XAPP639 (v1.0.1) March 31, 2004

HyperTransport Lite Interface for Virtex-II FPGAs

Summary

HyperTransport is a high-speed bus designed to move data from processors to peripherals at speeds up to 60 times faster than a 32-bit PCI bus operating at 66 MHz. The HyperTransport bus provides this performance enhancement while remaining compatible with PCI. A minimal version of the HyperTransport protocol called HyperTransport Lite has been developed and is described in this application note. The reference design is implemented in a Virtex™-II device and can run at a frequency of up to 400 MHz.

Introduction

The HyperTransport Lite reference design is a *minimal slave* interface for use at the end of a HyperTransport chain. Data can be written to the HyperTransport Lite interface using a part of the protocol known as posted writes. Posted writes are defined as write packets with set posted bits. The write request will not receive a response from the receiver if a posted bit is set. Thus, the buffer of the requester can be unallocated as soon as the write is transmitted. By initiating a posted write, data can be read from the HyperTransport Lite interface. It is also capable of generating interrupt packets. Currently, the reference design only supports an 8-bit link width running at 400 MHz. The HyperTransport Lite interface is compatible with HyperTransport devices, however, some features of the full HyperTransport protocol are not implemented in this design.

The purpose of HyperTransport Lite is to allow large amounts of data to be transferred from processors to peripherals, similar to a HyperTransport interface but with less overhead than a complete HyperTransport core. The HyperTransport Lite interface is designed to use as few resources as possible when fitted into an FPGA.

Interrupt packets are generated on the predefined events to report the status of the device. Designers can pre-program the vector and destinations associated with the vectors.

Functions

The HyperTransport Lite interface will perform following functions:

1. Link initialization
2. Link sync-error detection
3. CRC frame detection and insertion
4. CRC generation or checking
5. Packet framing or de-framing
6. Command decoding and generating
7. Buffering commands and data packets (for posted writes only)
8. Buffer releasing
9. End-of-chain responding

Functional Layers

The HyperTransport Lite interface has three layers of operation (link, node, and device). The lowest layer is the link layer. It carries an 8-bit HyperTransport link, takes care of CRC framing, generation and detection. The link layer also does link initialization and sync error detection.

The node layer performs command encoding, decoding, end-of-chain handling, and buffer management.

The device layer is the user layer. It connects to the intended function on the device. On the receive side, it decodes the HyperTransport command, 38-bit address, 32-bit or 64-bit data, 4-bit or 8-bit byte masks, and 1-bit control/status. On the transmit side, it takes a 40-bit address, 4-bit count, 32-bit or 64-bit data fields, and 3-bit control/status. Since the device only does posted double-word writes to double-word aligned addresses, it discards the lowest two address bits. Table 1 shows the complete architecture.

Table 1: **HyperTransport Lite Device Architecture**

<table>
<tr><td colspan="6">Address Decode
DATA BUFFER
RETURN DESCRIPTORS
Interrupt Registers</td><td>User Functions
Interrupt Generator
Transmit Machine</td><td>Device Layer</td></tr>
<tr><td>Address [39:2/0]</td><td>Data [63/31:0]</td><td>Count [3:0]</td><td>B Mask [7/3:0]</td><td>Control [1:0]</td><td>Status [1:0]</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td>Frame / de-frame
End-of-chain
Buffer Manager
Command Snooper</td><td>Node Layer</td></tr>
<tr><td colspan="6">PWR FIFO /PWRD FIFO
NOP RDREQ NON PW LATCH</td><td></td><td></td></tr>
<tr><td colspan="6">8-32 Demultiplexer/Multiplexer</td><td>Initialization
Sync Error Detection
CRC Framing
CRC Generation</td><td>Link Layer</td></tr>
</table>

Link Layer

The link layer carries the data path to the upper layers through a link synchronization unit and a CRC unit. The data path can be divided into a receive and a transmit path where the receive block accepts packets from a downstream device and the transmit block sends packets upstream.

CRC Unit

A cyclic redundancy check (CRC) receive is generated from the receive bits according to the HyperTransport specifications and compared against the CRC frame.

A CRC transmit is generated according to the HyperTransport specifications and transmitted every 512 bits.

The CRC uses the following polynomial:

$$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^{8} + X^{7} + X^{5} + X^{4} + X^{2} + X + 1.$$

More information is available in the HyperTransport specifications.

