_in     ( input0_data_out      ),
    .data1_in     ( input1_data_out      ),
    .data2_in     ( input2_data_out      ),
    .data3_in     ( input3_data_out      ),
    .data4_in     ( input4_data_out      ),
    .data5_in     ( input5_data_out      ),
    .data6_in     ( input6_data_out      ),
    .data7_in     ( input7_data_out      ),
    .credit_in    ( output3_credit       ),
    .credit_avail ( output_available[3]  ),
    .data_out     ( output3_packet       ),
    .data_out_v   ( output3_packet_v     )
    );

output_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) output4_block(
    .clk          ( clk                  ),
    .reset_       ( reset_               ),
    .input_sel    ( output4_decision     ),
    .sel_v        ( output_decision_v[4] ),
    .data0_in     ( input0_data_out      ),
    .data1_in     ( input1_data_out      ),
    .data2_in     ( input2_data_out      ),
    .data3_in     ( input3_data_out      ),
    .data4_in     ( input4_data_out      ),
    .data5_in     ( input5_data_out      ),
    .data6_in     ( input6_data_out      ),
    .data7_in     ( input7_data_out      ),
    .credit_in    ( output4_credit       ),
    .credit_avail ( output_available[4]  ),
    .data_out     ( output4_packet       ),
    .data_out_v   ( output4_packet_v     )
    );

output_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) output5_block(
    .clk          ( clk                  ),
    .reset_       ( reset_               ),
    .input_sel    ( output5_decision     ),
    .sel_v        ( output_decision_v[5] ),
    .data0_in     ( input0_data_out      ),
    .data1_in     ( input1_data_out      ),
    .data2_in     ( input2_data_out      ),
    .data3_in     ( input3_data_out      ),
    .data4_in     ( input4_data_out      ),
    .data5_in     ( input5_data_out      ),
    .data6_in     ( input6_data_out      ),
    .data7_in     ( input7_data_out      ),
    .credit_in    ( output5_credit       ),
```

```
    .credit_avail ( output_available[5]  ),
    .data_out     ( output5_packet       ),
    .data_out_v   ( output5_packet_v     )
    );

output_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) output6_block(
    .clk          ( clk                  ),
    .reset_       ( reset_               ),
    .input_sel    ( output6_decision     ),
    .sel_v        ( output_decision_v[6] ),
    .data0_in     ( input0_data_out      ),
    .data1_in     ( input1_data_out      ),
    .data2_in     ( input2_data_out      ),
    .data3_in     ( input3_data_out      ),
    .data4_in     ( input4_data_out      ),
    .data5_in     ( input5_data_out      ),
    .data6_in     ( input6_data_out      ),
    .data7_in     ( input7_data_out      ),
    .credit_in    ( output6_credit       ),
    .credit_avail ( output_available[6]  ),
    .data_out     ( output6_packet       ),
    .data_out_v   ( output6_packet_v     )
    );

output_block #(.SHIFT_WIDTH(SHIFT_WIDTH)) output7_block(
    .clk          ( clk                  ),
    .reset_       ( reset_               ),
    .input_sel    ( output7_decision     ),
    .sel_v        ( output_decision_v[7] ),
    .data0_in     ( input0_data_out      ),
    .data1_in     ( input1_data_out      ),
    .data2_in     ( input2_data_out      ),
    .data3_in     ( input3_data_out      ),
    .data4_in     ( input4_data_out      ),
    .data5_in     ( input5_data_out      ),
    .data6_in     ( input6_data_out      ),
    .data7_in     ( input7_data_out      ),
    .credit_in    ( output7_credit       ),
    .credit_avail ( output_available[7]  ),
    .data_out     ( output7_packet       ),
    .data_out_v   ( output7_packet_v     )
    );

//--------------------------
// Scheduler
//--------------------------
scheduler scheduler(
    .clk              ( clk               ),
    .reset_           ( reset_            ),
    .start            ( sched_start       ),
    .done             ( sched_done        ),
    .input0_req       ( input0_request    ),
    .input1_req       ( input1_request    ),
    .input2_req       ( input2_request    ),
    .input3_req       ( input3_request    ),
    .input4_req       ( input4_request    ),
    .input5_req       ( input5_request    ),
    .input6_req       ( input6_request    ),
    .input7_req       ( input7_request    ),
    .input0_decision  ( input0_decision   ),
    .input1_decision  ( input1_decision   ),
    .input2_decision  ( input2_decision   ),
    .input3_decision  ( input3_decision   ),
    .input4_decision  ( input4_decision   ),
    .input5_decision  ( input5_decision   ),
    .input6_decision  ( input6_decision   ),
    .input7_decision  ( input7_decision   ),
    .input_decision_v ( input_decision_v  ),
    .output0_decision ( output0_decision  ),
    .output1_decision ( output1_decision  ),
    .output2_decision ( output2_decision  ),
```

```
      .output3_decision  ( output3_decision  ),
      .output4_decision  ( output4_decision  ),
      .output5_decision  ( output5_decision  ),
      .output6_decision  ( output6_decision  ),
      .output7_decision  ( output7_decision  ),
      .output_decision_v ( output_decision_v ),
      .output_available  ( output_available  )
  );

  //--------------------------
  // Control logic. As soon as the scheduler finishes its last schedule, start
  // the next schedule
  //--------------------------
  always @(posedge clk)
  begin
    if(~reset_)
    begin
      sched_start <= 1;
    end
    else
    begin
      sched_start <= sched_done;
    end
  end

endmodule
```

```
//-------------------------------------------------------------------------------
// Module: scheduler
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//  This module implements a scheduler using the iSLIP scheduling algorithm.
//  The scheduler attempts to find a solution from 8 input ports to 8 output
//  ports.  The input ports send an occupancy vector indicating which outputs
//  they have pending packets for.  Over 8 iterations, the scheduler matches
//  inputs to outputs.  At the end of 8 iterations, the solution is sent to the
//  inputs to indicate which packet they should send, and to the outputs to
//  select which input to receive the packet from.
//
//  Scheduling Algorithm:
//
//  Step 1: Request.  Each unmatched input sends a request to every output
//          which for which it has a queued cell.
//
//  Step 2: Grant.  In an unmatched output receives any requests, it chooses
//          the one that appears next in a fixed, round-robin schedule
//          starting from the highest priority element.  The output notifies
//          each input whether or not its request was granted.  The pointer
//          g_i to the highest priority element of the round-robin schedule is
//          incremented (modulo N) to one location beyond the granted input
//          iff the grant is accepted in Step 3 of the first iteration.
//
//  Step 3: Accept.  If an unmatched input receives a grant, it accepts the one
//          that appears next in a fixed, round-robin schedule starting from
//          the highest priority element.  The pointer a_i to the highest
//          priority element of the round-robin schedule is incremented (modulo
//          N) to one location beyond the accepted output only if this input
//          was matched in the first iteration.
//
//-------------------------------------------------------------------------------
module scheduler(clk, reset_,
                 start,
                 done,
                 input0_req,
                 input1_req,
                 input2_req,
                 input3_req,
                 input4_req,
                 input5_req,
                 input6_req,
                 input7_req,
                 input0_decision,
                 input1_decision,
                 input2_decision,
                 input3_decision,
                 input4_decision,
                 input5_decision,
                 input6_decision,
                 input7_decision,
                 input_decision_v,
                 output0_decision,
                 output1_decision,
                 output2_decision,
                 output3_decision,
                 output4_decision,
                 output5_decision,
                 output6_decision,
                 output7_decision,
                 output_decision_v,
                 output_available
                );

  //--------------------
  // Parameters
  //--------------------
```

```
parameter IDLE       = 9'b000000001;
parameter ITERATION1 = 9'b000000010;
parameter ITERATION2 = 9'b000000100;
parameter ITERATION3 = 9'b000001000;
parameter ITERATION4 = 9'b000010000;
parameter ITERATION5 = 9'b000100000;
parameter ITERATION6 = 9'b001000000;
parameter ITERATION7 = 9'b010000000;
parameter ITERATION8 = 9'b100000000;

//--------------------
// Ports
//--------------------
input clk;
input reset_;

input start;
output done;

// Requests from the input ports
input [7:0] input0_req;
input [7:0] input1_req;
input [7:0] input2_req;
input [7:0] input3_req;
input [7:0] input4_req;
input [7:0] input5_req;
input [7:0] input6_req;
input [7:0] input7_req;

// Available inputs from the output ports.  Indicates that the output port
// has enough credit available to receive data.
input [7:0] output_available;

// 1-hot decisions for which input each output should select for its mux.
output [7:0] output0_decision;
output [7:0] output1_decision;
output [7:0] output2_decision;
output [7:0] output3_decision;
output [7:0] output4_decision;
output [7:0] output5_decision;
output [7:0] output6_decision;
output [7:0] output7_decision;

// Output decision valid
output [7:0] output_decision_v;

// 1-hot decisions for which output each input should send its packet to
output [7:0] input0_decision;
output [7:0] input1_decision;
output [7:0] input2_decision;
output [7:0] input3_decision;
output [7:0] input4_decision;
output [7:0] input5_decision;
output [7:0] input6_decision;
output [7:0] input7_decision;

// Input decision valid
output [7:0] input_decision_v;

// state registers
reg [8:0] state;
reg [8:0] next_state;
reg done ;
reg done_ns ;

wire first_iteration;
wire [7:0] input0_decision_ns;
wire [7:0] input1_decision_ns;
wire [7:0] input2_decision_ns;
wire [7:0] input3_decision_ns;
wire [7:0] input4_decision_ns;
```

```
wire [7:0] input5_decision_ns;
wire [7:0] input6_decision_ns;
wire [7:0] input7_decision_ns;


// Outputs from the grant arbiters
wire  [7:0] output0_gnt;
wire  [7:0] output1_gnt;
wire  [7:0] output2_gnt;
wire  [7:0] output3_gnt;
wire  [7:0] output4_gnt;
wire  [7:0] output5_gnt;
wire  [7:0] output6_gnt;
wire  [7:0] output7_gnt;

// Outputs from the accept arbiters
wire  [7:0] input0_acc;
wire  [7:0] input1_acc;
wire  [7:0] input2_acc;
wire  [7:0] input3_acc;
wire  [7:0] input4_acc;
wire  [7:0] input5_acc;
wire  [7:0] input6_acc;
wire  [7:0] input7_acc;

// store the decision made by the scheduler
reg [7:0] input0_decision;
reg [7:0] input1_decision;
reg [7:0] input2_decision;
reg [7:0] input3_decision;
reg [7:0] input4_decision;
reg [7:0] input5_decision;
reg [7:0] input6_decision;
reg [7:0] input7_decision;

// Requests grouped by output
wire [7:0] output0_req;
wire [7:0] output1_req;
wire [7:0] output2_req;
wire [7:0] output3_req;
wire [7:0] output4_req;
wire [7:0] output5_req;
wire [7:0] output6_req;
wire [7:0] output7_req;

// Group the output grants by input
wire [7:0] input0_gnt;
wire [7:0] input1_gnt;
wire [7:0] input2_gnt;
wire [7:0] input3_gnt;
wire [7:0] input4_gnt;
wire [7:0] input5_gnt;
wire [7:0] input6_gnt;
wire [7:0] input7_gnt;

// Indicate whether the input was matched to an output in the previous
// iteration
wire input0_matched;
wire input1_matched;
wire input2_matched;
wire input3_matched;
wire input4_matched;
wire input5_matched;
wire input6_matched;
wire input7_matched;

// Indicate whether the output was matched in the previous iteration.
// The output can't be matched if we are in the first iteration.
wire output0_matched;
wire output1_matched;
wire output2_matched;
```

```
wire output3_matched;
wire output4_matched;
wire output5_matched;
wire output6_matched;
wire output7_matched;

// Indicate whether the output got any accept.
wire output0_anyacc;
wire output1_anyacc;
wire output2_anyacc;
wire output3_anyacc;
wire output4_anyacc;
wire output5_anyacc;
wire output6_anyacc;
wire output7_anyacc;

// Group the accepts by output
wire [7:0] output0_acc;
wire [7:0] output1_acc;
wire [7:0] output2_acc;
wire [7:0] output3_acc;
wire [7:0] output4_acc;
wire [7:0] output5_acc;
wire [7:0] output6_acc;
wire [7:0] output7_acc;

// Masked off the reqs from the inputs.  Once an input has been matched, it
// should not submit any more requests.
wire [7:0] masked_input0_req;
wire [7:0] masked_input1_req;
wire [7:0] masked_input2_req;
wire [7:0] masked_input3_req;
wire [7:0] masked_input4_req;
wire [7:0] masked_input5_req;
wire [7:0] masked_input6_req;
wire [7:0] masked_input7_req;

assign masked_input0_req = ~{8{input0_matched}} & input0_req;
assign masked_input1_req = ~{8{input1_matched}} & input1_req;
assign masked_input2_req = ~{8{input2_matched}} & input2_req;
assign masked_input3_req = ~{8{input3_matched}} & input3_req;
assign masked_input4_req = ~{8{input4_matched}} & input4_req;
assign masked_input5_req = ~{8{input5_matched}} & input5_req;
assign masked_input6_req = ~{8{input6_matched}} & input6_req;
assign masked_input7_req = ~{8{input7_matched}} & input7_req;

// Group the input requests by requested output
swizzle input2output_req_swizzle(output7_req,
                                 output6_req,
                                 output5_req,
                                 output4_req,
                                 output3_req,
                                 output2_req,
                                 output1_req,
                                 output0_req,
                                 masked_input7_req,
                                 masked_input6_req,
                                 masked_input5_req,
                                 masked_input4_req,
                                 masked_input3_req,
                                 masked_input2_req,
                                 masked_input1_req,
                                 masked_input0_req
                                );

// Instantiate the output grant arbiters.  They are enabled if the output has
// not been matched yet, and can only update on the first iteration if their
// grant is accepted.
arbiter grant_arb0 (
   .clk          (clk),
   .reset_       (reset_),
```

```
    .arb_enable    (~output0_matched & output_available[0]),
    .update_enable (grant_arb0_update_enable),
    .req           (output0_req),
    .gnt           (output0_gnt),
    .anygnt        (output0_anygnt)
);
arbiter grant_arb1 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~output1_matched & output_available[1]),
    .update_enable (grant_arb1_update_enable),
    .req           (output1_req),
    .gnt           (output1_gnt),
    .anygnt        (output1_anygnt)
);
arbiter grant_arb2 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~output2_matched & output_available[2]),
    .update_enable (grant_arb2_update_enable),
    .req           (output2_req),
    .gnt           (output2_gnt),
    .anygnt        (output2_anygnt)
);
arbiter grant_arb3 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~output3_matched & output_available[3]),
    .update_enable (grant_arb3_update_enable),
    .req           (output3_req),
    .gnt           (output3_gnt),
    .anygnt        (output3_anygnt)
);
arbiter grant_arb4 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~output4_matched & output_available[4]),
    .update_enable (grant_arb4_update_enable),
    .req           (output4_req),
    .gnt           (output4_gnt),
    .anygnt        (output4_anygnt)
);
arbiter grant_arb5 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~output5_matched & output_available[5]),
    .update_enable (grant_arb5_update_enable),
    .req           (output5_req),
    .gnt           (output5_gnt),
    .anygnt        (output5_anygnt)
);
arbiter grant_arb6 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~output6_matched & output_available[6]),
    .update_enable (grant_arb6_update_enable),
    .req           (output6_req),
    .gnt           (output6_gnt),
    .anygnt        (output6_anygnt)
);
arbiter grant_arb7 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~output7_matched & output_available[7]),
    .update_enable (grant_arb7_update_enable),
    .req           (output7_req),
    .gnt           (output7_gnt),
    .anygnt        (output7_anygnt)
);

