RapidIO IP Library

Core descriptions

Content

Document history

Version	Date	Responsible	Description
1.0	130121	magro732	First version.

1 Introduction

The IP cores described in this document can be used to build both end points and switches. The chapters below describes the IP cores in more detail.

The function will be dependent on how the cores are assembled and connected to each other. To build a switch for example, a RioSwitch IP core is needed. It will need packet FIFOs connected on all of its ports.
If the port has a transmission channel to another device and it should be a reliable channel you would use a RioPacketBuffeWindow and attach a RioSerial module to it.
If the port is connected to an internal end point you would use RioPacketBuffer and attach a custom written module.

2 RioSwitch IP core

2.1 Overview

The main task of this component is to relay packets from an inbound port to one of its outbound ports based on the destination deviceId of the packet. The mapping from destination deviceId to destination port is done using a lookup table where the device identifier is used as an address into the table. The content of the lookup table is the port to send the packet to.

The switch itself can be configured using maintenance packets. Using these, it is possible to dynamically set and change routes and to do other network administrational tasks like enumeration and routing table setup.

A decomposition of the RioSwitch into sub components looks like this:

SwitchPortInterconnect
    A Wishbone compliant interconnect that makes it possible to transfer data between ports.
RouteTableInterconnect
    A Wishbone compliant interconnect that makes it possible to do deviceId to portNumber lookups in a shared routing table.
SwitchPort
    This subcomponent acts as a Wishbone master to transfer a packet to a destination port using the SwitchPortInterconnect. It contains both a master that handles the inbound packet and a slave that receives an outbound packet from a master of another SwitchPort.
SwitchPortMaintenance

This is a special port that receives **all** inbound maintenance packets from all ports to determine if the packet is targeted to this switch instance.

See later chapters for more details about the sub components.

2.2 Entity description

Generics	
SWITCH_PORTS	The number of ports to instantiate on the switch.
DEVICE_IDENTITY	The value to return as DeviceIdentity when a maintenance request is received on offset 0x000000 (DeviceIdentityCAR).
DEVICE_VENDOR_IDENTITY	The value to return as DeviceVendorIdentity when a maintenance request is received on offset 0x000000 (DeviceIdentityCAR).
DEVICE_REV	The value to return when a maintenance request is received on offset 0x000004 (DeviceInformationCAR).
ASSY_IDENTITY	The value to return as AssyIdentity when a maintenance request is received on offset 0x000008 (AssemblyIdentityCAR).
ASSY_VENDOR_IDENTITY	The value to return as AssyVendorIdentity when a maintenance request is received on offset 0x000008 (AssemblyIdentityCAR).
ASSY_REV	The value to return as AssyRev when a maintenance request is received on offset 0x00000c (AssemblyInformationCAR).
Signal interface	
clk	System clock.
areset_t	Asynchronous reset, active low.
portLinkTimeout_o	The value written to portLinkTimeoutControlCSR.
linkInitialized_i	Input value showing if the link connected to this port is initialized, i.e. if there is a link partner present.
outputPortEnable_o	Indicates if non-maintenance packets are accepted in the outbound direction.
inputPortEnable_o	Indicates if non-maintenance

	packets are accepted in the inbound direction.
localAckIdWrite_o	Indicates if an ackId write access is active.
clrOutstandingAckId_o	Indicate if the outstanding ackId should be cleared. Read when localAckIdWrite_o is asserted.
inboundAckId_o	Indicate the value to set the inbound ackId to. Read when localAckIdWrite_o is asserted.
outstandingAckId_o	Indicate the value to set the outstanding ackId to. Read when localAckIdWrite_o is asserted.
outboundAckId_o	Indicate the value to set the outbound ackId to. Read when localAckIdWrite_o is asserted.
inboundAckId_i	The current inbound ackId of a port.
outstandingAckId_i	The current outstanding ackId of a port.
outboundAckId_i	The current outbound ackId of a port.
RioPacketBuffer interface - See RioPacketBuffer description.	
configStb_o	Indicate that an access to the implementation defined config space is active.
configWe_o	Indicate if an implementation defined config space access is a write access or a read access.
configAddr_o	Indicate the address of an implementation defined config space access.
configData_o	If the access is a write, this signal contains the data to write.
configData_i	If the access is a read, this signal should contain what to return as a response on the access. It will be sampled one tick after the configStb_o has been asserted.

