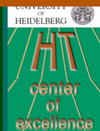
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Heidelberger Dokumentenserver



First International Workshop on
HyperTransport™ Research and Applications

Proceedings

of the

1st International Workshop on HyperTransport Research and Applications

WHTRA 2009

Editors
Holger Fröning
Mondrian Nüssle
Pedro Javier García García

ISBN: 978-3-00-027249-3

February 12th, 2009, Mannheim, Germany

University of Heidelberg, Computer Architecture Group

Proceedings of the
First International Workshop on HyperTransport Research and Applications (WHTRA2009)
Feb. 12th, 2009, Mannheim, Germany

HyperTransport 3 Core: A Next Generation Host Interface with Extremely High Bandwidth

Benjamin Kalisch, Alexander Giese, Heiner Litz, Ulrich Brüning
University of Heidelberg
Computer Architecture Group
{benjamin.kalisch, alexander.giese, heiner.litz, ulrich.bruening}@ziti.uni-heidelberg.de

Abstract

As the amount of computing power keeps increasing, host interface bandwidth to memory and input-output devices (I/O) becomes a more and more limiting factor. High speed serial host interface protocols like PCI-Express and HyperTransport (HT) have been introduced to satisfy the applications' ever increasing demands for more bandwidth. Recent applications in the field of General Purpose Graphic Processing Units (GPGPUs) and Field Programmable Gate Array (FPGA) based coprocessors are an example. In this Paper we present a novel implementation of an FPGA based HyperTransport 3 (HT3) host interface. To the best of our knowledge it represents the very first implementation of this type. The design offers an extremely high unidirectional bandwidth of up to 2.3 GByte/s. It can be employed in arbitrary FPGA applications and then offers direct access to an AMD Opteron processor via the HT interface. To allow the development of an optimal design, we perform a complexity and requirements analysis. The result is our proposed solution which has been implemented in synthesizable Hardware Description Language (HDL) code. Microbenchmarks are presented to show the feasibility and high performance of the design.

1. Introduction

Following Moore's Law, computing power has doubled every 18 months over the last years. While scaling the operating frequency of high end processors has come to an end, exponential gains in computing power are still anticipated through the use of parallel processing. In either case, providing the processor with enough duty will require the increase of I/O bandwidth significantly. In fact, the processor - I/O bandwidth performance gap has increased in the last years [1] making it even more crucial to improve I/O performance.

This fact was first realized by AMD which replaced the outdated front size bus (FSB), that is used to interconnect the CPU, memory and I/O devices, with a novel packet based point-to-point interconnect called HyperTransport (HT) in their Opteron processors. HT offers high bandwidth, extremely low latency [2] and can support cache coherency which makes it ideally suited for communication between CPUs, memory and IO. The high performance of the HT interconnect is the main reason for the much better scalability of Opteron based symmetric multiprocessor (SMP) with non unified memory architecture (NUMA) machines in comparison to Intel Xeon based SMPs [3].

HT supports variable link widths and up to 2 Gbit/s on each lane in protocol version 2.x, also referred to as Gen1. This leads to a maximum unidirectional bandwidth of 4 GByte/s for a 16 bit link. Apart from the Opteron CPUs, HT 2.x is successfully implemented by peripheral hardware devices like the Pathscale network interface [4], Cray's Seastar [5] and the Field Programmable Gate Array (FPGA) based rapid prototyping board [6]. HT devices can directly communicate with the processors without any intermediate bridges using the HyperTransport Extension (HTX) connector [7]. HTX is a PCI-Express slot like standard defined by the HyperTransport Consortium (HTC).

Recently, the HTC introduced HT 3.1, also referred to as Gen3, which increases the supported speeds to 6.4 Gbit/s on a lane equalling a theoretical unidirectional peak bandwidth of 12.8 GByte/s for a 16 bit link. The first integrated circuits (ICs) that will support HT 3.x are the Shanghai Opteron processors, however, no non Opteron implementations are currently available. The reason for this is, that currently no HTX3 capable mainboards are available and the lack of an open source HT3-Core like the HT2-Core [8]. To solve the latter issue, in this paper we present the very first high performance HT3-Core for FPGA implementations. The core provides very high bandwidth even for FPGA imple-

mentations, and therefore presents the ideal building block for high performance next generation I/O devices. Our solution promises to deliver a bidirectional bandwidth of up to 9.2 GByte/s for a 16 bit link. To the best of our knowledge this makes it the fastest host interface implementation currently available for FPGAs.

The rest of the paper is organized as follows. Section 2 will provide background information and define the requirements for an FPGA based HT3 core. Section 3 will present a complexity analysis and describe the challenges of such an implementation. Our proposed architecture is presented in Section 4. It is followed by an evaluation in Section 5 and we draw a conclusion in Section 6.

