Router Core

Specification Document v1.00

Overview

This document specifies the interface and behavior of the Router Core module for the ECE551 design project. The Core acts as the brain of our router system. It is responsible for analyzing packets, determining which link they should be sent on, and passing them on to the corresponding TX unit. The Core also communicates with a simulated Processor Node. If the Core receives a packet destined for its address, it delivers the packet to its Node. The Core must also be capable of receiving new packets from its Node. These new packets may represent new messages to be sent or may be used to configure the Core.

Each Router Core unit is attached to four TX/RX pairs, one pair per direction (North, South, East, West). For the purpose of designing the Router Core, you do not need to know the inner workings of the TX or RX. All communication with the Cores and between the TX and RX are governed by protocols described in this document. If your Router Core conforms to those protocols, it should work properly with any TX & RX units that conform to their specification, even those not designed by you. Therefore, it is important that you follow all communication protocol standards exactly.

Each Router Core is also connected to one Processor Node. For this project, the Processor Node will be simulated (i.e. part of a testbench-related module).

Diagrams showing the interconnection between the TX, RX, Router Core, and Processor Node are given in Fig. 1-2.

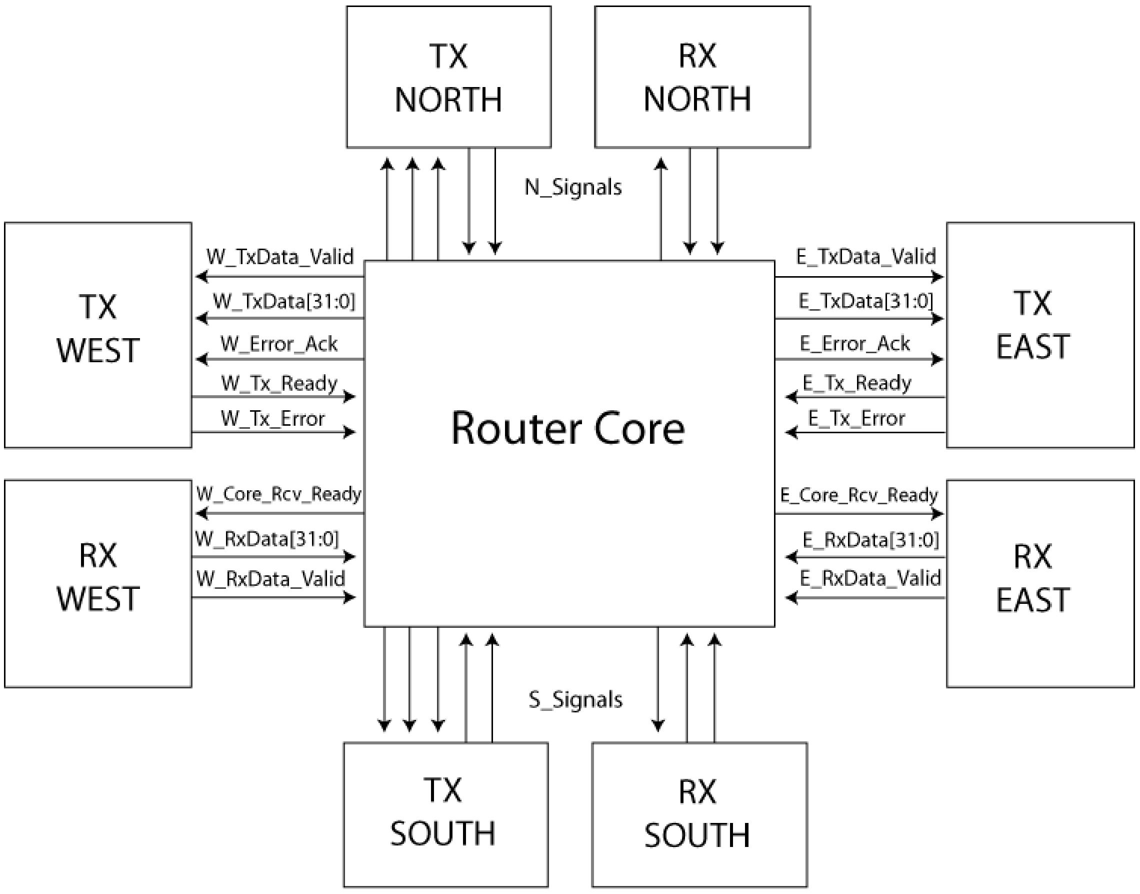


Fig. 1: Router Core interfaces to four TX & RX pairs. North, South, East, and West links each have the same interface.

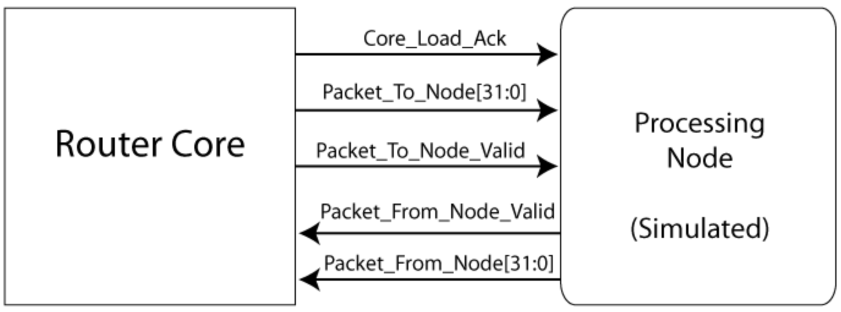


Fig. 2: Router Core interface to Processing Node.

Router Core Interface

Your Router Core unit should use the following interface. Please name your signals using the same names to ensure that your module will work properly with the testbenches used for grading. More detailed descriptions of the behavior of these signals are given later in this document.

Global Interface

input Clk_r – The global Core clock. This clock is synchronized between Core and Processor Node. The TX & RX do not necessarily use the same clock.

input Rst_n – The global asynchronous active-low reset signal.

Router Core /TX Interface

*Note: This interface appears 4 times in the Router Core. For the North interface, each signal should be preceded with **N_** as shown below. For the South, East, and West interfaces, it should be replaced with **S_, E_,** or **W_** respectively.*

output N_TxData_Valid – This signal serves as part of the TX/Core handshake. The Core should set TxData_Valid high when it has valid data ready on the output TxData and after the TX has asserted its associated Ready signal.

output [31:0] N_TxData – This signal represents a 32-bit packet for transmission.

output N_Error_Ack – This signal acknowledges that the Core has seen that Tx_Error was asserted.

input N_Tx_Ready – This signal serves as part of the TX/Core handshake. When Tx_Ready is high, it means that the TX is ready to accept a packet from the Core.

input N_Tx_Error – This signal is used to tell the Core that an error occurred while transmitting a packet on this link.