2.3 Sub components

2.3.1 SwitchPortInterconnect

The SwitchPortInterconnect is a Wishbone based interconnect. All ports are attached to it and issues bus transactions to access other ports.

The interconnect is hard coded to support 256 ports to simplify the VHDL code and all accesses are 32-bit. It is up to the synthesizer tool to optimize the design if the switch contains fewer ports.

Following the Wishbone standard makes it transparent to the user if the interconnect is a shared bus (only one master active at the same time) or a crossbar (multiple masters active at the same time). The current implementation is a shared bus to minimize the resources used in the design but it can be replaced with a more resource intensive crossbar whenever needed.

The total switching capacity of a shared bus implementation could roughly be calculated as (system clock frequency * the interconnect data bus width) which in our case equates to 25E6*32=800Mbit/s. Arbitration between ports is done using a Round-Robin scheduler. To avoid deadlocks it is not recommended to do a cut-through switch unless the interconnect is a crossbar.

The SwitchPort entities access one another by doing bus transactions. Packets are transferred from one port to another by addressing the destination port according to the address map found in the table below.
A port will send maintenance packets to its corresponding address and by setting the most significant bit in the address. Since each port has its own address space to write maintenance packets to, the maintenance switch port can return a response to the same port as the request was received on.

The below table shows the static address map that is hard wired to 256 ports.

Name	Address [9:0]			Write Content
Port 0 status	0	00000000	0	bit 0: writeFrame bit 1: writeFrameAbort
Port 0 data	0	00000000	1	bit 31-0: writeFrameContent
Port 1 status	0	00000001	0	…
Port 1 data	0	00000001	1	…
	…	…	…	…
Port 255 status	0	11111111	0	…
Port 255 data	0	11111111	1	…
Port 0 maintenance status	1	00000000	0	bit 0: writeFrame bit 1: writeFrameAbort
Port 0 maintenance data	1	00000000	1	bit 31-0: writeFrameContent
Port 1 maintenance status	1	00000001	0	…
Port 1 maintenance data	1	00000001	1	…
…	…	…	…	…
Port 255 maintenance status	1	11111111	0	…
Port 255 maintenance data	1	11111111	1	…

Table: Internal address mapping between switch ports.

2.3.2 RouteTableInterconnect

The RouteTableInterconnect is a Wishbone based interconnect used to share a common routing table between multiple switch ports. A priority scheduler has been used to minimize resources. A port requests an access by raising its stb_o-signal and places the destination deviceId as addr_o. It then waits for the port index to be valid when ack_i is asserted.
A lookup takes only one tick to perform so there should be no need to duplicate the routing table for each port.

2.3.3 SwitchPort

A switch port is divided into two parts, a master and a slave. The master part is responsible for reading a packet that is available and to transfer it to its destination port. It does this in the following sequence:

1. Wait for a new packet.
2. Check if the packet is a maintenance packet. If it is transfer it to this ports' address in the maintenance port. Otherwise, read the destination deviceId of the packet and lookup its destination port by accessing the routing table.
3. Wait for the destination port to be ready to accept the packet. This is done by reading the PortStatus and checking that writeFrameFull is deasserted. If writeFrameFull is asserted, the transfer is aborted and the bus cycle is terminated to let other ports use the interconnect. The a new cycle to transfer the frame is initiated.
4. Once the writeFrameFull has been found to be deasserted the packet is transferred to its destination port.
5. The packet is removed from the inbound queue. It has now been moved from an inbound packet queue to an outbound packet queue.