// Group the output grants by input
```

```
swizzle output2input_gnt_swizzle(input7_gnt,
                                 input6_gnt,
                                 input5_gnt,
                                 input4_gnt,
                                 input3_gnt,
                                 input2_gnt,
                                 input1_gnt,
                                 input0_gnt,
                                 output7_gnt,
                                 output6_gnt,
                                 output5_gnt,
                                 output4_gnt,
                                 output3_gnt,
                                 output2_gnt,
                                 output1_gnt,
                                 output0_gnt
                                );

// Instantiate the input accept arbiters.
arbiter accept_arb0 (
   .clk            (clk),
   .reset_         (reset_),
   .arb_enable     (~input0_matched),
   .update_enable  (accept_arb0_update_enable),
   .req            (input0_gnt),
   .gnt            (input0_acc),
   .anygnt         (input0_anyacc)
);
arbiter accept_arb1 (
   .clk            (clk),
   .reset_         (reset_),
   .arb_enable     (~input1_matched),
   .update_enable  (accept_arb1_update_enable),
   .req            (input1_gnt),
   .gnt            (input1_acc),
   .anygnt         (input1_anyacc)
);
arbiter accept_arb2 (
   .clk            (clk),
   .reset_         (reset_),
   .arb_enable     (~input2_matched),
   .update_enable  (accept_arb2_update_enable),
   .req            (input2_gnt),
   .gnt            (input2_acc),
   .anygnt         (input2_anyacc)
);
arbiter accept_arb3 (
   .clk            (clk),
   .reset_         (reset_),
   .arb_enable     (~input3_matched),
   .update_enable  (accept_arb3_update_enable),
   .req            (input3_gnt),
   .gnt            (input3_acc),
   .anygnt         (input3_anyacc)
);
arbiter accept_arb4 (
   .clk            (clk),
   .reset_         (reset_),
   .arb_enable     (~input4_matched),
   .update_enable  (accept_arb4_update_enable),
   .req            (input4_gnt),
   .gnt            (input4_acc),
   .anygnt         (input4_anyacc)
);
arbiter accept_arb5 (
   .clk            (clk),
   .reset_         (reset_),
   .arb_enable     (~input5_matched),
   .update_enable  (accept_arb5_update_enable),
   .req            (input5_gnt),
   .gnt            (input5_acc),
```

```
    .anygnt        (input5_anyacc)
);
arbiter accept_arb6 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~input6_matched),
    .update_enable (accept_arb6_update_enable),
    .req           (input6_gnt),
    .gnt           (input6_acc),
    .anygnt        (input6_anyacc)
);
arbiter accept_arb7 (
    .clk           (clk),
    .reset_        (reset_),
    .arb_enable    (~input7_matched),
    .update_enable (accept_arb7_update_enable),
    .req           (input7_gnt),
    .gnt           (input7_acc),
    .anygnt        (input7_anyacc)
);

// Indicate whether the input was matched to an output in the previous
// iteration
assign input0_matched = (|input0_decision) & ~first_iteration;
assign input1_matched = (|input1_decision) & ~first_iteration;
assign input2_matched = (|input2_decision) & ~first_iteration;
assign input3_matched = (|input3_decision) & ~first_iteration;
assign input4_matched = (|input4_decision) & ~first_iteration;
assign input5_matched = (|input5_decision) & ~first_iteration;
assign input6_matched = (|input6_decision) & ~first_iteration;
assign input7_matched = (|input7_decision) & ~first_iteration;

// Group the scheduler decisions by output
swizzle input2output_decision_swizzle(output7_decision,
                                      output6_decision,
                                      output5_decision,
                                      output4_decision,
                                      output3_decision,
                                      output2_decision,
                                      output1_decision,
                                      output0_decision,
                                      input7_decision,
                                      input6_decision,
                                      input5_decision,
                                      input4_decision,
                                      input3_decision,
                                      input2_decision,
                                      input1_decision,
                                      input0_decision
                                     );

// Indicate whether the output was matched in the previous iteration.
// The output can't be matched if we are in the first iteration.
assign output0_matched = (|output0_decision) & ~first_iteration;
assign output1_matched = (|output1_decision) & ~first_iteration;
assign output2_matched = (|output2_decision) & ~first_iteration;
assign output3_matched = (|output3_decision) & ~first_iteration;
assign output4_matched = (|output4_decision) & ~first_iteration;
assign output5_matched = (|output5_decision) & ~first_iteration;
assign output6_matched = (|output6_decision) & ~first_iteration;
assign output7_matched = (|output7_decision) & ~first_iteration;

// Update the accept arbiter if it is in the first iteration and it
// matches to any output.
assign accept_arb0_update_enable = first_iteration & input0_anyacc;
assign accept_arb1_update_enable = first_iteration & input1_anyacc;
assign accept_arb2_update_enable = first_iteration & input2_anyacc;
assign accept_arb3_update_enable = first_iteration & input3_anyacc;
assign accept_arb4_update_enable = first_iteration & input4_anyacc;
assign accept_arb5_update_enable = first_iteration & input5_anyacc;
assign accept_arb6_update_enable = first_iteration & input6_anyacc;
```

```
assign accept_arb7_update_enable = first_iteration & input7_anyacc;

// Update the grant arbiter if it is in the first iteration and the output is
// accepted by any input.
assign grant_arb0_update_enable = first_iteration & output0_anyacc;
assign grant_arb1_update_enable = first_iteration & output1_anyacc;
assign grant_arb2_update_enable = first_iteration & output2_anyacc;
assign grant_arb3_update_enable = first_iteration & output3_anyacc;
assign grant_arb4_update_enable = first_iteration & output4_anyacc;
assign grant_arb5_update_enable = first_iteration & output5_anyacc;
assign grant_arb6_update_enable = first_iteration & output6_anyacc;
assign grant_arb7_update_enable = first_iteration & output7_anyacc;

// Group the accepts by output
swizzle input2output_acc_swizzle(output7_acc,
                                 output6_acc,
                                 output5_acc,
                                 output4_acc,
                                 output3_acc,
                                 output2_acc,
                                 output1_acc,
                                 output0_acc,
                                 input7_acc,
                                 input6_acc,
                                 input5_acc,
                                 input4_acc,
                                 input3_acc,
                                 input2_acc,
                                 input1_acc,
                                 input0_acc
                                );

// Indicate whether the output got any accept.
assign output0_anyacc = |output0_acc;
assign output1_anyacc = |output1_acc;
assign output2_anyacc = |output2_acc;
assign output3_anyacc = |output3_acc;
assign output4_anyacc = |output4_acc;
assign output5_anyacc = |output5_acc;
assign output6_anyacc = |output6_acc;
assign output7_anyacc = |output7_acc;

always @(posedge clk)
begin
  if(!reset_)
  begin
    state <= IDLE;
    done <= 0;
    input0_decision <= 0;
    input1_decision <= 0;
    input2_decision <= 0;
    input3_decision <= 0;
    input4_decision <= 0;
    input5_decision <= 0;
    input6_decision <= 0;
    input7_decision <= 0;
  end
  else
  begin
    state <= next_state;
    done <= done_ns;
    input0_decision <= input0_decision_ns;
    input1_decision <= input1_decision_ns;
    input2_decision <= input2_decision_ns;
    input3_decision <= input3_decision_ns;
    input4_decision <= input4_decision_ns;
    input5_decision <= input5_decision_ns;
    input6_decision <= input6_decision_ns;
    input7_decision <= input7_decision_ns;
  end
end
```

```
  // If this is the first iteration, set the decision to the accept vectors for
  // each input.  Otherwise, or the new decision with the old decision.  Once a
  // decision is made, the accept arbiter will be disabled anyway.
  assign input0_decision_ns = (first_iteration) ? input0_acc : (input0_decision |
input0_acc);
  assign input1_decision_ns = (first_iteration) ? input1_acc : (input1_decision |
input1_acc);
  assign input2_decision_ns = (first_iteration) ? input2_acc : (input2_decision |
input2_acc);
  assign input3_decision_ns = (first_iteration) ? input3_acc : (input3_decision |
input3_acc);
  assign input4_decision_ns = (first_iteration) ? input4_acc : (input4_decision |
input4_acc);
  assign input5_decision_ns = (first_iteration) ? input5_acc : (input5_decision |
input5_acc);
  assign input6_decision_ns = (first_iteration) ? input6_acc : (input6_decision |
input6_acc);
  assign input7_decision_ns = (first_iteration) ? input7_acc : (input7_decision |
input7_acc);

  assign first_iteration = (state == ITERATION1);

  always @(*)
  begin
    done_ns = 0;
    next_state = IDLE;

    case(state) // full_case parallel_case
      IDLE:
      begin
        if(start)
        begin
          next_state = ITERATION1;
        end
      end
      ITERATION1: next_state = ITERATION2;
      ITERATION2: next_state = ITERATION3;
      ITERATION3: next_state = ITERATION4;
      ITERATION4: next_state = ITERATION5;
      ITERATION5: next_state = ITERATION6;
      ITERATION6: next_state = ITERATION7;
      ITERATION7: next_state = ITERATION8;
      ITERATION8:
      begin
        next_state = IDLE;
        done_ns = 1;
      end
      default: next_state = IDLE;
    endcase
  end

  assign input_decision_v[0] = done & (|input0_decision);
  assign input_decision_v[1] = done & (|input1_decision);
  assign input_decision_v[2] = done & (|input2_decision);
  assign input_decision_v[3] = done & (|input3_decision);
  assign input_decision_v[4] = done & (|input4_decision);
  assign input_decision_v[5] = done & (|input5_decision);
  assign input_decision_v[6] = done & (|input6_decision);
  assign input_decision_v[7] = done & (|input7_decision);

  assign output_decision_v[0] = done & (|output0_decision);
  assign output_decision_v[1] = done & (|output1_decision);
  assign output_decision_v[2] = done & (|output2_decision);
  assign output_decision_v[3] = done & (|output3_decision);
  assign output_decision_v[4] = done & (|output4_decision);
  assign output_decision_v[5] = done & (|output5_decision);
  assign output_decision_v[6] = done & (|output6_decision);
  assign output_decision_v[7] = done & (|output7_decision);

  // 1-hot assertions to make sure we don't grant more than 1 input to more
```

```
  // than one output.
  // synopsys translate_off
  assert_zero_1hot output0_assert(output0_decision,1'b1);
  assert_zero_1hot output1_assert(output1_decision,1'b1);
  assert_zero_1hot output2_assert(output2_decision,1'b1);
  assert_zero_1hot output3_assert(output3_decision,1'b1);
  assert_zero_1hot output4_assert(output4_decision,1'b1);
  assert_zero_1hot output5_assert(output5_decision,1'b1);
  assert_zero_1hot output6_assert(output6_decision,1'b1);
  assert_zero_1hot output7_assert(output7_decision,1'b1);

  assert_zero_1hot input0_assert(input0_decision,1'b1);
  assert_zero_1hot input1_assert(input1_decision,1'b1);
  assert_zero_1hot input2_assert(input2_decision,1'b1);
  assert_zero_1hot input3_assert(input3_decision,1'b1);
  assert_zero_1hot input4_assert(input4_decision,1'b1);
  assert_zero_1hot input5_assert(input5_decision,1'b1);
  assert_zero_1hot input6_assert(input6_decision,1'b1);
  assert_zero_1hot input7_assert(input7_decision,1'b1);
  // synopsys translate_on

endmodule
```

```
//-----------------------------------------------------------------------------
// Module: arbiter
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module impliments the arbiter described for the iSLIP scheduling
//   algorithm.  The arbiter is comprised of a programmable priority encoder
//   and a register containing the current programmed priority.  The arbiter
//   receives an 8-wide vector of requests and outputs a 1-hot signal
//   indicating the selected request.  If the update_enable and anygnt signals
//   are asserted, then the priority is updated to one beyond the request that
//   received the current grant.
//-----------------------------------------------------------------------------
module arbiter(clk, reset_, arb_enable, update_enable, req, gnt, anygnt);
  input        clk;
  input        reset_;

  output [7:0] gnt;
  output       anygnt;

  input        arb_enable;
  input        update_enable;
  input  [7:0] req;

  reg [2:0] priority;
  reg [2:0] priority_ns;

  wire [7:0] ppegnt;
  wire       ppe_anygnt;

  // Programmable priority encoder.
  ppe ppe(ppe_anygnt, ppegnt, req, priority);

  // If the arbiter is disabled, don't send any new grant signals out.
  assign gnt = (arb_enable) ? ppegnt : 8'h00;
  assign anygnt = (arb_enable) ? ppe_anygnt : 1'b0;

  // Update the priority
  always @(posedge clk)
  begin
    if(!reset_)
    begin
      priority <= 0;
    end
    else
    begin
      priority <= priority_ns;
    end
  end

  // Encode the grant from 1-hot to binary
  reg [2:0] gnt_enc;
  always @(*)
  begin
    case(gnt) // full_case parallel_case
      8'h01: gnt_enc = 0;
      8'h02: gnt_enc = 1;
      8'h04: gnt_enc = 2;
      8'h08: gnt_enc = 3;
      8'h10: gnt_enc = 4;
      8'h20: gnt_enc = 5;
      8'h40: gnt_enc = 6;
      8'h80: gnt_enc = 7;
      default: gnt_enc = 3'hx;
    endcase
  end

  // Determine the next priority value
  always @(*)
```

```
  begin
    // update the priority if arb & updates are enabled and an input was
    // granted.
    if(update_enable & arb_enable & anygnt)
      priority_ns = gnt_enc + 1;
    else
      priority_ns = priority;
  end

endmodule
```

```
//--------------------------------------------------------------------------------
// Module: assert_zero_1hot
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module errors if a signal is not either all zero or 1-hot.
//--------------------------------------------------------------------------------
module assert_zero_1hot(signal,valid);
  parameter WIDTH = 8;
  input [WIDTH-1:0] signal;
  input valid;

  integer count;
  integer i;

  always @(signal or valid)
  begin
    if(valid)
    begin
      count = 0;
      for(i=0; i<WIDTH; i=i+1)
      begin
        count = count + signal[i];
      end
      if(count != 0 && count != 1)
      begin
        $display("ERROR: %m signal not 0 or 1hot: %x",signal);
      end
    end
  end
endmodule
```