2. Background

To define the requirements of an HT3 core a short introduction to the HT protocol will be given. The HT specification defines the entire protocol stack ranging from the physical layer up to the transaction level layer. The physical layer defines the electrical parameters which have to be adhered by HT device implementations and include jitter, slew rate and common mode characteristics. Physical layer compliance is already provided by the physical layer device (PHY) and therefore out of scope of this paper. The PHY also takes care of serialization/deserialization (SERDES) of the high-speed serial data stream. For signalling HT defines 2, 4, 8, 16 and 32 bit command-address-data (CAD) busses which are accompanied by a set of control (CTL) lanes and clock (CLK) lanes. Most common are 8 or 16 bit configurations, whereby multiples of 8 CAD lanes, one CTL and one CLK lane are considered as a link. A link connects exactly two endpoints whereas switches have to be employed to realize topologies of multiple endpoints.

The transaction layer defines the packets which are transmitted over HT links. A transaction consists of a command packet and an optional data packet carrying 1-16 doublewords (32 bit) which allows to send maximum sized transactions of 64 Byte. This size is equivalent to a cacheline on current x86 systems. In Gen3 mode each transaction is also appended with a CRC. The specification defines a large number of commands with the main purpose of data movement. Therefore, write, read and response operations are defined. To avoid deadlocks, which may be caused by cyclic dependencies from split phase transactions, the different commands are assigned to different virtual channels (VC). This allows reordering of the data stream and breaking up cyclic dependencies. Furthermore, the commands implement low level functionality like flow control and fault tolerance. The required functionality which has to be provided by an HT3-Core implementation is therefore as follows:

- Packetization: Extracts Transactions from the data stream and sorts them into their according virtual channel queue and vice versa.
- Flow control: HT defines a credit based flow control which has to be supported by the core.
- Fault tolerance: HT3 defines an advanced CRC mechanisms for increased reliability
- Scrambling: To support clock data recovery in Gen3 mode, the data stream is scrambled.
- Low level initialization methods

As the HT2-Core presented in [8] implements the same functionality according to the HT2 specification and HT3 is downwards compatible, it is reasonable to analyze, whether a modified HT2-Core would be a sufficient solution. Therefore, it is useful to examine the novel features which have to be supported by Gen3 devices. The most important modifications are the increased frequency support of up to 3.2 GHz and the enhanced fault tolerance mechanic. Additionally to the periodic CRC, which can be used to detect, but not to correct errors, HT3 introduces a retry mechanism with per-transaction CRC. Every transaction sent out by the transmitter is appended with a CRC and stored into a retry buffer. On reception the receiver calculates the CRC again and in the case of a successful match sends an acknowledge back. In the case of a mismatch a nack packet is generated which leads to a retransmission of the original transaction. Implementation of the retry mechanism requires heavy modification of the HT2-Core. Even more significant, however, is the increased bandwidth that has to be supported internally. The HT2-Core supports an internal data width of 64 bit which requires an internal core clock frequency of 600-1600 MHz for a 16 bit link at Gen3 frequencies, and a frequency of 300-800 MHz for a 8 bit link.

Last but not least is the introduction of a new data sampling scheme for Gen3 devices. Instead of the source synchronous mechanism sampling incoming data with the link clock, HT3 devices use a clock data recovery (CDR) technique. The CDR circuit recovers a dedicated clock for each lane and uses it to sample the data. As static data patterns occurring in IDLE phases prohibit reliable clock recovery, a data scrambling mechanism is used in Gen3 mode.

The required change to a 128 bit internal interface and the additional modifications regarding retry mode demand for a complete redesign of the HT2-Core.

3. Complexity analysis

As mentioned before HT3 is a packet-based point to point interconnect, which operates on a minimum packet size of 32 bit, one doubleword (DW). To support the provided bandwidth of HT3 an analysis of the data stream is required.

The data stream of HT3 can be distinguished into three different DW types which are command (CMD), data (DATA), and a cyclic redundancy check (CRC) checksum. To keep the decoding of the data stream as simple as possible, an internal data width of 32 bit would be ideal, so every clock cycle one of only three different types of DWs must be interpreted. To support higher bandwidth on the HT link, the data stream has to be parallelized which leads to wider internal data buses, as the maximum frequency is the limiting factor in FPGAs. Due to the fact that the HT3 protocol does not allow all combinations of different DW types, the complexity does not increase quadratically, but as can be seen in Figure 1, the increase in complexity is significant. Multiple consecutive command DWs may belong to two separate command packets without a separating CRC due to command packet insertion. The number of combinations for a 256 bit wide data path is not depicted due to the large number of possible combinations.