Node Layer

The Node layer has a packet deframer, a framer end-of-chain unit, and buffer managers. The block diagram in Figure 1 describes the relationship of individual units within the node layer.

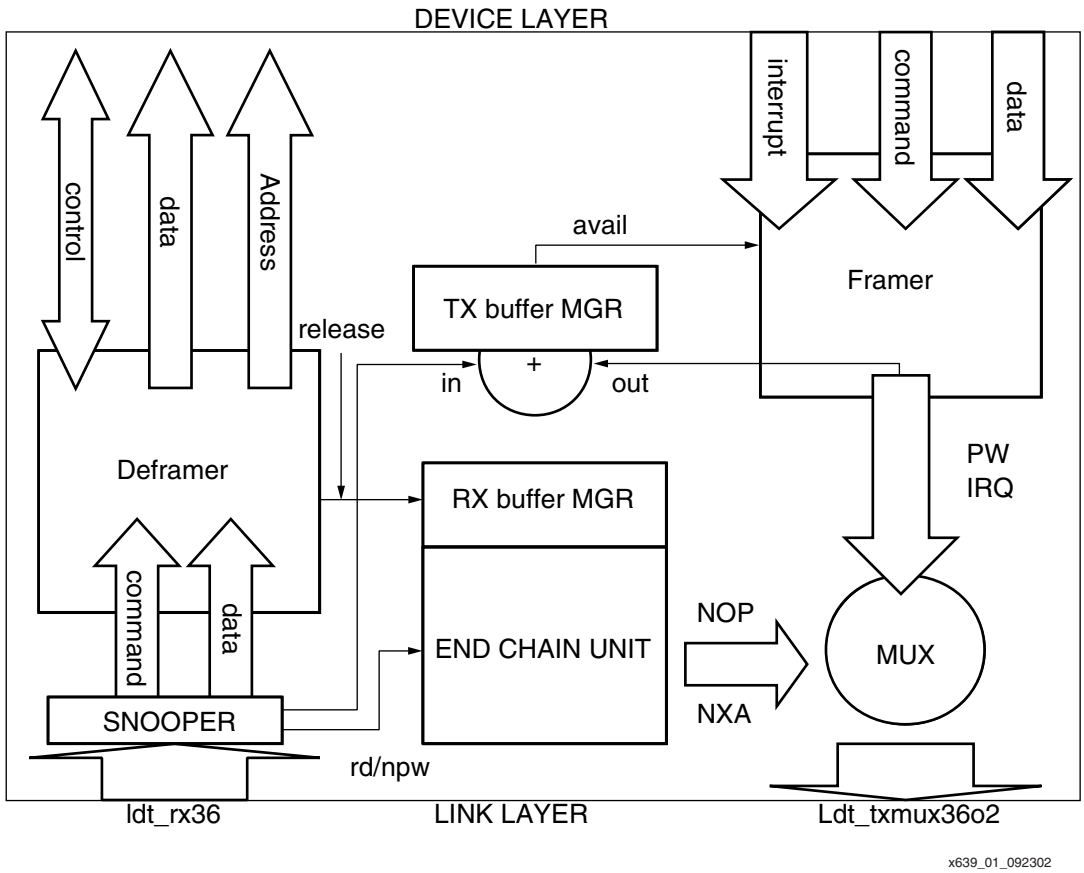

Figure 1: **Relationship of Individual Units in a Node Layer**

Packet Framer/Deframer

On the receive side, the deframer monitors the content of the received packet, extracts one packet at time from the receive FIFO, and submits it to the device. It also ensures that data packets are correctly attached to their command packets. On the transmit side, the packet framer generates posted double-word writes and posted byte writes.

End-of-Chain Unit

The end-of-chain unit generates the Read response signal for read requests and the Target Done signal for the non-posted writes to conform to HyperTransport specifications. It can also generate data packets with all ones of a requested size. If a Read response is received, the end-of-chain unit will discard the associated data packet since the HyperTransport Lite interface cannot handle a Read response. The intent of responding to a Read is to prevent configuration software from freezing up.

Buffer Manager

The transmit and receive units have separate buffer managers. Although the device does not take commands other than posted writes, it will logically maintain the buffer count of one for all other buffer types. It will maintain at least three buffers for posted write data and command. The buffer release manager will issue three posted command and data tokens after reset.