```
//----------------------------------------------------------------------------
// Module: ppe
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This is a programmable priority encoder.  It can be programmed with the
//   current highest priority, and this programmed priority determines which
//   input has the highest priority.  The programmed priority is input to the
//   module.
//
//   The priority encoder is implemented using two simple non-programmable
//   priority encoders.  The first encoder is a thermo-encoder.  The input to
//   this encoder is masked by a thermometer encoding based on the current
//   priority.  If the thermo-encoder detects a request, it's prioritized grant
//   is used.  If it does not detect a request, then the simple unmaked
//   priority encoder's output is used.  The use of two encoders saves the
//   complexity of wrapping the priority around if the request is lower then
//   the currently programmed priority.
//----------------------------------------------------------------------------
module ppe(anygnt, gnt, req, priority);
  output       anygnt;
  output [7:0] gnt;
  input  [7:0] req;
  input  [2:0] priority;

  reg [7:0] pri_thermo;

  // Thermo mask encoder
  always @(*)
  begin
    case(priority) // full_case parallel_case
      0: pri_thermo = 8'b0000_0000;
      1: pri_thermo = 8'b0000_0001;
      2: pri_thermo = 8'b0000_0011;
      3: pri_thermo = 8'b0000_0111;
      4: pri_thermo = 8'b0000_1111;
      5: pri_thermo = 8'b0001_1111;
      6: pri_thermo = 8'b0011_1111;
      7: pri_thermo = 8'b0111_1111;
    endcase
  end

  wire [7:0] thermo_req = ~pri_thermo & req;

  // simple priority encoder
  wire [7:0] gnt_smpl_pe;
  simple_pe simple_pe(anygnt, gnt_smpl_pe, req);

  // thermo priority encoder
  wire anygnt_thermo;
  wire [7:0] gnt_thermo_pe;
  simple_pe simple_thermo_pe(anygnt_thermo, gnt_thermo_pe, thermo_req);

  // Mux the output.  If the thermo-encoder granted, it takes precidence over
  // the simple encoder.
  assign gnt = (anygnt_thermo) ? gnt_thermo_pe : gnt_smpl_pe;

endmodule
```

```
//-----------------------------------------------------------------------------
// Module: simple_pe
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This implements a simple combinational 1-hot priority encoder.
//-----------------------------------------------------------------------------
module simple_pe(anygnt, gnt, req);
  output anygnt;
  output [7:0] gnt;
  input  [7:0] req;

  reg[7:0] gnt;

  // Simple priority encoder.  I thought about useing a parallel
  // prefix treee strucuture to be more efficient, but the design_vision
  // compiler did a nice optimization job on this.
  always @(*)
  begin
    casez(req) // full_case parallel_case
      8'b???????1: gnt = 8'b00000001;
      8'b??????10: gnt = 8'b00000010;
      8'b?????100: gnt = 8'b00000100;
      8'b????1000: gnt = 8'b00001000;
      8'b???10000: gnt = 8'b00010000;
      8'b??100000: gnt = 8'b00100000;
      8'b?1000000: gnt = 8'b01000000;
      8'b10000000: gnt = 8'b10000000;
      8'b00000000: gnt = 8'b00000000;
    endcase
  end

  // As long as there is at least one req, there will be a grant
  assign anygnt = |req;
endmodule
```

```
//-----------------------------------------------------------------------------
// Module: swizzle
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//  This module simply swizzle signals.
//
// Example:
//   Output0 - the 0'th bit of all of the inputs
//-----------------------------------------------------------------------------
module
swizzle(output7,output6,output5,output4,output3,output2,output1,output0,input7,input6,inp
ut5,input4,input3,input2,input1,input0);
  output [7:0] output7;
  output [7:0] output6;
  output [7:0] output5;
  output [7:0] output4;
  output [7:0] output3;
  output [7:0] output2;
  output [7:0] output1;
  output [7:0] output0;
  input [7:0] input7;
  input [7:0] input6;
  input [7:0] input5;
  input [7:0] input4;
  input [7:0] input3;
  input [7:0] input2;
  input [7:0] input1;
  input [7:0] input0;

  assign output7 = {input7[7],
                    input6[7],
                    input5[7],
                    input4[7],
                    input3[7],
                    input2[7],
                    input1[7],
                    input0[7]
                   };
  assign output6 = {input7[6],
                    input6[6],
                    input5[6],
                    input4[6],
                    input3[6],
                    input2[6],
                    input1[6],
                    input0[6]
                   };

  assign output5 = {input7[5],
                    input6[5],
                    input5[5],
                    input4[5],
                    input3[5],
                    input2[5],
                    input1[5],
                    input0[5]
                   };

  assign output4 = {input7[4],
                    input6[4],
                    input5[4],
                    input4[4],
                    input3[4],
                    input2[4],
                    input1[4],
                    input0[4]
                   };
```

```
  assign output3 = {input7[3],
                    input6[3],
                    input5[3],
                    input4[3],
                    input3[3],
                    input2[3],
                    input1[3],
                    input0[3]
                   };

  assign output2 = {input7[2],
                    input6[2],
                    input5[2],
                    input4[2],
                    input3[2],
                    input2[2],
                    input1[2],
                    input0[2]
                   };

  assign output1 = {input7[1],
                    input6[1],
                    input5[1],
                    input4[1],
                    input3[1],
                    input2[1],
                    input1[1],
                    input0[1]
                   };

  assign output0 = {input7[0],
                    input6[0],
                    input5[0],
                    input4[0],
                    input3[0],
                    input2[0],
                    input1[0],
                    input0[0]
                   };

endmodule
```

```verilog
//-----------------------------------------------------------------------------
// Module: input_block
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//  This module is the input side of a crossbar module.  It receives incoming
//  packets from one source for 8 destinations, queues them in a shared
//  linked-list style virtual output queue, and shifts them to the selected
//  destination port once indicated by the scheduler.  The module is comprised
//  of four memories for storing the virtual output queues and linked-list
//  information, a free list FIFO, and an output shift register.  The output
//  shift register shifts SHIFT_WIDTH bits at a time to the destination port.
//
//  Scheduling:  The module outputs an occupancy vector indicating which
//  outputs it has pending packets for.  The scheduler uses this occupancy
//  vector from the other input blocks to solve a schedule and sends a request
//  back to the input block indicating the destination it should send a packet
//  for.
//-----------------------------------------------------------------------------

//-----------------------------
// Defines
//-----------------------------
`define PACKET_SRC_RANGE  71:69
`define PACKET_DEST_RANGE 68:66
`define PACKET_DATA_RANGE 65:0

`define HEAD_TAIL_MEMINDEX_RANGE   5:1
`define HEAD_TAIL_VALID_RANGE      0
`define LINKED_LIST_MEMINDEX_RANGE 5:1
`define LINKED_LIST_VALID_RANGE    0

`define PACKET_ARRAY_ADDR_WIDTH 5
`define LINKED_LIST_ENTRY_WIDTH `PACKET_ARRAY_ADDR_WIDTH+1
`define HEAD_TAIL_ADDR_WIDTH    3
`define HEAD_TAIL_ENTRY_WIDTH   `PACKET_ARRAY_ADDR_WIDTH+1

module input_block(clk, reset_, packet_in, packet_v, credit, tx_dest, tx_dest_v, tx_out,
request_out);
  //-----------------------------
  // Parameters
  //-----------------------------
  parameter PACKET_SRC_WIDTH  = 3;
  parameter PACKET_DEST_WIDTH = 3;
  parameter PACKET_DATA_WIDTH = 66;
  parameter PACKET_WIDTH      = PACKET_SRC_WIDTH  +
                                PACKET_DEST_WIDTH +
                                PACKET_DATA_WIDTH;
  parameter MAX_REQUESTS      = 32;
  parameter SHIFT_WIDTH       = 1;

  //-----------------------------
  // Ports
  //-----------------------------
  input clk;
  input reset_;
  input [PACKET_WIDTH/2-1:0] packet_in; // Half of an incoming packet - 2 phases
  input                      packet_v;  // Packet port valid
  input [7:0]                tx_dest;   // send request from scheduler
  input                      tx_dest_v; // send request valid
  output                     credit;    // credit back to src once room is available
  output [SHIFT_WIDTH-1:0]   tx_out;    // Data from input_block to output_blk
  output [7:0]               request_out; // Occupancy vector of pending packets

  reg                        credit;

  //-----------------------------
  // Memory arrays
  //-----------------------------
```

```
// Packet array - main storage for packets shared between all destinations
reg  [`PACKET_ARRAY_ADDR_WIDTH-1:0] packet_array_waddr;
reg  [`PACKET_ARRAY_ADDR_WIDTH-1:0] packet_array_raddr;
reg                                 packet_array_wen;
reg                                 packet_array_ren;
reg  [PACKET_WIDTH-1:0]             packet_array_wdata;
wire [PACKET_WIDTH-1:0]             packet_array_rdata;

mem_32_1rp_1wp #(.WIDTH(PACKET_WIDTH)) packet_array (
  .clk    ( clk                ),
  .reset_ ( reset_             ),
  .waddr  ( packet_array_waddr ),
  .wen    ( packet_array_wen   ),
  .wdata  ( packet_array_wdata ),
  .raddr  ( packet_array_raddr ),
  .ren    ( packet_array_ren   ),
  .rdata  ( packet_array_rdata )
);

// Linked list array - chain of pointers to next packet for given dest
reg  [`PACKET_ARRAY_ADDR_WIDTH-1:0] linked_list_array_waddr;
reg  [`PACKET_ARRAY_ADDR_WIDTH-1:0] linked_list_array_raddr;
reg                                 linked_list_array_wen;
reg                                 linked_list_array_ren;
reg  [`LINKED_LIST_ENTRY_WIDTH-1:0] linked_list_array_wdata;
wire [`LINKED_LIST_ENTRY_WIDTH-1:0] linked_list_array_rdata;

mem_32_1rp_1wp #(.WIDTH(`LINKED_LIST_ENTRY_WIDTH)) linked_list_array (
  .clk    ( clk                     ),
  .reset_ ( reset_                  ),
  .waddr  ( linked_list_array_waddr ),
  .wen    ( linked_list_array_wen   ),
  .wdata  ( linked_list_array_wdata ),
  .raddr  ( linked_list_array_raddr ),
  .ren    ( linked_list_array_ren   ),
  .rdata  ( linked_list_array_rdata )
);

// Dest head array - head of linked list and valid signal
reg  [`HEAD_TAIL_ADDR_WIDTH-1:0]  dest_head_array_waddr;
reg  [`HEAD_TAIL_ADDR_WIDTH-1:0]  dest_head_array_raddr;
reg                               dest_head_array_wen;
reg                               dest_head_array_ren;
reg  [`HEAD_TAIL_ENTRY_WIDTH-1:0] dest_head_array_wdata;
wire [`HEAD_TAIL_ENTRY_WIDTH-1:0] dest_head_array_rdata;

mem_8_1rp_1wp #(.WIDTH(`HEAD_TAIL_ENTRY_WIDTH)) dest_head_array (
  .clk       ( clk                   ),
  .reset_    ( reset_                ),
  .waddr     ( dest_head_array_waddr ),
  .wen       ( dest_head_array_wen   ),
  .wdata     ( dest_head_array_wdata ),
  .raddr     ( dest_head_array_raddr ),
  .ren       ( dest_head_array_ren   ),
  .rdata     ( dest_head_array_rdata ),
  .valid_out ( request_out           )
);

// Dest tail array - tail of linked list and valid signal
reg  [`HEAD_TAIL_ADDR_WIDTH-1:0]  dest_tail_array_waddr;
reg  [`HEAD_TAIL_ADDR_WIDTH-1:0]  dest_tail_array_raddr;
reg                               dest_tail_array_wen;
reg                               dest_tail_array_ren;
reg  [`HEAD_TAIL_ENTRY_WIDTH-1:0] dest_tail_array_wdata;
wire [`HEAD_TAIL_ENTRY_WIDTH-1:0] dest_tail_array_rdata;

mem_8_1rp_1wp #(.WIDTH(`HEAD_TAIL_ENTRY_WIDTH)) dest_tail_array (
  .clk    ( clk                   ),
  .reset_ ( reset_                ),
  .waddr  ( dest_tail_array_waddr ),
```

```
  .wen    ( dest_tail_array_wen   ),
  .wdata  ( dest_tail_array_wdata ),
  .raddr  ( dest_tail_array_raddr ),
  .ren    ( dest_tail_array_ren   ),
  .rdata  ( dest_tail_array_rdata )
);

// Free list - this is a fifo of available PACKET_ARRAY locations
wire                                  freelist_full;
wire                                  freelist_empty;
wire [`PACKET_ARRAY_ADDR_WIDTH-1:0] freelist_rdata;
reg                                   freelist_ren;
reg  [`PACKET_ARRAY_ADDR_WIDTH-1:0] freelist_wdata;
reg                                   freelist_wen;

freelist_fifo freelist (
  .clk      ( clk            ),
  .reset_   ( reset_         ),
  .full     ( freelist_full  ),
  .empty    ( freelist_empty ),
  .rdata    ( freelist_rdata ),
  .read_en  ( freelist_ren   ),
  .write_en ( freelist_wen   ),
  .wdata    ( freelist_wdata )
);

// Data shifter - this shift register gets loaded with the packet that gets
// scheduled to be transmitted and shifts it out across the crossbar on the
// tx_out port
reg  [PACKET_WIDTH-1:0] shift_reg_data_in;
reg                     shift_reg_wen;
wire [SHIFT_WIDTH-1:0]  shift_reg_data_out;

shift_reg #(.WIDTH(PACKET_WIDTH),.SHIFT_WIDTH(SHIFT_WIDTH)) shift_reg (
  .clk       ( clk                ),
  .reset_    ( reset_             ),
  .data_in   ( shift_reg_data_in  ),
  .load_data ( shift_reg_wen      ),
  .shift_out ( shift_reg_data_out )
);

// For storing a temporary packet if there is a rx request in the same cycle
// as a tx request
reg [PACKET_WIDTH/2-1:0] packet_ph0_buffer;
reg [PACKET_WIDTH-1:0  ] delayed_packet_buffer;
reg                      delayed_packet_v;
reg                      packet_phase;

// Combinational Logic
assign tx_out = shift_reg_data_out;

// Pending packet bypass logic.  For simpler combo logic, we don't handle an
// incoming packet and a scheduled send request in the same clock.  If we
// receive a scheduled send request, we store the incoming packet in a
// delayed packet buffer for one cycle.  We are guaranteed not to receive
// send requests in consecutive clocks, so the incoming packet will be
// consumed on the next clock
wire [PACKET_WIDTH-1:0] active_packet_in;
wire packet_ready;
assign active_packet_in = (delayed_packet_v) ? delayed_packet_buffer :
                                               {packet_ph0_buffer,packet_in};
assign packet_ready = delayed_packet_v | packet_phase;

reg dest_head_valid;
reg dest_tail_valid;
reg [`PACKET_ARRAY_ADDR_WIDTH-1:0] writeIndex;
reg last_entry;

// Encode the 1-hot dest signal back to binary encoding for use in indexing
// the linked list
reg [`HEAD_TAIL_ADDR_WIDTH-1:0] dest_enc;
```