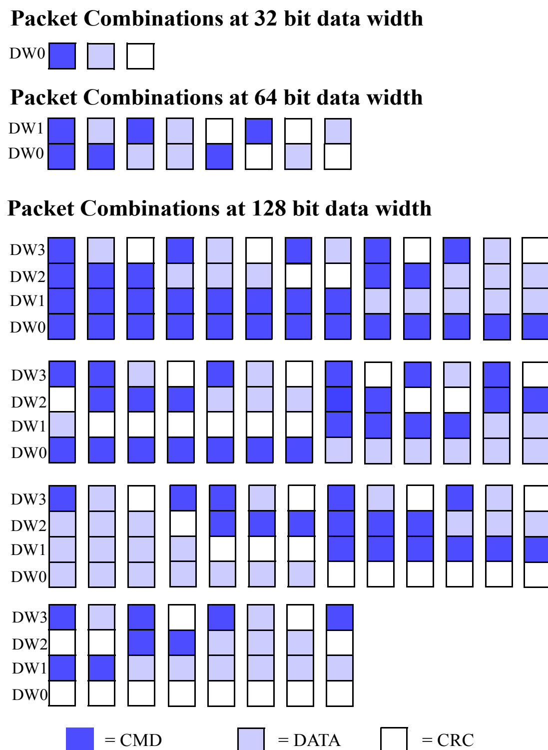


Figure 1: Complexity growth

HT3 has a minimum link frequency requirement of 1.2 GHz. Depending on link width and parallelization degree this results in different possible core frequencies shown in Table 1.

Table 1: Internal clock frequencies

At 1.2GHz link frequency		External Link Width	
		8bit	16bit
Internal Link Width	32bit	600MHz	1200MHz
	64bit	300MHz	600MHz
	128bit	150MHz	300MHz
	256bit	75MHz	150MHz

The target device is a state of the art Virtex 5 FPGA which can be clocked at a theoretical maximum frequency of 550 MHz for the core logic. For a design that contains complex logic as the HT3-Core a core frequency of 300 MHz is difficult to achieve but possible. This reduces the possible combinations of link width and parallelization degree that can be realized.

In the HT3-Core design an internal data width of 128 bit is implemented, as it provides the best combination between feasible core frequency and logic complexity. The core logic mainly consists of multiplexing structures which sort the DWs to form complete packets. Analysis of these multiplexing structures has shown that two different factors influence the reachable frequencies of such multiplexers. One is the number of the input bits of the multiplexer, the other is the number of control signals of the multiplexers. Figure 2 shows that increasing the multiplexer width reduces the achievable core frequency significantly. Doubling the width from two to four doublewords reduces the operating frequency by almost 100 MHz.

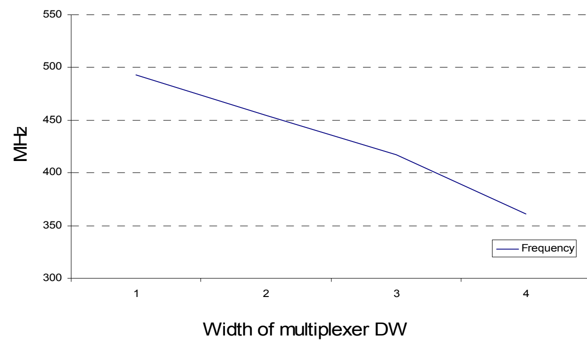


Figure 2: Multiplexer width influence

Increasing the number of control signals of the multiplexer also reduces the maximum operating frequency. This is shown in Figure 3, where all other parameters besides the control signals remain unmodified.

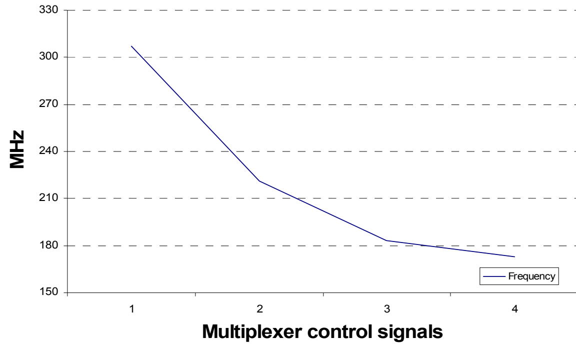


Figure 3: Multiplexer control signal influence

To handle all different traffic types, a multiplexer width of 128 bit which selects between two sets of seven DW wide registers is needed. The input and output path is four DWs wide, and if only one DW can be forwarded three DWs must be stored. If the multiplexer width would be increased to 256 bit the above explained factors for frequency decrease would take effect. Obviously, the input width would have to be doubled and also the number of control inputs would have to increase, as the decoding complexity increases due to different cases that have to be handled. These two combined factors result in a much larger amount and deeper hierarchy of multiplexers inside the design. Thereby the routing of the control signals to all multiplexers becomes so difficult that routing delay and fanout get extremely high and reduces the reachable frequency. This reduction outweighs the advantage gained through doubling the data path, which is a reduction of the necessary core frequency by a factor of two.