Every time a command is decoded or a packet of data is removed from the queue, a posted command and data buffer release NOP packet is generated.

Every time a buffer release is received at the receiver, the available buffer count is updated and will take a maximum of seven buffer releases.

Receiver Data Path

The receiver unit contains an 8-bit HyperTransport link and an 8-to-32 bit demultiplexer. As the packets are coming through the receiver unit, they are forwarded to the snooper unit. The purpose of the snooper unit is to determine if the incoming packet is a NOP packet. If it is, the snooper unit latches the *buffer-release tokens*. If the incoming packet is a posted write request packet, it is pushed onto the HyperTransport *command* FIFO. Otherwise, the data is ignored.

The HyperTransport Lite interface supports posted write requests only and does not support the read requests and non-posted write requests. This is one of the differences between HyperTransport Lite and the HyperTransport protocol. However, since the HyperTransport Lite interface is compatible with the HyperTransport protocol, appropriate flags are set whenever those commands are received. The HyperTransport protocol also requires returning all ones of a requested size. Therefore, the count field of a read request is also latched and forwarded to the end-of-chain unit. All data packets following posted writes are sent to the *data FIFO* in the device layer. Other data packets are simply dropped.

The data path is always connected to the CRC checker circuit. This CRC circuit keeps track of the CRC frame time and performs CRC checking according to the HyperTransport specification version 1.0.3 (**http://www.hypertransport.org/**)

As data progresses through the command and data FIFOs, it is fed into the deframer unit to determine if a data frame has byte masks. All fields are latched into separate registers.

This reference design contains switches to choose between big or little Endian format for data sent to the HyperTransport Lite interface, and a 32-bit or 64-bit internal data path. Eight byte-mask bits are provided for future use and are grouped into two 4-bit fields for each double word (32 bit). Bits in each group share same value.

Transmitter Data Path

The transmit data path has a 40-bit address, a 4-bit count field, and 32 or 64 data bits from the device layer along with some control signals.

The HyperTransport Lite command can either be a posted write request to a 40-bit addressed destination, an interrupt request packet, or an internal response packet. To conform to the HyperTransport protocol, response packets are generated in response to a read request or a non-posted write request. The size of a data packet can be up to one cache line, or 32 bytes per posted write request. Either 32-bit or 64-bit wide data can be arranged to big or little Endian format.

An interrupt packet is a posted write byte request with user-defined vector and destination fields. The vector and interrupt destination fields must be programmed before using the device. An end-of-chain unit generates the response packets to conform to the HyperTransport specifications.

There is a CRC unit to monitor CRC frame time and CRC bits. The transmitter generates CRC frames and inserts CRC frames according to the HyperTransport specification.

The buffer manager monitors available upstream and downstream buffers between the next hop device. Upstream buffer counts are updated whenever the receiver gets a valid NOP with a non-zero buffer release. Whenever a downstream packet is processed only count fields for posted write request fields, posted write request data, and downstream buffers are updated. Upstream includes all types of tokens since the device must keep at least one each buffer count to avoid deadlock of the channel.

User-Defined Layer

Device Layer

The device layer is modified by the user to performs the required tasks. At a minimum, it maintains the return descriptor FIFO to retain the necessary information to send data out of the device. The FIFO must store the return address and size of each unit of data plus any optional flags. The device layer must also have a set of event associated interrupt registers. This section of the application note describes all the necessary blocks of the device layer. The implementations are examples of how to design a device layer.

The device layer manages the following functions:

1. Address decoding
2. Accepting writes
3. Retrieving descriptors, issuing a series of writes
4. Generating interrupts
5. Routing data traffic

Return Descriptor

As data packets are sent from the FPGA, the information on where to send them and the size of the data packets is stored in the return descriptor FIFO. Return descriptors can be in the form of a FIFO or as a set of registers. This reference design uses a FIFO structure.

According to HyperTransport protocol, the return address field must be in the full 40-bit format. The two LSBs are ignored since the device only performs posted double-word writes. The device must break the data block into small packets of 32 bytes or less.

In this reference design, a 64-bit descriptor is used. It contains 40 address bits, an 18-bit count field, and a 6-bit flag field.

Interrupt Registers

Interrupt registers are 16-bits long. There are 16 registers associated with 16 IRQ pins in the node layer. Each register is divided into an 8-bit vector field and 8-bit interrupt destination bits. These bits are assembled into the interrupt packets by the packet framer in the node layer.