```
always @(*)
begin
  case(tx_dest) // full_case parallel_case
    8'b00000001 : dest_enc = 3'h0;
    8'b00000010 : dest_enc = 3'h1;
    8'b00000100 : dest_enc = 3'h2;
    8'b00001000 : dest_enc = 3'h3;
    8'b00010000 : dest_enc = 3'h4;
    8'b00100000 : dest_enc = 3'h5;
    8'b01000000 : dest_enc = 3'h6;
    8'b10000000 : dest_enc = 3'h7;
    default: dest_enc = 3'hx;
  endcase
end

// Linked list update logic
always @(*)
begin
  credit                  = 0;

  freelist_ren            = 0;
  freelist_wen            = 0;
  packet_array_wen        = 0;
  dest_head_array_wen     = 0;
  dest_tail_array_wen     = 0;
  linked_list_array_wen   = 0;
  shift_reg_wen           = 0;

  // Don't care
  writeIndex              = 5'hx;

  freelist_wdata          = 5'hx;

  packet_array_waddr      = 5'hx;
  packet_array_wdata      = {PACKET_WIDTH{1'bx}};
  packet_array_raddr      = 5'hx;
  packet_array_ren        = 1'hx;

  dest_head_array_ren     = 1'hx;
  dest_head_array_raddr   = 3'hx;
  dest_head_array_waddr   = 3'hx;
  dest_head_array_wdata   = 6'hx;

  dest_tail_array_ren     = 1'hx;
  dest_tail_array_raddr   = 3'hx;
  dest_tail_array_waddr   = 3'hx;
  dest_tail_array_wdata   = 6'hx;

  linked_list_array_ren   = 1'hx;
  linked_list_array_raddr = 5'hx;
  linked_list_array_waddr = 5'hx;
  linked_list_array_wdata = 6'hx;

  shift_reg_data_in       = {PACKET_WIDTH{1'bx}};

  last_entry              = 1'hx;
  dest_head_valid         = 1'hx;
  dest_tail_valid         = 1'hx;


  casez({packet_ready,tx_dest_v}) // full_case parallel_case
    // For an incoming packet request:
e   // 1. Find an empty entry in the packet array
    // 2. If there were no packets pending for this destination, initialize
    //    the DestHead array with this entry
    // 3. If there were packets pending for this destination, update the
    //    LinkedList and DestTail arrays to insert the new packet in the
    //    linked list for this destination
    2'b10:
    begin
      // For an incoming packet, the following sequence is executed
```

```
        // Get the next free entry in the packet array from the free list;
        writeIndex    = freelist_rdata;
        freelist_ren = 1'b1;

        // Load the packet into the packet array
        //   PacketArray[writeIndex] <= receive_packet;
        packet_array_waddr = writeIndex;
        packet_array_wdata = active_packet_in;
        packet_array_wen    = 1'b1;

        // Update the linked list for this destination
        //   If(!DestHead[receive_packet.dest].valid)
        //     DestHead[receive_packet.dest].memIndex = writeIndex;
        //     DestHead[receive_packet.dest].valid = 1;
        dest_head_array_ren    = 1'b1;
        dest_head_array_raddr = active_packet_in[`PACKET_DEST_RANGE];
        dest_head_valid        = dest_head_array_rdata[`HEAD_TAIL_VALID_RANGE];
        dest_head_array_wen    = ~dest_head_valid;
        dest_head_array_waddr = active_packet_in[`PACKET_DEST_RANGE];
        dest_head_array_wdata = {writeIndex,1'b1};

        //   If(!DestTail[receive_packet.dest].valid)
        //     LinkedListArray[DestTail[receive_packet.dest].memIndex].next <=
writeIndex;
        //     LinkedListArray[DestTail[receive_packet.dest].memIndex].valid <= 1;
        dest_tail_array_ren      = 1'b1;
        dest_tail_array_raddr    = active_packet_in[`PACKET_DEST_RANGE];
        dest_tail_valid          = dest_tail_array_rdata[`HEAD_TAIL_VALID_RANGE];
        linked_list_array_waddr = dest_tail_array_rdata[`HEAD_TAIL_MEMINDEX_RANGE];
        linked_list_array_wdata = {writeIndex,1'b1};
        linked_list_array_wen    = dest_tail_valid;

        //    DestTail[receive_packet.dest].valid <= 1;
        //    DestTail[receive_packet.dest].memIndex <= writeIndex;
        dest_tail_array_waddr = active_packet_in[`PACKET_DEST_RANGE];
        dest_tail_array_wdata = {writeIndex,1'b1};
        dest_tail_array_wen    = 1'b1;
      end
      // For a scheduled send request:
      // 1. Load the output shift register with the oldest packet (DestHead)
      //    for the selected destination.
      // 2. Update the LinkedList and DestHead to remove the packet from the
      //    linked list for the selected destination.
      // 3. De-allocate the PacketArray entry that was just moved to the output
      //    shift register, and return its index to the FreeList FIFO.
      // 4. Return a credit to the device attached to this input port,
      //    indicating there is room for another packet in the PacketArray.
      2'bz1:
      begin
        // Load the output shift register
        //   ShiftOut <= PacketArray[DestHead[dest]].memIndex];
        dest_head_array_raddr = dest_enc;
        packet_array_raddr = dest_head_array_rdata[`HEAD_TAIL_MEMINDEX_RANGE];
        shift_reg_data_in = packet_array_rdata;
        dest_head_array_ren = 1'b1;
        packet_array_ren = 1'b1;
        shift_reg_wen = 1'b1;

        // Update the linked list
        //   DestHead[dest] <= LinkedListArray[DestHead[dest].memIndex];
        //   LinkedListArray[DestHead[dest].memIndex].valid <= 0;
        linked_list_array_raddr = dest_head_array_rdata[`HEAD_TAIL_MEMINDEX_RANGE];
        dest_head_array_wdata = linked_list_array_rdata;
        dest_head_array_waddr = dest_enc;
        dest_head_array_wen = 1'b1;
        linked_list_array_waddr = dest_head_array_rdata[`HEAD_TAIL_MEMINDEX_RANGE];
        linked_list_array_wdata = 6'h00;
        linked_list_array_ren = 1'b1;
        linked_list_array_wen = 1'b1;

        // If this is the last packet for the delected destination, invalidate
```

```
        // the tail
        //   If(DestHead[dest].memIndex == DestTail[dest].memIndex)
        //     DestTail[dest].valid <= 0;
        dest_tail_array_raddr = dest_enc;
        dest_tail_array_ren = 1'b1;
        last_entry = (dest_tail_array_rdata[`HEAD_TAIL_MEMINDEX_RANGE] ==
dest_head_array_rdata[`HEAD_TAIL_MEMINDEX_RANGE]);
        dest_tail_array_waddr = dest_enc;
        dest_tail_array_wdata = 6'h00;
        dest_tail_array_wen = last_entry;

        // Put the PacketArray entry number back on the free list
        // FreeListFIFO.push(DestHead[dest].memIndex);
        freelist_wdata = dest_head_array_rdata[`HEAD_TAIL_MEMINDEX_RANGE];
        freelist_wen = 1'b1;

        // Return a credit to the device sending packets to this port
        credit = 1'b1;
      end
      default:
      begin
      end
    endcase
  end

  // We receive the packets in two phases.  The first phase is placed in the
  // packet_ph0_buffer.  When the second phase arrives, if there isn't a
  // conflicting scheduled send request, put the two phases in the PacketArray.
  // If there is a conflicting send request, move the packet to the
  // delayed_packet_buffer for one cycle.
  always @(posedge clk)
  begin
    if(~reset_)
    begin
      packet_phase <= 0;
      delayed_packet_v <= 1'b0;
    end
    else
    begin
      if(packet_v & packet_phase == 1'b0)
        packet_ph0_buffer <= packet_in;

      // If we get the second packet phase at the same time as a transfer
      // request, we put the packet in the delayed packet buffer and delay it
      // one cycle
      if(packet_phase & packet_v & tx_dest_v)
      begin
        delayed_packet_buffer <= {packet_ph0_buffer,packet_in};
        delayed_packet_v <= 1'b1;
      end

      // We can always clean out the delayed packet on a 0th packet phase
      if(~packet_phase)
        delayed_packet_v <= 1'b0;

      if(packet_v)
        packet_phase <= ~packet_phase;
    end
  end

endmodule
```

```
//---------------------------------------------------------------------------
// Module: freelist_fifo
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module impliments a FIFO that maintains a free list of locations in
//   the shared PacketArray. It is a simple FIFO with the exception that it is
//   initialed to be full, with the entries containing the consecutive values
//   of 0 thru 31.
//---------------------------------------------------------------------------
module freelist_fifo(rdata,full,empty,clk,reset_,read_en,write_en,wdata);
  //===================================
  // Parameter Declarations
  //===================================

  parameter WIDTH = 5;
  parameter DEPTH = 32;
  parameter PTR_WIDTH = 5;

  //===================================
  // Port Declarations
  //===================================

  output [WIDTH-1:0] rdata;
  output full;
  output empty;

  input clk;
  input reset_;
  input [WIDTH-1:0] wdata;
  input write_en;
  input read_en;

  //===================================
  // Internal Storage and Wires
  //===================================

  reg [WIDTH-1:0] data [0:DEPTH-1];

  reg [PTR_WIDTH-1:0] headptr;
  reg [PTR_WIDTH-1:0] tailptr;
  reg [PTR_WIDTH  :0] slots_used;

  assign full  = (slots_used == DEPTH) ? 1'b1 : 1'b0;
  assign empty = (slots_used == 0) ? 1'b1 : 1'b0;

  // Mux the output.  When read_en is set, that means the other device will
  // latch the rdata on this clock edge.  This eliminates one clock from the
  // passing of data.
  assign rdata = (read_en) ? data[tailptr] : {WIDTH{1'bz}};

  always @(posedge clk)
  begin
    if(~reset_)
    begin
      data[0] <= 0;
      data[1] <= 1;
      data[2] <= 2;
      data[3] <= 3;
      data[4] <= 4;
      data[5] <= 5;
      data[6] <= 6;
      data[7] <= 7;
      data[8] <= 8;
      data[9] <= 9;
      data[10] <= 10;
      data[11] <= 11;
      data[12] <= 12;
      data[13] <= 13;
```

```
      data[14] <= 14;
      data[15] <= 15;
      data[16] <= 16;
      data[17] <= 17;
      data[18] <= 18;
      data[19] <= 19;
      data[20] <= 20;
      data[21] <= 21;
      data[22] <= 22;
      data[23] <= 23;
      data[24] <= 24;
      data[25] <= 25;
      data[26] <= 26;
      data[27] <= 27;
      data[28] <= 28;
      data[29] <= 29;
      data[30] <= 30;
      data[31] <= 31;

      headptr    <= 0;
      tailptr    <= 0;
      slots_used <= 32; // All slots are ful
    end
    else
    begin
      // If we're doing a read
      if(read_en & ~empty)
      begin
        tailptr <= tailptr + 1;

        if(write_en)
        begin
          data[headptr] <= wdata;
          headptr <= headptr + 1;
          slots_used <= slots_used;
        end
        else
        begin
          slots_used <= slots_used - 1;
        end
      end

      // If we're doing a write
      if(write_en & ~full & ~read_en)
      begin
        data[headptr] <= wdata;
        headptr <= headptr + 1;
        slots_used <= slots_used + 1;
      end

      // synopsys translate_off
      if(write_en & full & ~read_en)
      begin
        $display("ERROR, writing to full fifo");
      end
      if(read_en & empty)
      begin
        $display("ERROR, reading from empty fifo");
      end
      // synopsys translate_on
    end
  end
endmodule
```

```verilog
//---------------------------------------------------------------------------
// Module: mem_8_1rp_1wp
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//  This implements an 8-entry memory with 1 write port and 1 read port
//---------------------------------------------------------------------------
module mem_8_1rp_1wp(clk,reset_,waddr,wen,wdata,raddr,ren,rdata,valid_out);
  parameter ADDR_WIDTH = 3;
  parameter NUM_ENTRIES = 8;
  parameter WIDTH = 6;

  input clk;
  input reset_;

  input [ADDR_WIDTH-1:0]   waddr;
  input                    wen;
  input [WIDTH-1:0]        wdata;

  input [ADDR_WIDTH-1:0]   raddr;
  input                    ren;
  output [WIDTH-1:0]       rdata;
  output [NUM_ENTRIES-1:0] valid_out;

  reg    [WIDTH-1:0]       rdata;
  reg    [NUM_ENTRIES-1:0] valid_out;

  reg [WIDTH-1:0] data [0:NUM_ENTRIES-1];

  integer i;

  // Synchronous writes
  always @(posedge clk)
  begin
    if(~reset_)
    begin
      for(i=0;i<NUM_ENTRIES;i=i+1)
        data[i] <= {WIDTH{1'b0}};
    end
    else
    begin
      if(wen)
      begin
        data[waddr] <= wdata;
      end
    end
  end

  // Asynchronous reads
  always @(*)
  begin
    if(ren)
      rdata = data[raddr];
    else
      rdata = {WIDTH{1'bz}};
  end

  // Valid signals
  always @(*)
  begin
    for(i=0;i<NUM_ENTRIES;i=i+1)
      valid_out[i] = data[i][0];
  end

endmodule
```

```
//------------------------------------------------------------------------------
// Module: mem_32_1rp_1wp
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This is a 32-entry memory with 1 write port and 1 read port.
//------------------------------------------------------------------------------
module mem_32_1rp_1wp(clk,reset_,waddr,wen,wdata,raddr,ren,rdata);
  parameter ADDR_WIDTH = 5;
  parameter NUM_ENTRIES = 32;
  parameter WIDTH = 72;

  input clk;
  input reset_;

  input [ADDR_WIDTH-1:0] waddr;
  input                  wen;
  input [WIDTH-1:0]      wdata;

  input [ADDR_WIDTH-1:0] raddr;
  input                  ren;
  output [WIDTH-1:0]     rdata;
  reg    [WIDTH-1:0]     rdata;

  reg [WIDTH-1:0] data [0:NUM_ENTRIES-1];

  integer i;

  // Synchronous writes
  always @(posedge clk)
  begin
    if(~reset_)
    begin
      for(i=0;i<NUM_ENTRIES;i=i+1)
        data[i] <= {WIDTH{1'b0}};
    end
    else
    begin
      if(wen)
      begin
        data[waddr] <= wdata;
      end
    end
  end

  // Asynchronous reads
  always @(*)
  begin
    if(ren)
      rdata = data[raddr];
    else
      rdata = {WIDTH{1'bz}};
  end

endmodule
```

```
//---------------------------------------------------------------------------
// Module: shift_reg
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module impliments a parallel-to-serial converter.  Parallel data is
//   input on the data_in port, and the module outputs the data SHIFT_WIDTH
//   bits at a time on shift_out.
//---------------------------------------------------------------------------
module shift_reg(clk,reset_,data_in,load_data,shift_out);
  parameter WIDTH = 32;
  parameter SHIFT_WIDTH = 1;

  input                    clk;
  input                    reset_;
  input [WIDTH-1:0]        data_in;
  input                    load_data;
  output [SHIFT_WIDTH-1:0] shift_out;

  reg [WIDTH-1:0] data;
  reg [WIDTH-1:0] tx_state;