These results show that an internal data width of 64 bit is not sufficient to reach a clock frequency which is feasible in today FPGAs. A multiplexer width of 256 bit increases the complexity of the logic nearly quadratically, which is a point where no advantage of the lower internal frequencies can be achieved due to the routing overhead. Therefore a multiplexer width of 128 bit was chosen for the design.

4. Proposed architecture

Due to the addition of a retry mode implementation for HT3 devices, as well as the increased internal data path width, a new architecture has been created to fulfill these needs. The increased data path width, necessary to handle the complexity, also results in an increased pipeline depth to reach timing closure.

Due to the nature of the HT protocol, it is necessary to support Gen1 operation as well as Gen3 operation. As the goal of the architecture is to operate in HT3 mode, Gen1 operation is only intended for configuration access following cold reset, to transition the controller into Gen3 operation.

The controller can be separated into two functional main components. One is handling the reception of incoming traffic (RX), while the other is responsible for creating and transmitting an outbound transaction stream (TX). These two entities largely operate independently from one another. Only the exchange of flow control credits links both components. An overview is given in Figure 4.

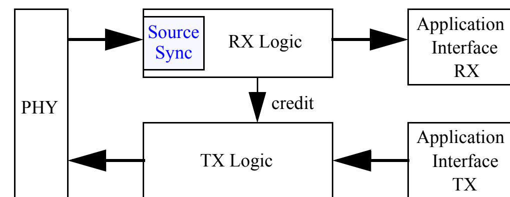


Figure 4: Top-level HT3-Core overview

The application interfaces consist of a number of traffic buffers, and support fully asynchronous clocking. This enables the application to run at an arbitrary frequency, independent of the link frequency. The interfaces contain separate command and data packet buffers for each VC. All contents of these buffers start at naturally aligned borders, whereby command packets are reordered to gain a continuous address field for transactions with address extension (64 bit addresses).

The PHY operates with a deserialization factor of 8. So for a 16 bit link, this results in 128 bits of CAD and 16 bits of CTL information each cycle. The core always operates on the same amount of data, independent of link width. This means that an 8 bit link only requires half the internal frequency of a 16 bit link.

The RX side architecture imposes no restrictions of command throttling, and permits command insertion. The TX architecture is more restrictive. Command insertion is not performed, and the number of command packets in each octaword is limited to one each cycle. If data transactions travel in the same VC, they can be streamed back-to-back.

RX and TX paths will be discussed in more detail in paragraphs 4.1 and 4.2 respectively. Two additional paragraphs highlight some of the more interesting implementation details. Paragraph 4.3 details the imple-

mentation of the user application interface, and paragraph 4.4 describes the CRC implementation.

4.1 RX path

The RX path reorders and decodes the incoming transaction stream, so that it can forward octaword aligned command and data packets. Such an alignment is rarely given in the data stream itself. It is further responsible for handling some low-level signaling (initialization), and must handle no-operation (NOP) packets.

All RX functionality can be divided into five major functional blocks, shown in Figure 5, and further described below. Each block contains multiple pipeline stages.

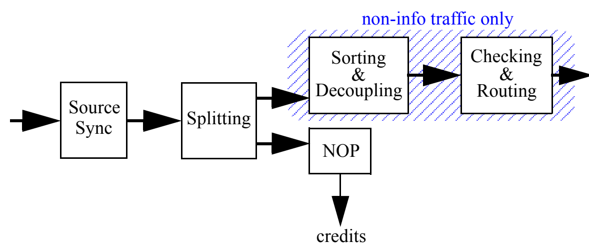


Figure 5: Functional pipeline of the RX path

The *Source Sync block* operates at the frequency of the recovered link clock. Its functionality includes handling the Gen1 initialization and the Gen3 training sequences. These sequences are used to communicate the start of the first DW between two participants in an HT chain. Due to the deserialization factor of 8, it might further be necessary to align the 8 bit received from each lane to reflect the DW alignment. During Gen3 operation the individual lanes are deskewed to return the same link bit-time, and the data stream is also descrambled. The last function this block fulfills is to check the periodic CRC DW and remove it from the data stream. The data stream is then stored in an asynchronously clocked FIFO to leave the source synchronous clock domain.