RX / Router Core Interface

Note: This interface appears 4 times in the Router Core. For the North interface, each signal should be preceded with ***N_*** *as shown below. For the South, East, and West interfaces, it should be replaced with* ***S_, E_,*** *or* ***W_*** *respectively.*

output N_Core_Rcv_Ready – This signal serves as part of the RX/Core handshake. When Core_Rcv_Ready is high, it means that the Core is capable of accepting a packet from the receiver.

input N_RxData_Valid – This signal serves as part of the RX/Core handshake. When RxData_Valid is high, it means that the input RxData has valid data available on it.

input [31:0] N_RxData – This signal represents a 32-bit packet which has been received at the RX from a different router, de-serialized, and is now presented to the Core.

Router Core / Processor Node Interface

input [31:0] Packet_From_Node – A 32-bit parallel interface for transferring packets from the Node to Core.

input Packet_From_Node_Valid – If this signal is high, it indicates that the Node has new, valid data on the Packet_From_Node interface.

output Core_Load_Ack – Once the Core has loaded data from the Node, it will assert Core_Load_Ack for one cycle of Clk_r.

output [31:0] Packet_To_Node – A 32-bit parallel interface for transferring packets from the Core to Node.

output Packet_To_Node_Valid – When the Core puts valid data on Packet_To_Node, it should assert this signal for one cycle of Clk_r.

Handshaking

The Router Core and Processor Node operate on a single clock, **Clk_r**. The TX/RX, however, will operate on a different clock that may run at a different speed from the Router Core clock. Therefore, any signals on the interface between the TX & Core or the RX & Core are said to be *crossing clock domains*.

When signals cross clock domains it is no longer possible for their behavior to be described easily using a clock, since an operation that takes one cycle in a certain clock domain may take many cycles in another domain. This becomes an issue when data needs to be passed between domains, because it is important for the sender of the data to know when the receiver has finished reading the data, otherwise the sender might start changing the data before the receiver has had enough time to read it. *Note that the term 'receiver' is being used in a generic sense in this section, rather than referring to the RX unit.* To overcome this problem, signals that cross clock domains need to operate using a handshaking protocol.

In this project, handshaking is generally done using two signals: a *valid* signal from the sender indicates to the receiver that there is data it needs to read. A *ready* signal from the receiver indicates to the sender that the receiver is capable of reading new data. The handshaking process goes as follows (assume that both READY and VALID start low after reset):

- When the receiver has a buffer available to load new data it raises the READY signal.
- If the sender sees the READY signal is high *and* the sender has data it needs to send, it raises the VALID signal.
- Once the receiver has loaded the data from the sender (usually into a register operating on the receiver's clock domain), it lowers the READY signal.
- When the sender sees that the READY signal is low, it lowers the VALID signal.

Under no circumstances should you change the order of the handshake. To make sure you avoid any potential pitfalls in handshaking, you need to enforce strict ordering of the handshake process. Make sure you follow these rules:

- **Never** raise READY unless VALID is low.
- **Never** drop READY unless VALID is high.
- **Never** raise VALID unless READY is high.
- **Never** drop VALID unless READY is low.

A waveform showing a generic handshaking exchange is shown in Fig. 3.

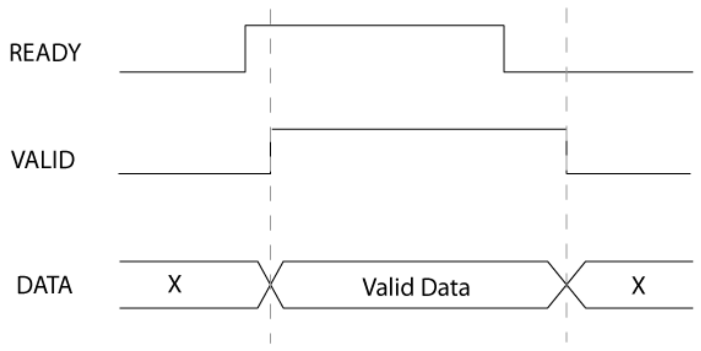


Fig. 3: Generic handshaking example

The Router Core takes part in three different handshake protocols on each of its four links.

TxData Handshake

This handshake takes place between the TX and Router Core, and is used to load packets from the Core into the TX. The Core acts as sender using the **TxData_Valid** and **TxData** signals, and the TX acts as the receiver using the **Tx_Ready** signal. The TX will wait until it has a buffer free to load a packet before initiating the handshake by raising **Tx_Ready**.

RxData Handshake

This handshake takes place between the RX and Router Core, and is used to load packets from the RX into the Router Core. The RX acts as sender using the **RxData_Valid** and **RxData** signals, and the Core acts as the receiver using the **Core_Rcv_Ready** signal. The Core should wait until it has a buffer free to load the packet from the RX before initiating the handshake by raising **Core_Rcv_Ready**.

TxError Handshake

This handshake also takes place between the TX and Router Core, and is used to acknowledge that the Core has received the parity error signal that originally comes from the RX unit. The process works as follows:

1. The TX sees raises **Tx_Error** high.
2. When the Core sees that **Tx_Error** is high, it raises **Error_ACK**.
3. When the TX sees that **Error_ACK** is high, it lowers **Tx_Error**.
4. When the Core sees that **Tx_Error** has been lowered, it lowers **Error_ACK**.

Core/Node Communication Protocol

The Router Core and Processor Node operate on a synchronous clock (**Clk_r**). Moreover, we will guarantee that the Processor Node always has resources available to accept packet from the Core. This allows us to use a simpler protocol between the Core and Node.

Sending a Packet to the Processor Node

When the Core needs to deliver a packet to the Node, it should place the data on the **Packet_To_Node** bus and drive **Packet_To_Node_Valid** high for one cycle of **Clk_r**. We will guarantee that the Node has stored the data from **Packet_To_Node** by the end of that cycle. It is possible to leave **Packet_To_Node_Valid** high for multiple cycles to send multiple consecutive packets to the Node. However, for every cycle that **Packet_To_Node_Valid** is high, there must be new valid data on **Packet_To_Node**. An example is shown in Fig. 4.