  // Shift out from MSB side.
  assign shift_out = data[WIDTH-1:WIDTH-SHIFT_WIDTH];

  always @(posedge clk)
  begin
    if(!reset_)
    begin
      data <= {WIDTH{1'b0}};
      tx_state <= {WIDTH{1'b0}};
    end
    else
    begin
      if(load_data)
      begin
        tx_state <= 1;
        data <= data_in;

        // Assertion
        //synopsys translate_off
        if(tx_state)
          $display("ERROR: tx_state != 0 and load_data asserted");
        //synopsys translate_on
      end
      else
      begin
        tx_state <= tx_state << SHIFT_WIDTH;
        data <= data << SHIFT_WIDTH;
      end
    end
  end

endmodule
```

```
//----------------------------------------------------------------------------
// Module: output_block
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This is the output block for the 8x8 crossbar.  The block receives inputs
//   from each of the 8 input ports, a select signal from the scheduler, and a
//   schedule valid signal.  The block registers the select signal to mux
//   select the appropriate input port.  If then shifts in the receive data
//   from the input port over the next several clocks SHIFT_WIDTH bits per
//   clock.  Once it receives all of the packet, it sends the packet in two
//   half-phases to the output device.
//
//   The module also maintains a count of available downstream credit.  When it
//   runs out of credit, it sends a signal to the scheduler to omit it from the
//   next scheduling cycle.  The downstream device will send more credit when
//   it is able to receive another packet.
//----------------------------------------------------------------------------
module output_block(clk, reset_, input_sel, sel_v, data0_in, data1_in, data2_in,
data3_in, data4_in, data5_in, data6_in, data7_in, credit_in, credit_avail, data_out,
data_out_v);

  //--------------------------
  // Parameters
  //--------------------------
  parameter SHIFT_WIDTH = 1;
  parameter PACKET_WIDTH = 72;

  //--------------------------
  // Ports
  //--------------------------
  input       clk;
  input       reset_;

  input [7:0] input_sel; // Input selection
  input       sel_v;     // Input selection valid

  input [SHIFT_WIDTH-1:0] data0_in;   // Input data
  input [SHIFT_WIDTH-1:0] data1_in;   // Input data
  input [SHIFT_WIDTH-1:0] data2_in;   // Input data
  input [SHIFT_WIDTH-1:0] data3_in;   // Input data
  input [SHIFT_WIDTH-1:0] data4_in;   // Input data
  input [SHIFT_WIDTH-1:0] data5_in;   // Input data
  input [SHIFT_WIDTH-1:0] data6_in;   // Input data
  input [SHIFT_WIDTH-1:0] data7_in;   // Input data
  input                   credit_in;  // Credit from downstream device

  output                      credit_avail;
  output [PACKET_WIDTH/2-1:0] data_out;  // Output data
  output                      data_out_v;
  reg    [PACKET_WIDTH-1:0]   shift_data;

  reg   [SHIFT_WIDTH-1:0]   data_in; // Muxed data from the selected input
  reg   [5:0]               data_cnt;

  reg [5:0] credits;
  assign credit_avail = |credits;

  wire phase0_v;
  reg phase1_v;

  // Select the appropriate phase to send to the device on the output
  reg [PACKET_WIDTH/2-1:0] phase1_data;
  assign data_out = (phase1_v) ? phase1_data : shift_data[PACKET_WIDTH-1:PACKET_WIDTH/2];

  // State logic to receive the shift-in data and send the 2-phase packet
  // downstream
  reg [7:0] saved_sel;
  always @(posedge clk)
```

```
  begin
    if(!reset_)
    begin
      saved_sel <= 8'h00;
      credits <= 32;
      data_cnt <= 33;
    end
    else
    begin
      // If the select_valid is high, Register the mux select until it changes
      if(sel_v)
      begin
        saved_sel <= input_sel;
        data_cnt <= 0; // Start a new data count
      end
      else
      begin
        saved_sel <= saved_sel;
        if(data_cnt <= PACKET_WIDTH/SHIFT_WIDTH)
          data_cnt <= data_cnt + 1; // Increment the data count
      end

      // Shift in the new data
      shift_data <= {shift_data,data_in};

      //Manage credits.  Increment if we get a new credit, decrement if we get
      //a new send request, otherwise leave unchanged
      case({sel_v,credit_in}) // full_case parallel_case
        2'b10: credits <= credits-1;
        2'b01: credits <= credits+1;
        default: credits <= credits;
      endcase

      phase1_v <= phase0_v;

      if(phase0_v)
        phase1_data <= shift_data [PACKET_WIDTH/2-1:0];
    end
  end

  // The data is valid when we have received the correct number of incoming
  // packets shifts.
  assign phase0_v = (data_cnt == PACKET_WIDTH/SHIFT_WIDTH);

  assign data_out_v = phase0_v | phase1_v;

  // Mux the correct input to the shift register
  always @(*)
  begin
    case (saved_sel) // full_case parallel_case
      8'b00000001: data_in = data0_in;
      8'b00000010: data_in = data1_in;
      8'b00000100: data_in = data2_in;
      8'b00001000: data_in = data3_in;
      8'b00010000: data_in = data4_in;
      8'b00100000: data_in = data5_in;
      8'b01000000: data_in = data6_in;
      8'b10000000: data_in = data7_in;
      default: data_in = {SHIFT_WIDTH{1'b0}};
    endcase
  end

endmodule
```

10. Appendix B -- Design Layout

The layout was performed using the Silicon Ensemble Auto Place and Route tool. The design could not be imported back into the Cadence design tools, so a screen shot of the layout is shown below.

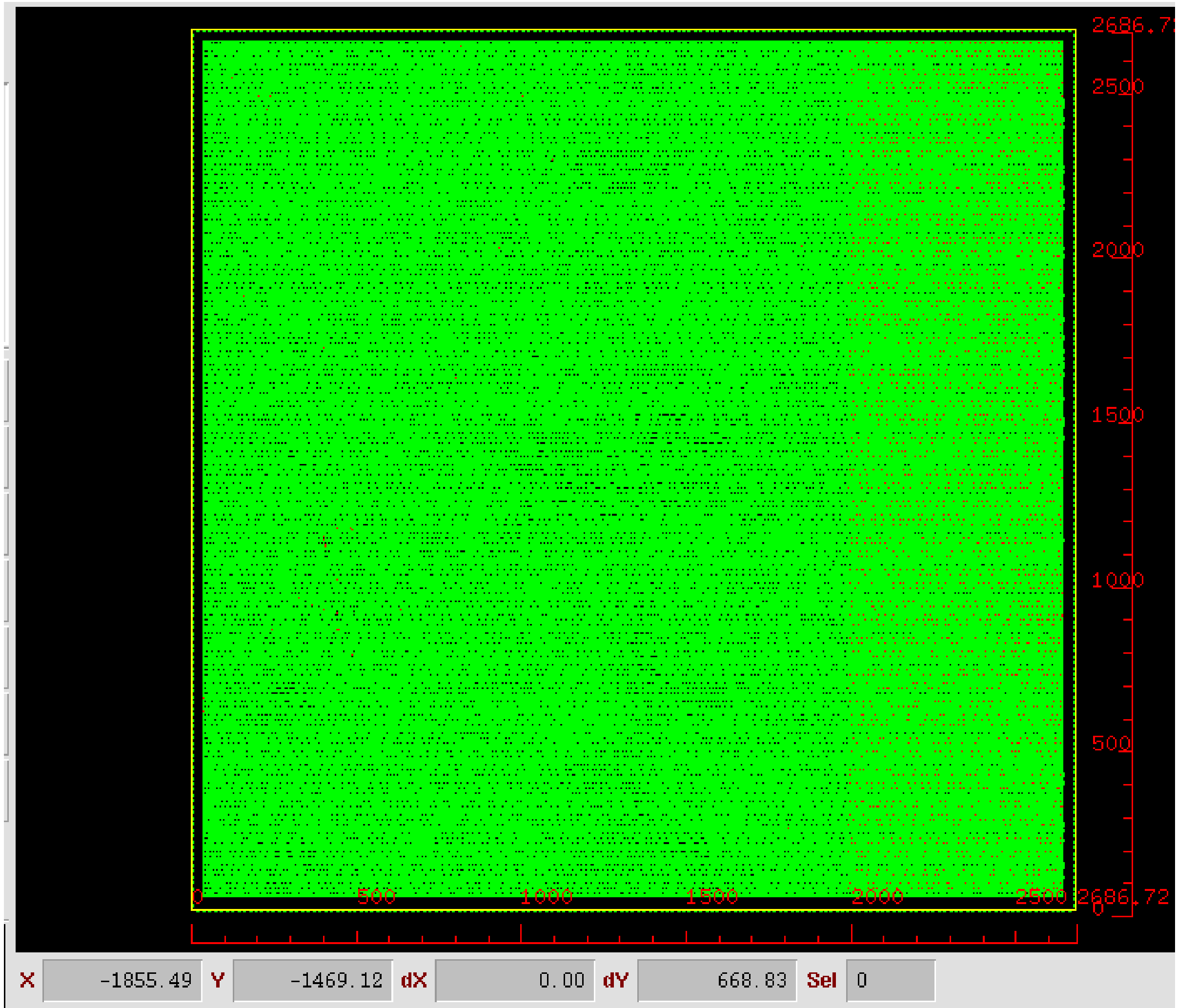


Figure 8 - 8x8 interconnect Layout

11. APPENDIX C – TESTBENCH SOURCE CODE

The following is a listing of the testbench source code for system and unit-level testing. The check_results.pl post-processing script is also provided.

```
//---------------------------------------------------------------------------
// Module: interconnect_tb
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module is the testbench for the top-level interconnect.  The
//   testbench is comprised of 8 input_block masters which drive random packet
//   requests, 8 output_block slaves which receive the destination data and
//   randomly return credits, and the interconnect.  Results are checked
//   post-simulation using a perl script that parses the output log.
//---------------------------------------------------------------------------
module interconnect_tb();
  //------------------------
  // Parameters
  //------------------------
  parameter PACKET_WIDTH = 72;
  parameter SHIFT_WIDTH = 9;

  reg                       clk;
  reg                       reset_;

  //------------------------
  // Wires
  //------------------------
  wire [PACKET_WIDTH/2-1:0] input0_packet;
  wire                      input0_packet_v;
  wire                      input0_credit;
  wire [PACKET_WIDTH/2-1:0] output0_packet;
  wire                      output0_packet_v;
  wire                      output0_credit;
  wire [PACKET_WIDTH/2-1:0] input1_packet;
  wire                      input1_packet_v;
  wire                      input1_credit;
  wire [PACKET_WIDTH/2-1:0] output1_packet;
  wire                      output1_packet_v;
  wire                      output1_credit;
  wire [PACKET_WIDTH/2-1:0] input2_packet;
  wire                      input2_packet_v;
  wire                      input2_credit;
  wire [PACKET_WIDTH/2-1:0] output2_packet;
  wire                      output2_packet_v;
  wire                      output2_credit;
  wire [PACKET_WIDTH/2-1:0] input3_packet;
  wire                      input3_packet_v;
  wire                      input3_credit;
  wire [PACKET_WIDTH/2-1:0] output3_packet;
  wire                      output3_packet_v;
  wire                      output3_credit;
  wire [PACKET_WIDTH/2-1:0] input4_packet;
  wire                      input4_packet_v;
  wire                      input4_credit;
  wire [PACKET_WIDTH/2-1:0] output4_packet;
  wire                      output4_packet_v;
  wire                      output4_credit;
  wire [PACKET_WIDTH/2-1:0] input5_packet;
  wire                      input5_packet_v;
  wire                      input5_credit;
  wire [PACKET_WIDTH/2-1:0] output5_packet;
  wire                      output5_packet_v;
  wire                      output5_credit;
  wire [PACKET_WIDTH/2-1:0] input6_packet;
  wire                      input6_packet_v;
  wire                      input6_credit;
  wire [PACKET_WIDTH/2-1:0] output6_packet;
  wire                      output6_packet_v;
  wire                      output6_credit;
  wire [PACKET_WIDTH/2-1:0] input7_packet;
  wire                      input7_packet_v;
  wire                      input7_credit;
```

```
wire [PACKET_WIDTH/2-1:0] output7_packet;
wire                      output7_packet_v;
wire                      output7_credit;

//------------------------
// Input block BFMs
//------------------------
input_block_master #(.PACKET_WIDTH(PACKET_WIDTH), .SRC_ID(0)) input0_master(
    .clk(clk),
    .reset_(reset_),
    .packet_out(input0_packet),
    .packet_v(input0_packet_v),
    .credit_in(input0_credit)
);
input_block_master #(.PACKET_WIDTH(PACKET_WIDTH), .SRC_ID(1)) input1_master(
    .clk(clk),
    .reset_(reset_),
    .packet_out(input1_packet),
    .packet_v(input1_packet_v),
    .credit_in(input1_credit)
);
input_block_master #(.PACKET_WIDTH(PACKET_WIDTH), .SRC_ID(2)) input2_master(
    .clk(clk),
    .reset_(reset_),
    .packet_out(input2_packet),
    .packet_v(input2_packet_v),
    .credit_in(input2_credit)
);
input_block_master #(.PACKET_WIDTH(PACKET_WIDTH), .SRC_ID(3)) input3_master(
    .clk(clk),
    .reset_(reset_),
    .packet_out(input3_packet),
    .packet_v(input3_packet_v),
    .credit_in(input3_credit)
);
input_block_master #(.PACKET_WIDTH(PACKET_WIDTH), .SRC_ID(4)) input4_master(
    .clk(clk),
    .reset_(reset_),
    .packet_out(input4_packet),
    .packet_v(input4_packet_v),
    .credit_in(input4_credit)
);
input_block_master #(.PACKET_WIDTH(PACKET_WIDTH), .SRC_ID(5)) input5_master(
    .clk(clk),
    .reset_(reset_),
    .packet_out(input5_packet),
    .packet_v(input5_packet_v),
    .credit_in(input5_credit)
);
input_block_master #(.PACKET_WIDTH(PACKET_WIDTH), .SRC_ID(6)) input6_master(
    .clk(clk),
    .reset_(reset_),
    .packet_out(input6_packet),
    .packet_v(input6_packet_v),
    .credit_in(input6_credit)
);
input_block_master #(.PACKET_WIDTH(PACKET_WIDTH), .SRC_ID(7)) input7_master(
    .clk(clk),
    .reset_(reset_),
    .packet_out(input7_packet),
    .packet_v(input7_packet_v),
    .credit_in(input7_credit)
);

//------------------------
// Output block BFMs
//------------------------
output_block_slave #(.PACKET_WIDTH(PACKET_WIDTH), .DEST_ID(0)) output0_slave(
    .clk(clk),
    .reset_(reset_),
    .packet_in(output0_packet),
```