2.3.4 SwitchPortMaintenance

A switch port maintenance is divided into four parts, a slave, a master, a config space handler and the routing lookup table. The slave is responsible for receiving new packets from other ports and the master generates responses to these. The config space handler responds to read and write accesses into the switch' maintenance space and the result from this is returned by the master.
The SwitchPortMaintenance module executes according to the steps below:

1. A packet is received by the slave from the SwitchPortInterconnect. The interface towards it is the same as an ordinary SwitchPort.
2. The slave parses the incoming packet and stores its content. When the packet is fully received, it is indicated to the master. The slave will now be locked until the master has completed its processing of the new packet. If another maintenance packet is received by a port it will remain in the inbound packet queue until the master has processed the current packet.
3. The master checks the content of the packet and determines if the packet is for this switch instance. If not, it decrements the hop-field, recalculates the CRC, lookup the destination deviceId to get the destination port and forwards the packet to that port. If the packet *is* aimed at this switch instance, it is converted

into an access into the internal configuration space and a response is sent back to the source port of the packet.
4. The master unlocks the slave to allow it to receive a new packet.

This module also maintains the routing lookup table since this is accessed through maintenance packets. The lookup table interface is connected to the routing table interconnect to be able to share it between all the ports in the system.
The routing table currently supports routing of 2048 deviceIds. It could be increased if it is hard-wired to less than 8-bit port numbers. The routing table is designed to fit into one block RAM in a Xilinx Spartan 6.

2.4 Limitations

There are some limitations on the implementation:

- Only 16-bit deviceIds are supported.
- Currently, only the store-and-forward switch philosophy is supported. It is however a minor task to make it support a cut-though implementation.
- The SwitchPortInterconnect is a shared bus which means that only one port can transfer packets to another port at a time. This decision has been made to lower the amount of used resources. It is however a simple task to replace the shared-bus interconnect with a crossbar interconnect without changing any interfaces.
- Only 32-bit maintenance accesses are supported. That also applies to packets that are only routed though.
- No packet priority is supported.
- No parity is added to packets internally. This may result in a hanging port if an error occurs internally when a packet is transferred between the internal block memories or while stored in an internal block memory.
- No maximum switch survival time is implemented. A packet can survive inside the switch forever until it is transmitted.
- Only the standard route table configuration is supported.

3 RioPacketBuffer IP core

The RioPacketBuffer contains logic to store a configurable number of packets in a memory. It can be configured to have a maximum memory size and a maximum number of packets stored in that memory size. It has a signal interface that allows a packet to be aborted and to detect this at the other side. It is also possible to start reading a packet before it is completely written into the memory.

The normal usage is to write a packet using writeContent and writeContentData. The signal writeFrame is asserted for one tick to indicate that the packet is completely written. When a packet is available for reading the signal readFrameEmpty is deasserted. The content at the read side is updated by asserting readContent and reading readContentData. The signal readContentEnd is asserted when there is no more packet content to read. Note that when this happens, readContentData is not valid anymore. See figure below for signal timing.

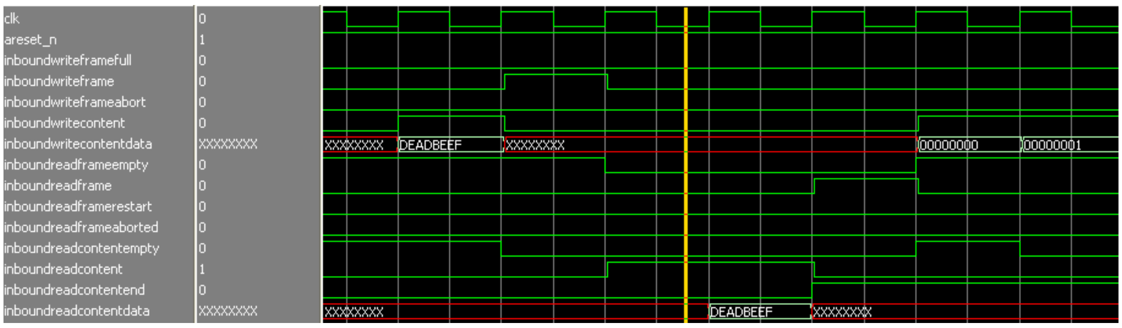

Figure: The above figure shows a frame where the content is one word equal to 0xdeadbeef.