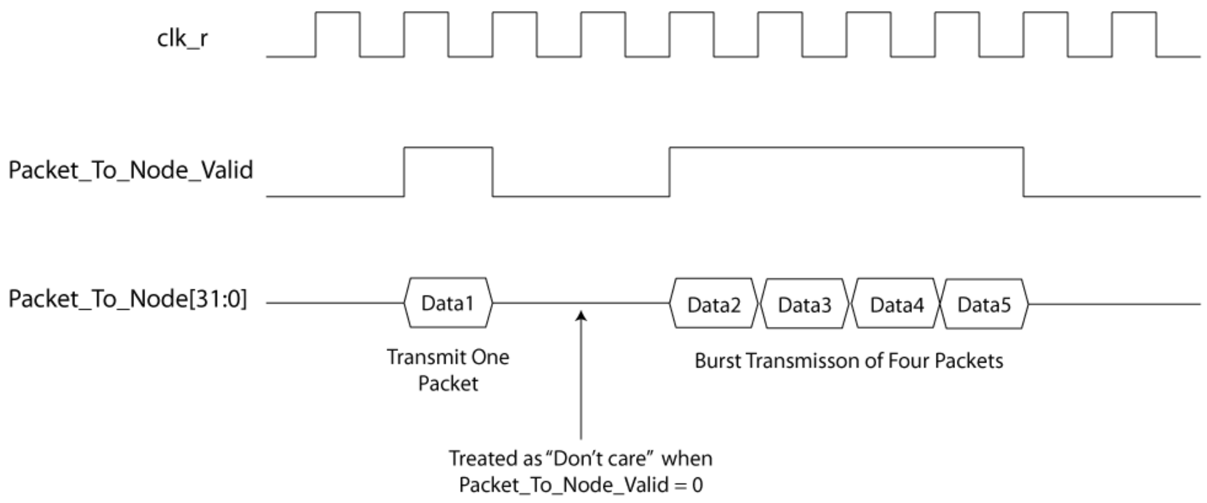


Fig. 4: Transmission of packets from Core to Node

Receiving a Packet from the Processor Node

We cannot guarantee that the Core will always have sufficient buffers free to be able to accept a new packet on any cycle, therefore the protocol for receiving data from the Node uses a partial handshake.

1. When the Node has new data for the Core, it will place it on **Packet_From_Node** and drive **Packet_From_Node_Valid** high.

2. To indicate that the Core has stored the packet, it should drive **Core_Load_Ack** high for one cycle.
3. On the next cycle, the Node will either drop **Packet_From_Node_Valid** if it has no more packets to send or will keep it high if there is new valid data available on **Packet_From_Node.**

An example of this protocol in use is shown in Fig. 5

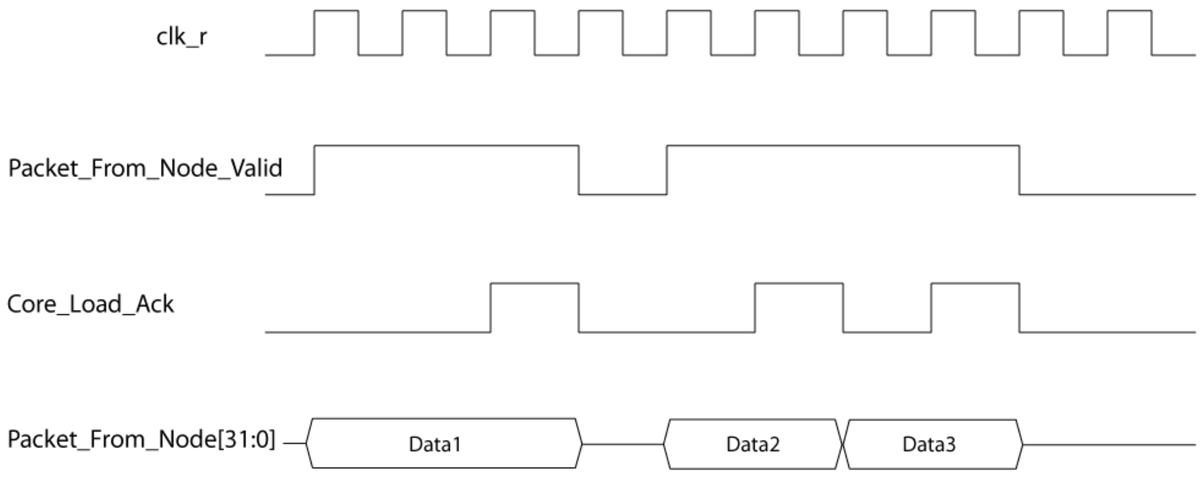


Fig. 5: Reception of packets from the Processor Node.

Packet Format

Each packet used in the router is 32-bits long. The 32-bits are organized into three fields, as shown in Fig. 6. The packets can be broken up into two major categories.

- Command Packets are used to configure a Router Core. Command Packets always come from the attached Node, not from a different router.
- Message Packets are the packets that are sent from router to router. A Core may receive a Message Packet via either an RX link or via the attached Node.

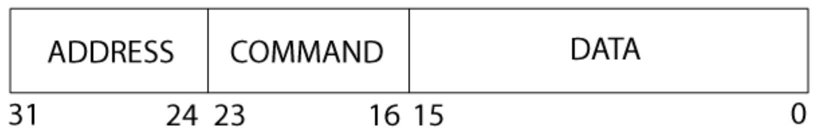


Fig. 6: A generic packet from the Processing Node.

Command Packets

Command Packets are characterized by an ADDRESS field that contains “1111_1111” in bits [31:24] of the packet. In Command Packets, bits [23:16] (COMMAND field) will contain the Command Code. Bits [15:0] (DATA field) contain command-specific data. Once you have executed the command contained in the packet, the packet can be discarded.

Command Code Listing:

8’d0 : Address Assignment
8’d1 : Link Configuration
8’d2 ~ 8’d255 : Reserved for future use

<u>*Address Assignment Command*</u>

The Address Assignment command is used to assign an address to each Router Core. This is the method the Core uses to learn its own address. The Core should store this address in an internal register. It is important for the Core to know its address, since this is necessary when determining routing. In the Address Assignment command, the 8 most-significant bits of the command-specific DATA field of the packet are Don’t Care, and the 8 least-significant bits contain the new address to be assigned. An example Address Assignment command packet is shown in Fig. 7.