```
        .packet_v(output0_packet_v),
        .credit(output0_credit)
    );
    output_block_slave #(.PACKET_WIDTH(PACKET_WIDTH), .DEST_ID(1)) output1_slave(
        .clk(clk),
        .reset_(reset_),
        .packet_in(output1_packet),
        .packet_v(output1_packet_v),
        .credit(output1_credit)
    );
    output_block_slave #(.PACKET_WIDTH(PACKET_WIDTH), .DEST_ID(2)) output2_slave(
        .clk(clk),
        .reset_(reset_),
        .packet_in(output2_packet),
        .packet_v(output2_packet_v),
        .credit(output2_credit)
    );
    output_block_slave #(.PACKET_WIDTH(PACKET_WIDTH), .DEST_ID(3)) output3_slave(
        .clk(clk),
        .reset_(reset_),
        .packet_in(output3_packet),
        .packet_v(output3_packet_v),
        .credit(output3_credit)
    );
    output_block_slave #(.PACKET_WIDTH(PACKET_WIDTH), .DEST_ID(4)) output4_slave(
        .clk(clk),
        .reset_(reset_),
        .packet_in(output4_packet),
        .packet_v(output4_packet_v),
        .credit(output4_credit)
    );
    output_block_slave #(.PACKET_WIDTH(PACKET_WIDTH), .DEST_ID(5)) output5_slave(
  .clk(clk),
        .reset_(reset_),
        .packet_in(output5_packet),
        .packet_v(output5_packet_v),
        .credit(output5_credit)
    );
    output_block_slave #(.PACKET_WIDTH(PACKET_WIDTH), .DEST_ID(6)) output6_slave(
        .clk(clk),
        .reset_(reset_),
        .packet_in(output6_packet),
        .packet_v(output6_packet_v),
        .credit(output6_credit)
    );
    output_block_slave #(.PACKET_WIDTH(PACKET_WIDTH), .DEST_ID(7)) output7_slave(
        .clk(clk),
        .reset_(reset_),
        .packet_in(output7_packet),
        .packet_v(output7_packet_v),
        .credit(output7_credit)
    );

    //------------------------
    // 8x8 crossbar
    //------------------------
    interconnect #(.PACKET_WIDTH(PACKET_WIDTH), .SHIFT_WIDTH(SHIFT_WIDTH)) interconnect(
        .clk              ( clk              ),
        .reset_           ( reset_           ),
        .input0_packet    ( input0_packet    ),
        .input0_packet_v  ( input0_packet_v  ),
        .input0_credit    ( input0_credit    ),
        .output0_packet   ( output0_packet   ),
        .output0_packet_v ( output0_packet_v ),
        .output0_credit   ( output0_credit   ),
        .input1_packet    ( input1_packet    ),
        .input1_packet_v  ( input1_packet_v  ),
        .input1_credit    ( input1_credit    ),
        .output1_packet   ( output1_packet   ),
        .output1_packet_v ( output1_packet_v ),
        .output1_credit   ( output1_credit   ),
```

```
      .input2_packet    ( input2_packet    ),
      .input2_packet_v  ( input2_packet_v  ),
      .input2_credit    ( input2_credit    ),
      .output2_packet   ( output2_packet   ),
      .output2_packet_v ( output2_packet_v ),
      .output2_credit   ( output2_credit   ),
      .input3_packet    ( input3_packet    ),
      .input3_packet_v  ( input3_packet_v  ),
      .input3_credit    ( input3_credit    ),
      .output3_packet   ( output3_packet   ),
      .output3_packet_v ( output3_packet_v ),
      .output3_credit   ( output3_credit   ),
      .input4_packet    ( input4_packet    ),
      .input4_packet_v  ( input4_packet_v  ),
      .input4_credit    ( input4_credit    ),
      .output4_packet   ( output4_packet   ),
      .output4_packet_v ( output4_packet_v ),
      .output4_credit   ( output4_credit   ),
      .input5_packet    ( input5_packet    ),
      .input5_packet_v  ( input5_packet_v  ),
      .input5_credit    ( input5_credit    ),
      .output5_packet   ( output5_packet   ),
      .output5_packet_v ( output5_packet_v ),
      .output5_credit   ( output5_credit   ),
      .input6_packet    ( input6_packet    ),
      .input6_packet_v  ( input6_packet_v  ),
      .input6_credit    ( input6_credit    ),
      .output6_packet   ( output6_packet   ),
      .output6_packet_v ( output6_packet_v ),
      .output6_credit   ( output6_credit   ),
      .input7_packet    ( input7_packet    ),
      .input7_packet_v  ( input7_packet_v  ),
      .input7_credit    ( input7_credit    ),
      .output7_packet   ( output7_packet   ),
      .output7_packet_v ( output7_packet_v ),
      .output7_credit   ( output7_credit   )
  );

  initial
  begin
    $fsdbDumpvars;
    $fsdbDumpon;

    clk = 0;
    forever
      #5 clk = ~clk;
  end

  initial
  begin
    reset_ = 0;
    @(posedge clk);
    @(posedge clk);
    reset_ = 1;

    repeat (100000) @(posedge clk);
    $finish;
  end

endmodule
```

```
//---------------------------------------------------------------------------------
// Module: input_block_master
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module implements a source BFM for testing the 8x8 crossbar.  It
//   randomly chooses destinations to send packets to and generates random data
//   for the packets.  When it sends the packets, it prints a message to the
//   log file for use in the post-processing perl scripts that check results.
//---------------------------------------------------------------------------------
module input_block_master(clk,reset_,packet_out,packet_v,credit_in);
  parameter SRC_ID = 0;
  parameter PACKET_WIDTH = 72;

  input         clk;
  input         reset_;
  output [PACKET_WIDTH/2-1:0] packet_out;
  output        packet_v;
  input         credit_in;

  integer credits;
  integer rand_weight;

  reg [PACKET_WIDTH/2-1:0] packet_out;
  reg        packet_v;
  reg        send_in_progress;

  always @(posedge clk)
  begin
    if(~reset_)
    begin
      credits = 32;
      packet_out <= 35'h0;
      packet_v <= 1'b0;
      send_in_progress = 0;
    end
    else
    begin
      packet_v <= 1'b0;

      if(credit_in)
        credits = credits + 1;

      if(credits > 0 && !send_in_progress)
      begin
        // The random component introduces delay so it doesn't submit a packet
        // on every single cycle
        rand_weight = $random % 100;
        if(rand_weight < 0)
          rand_weight = -rand_weight;

        if(rand_weight < 40 && rand_weight > 0)
        begin
          send_packet(SRC_ID,$random,{$random,$random,$random});
          credits = credits - 1;
        end
      end
    end
  end

  task send_packet;
    input [2:0] src;
    input [2:0] dest;
    input [65:0] data;
    reg [PACKET_WIDTH-1:0] packet_data;

    begin
      send_in_progress <= 1;
```

```
      $display("%t Input %0d submitting packet. src=%x dest=%x
data=%x",$time,SRC_ID,src,dest,data);
      packet_data = {src,dest,data};

      packet_out <= packet_data[PACKET_WIDTH-1:PACKET_WIDTH/2];
      packet_v <= 1'b1;
      @(posedge clk)
      packet_out <= packet_data[PACKET_WIDTH/2-1:0];
      packet_v <= 1'b1;
      send_in_progress <= 0;
    end
  endtask
endmodule
```

```
//-----------------------------------------------------------------------------
// Module: output_block_slave
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module models a destination device that would be connected on an
//   output port of the crossbar switch.  It receives 2-phase packets from the
//   crossbar and returns credits at random intervals.  When it receives a
//   packet, it prints the packet information to the log file for
//   post-processsing and checking.
//-----------------------------------------------------------------------------
module output_block_slave (clk,reset_,packet_in,packet_v,credit);
  parameter DEST_ID = 0;
  parameter PACKET_WIDTH = 72;

  input clk;
  input reset_;
  input [PACKET_WIDTH/2-1:0] packet_in;
  input packet_v;
  output credit;

  reg credit;

  integer pending_credit_cnt;

  reg packet_phase;
  reg [PACKET_WIDTH-1:0] packet_data;

  always @(posedge clk)
  begin
    if(~reset_)
    begin
      packet_phase <= 0;
      pending_credit_cnt <= 0;
    end
    else
    begin
      if(packet_v)
      begin
        packet_phase <= ~packet_phase;
        packet_data = {packet_data,packet_in};
        if(packet_phase)
        begin
          pending_credit_cnt = pending_credit_cnt + 1;
          $display("%t Output %0d read src=%x dest=%x
data=%x",$time,DEST_ID,packet_data[71:69],packet_data[68:66],packet_data[65:0]);
        end
      end

      if(pending_credit_cnt)
      begin
        credit <= 1'b1;
        pending_credit_cnt = pending_credit_cnt - 1;
      end
      else
      begin
        credit <= 1'b0;
      end
    end
  end
endmodule
```

```perl
##------------------------------------------------------------------------------
## Program: check_results.pl
## Author: John D. Pape
## Date: December 1, 2006
##
## Description:
##   This script post processes the simulation log of the interconnect
##   testbench to determine whether the interconnect operated correctly.  The
##   script contains the following checks.
##
##   Checks:
##   1.  Source ID matches input port
##   2.  Dest ID matches output port
##   3.  Dest received packet from source in same order packets were sent to
##       dest from the source.
##------------------------------------------------------------------------------
: # -*- perl -*-
eval 'exec perl -wS $0 ${1+"$@"}'
   if 0;

use Getopt::Long;

$Getopt::Long::ignorecase = 0;
%options = ();
GetOptions(
  \%options,
  "debug",
#   "h",
);

$debug         = 1 if defined $options{debug};

open(FILE,"sim.log");
while($line = <FILE>)
{
  if($line =~ /Input (\d+) submitting packet.*src=(\d+) dest=(\d+) data=(\w+)/)
  {
    if($1 ne $2)
    {
      print "ERROR: Src mismatch\n$line";
    }

    push(@{$data{$2}{$3}},$4);
  }
  elsif($line =~ /Output (\d+) read src=(\d+) dest=(\d+) data=(\w+)/)
  {
    if($1 ne $3)
    {
      print "ERROR: Dest mismatch\n$line";
    }

    if($check_data = shift(@{$data{$2}{$3}}))
    {
      if($check_data ne $4)
      {
        print "ERROR: Data mismatch. src=$2 dest=$3 data=$4 exp_data=$check_data\n";
      }
    }
    else
    {
      print "ERROR: Data for src=$2 and dest=$3 does not exist\n";
    }
  }
}

# ****************************************************************************
# $Log: check_results.pl,v $
# Revision 1.1  2006/12/09 23:35:01  jpape
# Woo Hoo
#
```

```
# Revision 1.1  2006/12/09 05:16:02  jpape
# Added debussy waves dumping and result-checking script
#
#
```

```verilog
//---------------------------------------------------------------------------
// Module: input_block_tb
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This is a standalone testbench for testing the input_block module.  It has
//   bfms for modeling source devices and output_blocks.  The BFM generate
//   random behavior, and the log file is post processed to ensure that the
//   packets were sent in the correct order.
//---------------------------------------------------------------------------
module input_block_tb();
  parameter PACKET_WIDTH = 72;
  parameter SHIFT_WIDTH = 9;

  reg                        clk;
  reg                        reset_;
  wire [PACKET_WIDTH/2-1:0] packet_in;
  wire                       packet_v;
  wire                       credit;
  wire [7:0]                 tx_dest;
  wire                       tx_dest_v;
  wire [SHIFT_WIDTH-1:0]     tx_out;
  wire [7:0]                 request_out;

  input_block #(.SHIFT_WIDTH(SHIFT_WIDTH),.PACKET_WIDTH(PACKET_WIDTH)) input_block(
      .clk         ( clk         ) ,
      .reset_      ( reset_      ) ,
      .packet_in   ( packet_in   ) ,
      .packet_v    ( packet_v    ) ,
      .credit      ( credit      ) ,
      .tx_dest     ( tx_dest     ) ,
      .tx_dest_v   ( tx_dest_v   ) ,
      .tx_out      ( tx_out      ) ,
      .request_out ( request_out )
  );

  input_block_master #(.PACKET_WIDTH(PACKET_WIDTH)) master(
    .clk          ( clk        ),
    .reset_       ( reset_     ),
    .packet_out   ( packet_in  ),
    .packet_v     ( packet_v   ),
    .credit_in    ( credit     )
  );

  input_block_slave #(.SHIFT_WIDTH(SHIFT_WIDTH),.PACKET_WIDTH(PACKET_WIDTH)) slave(
    .clk            ( clk         ),
    .reset_         ( reset_      ),
    .shift_in       ( tx_out      ),
    .tx_dest        ( tx_dest     ),
    .tx_dest_v      ( tx_dest_v   ),
    .request        ( request_out )
  );

  initial
  begin
    $fsdbDumpvars;
    $fsdbDumpon;

    clk = 0;
    forever
      #5 clk = ~clk;
  end

  integer i;

  initial
  begin
    reset_ = 1'b0;
    @(posedge clk);
```

```
    @(posedge clk);
    reset_ = 1'b1;

    repeat (100000) @(posedge clk);
    $finish;
  end

endmodule
```

```
//---------------------------------------------------------------------------
// Module: input_block_slave
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module models an output_block for testing the input_block in a
//   standalone environment.  The module receives shift data from the input and
//   prints it to a log file when the shifted data is complete.
//---------------------------------------------------------------------------
module input_block_slave (clk, reset_, shift_in, tx_dest, tx_dest_v, request);
  parameter SHIFT_WIDTH = 1;
  parameter PACKET_WIDTH = 72;
  input                    clk;
  input                    reset_;
  input  [SHIFT_WIDTH-1:0] shift_in;
  output [7:0]             tx_dest;
  output                   tx_dest_v;
  input  [7:0]             request;

  reg [7:0] tx_dest;
  reg       tx_dest_v;

  reg [PACKET_WIDTH-1:0] rx_data;

  reg [2:0] random_dest;
  reg       matched;
  reg       rx_in_progress;

  always @(posedge clk)
  begin
    if(~reset_)
    begin
      tx_dest = 8'h0;
      tx_dest_v = 1'b0;
      rx_in_progress = 0;
    end
    // Pick a new request and start a new transfer
    else if(!rx_in_progress && request != 8'h0)
    begin
      random_dest = 3'h0;
      matched = 0;
      while(!matched)
      begin
        random_dest = $random;
        if(request & (1'b1 << random_dest))
        begin
          //$display("Matched on request=%b random_dest=%d",request,random_dest);
          matched = 1;
        end
      end
      read_packet(random_dest);
    end
  end

  task read_packet;
    input [2:0] dest;

    begin
      rx_in_progress = 1;
      $display("%t reading packet. dest=%x",$time,dest);
      tx_dest = 8'h1 << dest;
      tx_dest_v = 1'b1;
      @(posedge clk);
      tx_dest_v = 1'b0;
      // Wait for packet to finish transmitting
      repeat(PACKET_WIDTH/SHIFT_WIDTH)
      begin
        @(posedge clk);
        rx_data = {rx_data,shift_in};
```