Address Decoder

The device layer further decodes the address to determine if a Write is for the descriptor FIFO, the data FIFO, or the on board registers (such as the interrupt registers). The address decoder can ignore unused addresses.

Programming Rules

A CPU can write to the port using un-cached accelerated writes or using the data mover to DMA packets out of memory. Un-cached accelerated writes allow a bus unit to merge write packets. Some rules should be observed in using the device for data streaming.

- There must be a sufficient amount of data in the data buffer/FIFO before writing a descriptor to keep the data FIFO from underflow.
- The HyperTransport Lite unit can mix descriptor writes with the data mover writes of earlier packets.
- Software must keep track of outstanding descriptors to avoid FIFO overflow.

Device Layer Operation

Receiver

The address decoder in the device layer decodes the address from the receive address field in the descriptor FIFO. Whenever the write bit is true, if a function is selected, the receiver determines the amount of data to be transferred. Writes can be double-word sized and up to one cache line (eight double words or 32 bytes). If writing to an interrupt register, it can be double word or quad word. Extra bits are ignored. If an address does not select any device layer commands, the command is ignored.

One set of control and status registers exists for transmit and receive channels. These bits are signals between the device layer and the node layer. Control signals originate in the device layer and status signals originate in the node layer. These signals are used to interface the node layer and the device layer functions.

Receiver Control and Status

Table 2 shows the receiver status signal. Table 3 shows the receiver control signal. If the device is idle, it asserts RX_NXCMD to allow the deframer to submit a command. A valid command is recognized if RX_WR_L is Low. Both command and data are then available to the device. The device can de-assert RX_NXCMD to enter a busy state. Otherwise an address decoder generates register selects and the register loading takes place. The node layer may hold RX_WR_L to repeat writes if RX_NXCMD is High. Figure 2 shows the receiver waveforms.

Table 2: **Receiver Status Signals**

Signal Name	Status Name	Description
RX_WR_L	Write Request	A posted Write request received

Table 3: **Receiver Control Signals**

Signal Name	Command Name	Description
RX_NXCMD	Get Next	Set to Low to enable framer to unload next command packet from HyperTransport Lite queue

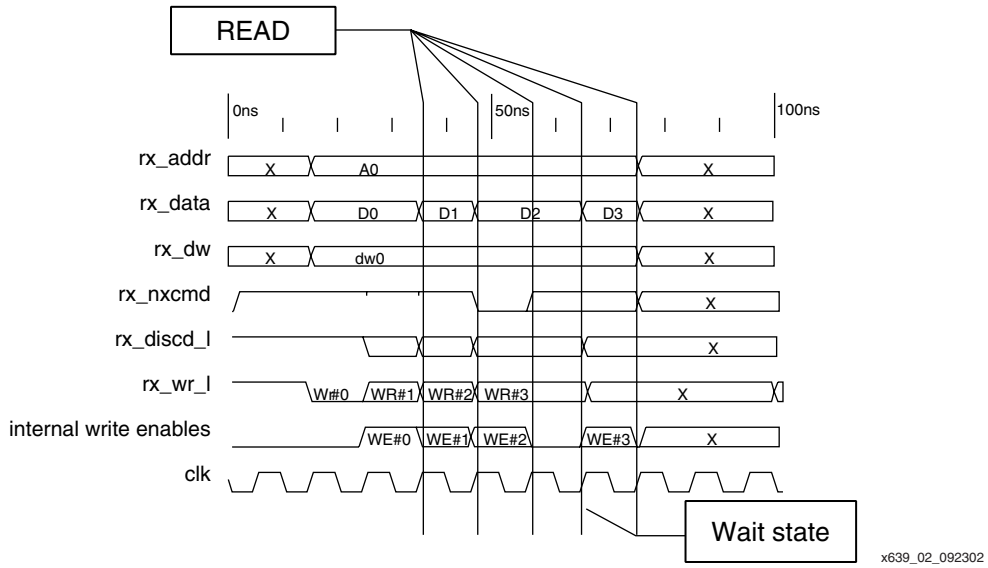

Figure 2: **Receiver Waveforms**

Transmitter

Whenever a block of data is ready to be sent back to the host, the transmit state machine retrieves the descriptor from the FIFO. It then determines if it is necessary to generate multiple writes.