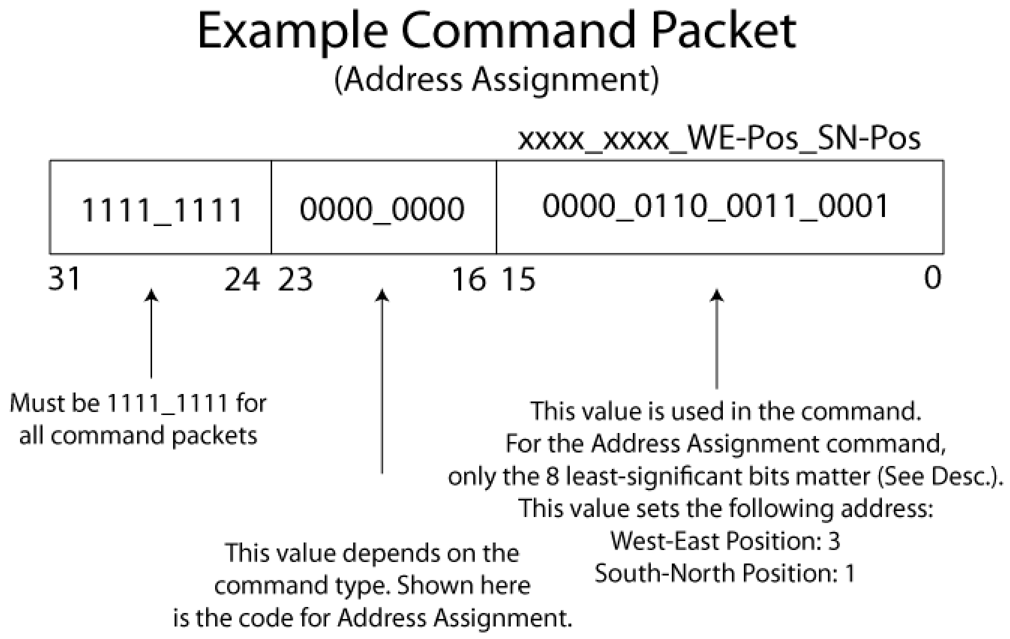

Example Command Packet
(Address Assignment)
xxxx_xxxx_WE-Pos_SN-Pos
1111_1111
0000_0000
0000_0110_0011_0001
31
24
23
16
15
0
Must be 1111_1111 for all command packets
This value is used in the command. For the Address Assignment command, only the 8 least-significant bits matter (See Desc.). This value sets the following address: West-East Position: 3 South-North Position: 1
This value depends on the command type. Shown here is the code for Address Assignment.

Fig. 7: Example Address Assignment Command Packet

Each router has an 8-bit address. This address can be broken into two 4-bit components. The first 4 bits specify the router's position on the West-East axis. Higher values are further East. Lower values are further West. The second 4 bits specify the position on the South-North axis. Higher values are further North. Lower values are further South. Values begin at zero. For example, in a mesh of size 5x5, the router located in the Southwest corner of the mesh would have the address "0000_0000" (<0,0>), the router located in the Northeast corner will have the address "0100_0100" (<4,4>), Northwest: "0000_0100" (<0,4>), Southeast: "0100_0000" (<4,0>).

Since we are using a symmetrical 2-D mesh and since the address "1111_1111" is unusable, you may assume that your Core will never be assigned any address of the form "1111_xxxx" or "xxxx_1111". Therefore the maximum mesh size we will support is up to 15x15.

Address Assignment will be the first packet each Core receives after reset, and will not be sent again until the next reset.

Link Configuration Command

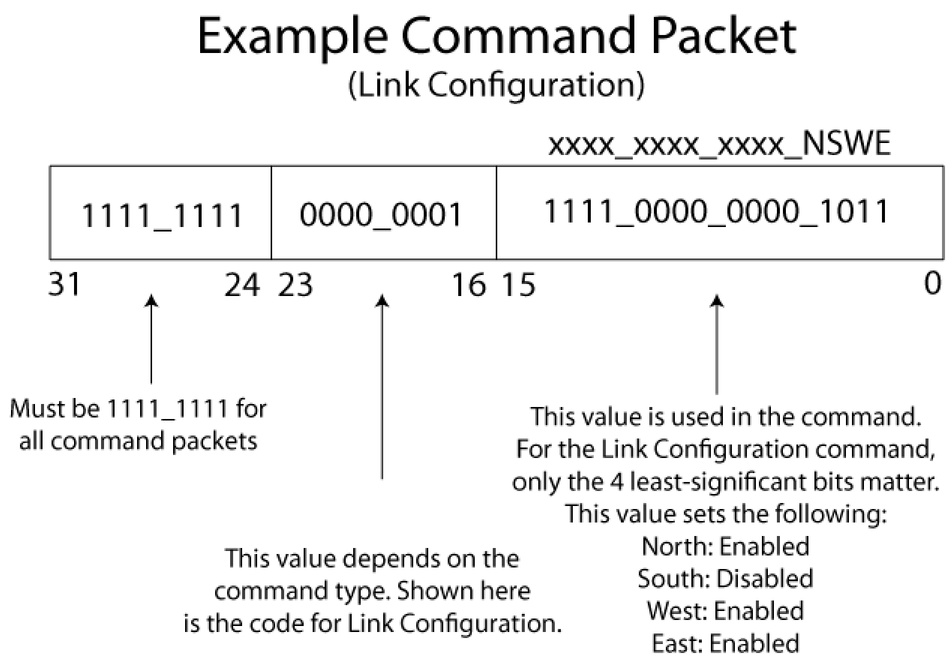
The Link Configuration command is used to tell the Router Core whether or not it should use any of its four links. This is necessary because some routers will be located on the edge of the mesh and might not, for example, have a West neighbor. It could also be used to disable a link if the Processor Node knew that the router on that link was out of service. The Link Configuration command only uses the four least significant bits of the command-specific DATA field of the packet. These four bits represent, in order of significance, the North, South, West, and East enable bits. If the corresponding bit is set to 0, that means that your Router Core should not attempt to route any packets to that link. **You do not need to worry about aborting any packets that might currently be sending or buffered in your TX –** we will only send the Link Configuration command once, before we start injecting Message Packets. **After reset, ALL links should start up disabled. Therefore, a Link Configuration packet must be sent to each Core before it can start routing messages**.

An example Link Configuration Packet is shown in Fig. 8.