The *Splitting block* separates the incoming transaction stream into info and non-info traffic. Info traffic refers to NOPs and credits, whereas non-info traffic consists of all other transactions. This is necessary, as the following block buffers the non-info transactions. Buffering is enabled by the fact that VC traffic is flow controlled and therefore limited, whereas info traffic is not limited. Info packets are handled in the NOP block in parallel to the non-info transaction processing.

The *NOP block* evaluates received info packets. This includes extraction of flow control credits, as well as evaluation of other info packet fields, such as LDTSTOP/retry indication. During Gen3 operation the per-transaction CRC of the NOP packets is also checked and the acknowledge count included in the NOP is used to remove the acknowledged transactions from the retry logic.

The *Sorting and Decoupling stage* contains three major blocks. The first separates the incoming transaction stream into the basic transaction building blocks, which are command packets, data packets, and CRC packets. These packets are then stored in independent buffers. This buffering allows the remaining controller logic to operate at reduced bandwidth in cases where the input stream contains more than one command packet every octaword. A worst-case maximum of three command packets can be located inside of an octaword. The logic can compensate the reduced bandwidth if data transactions are processed. This is possible as commands and data, of up-to 128 bit size each, are processed simultaneously.

The *Checking and Routing block* of the RX path implements forwarding of the decoded transactions to their corresponding VC buffer within the application interface. During Gen3 operation, it calculates the per-transactions CRC of the forwarded transaction and indicates a successful check to the VC buffers. The VC buffers are implemented in such a way that a stored transaction only becomes visible to the application after it has been validated.

4.2 TX path

The TX path creates a HT compliant transaction stream from the command and data packets provided by the application interface. If no transactions are available in the user buffers, NOP packets are transmitted. The retry functionality required for Gen3 operation is not implemented with an explicit retry buffer, but reuses the TX application interface buffers to reduce complexity and area in terms of SRAM.

TX functionality can be divided into four major functional blocks, shown in Figure 6, which are further described below. Each TX block contains multiple pipeline stages.

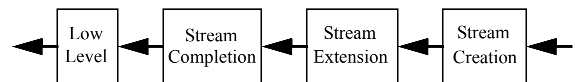


Figure 6: Functional pipeline of the TX path

The *Stream Creation block* merges the command and data packets from the application interface buffers and creates a rudimentary transaction stream for each VC, which is limited to one command packet per octaword. These streams do not yet contain the per-transaction CRC or any info packets. The VC transaction-

streams are multiplexed into a single stream via round robin arbitration. The order in which transactions from different VCs are transmitted is tracked as well. This allows to correctly assign received acknowledges to the corresponding VCs. The arbitrated transaction stream is stored into a decoupling buffer to ease implementation of backward flow control between the complex pipeline stages.

The *Stream Extension block* adapts the transactions retrieved from the decoupling buffer to the actual link width. During this adjustment, it also appends a per-transaction CRC placeholder after each transaction, independent of the operation mode, and is filling possible gaps between transactions with NOP placeholders.

The *Stream Completion block* is filling the placeholders inserted by the previous block with the required information. This means that it is computing the per-transaction CRC during Gen3 operation and inserting it into the CRC placeholder. NOP placeholders are filled with valid information, including the release of flow-control credits. During Gen1 operation all CRC placeholders are replaced with empty NOP packets, as this helps reduce the amount of necessary complexity for Gen1 operation in prior pipeline stages.

The *Low Level stage* implements a multiplexer between the assembled transaction stream and special low-level signaling schemes. The low-level signaling includes Gen1 initialization, Gen3 training, as well as sync-flooding. During Gen3 operation the transaction stream is also scrambled in this block, before it is forwarded to the PHY.

4.3 Application interface buffers

During Gen3 operation the core must support the retry mode defined by HT. This retry mode secures every transaction with a per-transaction CRC, and introduces two requirements to the architecture:

a) Received transactions are only forwarded after their CRC has been successfully verified
b) Transmitted transactions, barring info traffic, must be stored to allow a retransmission

Point *a)* is solved by the use of a special buffer that stores unverified transactions until their CRC has been checked. Entries in the buffer only become visible to the application after the CRC check was successful. This allows the decoding of the transaction to continue concurrently with the verification of the transaction's CRC, as even unverified transactions can already be added to the buffer and get validated later on. It furthermore, reduces area as transactions do not have to be intermittently stored in registers, but can be forwarded to the buffers immediately. To keep the interface simple and easy to use, while still fulfilling all needs of the retry mode, the RX application buffers are implemented as FIFO buffers with an additional *validated* write pointer (*ValWr*). Stored values are written to the write pointer (*Wr*) address, whereas the output is read from the read pointer (*Rd*) address. The operation of this FIFO is illustrated in Figure 7. Entries located between the *ValWr* pointer and the write pointer are unvalidated (shaded dark) and not visible to the application. Entries located between the read and the *ValWr* pointer are validated entries that the application can access through the defined mechanism. If a retry is executed, the write pointer will be set to the current *ValWr* pointer address, thereby removing all unvalidated entries.