To support the RioSerialTransmitter transmission window there is a special version of this component called RioPacketBufferWindow. This component has three additional signals called readWindowEmpty, readWindowReset and readWindowNext. Using these signals it is possible to read new packets without removing them from the memory. The packet can be removed later once a packet-accepted control symbol has been received.

If writeFrameFull is deasserted, it is always possible to write a full sized packet. This is to guarantee that a packet transfer will always be successful once it has started. That means that at most 68 words (the maximum size of a RapidIO packet) can be unavailable to be used in a worst-case scenario.

The packets are stored in a continuous memory (that could be block RAM) using four pointers. One pointer points to the first available memory position, one pointer to the first unavailable position, one pointer the current read position and one the current write position. An additional memory (not block RAM) stores the position of the packets stored in the packet memory. This information is used to know when the end of a packet has been reached.

4 RioSerial IP core

RioSerial implements the hardware independent protocol parts of the LP-serial physical layer described in RapidIO 2.2 part 6 chapters 2, 3, 5 and parts of chapter 6. Note that the other chapters are **not** implemented in this module. It is implemented in the hardware dependent PCS layer.

The interface between the PCS layer and this module handle an abstract object here called a symbol. A symbol is defined as a 34 bit long array of bits where the 2 MSB bits sets it as one of four flavors, IDLE=00, CONTROL=01, DATA=11 and ERROR=10. The other 32 bits are data bits associated to the symbol.
An IDLE symbol is transmitted when there is nothing else to transmit. A PCS layer can send this to indicate that the link is up and running but there is nothing

to process. It does not have any associated symbol data so the data bits are not used for this type.
A CONTROL symbol is 24 bits and follows the description of a short control symbol described in the standard. The 24 bits are left adjusted in the 32 bit of symbol data.
A DATA symbol contains a 32-bit fragment of a RapidIO packet.
An ERROR symbol is used to indicate that the PCS layer encountered data on the physical transmission channel that could not be mapped to a valid symbol. It is never transmitted, only received by RioSerialReceiver. It does not have any associated symbol data so the data bits are not used for this type.

The symbol concept is not clear in the standard but it is not that hard to see that is is there. It was however necessary to clearly define it when the PCS functionality was moved out from the channel independent parts.

RioSerial is divided into RioSerialTransmitter and RioSerialReceiver and two control-symbol FIFOs in between.
RioSerialTransmitter splits packets into data-symbols, inserts control-symbols to delimit them and keeps track of packet-accepted control symbols received from its link partner. Idle-symbols are transmitted when no packet is available to send.
The RioSerialReceiver reassembles the data-symbols into packets at the other side and checks the checksum. Errors are reported to its link partner using packet-not-accepted control-symbols. The protocol will thereby provide a "secure" transmission path to its link partner. Any error results in a retransmission.

The RapidIO specification allows for a transmission window to increase the performance of the link. That means that up to 32 packets can be transmitted without receiving a packet-accepted. The RioSerialTransmitter has support to handle the full size of a transmission window.

There are two ways of dealing with link flow control in the standard. This module implements the receiver controlled flow control mechanism. All packets are accepted until the reception buffer is full. Then a packet-retry is issued and all packets are discarded until the retry situation is acknowledged. The packets may then be retransmitted and hopefully be accepted.

Link initialization follows the procedure described in the standard. It waits for the successful reception of eight status control-symbols without errors in between. Once received, the link is considered initialized and packets may be transferred. The transmitter has to send at least 16 status control symbols before it can start sending packets. To speed up the initialization, the number of idle symbols in between the status control symbols is reduced to 16 instead of 256 at startup.

The timeout value waiting for a packet-accepted control-symbol in the transmitter is input using the portLinkTimeout_i.

4.1 Limitations

- Handles short control symbols only.
- No STYPE0 and STYPE1 merging.
- Not all bits in the port control registers described in the standard are supported.
- All inbound packets are checked for errors but if an error occurs internally, inside the device, a packet will be retransmitted forever A mechanism to protect against this should be implemented.
- The embedded in-frame CRC placed in bytes 80 and 81 are not checked.
- Cut-through switching is not supported but could quite easily be implemented.