Link Configuration will be the second packet each Core receives after reset (following Address Assignment), and will not be sent again until the next reset.

Other Commands

If you receive a command packet containing any of the reserved command codes, you should do nothing with it; you can simply ignore (drop) the packet.



Example Command Packet
(Link Configuration)
xxxx_xxxx_xxxx_NSWE
1111_1111
0000_0001
1111_0000_0000_1011
31
24
23
16
15
0
Must be 1111_1111 for all command packets
This value depends on the command type. Shown here is the code for Link Configuration.
This value is used in the command. For the Link Configuration command, only the 4 least-significant bits matter. This value sets the following:
North: Enabled
South: Disabled
West: Enabled
East: Enabled

Fig. 8: Link Configuration Command Packet

Message Packets

In a Message Packet, the ADDRESS field contains any values other than “1111_xxxx” and “xxxx_1111”. This address represents the recipient address, and matches the address format used in Address Assignment packets. The ADDRESS field will be used in the routing algorithm. **You may assume that we will not send you any addresses that don’t exist** (such as the address <9,4> in a 6x6 mesh).

The COMMAND field is reserved for your use. You may choose to store metadata in this field to implement extra features for your routing algorithm or you may choose to do nothing with it. Our testbenches will treat the contents of the COMMAND field as Don’t Care when checking for errors.

The DATA field contains the message body. You should pass this on to the recipient unchanged.

An example Message Packet is shown in Fig. 9.

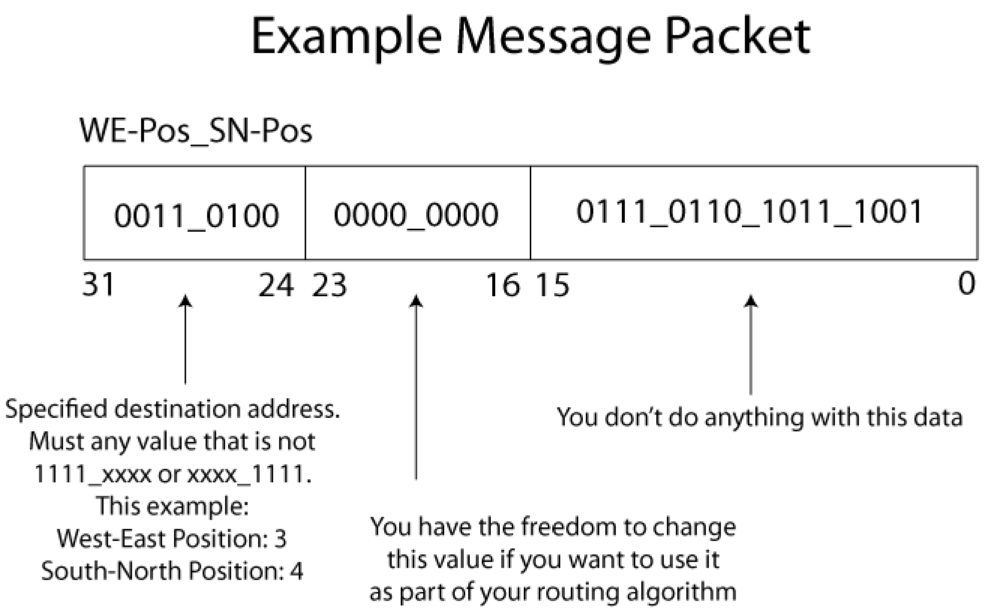

Example Message Packet
WE-Pos_SN-Pos
0011_0100
0000_0000
0111_0110_1011_1001
31
24
23
16
15
0
Specified destination address. Must any value that is not 1111_xxxx or xxxx_1111. This example: West-East Position: 3 South-North Position: 4
You have the freedom to change this value if you want to use it as part of your routing algorithm
You don’t do anything with this data

Fig. 9: Example Message Packet

Unrecognized Packet Types

If you receive a packet with an ADDRESS field that does not match that of a Message Packet or Command packet (for example, the address 1111_0000), you should do nothing with it; you can simply ignore the packet.

Router Core Components

The Router Core will consist of four primary components: (1) A set of state machines that manage the handshake protocols needed for loading and unloading packets between the Core and attached modules, (2) logic for computing a Message Packet’s route (i.e. the routing algorithm), (3) a set (or sets) of buffer queues used to store packets before they are routed, and (4) logic that processes Command Packets and handles Core configuration.

You are given a wide range of freedom on how you choose to organize and partition your internal components. In particular, you should spend time thinking about how you want to partition your state machines and buffers, as this can have important implications on the complexity and performance of the router.

Buffer Queue Architecture

You will want to include buffer queues in your Router so it will be able to load multiple message packets without stalling. There are many options for your buffers. You may choose to have a queue at each input (RX), each output (TX), and/or a shared central queue. Sharing a central queue may allow you to use fewer buffer registers than having individual queues per link. However, this may be more complex to use than dedicated queues. You will also need to think about how large you should make your queues. Having too few buffers can cause one or more links to stall, hurting your throughput. However, more buffers means more area.

Routing State Machine Partitioning

When designing the state machines that handle the routing of packets to their destination links, you should think about whether you want to use a single centralized state machine that handles all links or distributed state machines that work in parallel.

For example, you might choose to have a single state machine that processes packets from each of the 4 input links in a round-robin fashion. This sort of design is simpler and allows you to share the same routing algorithm logic for all four links, but doesn't exploit any parallelism.

You might instead choose to use 4 routing state machines, with one dedicated to each input link. This approach exploits parallelism and could significantly improve your throughput. On the other hand, this approach uses more area because you need to duplicate your routing logic. It may also make some things more complex. For example, if two state machines want to forward a packet to the same link on the same clock cycle, you will need to think about how to resolve such a conflict.

Tips For Getting Started

If you are not sure how to start, it is best to try to design the Router Core in the simplest way you can. You can always add more once you have the simple design working properly. If you want a simple design, try using a single routing state machine that processes packets from each link in a round-robin fashion. For example, try to route a packet from the North link on cycle 1, West link on cycle 2, South on cycle 3, East on cycle 4, and new packets from the Node on cycle 5, then repeat. This is slower than using multiple routing state machines, but avoids many complexities.

You may also find it easier to have dedicated input & output buffer queues at each link and may want to start with a single buffer in each queue. Again, this may not be very efficient, but will be more straightforward.

An example of the various logic used to implement this "simple" design is shown in Fig. 10.