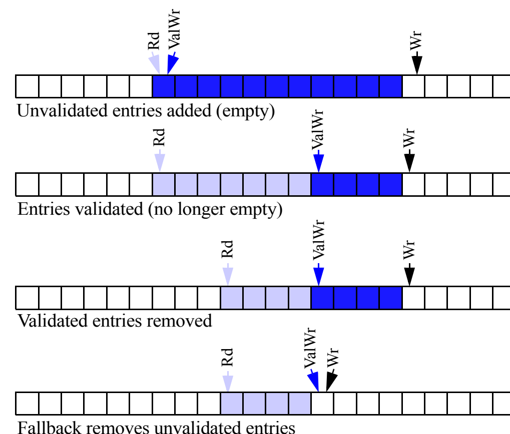


Figure 7: RX application buffer operation

A solution to issue *b)* would be the addition of a retry buffer that stores all transmitted non-info transactions. Our proposed implementation avoids this additional retry buffer by reusing the already existing TX application buffers. This is possible as HT makes no assumption about the order in which unacknowledged transactions are replayed in case of an error.

The TX application buffers are implemented as FIFO buffers with a second *unacknowledged* read pointer (*URd*). Whenever a transaction gets acknowledged by the remote device, the *URd* pointer is incremented and thereby the addressed transaction is effectively removed from the retry buffer. Figure 8 illustrates the operation of this FIFO. All transactions located between the *URd* pointer and the read pointer resemble the retry buffer, as they are unacknowledged (shaded dark). Entries located between the read and the write pointer resemble the application interface buffer with transactions that still have to be transmitted (lightly shaded). During a retry, the read pointer is reset to the current value of the *URd* pointer. As the VC multiplexing in TX is done behind the application buffers,

this means that retried transactions are not sent in the same order they were initially sent.

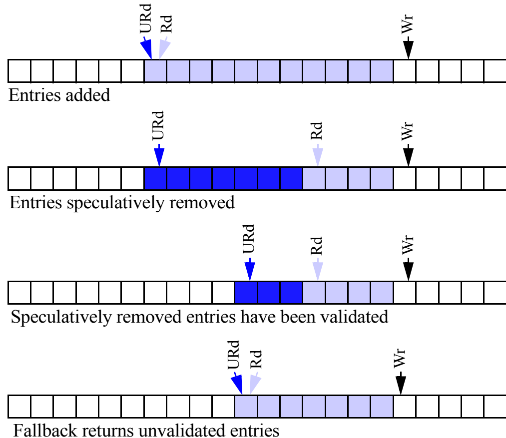


Figure 8: TX application buffer operation

To maintain the simple FIFO interface, both additional pointers can be incremented in single steps via an additional input signal to the buffers. Incrementing of the additional pointers is done after a successful packet-CRC check for RX, and the additional TX pointer is incremented whenever a new acknowledge counter is received from the remote device.

4.4 CRC calculation of non-info transactions

Calculation of the 32 bit per-transaction CRCs used for the retry mode during Gen3 operation is dependent on the degree of used data parallelism. The CRC calculation is commonly implemented as linear feedback shift registers (LFSR) for the polynomial division. Figure 9 depicts an example of a CRC calculation implemented as LFSR, where in each cycle one bit of data is serially added to the checksum. The calculation shown is based on the polynomial x^4+x^2+x+1.

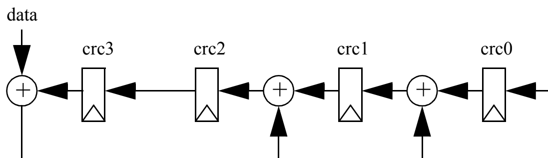


Figure 9: CRC LFSR example

The operation performed by the LFSR can be expressed through the following formulas, where t identifies time (cycles).

$$
\begin{aligned}
crc3_{t+1} &= crc2_t \\
crc2_{t+1} &= crc1_t + crc3_t + data_t \\
crc1_{t+1} &= crc0_t + crc3_t + data_t \\
crc0_{t+1} &= crc3_t + data_t
\end{aligned}
$$

These formulas describe the addition of one bit of data to a checksum. It can also be seen that each new result directly depends upon the previous cycle's result. More practical formulas, describing how multiple bits are added to a checksum in parallel, can be produced by recursively iterating these formulas.