For packets less than 32 bytes long, the transmitter loads the transmit address from the FIFO address field, sets the count field, and lets the packet framer go. For longer packets, it breaks the transfer into multiple writes where the number of writes is derived from the count field bits [17:6]. The transmitter repeats writes while decrementing write count and adjusting the address.

Special cycles must be generated if a return buffer start address is not cache-line aligned. The transmit machine writes out all the words to the next aligned address and continues writing the rest of data block.

Only if the buffer manager reports available posted request channel buffers at the next hop, can the transmitter issue a Write command. The device layer must ensure sufficient buffering within itself to handle worst-case traffic.

Transmitter Control and Status

Whenever there is a block of data to send, and the TX_IDLE is High, the transmitter state machine starts by popping the descriptor FIFO. When TX_IDLE is High, at least one posted-write request packet and a data packet up to 32 bytes can be sent. The first write takes place three clocks after sensing TX_IDLE High. The device layer uses these clock times to get the return address, count, and data. It then asserts TX_WR_L and TX_DS_L on the following clock.

TX_WR_L loads the address and count fields into the framer. TX_DS_L loads the data into data buffer. If the device has more than 32 bits or 64 bits of data, it holds TX_DS_L for up to four clocks in a 64-bit system, eight clocks in a 32-bit system, or until the double word or quad word count is zero.

After the whole write packet is sent to the node layer, the device goes into the "next packet" state where it updates address and count fields. If the node layer still has available write buffers, it will assert TX_IDLE at this time. If TX_IDLE is High, the device continues sending the next write packet on the following clock. The transmitter repeats generating write packets until the computed write-packet-count field is zero.

Table 4 and Table 5 show the transmitter status and control signal definitions. Figure 3 shows the transmitter waveforms.

Table 4: **Transmitter Status Signal Definition**

Signal Name	Status Name	Description
TX_IDLE	Posted write request buffers available	At least one PWR and PWRD buffer is available. If this bit is zero the device must not assert TX_WR_L.

Table 5: **Transmitter Control Signals Definition**

Signal Name	Command Name	Description
TX_WR_L	Generate Write Packet	Set to Low to let framer go. Loads count field and address fields.
TX_DS_L	Data Strobe	Device sets this bit to load more data.

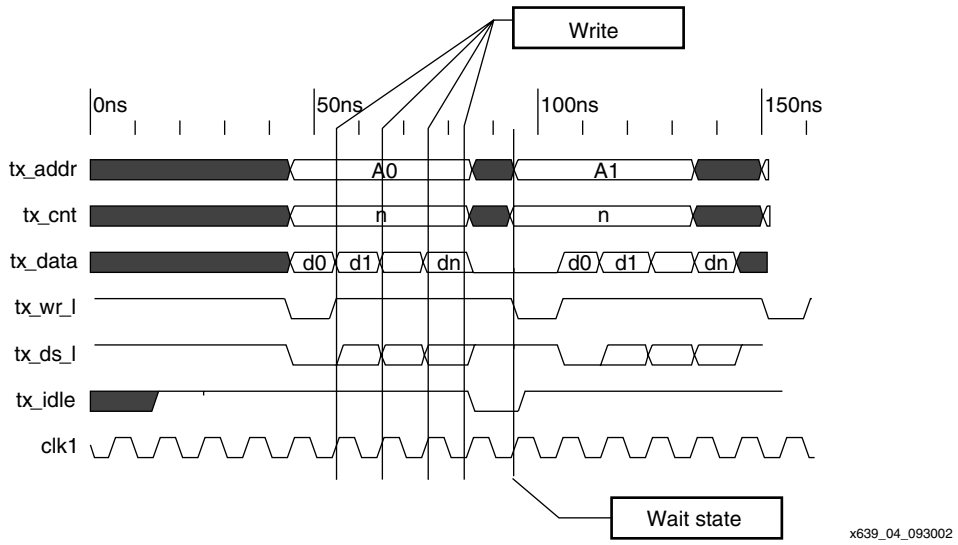

Figure 3: **Transmitter Waveforms**

Device Layer Interrupt Mechanism

Interrupts can be generated to report the status of the device. At the least, there should be interrupt signals defined to alert the system when the descriptor FIFOs are Full and Empty. In this reference design, 16 registers and interrupt pins. An interrupt event will set a bit in the interrupt pin field and cause the device layer to generate an interrupt packet using the corresponding interrupt register. The interrupt register must be pre-programmed with the appropriate vector number and destination field. If the interrupt register that is used is not initialized, device will generate a non-maskable interrupt (NMI).