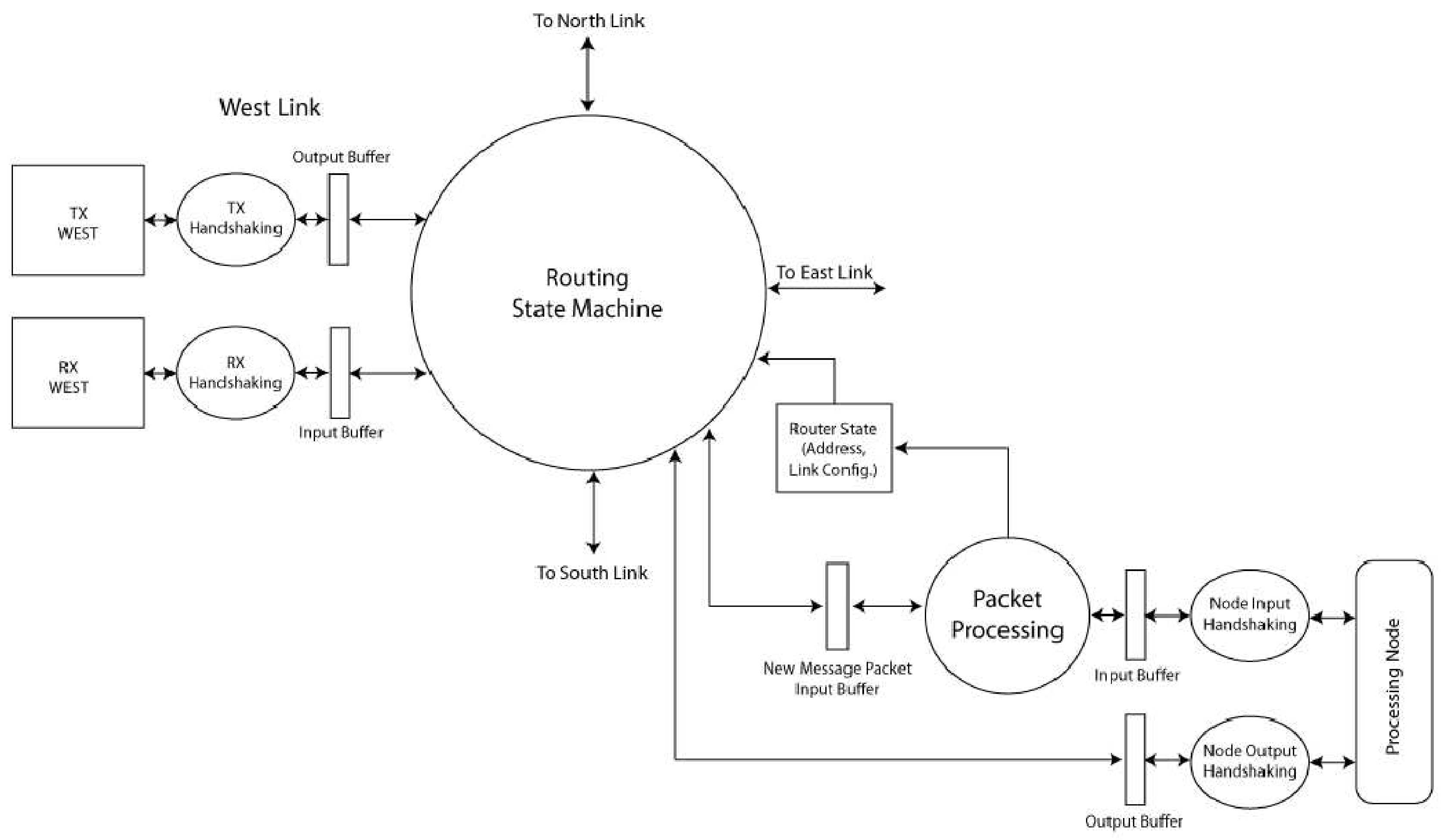


Fig. 10: This diagram shows the interconnections of logic needed to design a Router Core that utilizes a single centralized Routing State Machine and dedicated Input and Output buffer queues (with queue size = 1). This may be a good starting point for you to think about how you want to organize your design.

Initially, the router design may seem intimidating. To make it more manageable, I encourage you to utilize a hierarchal design to break the router up into more manageable pieces. Fig. 11 shows the design from Fig. 10 represented as a hierarchal design that re-uses modules.