The complexity of the formulas increases with more data to be included into the calculation. They can always be expressed as XOR combinations of the input data and the state of the CRC register from the previous cycle. This determines the maximum number of parameters one formula can include to be $parameter_limit = CRC_size + data_size$ and the worst case number of necessary XOR operations is $xor_limit = parameter_limit - 1$. So each formula grows linear with both CRC and data size. All formulas together grow quadratically with the CRC size and linear with the data size, because there are as many formulas as there are bits in the CRC.

In HT, the minimum transaction unit (mTU) is one DW of CAD plus 4 bit of CTL, which are all covered by the per-transaction CRC. The maximum size of an HT transaction that is supported by the proposed architecture is 19 mTUs, excluding the per-transaction CRC. Such a transaction contains a 3 mTU command packet with address extension plus a 16 mTU data packet. Any received HT transaction can have an arbitrary size ranging from 1 to 19 mTUs. The CRC calculation must be capable of calculating the per-transaction CRC for all possible transaction sizes. For the given architecture operating on four mTUs per cycle, this means that in each clock cycle 0 to 4 mTUs may be added to the calculation of one per-transaction CRC. This results in four sets of different CRC formulas, for adding 1 to 4 mTUs that all operate on the same 32 bit CRC register.

A formula f used to calculate one bit of a CRC for one parallel data input combination can be divided into a recursive function g and a non-recursive function h.

$$crc_{t+1} = f(crc_t, data) = g(crc_t) + h(data)$$

This reduces the impact of the cycle-to-cycle dependency on the CRC calculation and relaxes timing, as functions g and h can be implemented in different pipeline stages. This is especially attractive for large amounts of data, as function h can be further pipelined. Function g however contains the cycle-to-cycle dependency of the CRC calculation and cannot directly be pipelined further, which also means that it includes the critical timing path.

This approach was used to implement the CRC calculation of non-info transactions in the proposed architecture. As the architecture must handle 1 to 4 mTUs, four different g and h formulas exist for each CRC bit. These are multiplexed in the last pipeline stage which

then contains all recursive logic. Figure 10 gives an overview of the CRC calculation pipelining.

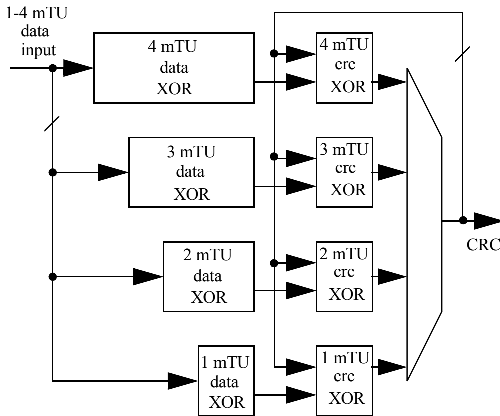


Figure 10: CRC pipelining overview

5. Evaluation

The bandwidth results shown in this paragraph were gathered from simulation of the synthesizable HDL description of the core. An implementation of the proposed architecture, using an 8 bit link operating at 2.4 Gbit/s. The bandwidth was measured through the transmission of 2,000 write transactions which introduces 3 DWs of overhead to the data payload. One DW per-transaction CRC and two DWs command packet containing a 40 bit address. The simulation have been repeated for all sizes of data packets to show the overall performance of the core.

Figures 11 and 12 show the measurement results of the RX and TX path. Both paths were simulated separately to avoid performance influences. In these figures Transaction Bandwidth refers to the bandwidth being used for non-info transactions, including command packets, data packets and per-transaction CRCs, calculated as *(((DATA DW + CMD DW + CRC DW) * 2000 * 4 * (1 / time)) / 10^9) GByte/s*. Transaction Bandwidth without CRC does not count the CRCs, calculated as *(((DATA DW + CMD DW) * 2000 *4 * (1 / time)) / 10^9) GByte/s*. Lastly Payload Bandwidth shows the effective bandwidth that is used for data forwarding, excluding command packets and CRCs, calculated as *((DATA DW * 2000 * 4 * (1 / time)) / 10^9) GByte/s*.

For sufficient payload sizes the architecture reaches a total bandwidth of 2.38 GByte/s. The periodic CRC slot accounts for a bandwidth loss of about 0.775% or 0,0186 GByte/s on an 8 bit link as it is recommended by HT so every 512 bit times a 4 bit CRC has to be transmitted. If this bandwidth loss is added to the Transaction Bandwidth it results in a total utilized bandwidth of 2.3986 GByte/s which is extremely close to the theoretical maximum of 2.4 GByte/s for an 8 bit link.