Interrupt Signals

Interrupt events are generally generated by the device layer (Table 6). A device layer may submit a vector to the node layer to generate an interrupt packet. An interrupt vector is 16 bits: an 8-bit vector number and an 8-bit interrupt destination. It checks if the interrupt FIFO is not full in the node layer by polling TX_NOINTR. If TX_NOINTR is High, the device raises the TX_LD_IVECT to load the 16-bit vector TX_IVECT.

Every interrupt register has an extra bit to reset on LOGIC_RESET_L and set on write hits. This bit is fed to the TX_NMI signal when a corresponding interrupt occurs. If this bit is Low, NMI is generated, and the host ignores the vector bits.

If producing more than one interrupt at a time is likely, the device should also implement an interrupt arbitration mechanism. Interrupt logic must ensure that only one interrupt packet is generated for an event, otherwise it will flood the node layer with interrupt request (IRQ) packets. The interrupt logic is shown in Figure 4.

Table 6: **Interrupt Signal Definitions**

Signal Name	Command Name	Direction from Device	Description
TX_NMI_L	Non-Maskable Interrupt	Output	If this bit is clear, a NMI packet is generated instead of a fixed physical type.
TX_IVECT[15:0]	Interrupt Vector Number Bits [15:8]	Output	If corresponding register is preprogrammed, it will generate a fixed interrupt in physical mode. Otherwise, an NMI will be generated.
	Destination Field Bits [7:0]		
TX_LD_IVECT	Load interrupt vector	Output	Signal to node layer to load the TX_VECT bits.
TX_NOINTR	No interrupts pending	Input	Set High if the interrupt packet framer is idle and ready to receive a new vector.

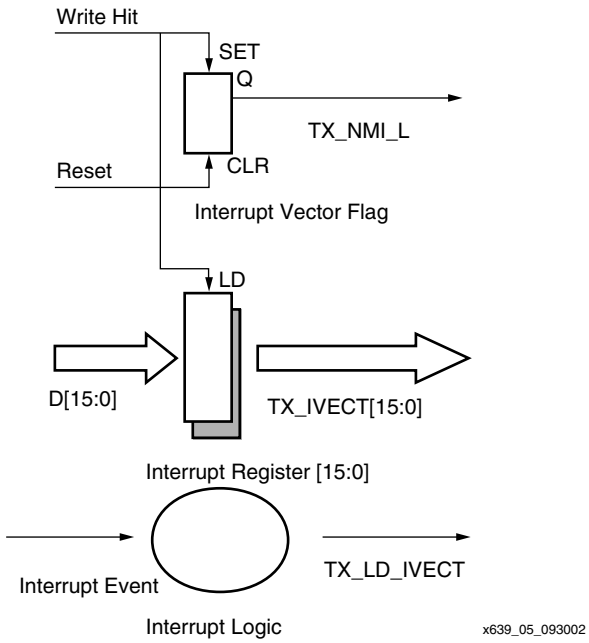

Figure 4: **Interrupt Logic**

Clock Requirements

There are two major clock sources, the HyperTransport receive clock (driven by the adjacent HyperTransport device), and the FPGA clock (generated on board). The HyperTransport specification requires all nodes to run at least at 200 MHz. This reference design uses two DCMs and four global clock lines. The reference design will run at a 400 MHz link speed with a 200 MHz core clock.

The link layer divides the LDT_RX_CLKI clock by two to generate clock CLK100 for the CRC checker circuitry. The transmit CRC circuit uses CLK100 and it is multiplied by two for the TX clock. The two times TX clock also has a 180° phase-shifted version to drive the FDDR cells.

Receive Clocks

The receive clock LDT_RX_CLKI will clock in the command-address and data bus (CAD) and the control (CTL) bits into a receiver.

This clock is driven from the next-hop node and is fed to a clock phase adjustment circuit (aligns the phase of the clock to the data). The phase adjusted version of receive clock is called LDT_CLK200.

Receive CLK100

As the internal data path is 32-bits wide, it can be processed at half the speed of LDT_RX_CLKI. CLK100 is the name of the clock that drives the internal data path. It drives the latches, the command snooper, and the FIFO input as well as the NOP latch and the other-commands latch.