```
      end
      $display("%t read src=%x dest=%x
data=%x",$time,rx_data[71:69],rx_data[68:66],rx_data[65:0]);
      rx_in_progress = 0;
    end
  endtask

endmodule
```

```
//---------------------------------------------------------------------------
// Module: scheduler_testbench
// Author: John D. Pape
// Date: December 1, 2006
//
// Description:
//   This module impliments a simple testbench for the scheduler module. It
//   applies test vectors to the input[0-7] request ports and produces waves
//   for debugging.
//---------------------------------------------------------------------------
module scheduler_testbench;
  reg clk;
  reg reset_;

  reg  [7:0] input0_req;
  reg  [7:0] input1_req;
  reg  [7:0] input2_req;
  reg  [7:0] input3_req;
  reg  [7:0] input4_req;
  reg  [7:0] input5_req;
  reg  [7:0] input6_req;
  reg  [7:0] input7_req;

  reg  [7:0] output_available;

  wire [7:0] input0_decision;
  wire [7:0] input1_decision;
  wire [7:0] input2_decision;
  wire [7:0] input3_decision;
  wire [7:0] input4_decision;
  wire [7:0] input5_decision;
  wire [7:0] input6_decision;
  wire [7:0] input7_decision;
  wire [7:0] input_decision_v;

  wire [7:0] output0_decision;
  wire [7:0] output1_decision;
  wire [7:0] output2_decision;
  wire [7:0] output3_decision;
  wire [7:0] output4_decision;
  wire [7:0] output5_decision;
  wire [7:0] output6_decision;
  wire [7:0] output7_decision;
  wire [7:0] output_decision_v;

  reg sched_start;
  wire sched_done;

  scheduler scheduler(
    .clk (clk),
    .reset_ (reset_),
    .start (sched_start),
    .done  (sched_done),
    .output_available (output_available),
    .input0_req (input0_req),
    .input1_req (input1_req),
    .input2_req (input2_req),
    .input3_req (input3_req),
    .input4_req (input4_req),
    .input5_req (input5_req),
    .input6_req (input6_req),
    .input7_req (input7_req),
    .input0_decision (input0_decision),
    .input1_decision (input1_decision),
    .input2_decision (input2_decision),
    .input3_decision (input3_decision),
    .input4_decision (input4_decision),
    .input5_decision (input5_decision),
    .input6_decision (input6_decision),
    .input7_decision (input7_decision),
```

```
    .input_decision_v (input_decision_v),
    .output0_decision (output0_decision),
    .output1_decision (output1_decision),
    .output2_decision (output2_decision),
    .output3_decision (output3_decision),
    .output4_decision (output4_decision),
    .output5_decision (output5_decision),
    .output6_decision (output6_decision),
    .output7_decision (output7_decision),
    .output_decision_v (output_decision_v)
  );

  initial
  begin
    $dumpvars;
    $dumpon;

    clk = 0;
    forever
       #5 clk = ~clk;
  end

  initial
  begin
    reset_ = 0;
    @(posedge clk);
    @(posedge clk);
    reset_ = 1;
    output_available = 8'b11111111;
    input0_req = 8'b00000001;
    input1_req = 8'b00000010;
    input2_req = 8'b00000100;
    input3_req = 8'b00001000;
    input4_req = 8'b00010000;
    input5_req = 8'b00100000;
    input6_req = 8'b01000000;
    input7_req = 8'b10000000;
    sched_start = 1'b1;
    @(posedge clk);
    sched_start = 1'b0;
    @(posedge sched_done);
    @(posedge clk);
    repeat(10)
    begin
      input0_req = 8'b11111111;
      input1_req = 8'b11111111;
      input2_req = 8'b11111111;
      input3_req = 8'b11111111;
      input4_req = 8'b11111111;
      input5_req = 8'b11111111;
      input6_req = 8'b11111111;
      input7_req = 8'b11111111;
      sched_start = 1'b1;
      @(posedge clk);
      sched_start = 1'b1;
      @(posedge sched_done);
      @(posedge clk);
    end

    repeat(10)
    begin
      input0_req = 8'b00000001;
      input1_req = 8'b00000001;
      input2_req = 8'b00000001;
      input3_req = 8'b00000001;
      input4_req = 8'b00000001;
      input5_req = 8'b00000001;
      input6_req = 8'b00000001;
      input7_req = 8'b00000001;
      sched_start = 1'b1;
      @(posedge clk);
```

```
      sched_start = 1'b1;
      @(posedge sched_done);
      @(posedge clk);
    end

    input0_req = 8'b00000001;
    input1_req = 8'b00000001;
    input2_req = 8'b00000001;
    input3_req = 8'b00000001;
    input4_req = 8'b10000000;
    input5_req = 8'b10000000;
    input6_req = 8'b10000000;
    input7_req = 8'b10000000;
    sched_start = 1'b1;
    @(posedge clk);
    sched_start = 1'b1;
    @(posedge sched_done);
    @(posedge clk);
    $finish;
  end
endmodule
```

12. Appendix D – Synthesis Reports

The following synthesis reports are provided:

Interconnect timing
Interconnect area
Interconnect power – no leakage characterization was provided by the library.

Input_block area
Output_block area
Scheduler area
Scheduler timing

Interconnect timing_report.txt

```
Information: Updating design information... (UID-85)
Warning: Design 'interconnect' contains 1 high-fanout nets. A fanout number of 1000 will
be used for delay calculations involving these nets. (TIM-134)

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
        -sort_by group
Design : interconnect
Version: V-2004.06-SP2
Date   : Sun Dec 10 01:26:42 2006
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: WCCOM   Library: HT018
Wire Load Model Mode: top

  Startpoint: input7_block/freelist/tailptr_reg[0]
              (rising edge-triggered flip-flop clocked by vclk)
  Endpoint: input7_block/packet_array/data_reg[9][67]
            (rising edge-triggered flip-flop clocked by vclk)
  Path Group: vclk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  interconnect       C10_A50K              HT018

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock vclk (rise edge)                                  0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  input7_block/freelist/tailptr_reg[0]/CLK (dff)          0.00 #     0.00 r
  input7_block/freelist/tailptr_reg[0]/Q (dff)          138.01     138.01 f
  input7_block/freelist/C2674/S0 (freelist_fifo_MUX_OP_32_5_5_0)
                                                          0.00     138.01 f
  input7_block/freelist/C2674/U171/Z (inv4)              32.20     170.21 r
  input7_block/freelist/C2674/U172/Z (inv4)              95.53     265.74 f
  input7_block/freelist/C2674/U155/Z (mux2)             124.29     390.03 r
  input7_block/freelist/C2674/U153/Z (mux2)              91.36     481.39 r
  input7_block/freelist/C2674/U149/Z (mux2)              90.52     571.92 r
  input7_block/freelist/C2674/U141/Z (mux2)              90.93     662.85 r
  input7_block/freelist/C2674/U125/Z (mux2)              87.32     750.17 r
  input7_block/freelist/C2674/Z_4 (freelist_fifo_MUX_OP_32_5_5_0)
                                                          0.00     750.17 r
  input7_block/freelist/U1461/Z (inv4)                   44.68     794.85 f
  input7_block/freelist/rdata_tri[0]/Z (tristate_inv1)
                                                        102.99     897.84 r
  input7_block/freelist/rdata[0] (freelist_fifo_0)        0.00     897.84 r
  input7_block/packet_array/waddr[0] (mem_32_1rp_1wp_WIDTH72_0)
                                                          0.00     897.84 r
  input7_block/packet_array/U7181/Z (inv1)              114.76    1012.60 f
  input7_block/packet_array/U15796/Z (nor3x1)            59.05    1071.65 r
  input7_block/packet_array/U9978/Z (and2x1)             64.33    1135.98 r
  input7_block/packet_array/U13380/Z (or2x2)            105.48    1241.46 r
  input7_block/packet_array/U15739/Z (buf4)              88.28    1329.75 r
  input7_block/packet_array/U8543/Z (and2x2)             57.59    1387.34 r
  input7_block/packet_array/U15761/Z (buf4)             177.75    1565.09 r
  input7_block/packet_array/U10726/Z (and2x1)            67.96    1633.06 r
  input7_block/packet_array/U10727/Z (or2x1)            107.79    1740.85 r
  input7_block/packet_array/data_reg[9][67]/D (dff)       0.00    1740.85 r
  data arrival time                                               1740.85

  clock vclk (rise edge)                               1800.00    1800.00
  clock network delay (ideal)                             0.00    1800.00
  input7_block/packet_array/data_reg[9][67]/CLK (dff)     0.00    1800.00 r
```

```
library setup time                                          -59.10    1740.90
data required time                                                    1740.90
---------------------------------------------------------------------------
data required time                                                    1740.90
data arrival time                                                    -1740.85
---------------------------------------------------------------------------
slack (MET)                                                              0.05
```

interconnect area_report.txt

```
****************************************
Report : area
Design : interconnect
Version: V-2004.06-SP2
Date   : Sun Dec 10 01:28:01 2006
****************************************

Library(s) Used:

    HT018 (File: /home/ecelrc/students/turo/public_html/vlsi1/lab3/synopsys/HT018.db)

Number of ports:              610
Number of nets:               902
Number of cells:               20
Number of references:          20

Combinational area:       2859715.750000
Noncombinational area:    1915617.375000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          4774748.000000
Total area:                 undefined
```

Interconnect power_report.txt

```
****************************************
Report : power
        -analysis_effort low
Design : interconnect
Version: V-2004.06-SP2
Date   : Sun Dec 10 01:33:00 2006
****************************************


Library(s) Used:

    HT018 (File: /home/ecelrc/students/turo/public_html/vlsi1/lab3/synopsys/HT018.db)

Warning: The library cells used by your design are not characterized for internal power.
(PWR-26)

Operating Conditions: WCCOM   Library: HT018
Wire Load Model Mode: top

Design        Wire Load Model            Library
------------------------------------------------
interconnect           C10_A50K          HT018


Global Operating Voltage = 2.35
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ps
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = Unitless


  Cell Internal Power  =   0.0000 mW    (0%)
  Net Switching Power  = 116.1466 mW  (100%)
                         ---------
Total Dynamic Power    = 116.1466 mW  (100%)

Cell Leakage Power     =   0.0000
```

input_block_area_report.txt

```
Information: Updating design information... (UID-85)
Warning: Design 'input_block_SHIFT_WIDTH9_0' contains 1 high-fanout nets. A fanout number
of 1000 will be used for delay calculations involving these nets. (TIM-134)

****************************************
Report : area
Design : input_block_SHIFT_WIDTH9_0
Version: V-2004.06-SP2
Date   : Sun Dec 10 18:42:43 2006
****************************************

Library(s) Used:

    HT018 (File: /home/ecelrc/students/turo/public_html/vlsi1/lab3/synopsys/HT018.db)

Number of ports:               66
Number of nets:              1087
Number of cells:              933
Number of references:          29

Combinational area:       333486.906250
Noncombinational area:    228335.984375
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:          561836.562500
Total area:                 undefined
```

output_block_timing_report.txt

```
Information: Updating design information... (UID-85)

****************************************
Report : area
Design : output_block_SHIFT_WIDTH9_0
Version: V-2004.06-SP2
Date   : Sun Dec 10 18:44:21 2006
****************************************

Library(s) Used:

    HT018 (File: /home/ecelrc/students/turo/public_html/vlsi1/lab3/synopsys/HT018.db)

Number of ports:              122
Number of nets:               956
Number of cells:              857
Number of references:          19

Combinational area:       15170.675781
Noncombinational area:    9984.601562
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:          25155.160156
Total area:                 undefined
```

scheduler_area_report.txt

```
Information: Updating design information... (UID-85)

****************************************
Report : area
Design : scheduler
Version: V-2004.06-SP2
Date   : Sun Dec 10 18:43:31 2006
****************************************

Library(s) Used:

    HT018 (File: /home/ecelrc/students/turo/public_html/vlsi1/lab3/synopsys/HT018.db)

Number of ports:              220
Number of nets:              1276
Number of cells:              829
Number of references:          47

Combinational area:       66436.484375
Noncombinational area:    9442.798828
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:          75879.921875
Total area:                 undefined
```

scheduler_timing_report.txt

```
Information: Updating design information... (UID-85)

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
        -sort_by group
Design : scheduler
Version: V-2004.06-SP2
Date   : Sun Dec 10 18:45:38 2006
****************************************

Operating Conditions: WCCOM   Library: HT018
Wire Load Model Mode: top

  Startpoint: done_reg (rising edge-triggered flip-flop)
  Endpoint: output_decision_v[7]
            (output port)
  Path Group: (none)
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  done_reg/CLK (dff)                       0.00       0.00 r
  done_reg/Q (dff)                        91.81      91.81 f
  U1000/Z (inv4)                          21.61     113.42 r
  U687/Z (buf1)                           59.13     172.55 r
  U564/Z (inv1)                          293.53     466.09 f
  U702/Z (and2x2)                         35.95     502.04 f
  output_decision_v[7] (out)               0.00     502.04 f
  data arrival time                                 502.04
  -----------------------------------------------------------
  (Path is unconstrained)
```

13. APPENDIX E – SIMULATION RESULTS

Below is an excerpt from the final simulation log of the interconnect and the start of its corresponding waveform:

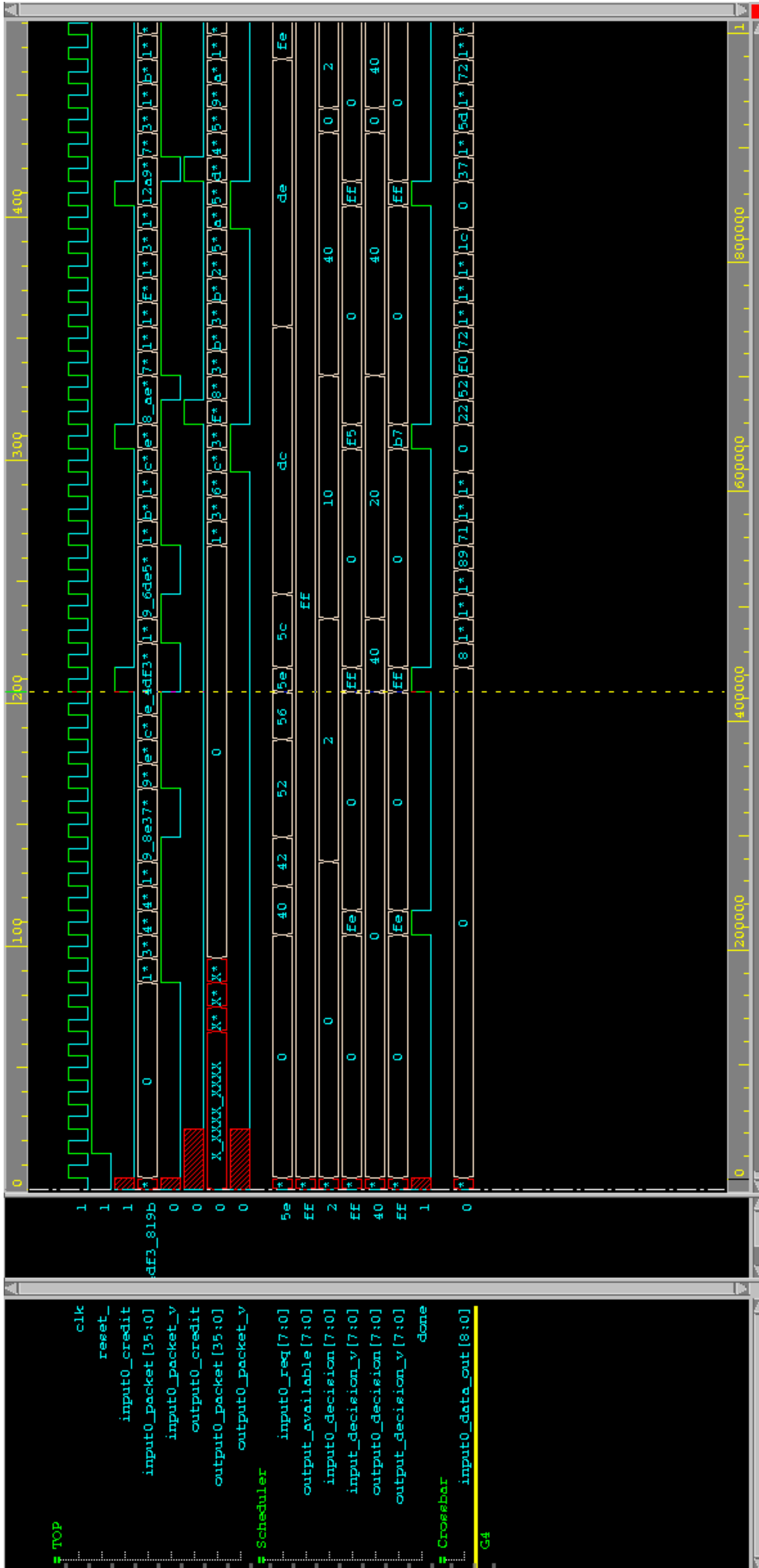
sim.log:

```
Command: ./simv -l sim.log
Chronologic VCS simulator copyright 1991-2005
Contains Synopsys proprietary information.
Compiler version X-2005.06-SP1-15; Runtime version X-2005.06-SP1-15;  Dec  9 22:58 2006

Novas FSDB Dumper for VCS2005.06-DKI, Release 6.0v4 (Linux) 10/25/2005
Copyright (C) 1996 - 2005 by Novas Software, Inc.
*Novas* Create FSDB file 'verilog.fsdb'
*Novas* Enable the struct and array variables dumping.
*Novas* Begin dumping the top modules, layer(0).
*Novas* End dumping the top modules.
                  25 Input 5 submitting packet. src=5 dest=3 data=146df998db2c28465
                  25 Input 4 submitting packet. src=4 dest=1 data=13b23f1761e8dcd3d
                  25 Input 3 submitting packet. src=3 dest=4 data=1e33724c6e2f784c5
                  35 Input 7 submitting packet. src=7 dest=7 data=2e77696cef4007ae8
                  35 Input 1 submitting packet. src=1 dest=5 data=1b1ef62630573870a
                  45 Input 6 submitting packet. src=6 dest=2 data=1cb203e968983b813
                  45 Input 5 submitting packet. src=5 dest=3 data=3eaa62ad581174a02
                  45 Input 4 submitting packet. src=4 dest=5 data=3118449230509650a
                  45 Input 3 submitting packet. src=3 dest=4 data=2452e618a20c4b341
                  55 Input 2 submitting packet. src=2 dest=6 data=2571513aede7502bc
                  65 Input 4 submitting packet. src=4 dest=2 data=20aaa4b1578d99bf1
                  65 Input 1 submitting packet. src=1 dest=4 data=347b9a18f7c6da9f8
                  75 Input 6 submitting packet. src=6 dest=3 data=1a4ae3249e8233ed0
                  75 Input 5 submitting packet. src=5 dest=1 data=2061d7f0ce12ccec2
                  75 Input 3 submitting packet. src=3 dest=7 data=1090cdb12bf05007e
                  85 Input 7 submitting packet. src=7 dest=3 data=1bc1488782dda595b
                  85 Input 2 submitting packet. src=2 dest=2 data=0c33f3886c71a0c8e
                  85 Input 0 submitting packet. src=0 dest=6 data=3d18bb4a39799a82f
                  95 Input 6 submitting packet. src=6 dest=7 data=07bf8fdf7e59b36cb
                  95 Input 4 submitting packet. src=4 dest=1 data=1ed536cdab29fb665
                  95 Input 3 submitting packet. src=3 dest=7 data=12231ff44e8740cd0
                  95 Input 1 submitting packet. src=1 dest=3 data=26e5daddccd5ebc9a
                 105 Input 5 submitting packet. src=5 dest=6 data=2b3d976678531340a
                 105 Input 0 submitting packet. src=0 dest=1 data=04a74bf9449c65d93
                 115 Input 2 submitting packet. src=2 dest=5 data=2653b49ca5b172db6
                 125 Input 7 submitting packet. src=7 dest=4 data=334980769da6ebab4
                 125 Input 0 submitting packet. src=0 dest=4 data=1149e07298e37901c
                 135 Input 4 submitting packet. src=4 dest=6 data=0b9f504735d7199ba
                 135 Input 1 submitting packet. src=1 dest=2 data=11b8761374b273796
                 145 Input 7 submitting packet. src=7 dest=6 data=23e99837d6e5f0fdc
                 155 Input 6 submitting packet. src=6 dest=2 data=10b940917d0f578a1
                 155 Input 5 submitting packet. src=5 dest=0 data=19ab48835949a8a29
                 165 Input 7 submitting packet. src=7 dest=2 data=2d44b80a8549efda9
                 165 Input 0 submitting packet. src=0 dest=2 data=14f75ff9e9c6de638
                 175 Input 6 submitting packet. src=6 dest=0 data=209ff411335a0c96b
                 175 Input 5 submitting packet. src=5 dest=6 data=26216abc45c8295b9
                 175 Input 4 submitting packet. src=4 dest=4 data=3c33390867d6df5fa
                 185 Input 7 submitting packet. src=7 dest=4 data=254a879a9d095a8a1
                 185 Input 1 submitting packet. src=1 dest=7 data=11b60e536bab14875
                 185 Input 0 submitting packet. src=0 dest=3 data=0d73fb4ae4df3819b
                 195 Input 6 submitting packet. src=6 dest=0 data=125b75f4be169b0c2
                 195 Input 4 submitting packet. src=4 dest=4 data=068ae1bd1c3761c86
                 205 Input 7 submitting packet. src=7 dest=3 data=2adac225bf166fae2
                 205 Input 5 submitting packet. src=5 dest=3 data=0093e4d12dc0344b8
                 215 Output 7 read src=7 dest=7 data=2e77696cef4007ae8
                 215 Output 6 read src=2 dest=6 data=2571513aede7502bc
                 215 Output 5 read src=1 dest=5 data=1b1ef62630573870a
                 215 Output 4 read src=3 dest=4 data=1e33724c6e2f784c5
                 215 Output 3 read src=5 dest=3 data=146df998db2c28465
```

```
215 Output 2 read src=6 dest=2 data=1cb203e968983b813
215 Output 1 read src=4 dest=1 data=13b23f1761e8dcd3d
215 Input 4 submitting packet. src=4 dest=7 data=18f8c6e1f82223a04
215 Input 2 submitting packet. src=2 dest=4 data=2cb227096d8ace2b1
225 Input 0 submitting packet. src=0 dest=7 data=1a4da56496de5bbdb
235 Input 7 submitting packet. src=7 dest=1 data=11546dd2a7d2a45fa
235 Input 4 submitting packet. src=4 dest=2 data=2b455f268b7dfaa6f
235 Input 1 submitting packet. src=1 dest=0 data=07b0da9f6e2bf1ac5
245 Input 6 submitting packet. src=6 dest=4 data=1d86a6ab01e1c873c
255 Input 7 submitting packet. src=7 dest=3 data=1c33604861297cb25
265 Input 6 submitting packet. src=6 dest=3 data=2fbdfc2f7cf14ce9e
265 Input 4 submitting packet. src=4 dest=0 data=039600972da3d8cb4
265 Input 0 submitting packet. src=0 dest=6 data=1bde0d27b47e2738f
275 Input 1 submitting packet. src=1 dest=5 data=0f6a178ed297a1552
285 Input 7 submitting packet. src=7 dest=5 data=24219e784236afd46
285 Input 6 submitting packet. src=6 dest=0 data=3352d616a427b5784
285 Input 5 submitting packet. src=5 dest=4 data=05d4a4dbac5a1608b
285 Input 3 submitting packet. src=3 dest=2 data=2a48f7c498a64b014
285 Input 2 submitting packet. src=2 dest=3 data=023907547433e9786
285 Input 0 submitting packet. src=0 dest=4 data=09622502c467c458c
305 Input 5 submitting packet. src=5 dest=3 data=011fe0523520eefa4
305 Input 3 submitting packet. src=3 dest=7 data=0338365674ea0419d
305 Input 2 submitting packet. src=2 dest=2 data=1ecb91ad91000b720
305 Input 0 submitting packet. src=0 dest=3 data=2e471f8c8aed72e5d
315 Output 7 read src=3 dest=7 data=1090cdb12bf05007e
315 Output 6 read src=5 dest=6 data=2b3d976678531340a
315 Output 5 read src=4 dest=5 data=3118449230509650a
315 Output 4 read src=7 dest=4 data=334980769da6ebab4
315 Output 3 read src=1 dest=3 data=26e5daddccd5ebc9a
315 Output 2 read src=2 dest=2 data=0c33f3886c71a0c8e
315 Output 1 read src=0 dest=1 data=04a74bf9449c65d93
315 Output 0 read src=6 dest=0 data=209ff411335a0c96b
315 Input 6 submitting packet. src=6 dest=3 data=1897f1c12a97f0052
315 Input 4 submitting packet. src=4 dest=0 data=24bf529976d8b87db
315 Input 1 submitting packet. src=1 dest=3 data=087e44c0fb4e8d669
335 Input 7 submitting packet. src=7 dest=1 data=32523654aec3758d8
335 Input 1 submitting packet. src=1 dest=7 data=16a15f5d403878707
335 Input 0 submitting packet. src=0 dest=1 data=300f25f016d808bdb
345 Input 3 submitting packet. src=3 dest=7 data=257fbb9aff4d86ee9
355 Input 7 submitting packet. src=7 dest=7 data=1cab47c95f7723eee
355 Input 5 submitting packet. src=5 dest=0 data=014b43729f0eeaee1
355 Input 4 submitting packet. src=4 dest=6 data=16a9fb9d53437d568
355 Input 2 submitting packet. src=2 dest=2 data=3f8dc48f1be9bbc7d
355 Input 0 submitting packet. src=0 dest=4 data=177ebb1efade7d05b
365 Input 3 submitting packet. src=3 dest=1 data=309c8351332dc4165
365 Input 1 submitting packet. src=1 dest=7 data=19d12083ab8ea3a71
375 Input 7 submitting packet. src=7 dest=4 data=2beda447d2cee5f59
375 Input 6 submitting packet. src=6 dest=5 data=1a0aecc4157c1d1af
375 Input 5 submitting packet. src=5 dest=5 data=08326d406d14820a2
375 Input 2 submitting packet. src=2 dest=4 data=0bd86f47b929d5825
375 Input 0 submitting packet. src=0 dest=6 data=011cc9b233cb3ab79
385 Input 3 submitting packet. src=3 dest=6 data=30671030ce70f98ce
395 Input 7 submitting packet. src=7 dest=1 data=37a4fbff4baf4e275
395 Input 5 submitting packet. src=5 dest=2 data=102fbf905271c434e
395 Input 1 submitting packet. src=1 dest=5 data=3747331e848590990
395 Input 0 submitting packet. src=0 dest=4 data=3a005a64012a90325
415 Output 7 read src=6 dest=7 data=07bf8fdf7e59b36cb
415 Output 5 read src=2 dest=5 data=2653b49ca5b172db6
415 Output 4 read src=0 dest=4 data=1149e07298e37901c
415 Output 2 read src=7 dest=2 data=2d44b80a8549efda9
415 Output 1 read src=4 dest=1 data=1ed536cdab29fb665
415 Output 0 read src=5 dest=0 data=19ab48835949a8a29
415 Input 3 submitting packet. src=3 dest=5 data=17a87aff5aeeacc5d
415 Input 2 submitting packet. src=2 dest=2 data=0dc4308b8cf309c9e
415 Input 1 submitting packet. src=1 dest=6 data=3c9cbbc93aada7455
425 Input 5 submitting packet. src=5 dest=6 data=08f63e41ec838f490
425 Input 0 submitting packet. src=0 dest=1 data=36826d9d037b9656f
435 Input 4 submitting packet. src=4 dest=3 data=06e1e1fdcd27f0aa4
445 Input 7 submitting packet. src=7 dest=0 data=02e36435c033a4506
445 Input 6 submitting packet. src=6 dest=0 data=15515d1aa0d12031a
445 Input 3 submitting packet. src=3 dest=4 data=161dafdc375ad73eb
```

```
445 Input 2 submitting packet. src=2 dest=2 data=1378c736f0de14b1b
445 Input 0 submitting packet. src=0 dest=5 data=355dd8dab8d94d21b
465 Input 6 submitting packet. src=6 dest=6 data=2bce32879f24baee4
465 Input 4 submitting packet. src=4 dest=0 data=24665378cb8c0c271
465 Input 3 submitting packet. src=3 dest=4 data=00bec5717e0e004c1
465 Input 0 submitting packet. src=0 dest=7 data=3d8462ab0e9b49ad3
475 Input 5 submitting packet. src=5 dest=2 data=1eb8804d787628e0e
475 Input 2 submitting packet. src=2 dest=5 data=10e3aeb1c4c18c798
485 Input 4 submitting packet. src=4 dest=3 data=2feaddcfd9f7a0e3e
485 Input 1 submitting packet. src=1 dest=0 data=209164d12b46afc68
495 Input 7 submitting packet. src=7 dest=6 data=37dddabfbb58d7c6b
495 Input 6 submitting packet. src=6 dest=7 data=01500052ad3666ea6
505 Input 3 submitting packet. src=3 dest=5 data=1e4df16c9b05f8e60
505 Input 2 submitting packet. src=2 dest=4 data=32c0c555839d65773
515 Output 7 read src=4 dest=7 data=18f8c6e1f82223a04
515 Output 6 read src=0 dest=6 data=3d18bb4a39799a82f
515 Output 5 read src=7 dest=5 data=24219e784236afd46
515 Output 4 read src=1 dest=4 data=347b9a18f7c6da9f8
515 Output 3 read src=2 dest=3 data=023907547433e9786
515 Output 2 read src=3 dest=2 data=2a48f7c498a64b014
515 Output 1 read src=5 dest=1 data=2061d7f0ce12ccec2
515 Output 0 read src=6 dest=0 data=125b75f4be169b0c2
```


TOP
clk
reset_
input0_credit
input0_packet[35:0]
input0_packet_v
output0_credit
output0_packet[35:0]
output0_packet_v
Scheduler
input0_req[7:0]
output_available[7:0]
input0_decision[7:0]
input_decision_v[7:0]
output0_decision[7:0]
output_decision_v[7:0]
done
Crossbar
input0_data_out[8:0]
G4

14. APPENDIX F – SELECTED DESIGN SCHEMATICS

The following are selected schematics from the synthesized RTL. Most of the schematics were too dense to produce meaningful plots, so only a few of the smaller blocks are provided.