Due to the increased number of pipeline stages the bandwidth drops with lower data payloads. This happens because of credit starvation. Then all available credits can be in use. But if high bandwidth is required, sending smaller data payloads is counterproductive because the further command overhead will decrease the usable bandwidth additionally.

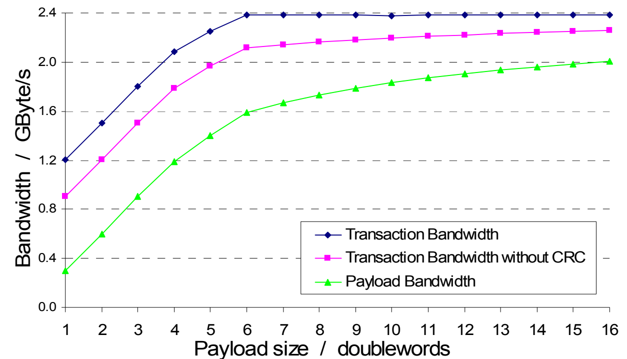


Figure 11: RX bandwidth for data transfers

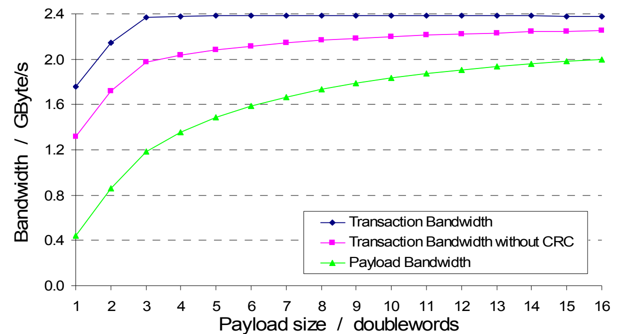


Figure 12: TX bandwidth for data transfers

Due to the increased datapath of the core and the fact that this increase also adds complexity, the resource usage of the HT3-Core is higher than it was for the HT2-Core. Table 2 shows the total and percentage resource usage of the core on a Xilinx Virtex-5 LX110T FPGA device.

Table 2: Resource usage on LX110T

Resource	Used	Percentage
Slice Registers	18,905	27%
Slice LUTs	37,094	53%
Occupied Slices	11,098	64%
Block RAMs	78	52%

6. Conclusion and outlook

We have proposed a novel architecture of an HT3 controller for FPGAs. To the best of our knowledge this is the first implementation for such reconfigurable platforms. We have performed a complexity analysis and shown the issues of modern high bandwidth I/O technologies like HT. Solutions for these problems and a very efficient implementation of the core is provided. The benchmarks show the excellent performance of the architecture with maximum achievable bandwidth of 2.3 GByte/s, which is close to the theoretical optimum.

Our future work will focus on the bringup of the design in real world systems using various FPGA technologies. The architecture will also be further improved as more real world data can be gathered. A 16 bit link version, currently in development, promises to double the achievable bandwidth to 4.6 GByte/s. There are also plans to push the 8 bit link version beyond the current lane rate of 2.4 Gbit/s.

7. References

[1] Mahapatra, N. R. and Venkatrao, B.: The processor-memory bottleneck: problems and solutions. Proc. Crossroads 5, 3, 1999

[2] Bees, D. and Holden B. 2004. HyperTransport reduces delays in some applications.

[3] C. Guiang, K. Milfeld, A. Purkayastha, J. Boisseau. "Memory Performance on Dual-Processor Nodes: Comparison of Intel Xeon and AMD Opteron Memory Subsystem Architectures," Proceedings of the ClusterWorld Conference and Expo, San Jose, CA, June 24-26, 2003.

[4] R. Brightwell, D. Doerfler, K.D. Underwood, "A preliminary analysis of the InfiniPath and XD1 network interfaces," ipdps,pp.311, Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, 2006

[5] Ron Brightwell, Kevin T. Pedretti, Keith D. Underwood, Trammell Hudson, "SeaStar Interconnect: Balanced Bandwidth for Scalable Performance," IEEE Micro, vol. 26, no. 3, pp. 41-57, May/Jun, 2006

[6] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, U. Brüning: The HTX-Board: A Rapid Prototyping Station. Proc. of 3rd annual FPGAworld Conference, Nov. 16, 2006, Stockholm, Sweden.

[7] HyperTransport Consortium: The Future of High-Performance Computing: Direct Low Latency CPU-to-Subsystem Interconnect. HTC whitepaper, 2008

[8] David Slogsnat, Alexander Giese, Mondrian Nüssle, Ulrich Brüning: An Open-Source HyperTransport Core. ACM Transactions on Reconfigurable Technology and Systems (TRETS), Vol. 1, Issue 3, p. 1-21, Sept. 2008.