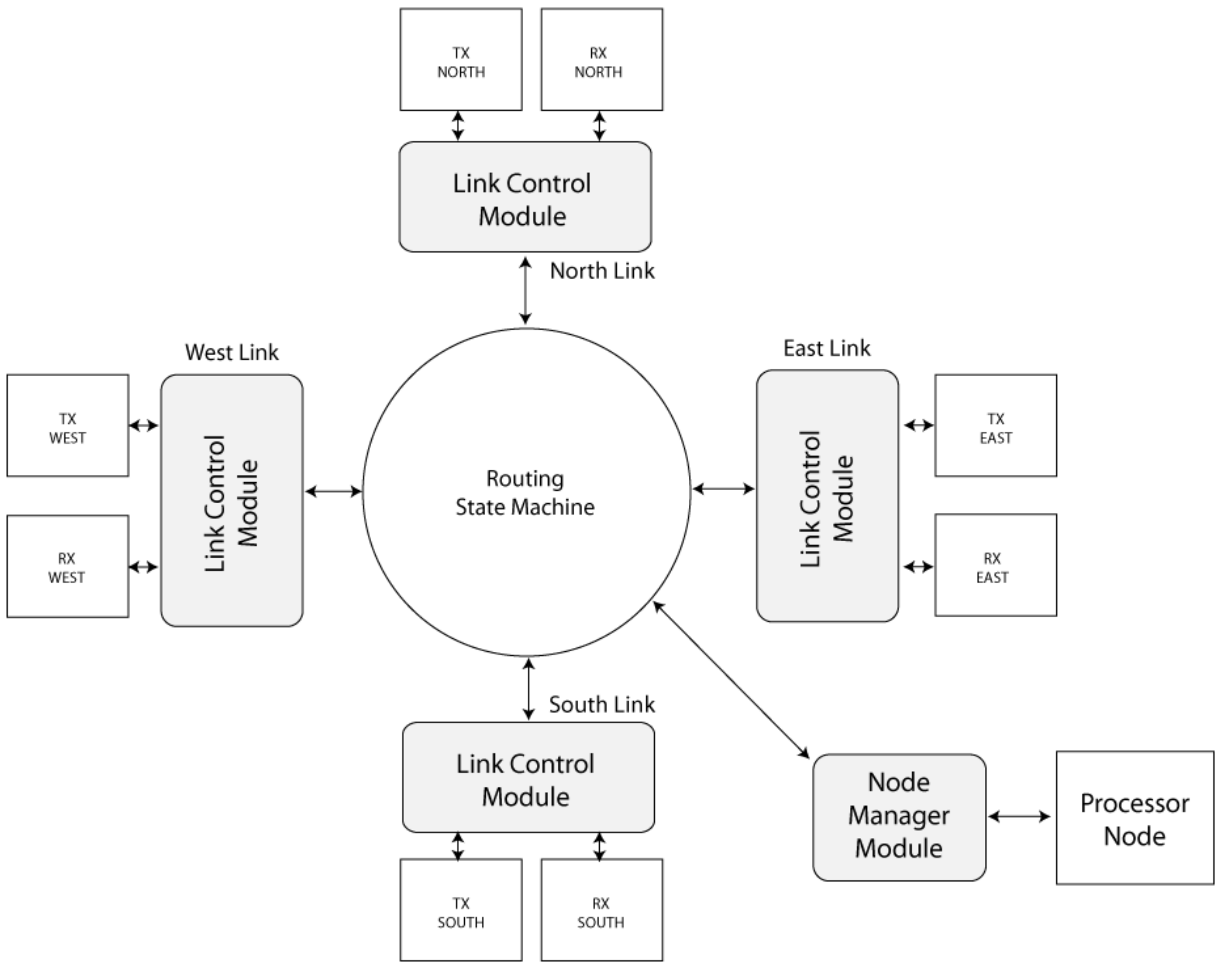


Fig. 11: Hierarchal look at the router design from Fig. 10. Using a hierarchal design approach can make the design of complex systems much more manageable.

From this view, you only need to worry about designing three components: the Link Control module, the Node Manager module, and the Routing State Machine. Of course, this approach is optional. Feel free to use whatever design approach you prefer.

The last design tip concerns the buffers. In many portions of the router design, it is important to know if a buffer is full or not. An easy way to keep track of whether a buffer is full is to add an extra bit to it that indicates whether or not the buffer contains data. When one state machine writes data to the buffer, it sets the full bit to ‘1’. When another (or possibly the same) state machine reads data from the buffer, it sets the full bit to ‘0’, to indicate that new data can be stored in the buffer.

The key here is that you may try to write to the full bit from more than one state machine. If you do so, make sure that you do not try to use multiple drivers to the input of the full bit flip-flop. Remember the rule of thumb: Don’t try to write to the same variable in multiple **always** blocks when using Behavioral Verilog!

Fig. 12 shows a diagram of how you might connect logic to use a buffer with a full bit.

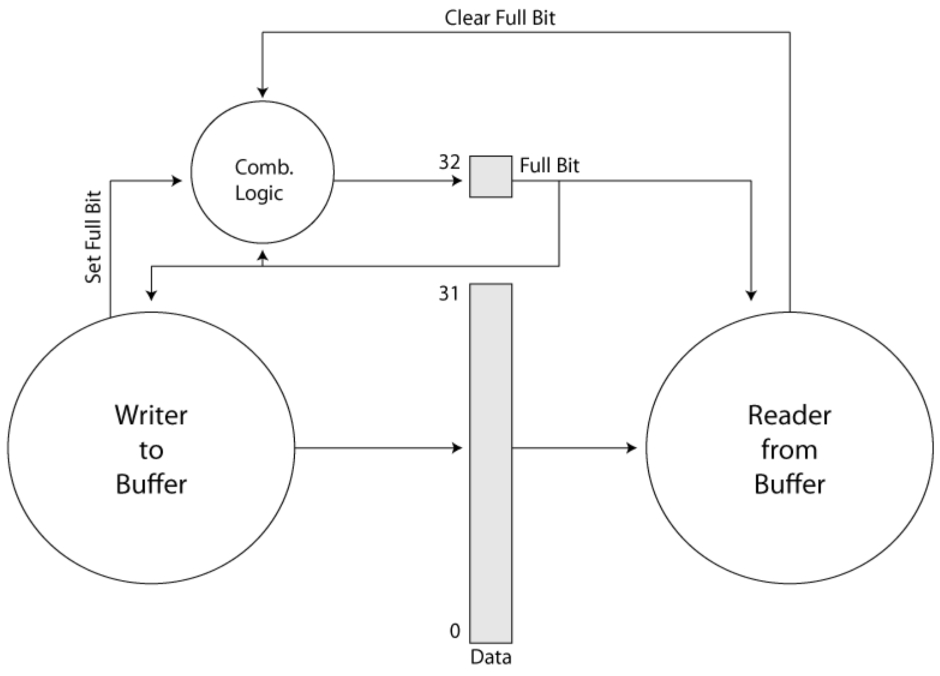


Fig. 12: Writing to and Reading from a Buffer with Full bit.

Routing Algorithm

There is no fixed standard for your routing algorithm; the important thing is that your algorithm is capable of getting *most* of the messages to their destination on *some* route. (Remember that we say most because we are using a best-effort network. It is OK for you to drop some packets, but your design will be evaluated based on your drop rate. Your router should very rarely drop packets under low-traffic conditions where all routers and links are functioning properly.)

In general, your routing should be based on the address of the current router, the destination address of the packet, and the state of each of your transmission links (enabled or disabled). For performance reasons, you may also want to consider whether a link is currently ready to accept a new packet or how many packets are waiting in a link's queue (if you choose to use an output queue).

Here is pseudocode for an "easy" routing algorithm. Further complexity that you add to your algorithm can be counted as a project "extra".

```
IF (destination address matches my address)
      Send to Processor Node;
ELSE IF (destination address is further North & North link not disabled)
      IF(North link buffer is available)
            Route North;
      ELSE
            Wait;
ELSE IF (destination address is further South & South link not disabled)
      IF(South link buffer is available)
            Route South;
      ELSE
            Wait;
ELSE IF (destination address is further West & West link not disabled)
      IF(West link buffer is available)
            Route West;
      ELSE
            Wait;
ELSE IF (destination address is further East & East link not disabled)
      IF(East link buffer is available)
            Route East;
      ELSE
            Wait;
ELSE
      Drop Packet;
```

Listing I: Pseudocode for a simple fixed priority routing algorithm

Some of the weaknesses of this fixed priority algorithm are that it will overload the high-priority North link relative to the low-priority East link, causing lower throughput under high-load scenarios. It is, however a good starting point to think about how you can design algorithms that will perform better under load.