Transmit Clocks

Transmit CLK100

A copy of a 100 MHz clock to drive the CRC unit, the link synchronization unit, and the transmit slot machine. It is generated from CLK100 and can also be generated from the FPGA clock DEV_CCLKG.

LDT_TX_CLKO

The transmit clock is derived from the phase-adjusted input clock. It is used to drive the HT_TX_CAD/CTL IOBs. It is a 180° apart two-phase clock. It is required for the FDDR cells.

As the clock is in the middle of the data window, the clock driving the LDT_TX_CLKO pin has to be phase shifted 90° from the clock driving DATA FDDR cells. It is driven through another FDDR cell to ensure a clock-to-data phase relationship is maintained at the pins. This clock is also a dual phase clock (separated by 180°).

DEV_CCLKG

DEV_CCLKG is a 200 MHz clock used by the node layer. It clocks deframer, framer, and the end-chain unit.

The device clock DEV_CLK may be 100 MHz if a 64-bit data path is used, and 200 MHz with a 32-bit data path. Figure 5 shows all the clock domain descriptions.

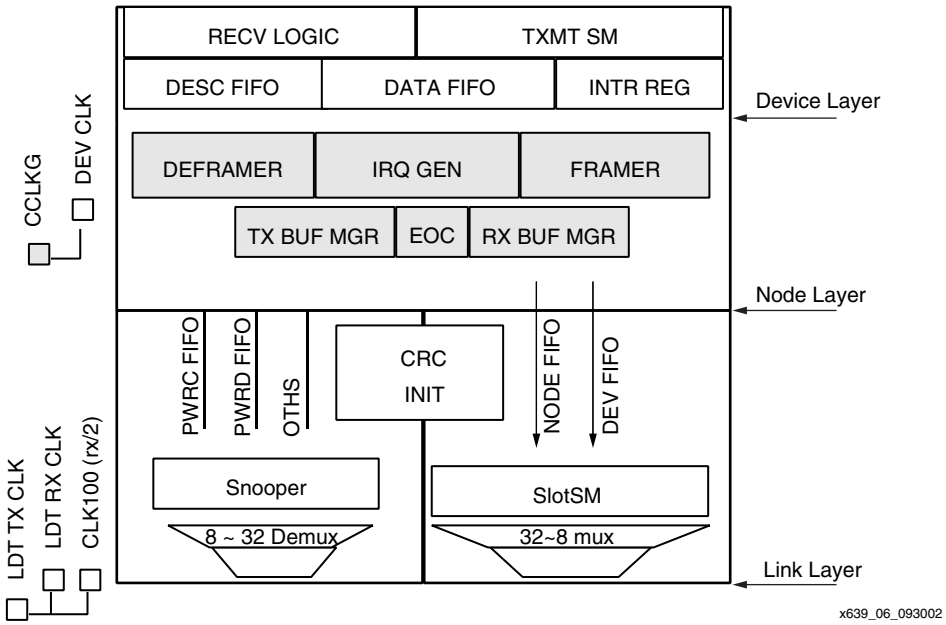

Figure 5: **Clock Domains**

Reset

There are two resets, FPGA reset and HyperTransport link reset. The FPGA reset asserts the LOGIC_RESET_L signal to the logic to reset FPGA registers and state machines to a known state. The HyperTransport reset follows the HyperTransport specifications. It has two HyperTransport signals, LDT_RESET_L and LDT_PWROK. LDT_RESET_L is held Low during FPGA reset.

When HyperTransport link is in reset, there is no valid clock signal on LDT_RX_CLK, so the DLL must be held in reset mode. After LDT_PWROK is deasserted, the DLL will be released from reset, re-synchronized to its input clock (LDT_RX_CLK), and will also restart the phase detection logic as described in **Receive Clocks**.

LDT_RESET_L resets the receive command snooper and transmit slot state machines and all related latches. LOGIC_RESET resets the node layer and the device layer.

The device can also be reset in software by writing to an address, `e0_00ff_0000` in this reference design.

Signal Definition

Table 7 and Table 8 describe signals for the HyperTransport Lite interface.