Router Start-up

To make life easier for you, we will follow a standard start-up procedure when testing your router.

1. After Reset, every router will first receive an Address Assignment packet.
2. Next, every router will receive a Link Configuration packet.
3. Your router is now ready to start routing.

We guarantee that every Router Core will always receive these two packets from its Node before getting any Message Packets to route. **Furthermore, we will guarantee that once your router begins normal routing, we will not send the Address Assignment or Link Configuration packets again, until after the next reset.** You do not need to worry about having a link be disabled while you are using it or having your address change while computing a route.

Optional "Project Extras" Features

The sections that follow describe features that are not required in your design, but may be added to improve your score. Don't feel pressured to implement any of these features; you may find that doing extras related to testbench design is easier than adding router features since the router is already fairly complex. You should also feel free to come up with your own interesting features instead (but it might be a good idea to run them by the instructor or TA before you start implementing them). Keep in mind that you might want to wait until *after* you have the basic functions needed for Milestone 2 to add extra features. Your extras will not be graded until the final project demo/report.

Handling Transmission Errors

Your Router Core is **not** required to do anything with the error data it receives from the TxError handshakes (it is however, required to complete the handshakes). Error-handling is left as an **optional** feature you can implement if you desire.

Because it is difficult to synchronize timing between the Core and TX/RX, it is probably not easily feasible to save old packets and try resending them if an error occurs. However, in the Router Core you might want to track the frequency of errors on a particular link. If you notice that a link is frequently producing errors, you might want to disable communication on that link and try to send your packets on a different route.

During the performance tests of the final design, one of the tests we run will simulate intermittent and permanent faults on links between some of your routers. If you implement adaptive routing based on error detection, your router may exhibit a lower drop rate on this test. It will also count toward the Extras portion of the project grade. However, adding error-handling will add additional complexity into your Router Core and will require extra area. It is up to you to decide whether it is worthwhile to add it to your design.

Deadlock & Livelock Handling

Dealing with Deadlock and/or Livelock is optional, and considered a project "extra".

Deadlock is always an important concern when designing networks. Deadlock is a term that describes a state in which no forward progress can be made in your network. Deadlock tends to occur when your network is under high load, with a large number of packets being injected and forwarded. A typical deadlock scenario might occur as follows: Assume you have Node A and Node B. Node A has many packets to send to B, but can't send any because all of B's buffers are full of packets destined for A. Node B, similarly wants to send to A, but can't because all of A's buffers are full of packets for B. Therefore neither is able to send.

Because we are using a best-effort network, we have two options for dealing with deadlock: (1) Deadlock avoidance and (2) Deadlock detection and recovery.

Deadlock avoidance attempts prevent deadlock from ever happening. There are multiple deadlock avoidance mechanisms out there that you can investigate if you are interested in implementing deadlock avoidance. One popular method that you might use as a starting point is called “Virtual Channels”.

Deadlock detection and recovery does not worry about preventing deadlock, but rather about how to get the network running again if deadlock happens. Since this is a best-effort network, one option for deadlock recovery is for a router to drop any packets that it has been holding for a long time or to simply drop all the packets it is buffering. Although this leads to a higher drop rate than deadlock avoidance, it is still much better than entering deadlock and not being able to deliver any packets afterward.

Livelock is a problem that is similar but less severe than deadlock and is typically related to the concept of Quality of Service (QoS) in a network. Livelock occurs when one packet is unable to ever make forward progress, despite the fact that other packets are making progress. This might happen when a network is under high contention and is using a routing algorithm that prioritizes certain packets over others. It might also happen if you are using a routing algorithm that allows a packet to be continuously routed back and forth or in a loop during high contention conditions

If you need to detect deadlock or livelock, one option is to make use of the router-reserved COMMAND field of each packet. You could fill this field with a Time to Live (TTL). TTLs are used to keep track of how long a packet has been waiting or how many times it has been forwarded around. One potential way you could use TTL to detect deadlock would be to set the TTL value to zero when a new message reaches the front of a TX buffer queue, then increment the TTL value of the packet each cycle it is stuck waiting in the queue. If the value exceeds a certain threshold (say the packet has been stuck waiting for more than 50 cycles), you could drop some or all of the packets in that queue. This is just one possibility, and you are free to develop your own.

Design Tips

- **Read this document thoroughly and follow it to the letter.** The easiest way to lose points is to leave out a required feature or deviate from the specified protocols.
- **Spend some time planning your design before you start coding**. Draw out the hardware you will need and make state diagrams for any state machine. You can include these in your project report later. The Router Core is particularly complex, as it has multiple communication links and state machines. It will be very important for you to think about how you want to partition the design before you start.

- **Exploit design re-use.** The Router Core interfaces with four identical sets of TX/RX pairs. Design the logic that manages handshaking with the TX & RX as a separate module, then use four instances of that module in your core. It will make your router simpler and easier to test.
- **Start Simple.** If you have a choice of implementing something in a simple way and a more complex way, do the simple implementation first. Once you have tested it and seen it working, you can add complexity piece by piece and monitor changes in the test results. This is much easier than starting out with a very complex design and trying to debug it. You might want to start by using a simple routing algorithm, a centralized routing state machine that handles packets from the different links in a round-robin fashion, and dedicated input & output buffers with a buffer size of one. You can always add more functionality after Milestone 2.
- **Divide and conquer**. Use a hierarchal design for the router. Divide functions such as management of the TX/RX links and processing of new packets from the Node into separate modules. It's usually easier to test several small modules rather than one large one, even if it means writing more testbenches. Why? Because a small module is much more likely to work correctly the first time and is simpler to fix if it doesn't. It also makes it easier for your team members to work in parallel.
- **It's never too early to start thinking about synthesis**. When you are planning your design, try drawing out the hardware that would be synthesized. Once you've finished writing the module, load it up in the synthesis tool and see if you get any errors. Running Analyze and Check Design in DesignVision can quickly uncover certain types of mistakes that might take a long time to find in ModelSim.

Document History

This document will be updated as necessary to address frequently-asked questions and fix any errors. The latest version of the document will always be available on the course website. You can find a summary of the version history here.

V1.00 – Initial release

Author: T. Gregerson