Table 7: **HyperTransport Lite Top Layer Signal Definitions**

Signal Name	Pin Type	Description
LDT_RX0_CTL	DDR-DS[1]	LDT receive CTL pin
LDT_RX0_CAD[7:0]	DDR-DS[1]	LDT receive command/data pins
LDT_RX0_CLK	DDR-DS[1]	LDT receive clock
LDT_TX0_CTL	DDR-DS[1]	LDT transmit CTL pin
LDT_TX0_CAD[7:0]	DDR-DS[1]	LDT transmit command/data pins
LDT_TX0_CLK	DDR-DS[1]	LDT transmit clock
LDT_RESET_L	OD/PP LVCMOS[2]	LDT reset
LDT_PWROK	OD/PP LVCMOS[2]	Power OK

Notes:

1. DDR-DS = dual data rate differential signaling according to HyperTransport electrical specifications version 0.77.
2. OD/LVCMOS = Open Drain signal is used, LVCMOS signal allowed for compliance. Must have pull-up to 2.5V.

Table 8: **Device Layer Signal Definition**

Signal Name	Direction from Device Layer	Description
Transmit Side (Uplink)		
TX_Addr[39:0]	Output	Address bits
TX_Data[31:0]	Output	Data bits, byte lane order depends on jumper setting Endian.
TX_Data[63:32]	Output	More data bits if 64 bit is selected by jumper. Byte lane order depends on jumper setting Endian.
TX_CNT[3:0]	Output	Count field value for the Write packet.
TX_WR_L	Output	Device sets Low to start a Write cycle.
TX_DS_L	Output	Device sets Low to pass more data to node layer.
TX_IDLE	Input	Node layer is ready to receive write.
Interrupt		
TX_IVECT[15:0]	Output	IRQ pins, vector and interrupt destination will be extracted from pre-programmed registers.
TX_LD_IVECT	Output	Load vector
TX_NMI_L	Output	If clear, NMI will be generated instead of a fixed type.
TX_NOINTR	Input	IRQ acknowledge, Resets the IRQ FF in device.
Receive Side (Downlink)		
RX_Addr[39:2]	Input	Address bits
RX_Data[31:0]	Input	Data bits, Endian based on jumper setting.
RX_Byte Mask[3:0]	Input	Four mask bits are provided for future use. All bits carry same value. A High indicates valid data.
RX_Data[63:32]	Input	More data bits if 64 bit is selected by a jumper. Byte lane order depends on Endian.
RX_Byte Mask[7:4]	Input	More byte mask bits, if 64 bit mode is selected by jumper. Four mask bits are provided for future use. All bits carry same value. A High indicates valid data.
RX_NXCMD	Output	Sets to get next command reset to enter busy state.
RX_WR_L	Input	Valid Write command strobe
RX_DW	Input	Bit 2 of Write command 1 = Double word 0 = byte sized.

Reference Design

Table 9 lists the device utilization of the HyperTransport Lite reference design. The HyperTransport Lite interface reference design is available on the Xilinx site at: http://www.xilinx.com/bvdocs/appnotes/xapp639.zip. It includes Verilog source code, constraints files, and a pre-synthesized EDIF. The design was tested on a Virtex-II XC2V1000FG256-6 at 400 MHz link speed.

Table 9: **HyperTransport Lite Reference Design Device Utilization**

Tested link speed	400 MHz in a Virtex-II XC2V1000FG256-6 200 MHz in a Virtex-II XC2V1000FG256-4
Number of slices	1900
Number of DCMs	2
Number of global clocks	4

Conclusion

The HyperTransport protocol is designed to deliver a high-performance and scalable interconnect between CPU, memory, and I/O devices. The protocol is engineered to operate with a top signaling rate of 1.6 GHz on each wire pair supporting a peak aggregate bandwidth of 12.8 GBytes/s. However, when implemented in an FPGA, the full-sized core takes up nearly 7,000 slices, or nearly half of a Virtex-II XC2V3000. A simplified version of the HyperTransport core, called HyperTransport Lite, has been developed as a reference design to perform the basic data transfer functions of the original HyperTransport core. This HyperTransport Lite reference design is compatible (although not fully compliant) with the HyperTransport specifications albeit using less device resources. The XAPP639 reference design currently uses less than 2000 slices and occupies about one-quarter of a Virtex-II XC2V1000 device. The design is verified with RTL and full-timing simulations to run at a 400 MHz link speed.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/07/03	1.0	Initial Xilinx release.
03/31/04	1.0.1	Updated the link to the reference design.