Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Master's Thesis

Serial ATA core with cryptography support

Bc. Martin Chloupek

Supervisor: Ing. Pavel Kubalík, Ph.D.

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

May 22, 2009

Aknowledgements

I would like to thank Ing. Pavel Kubalík, Ph.D. for his efforts he gave supervising this work and special thanks to my family for their patience during my work as well.

Declaration

Abstract

This work deals with design and implementation of a Serial ATA core with cryptography support in HDL language. The core is implemented and tested on Xilinx Virtex-5 FXT FPGA, Xilinx ML-507 development board. The system is controlled by using Ethernet frames. Cryptographic AES-128 core is placed between Ethernet and SATA core, making possible to crypt stream of transferred data. The design uses Xilinx Serial ATA PHY and Xiling TEMAC hard cores. First part of this work deals with analysis of Serial ATA protocol and cryptography as well, next part deals with overall system design, the last one describes system implementation and testing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's hard drives are the most common equipment for storing large amounts of data. They have been here for quite a long time, starting with several megabytes, nowdays their capacity is measured in terabytes. Their interfaces has been developing with their size. Nowdays we can choose from several interfaces, beginning with Parallel ATA, Ultra Wide SCSI, Serial ATA and Serial Attached SCSI (SAS).

In area of personal computers, laptops and low-end servers the Serial ATA interface and protocol is the most used one. Because of its popularity, SATA now appears more and more in systems on chip, embeedded systems and other areas where high capacity of non-volatile memory is needed.

In area of FPGA and ASIC design, lot of companies like Mentor Graphics, Synopsys, Altera, Xilinx, Aeuluros provide hard IP cores realizing physical layer of SATA. These companies provide their designs in many nanometer technologies and also as burned hard cores on FPGA chips.

While Xilinx company provides Serial ATA PHY core so I decided to implement another layers of Serial ATA protocol. This could be very useful for other FPGA/ASIC application development, where large amounts of data are generated or consumed. For example EDK generated system with Linux operation system would use SATA core at most.

Hard drives often contain very private data. Because hard drives can be stolen, it is necessary to encrypt data stored on hard drive. This Master's Thesis also handles this situation by using encryption/decryption core in dataflow process.

My task is to design and realize system in hardware that writes and reads data from Serial ATA hard drive with possibility of encrypting and decrypting data stream. System receives control and input data from gigabit Ethernet interface.

Chapter 2

Problem description, goal specification

Target platform on which the system was developed and tested is Xilinx ML-507 development board with Xilinx Virtex-5 XC5VFX70T FPGA chip in the center. This chip contains GTX transceiver/receiver hard core, that is basically high speed serial to parallel converter with additional protocol required features. The GTX core supports many serial communication protocols like PCI express, fiber channel, XAUI, SATA generation one and two and others. In this thesis the GTX core with SATA configuration is used. For fully working physical layer of SATA protocol there is necessity of implementing control circuits for correct SATA speed negotiation. Reference design is provided from Xilinx company, see [1].

Early versions of this core appeared also on older FPGA chips called Virtex-II, however this version couldn't be used with regular SATA hard drives due to lack off Out Of Band signal support, that is necesary for Serial ATA to work.

The Virtex-5 FPGA contains among others also Tri-Mode Ethernet Mac (TEMAC) hard core that implements MAC layer of Ethernet and therefore the development board is equipped with chip that realizes Ethernet physical layer. It supports 10Mb, 100Mb and 1Gb Ethernet transfer rates, however only 1Gb configuration is used in this design due to SATA rates that are 150MB for SATA generation 1 and 300MB for SATA generation 2. Reference design, how to operate this core is available from Xilinx company.

Situation regarding IP cores which implement transport and link layers of SATA is very various on the market. Company Synopsys in their product DesignWare Cores SATA AHCI [9] provides among SATA transport and link layer implementation also AHCI interface, that is used in operation systems like Microsoft Windows Vista and Linux. Company Nuvation [10] also provides implementation of previously mentioned SATA layers with AHCI interface for product of Altera company. Xilinx company also provides implementation for their products. Detailed market survey can be found on site [7], but the analysis is paid. I was not succesful in finding free/open-source implementation of transport and link layers of SATA protocol.

After consulting with supervisor, we decided, that using of old ciphers like DES or Triple-DES that are not so hard to implement is obsolete today. Today the most used symmetric block cipher is AES. AES was announced by National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197 (FIPS 197) on November 26, 2001. AES is in my opinion little bit harder to implement than ciphers previously mentioned, and because there are many implementations on the web in hardware and software, we decided, that it would be best to choose one of implementations freely available rather than implement one from scratch. More details in this manner can be found in chapter Cryptographic analysis 3.2, where I compare ciphers that I found and comment the implementation I use in this Master's thesis and also comment the decision making process.

My task is to design and implement system, which on one side receives control data, transmits status data and receives/transmits payload data through 1Gb Ethernet interface, then if desired encrypts/decrypts payload data using symmetric block cipher and on the other side it communicates with SATA hard drive, where the payload data are written or read. Other result of this thesis is SATA protocol analyzer using Chipscope Pro software from Xilinx company.

Chapter 3

Analysis

This part of the thesis is devoted to analysis of SATA protocol in scope necessary for implementation of desired system, then it focuses on analysis of available implementations of AES cipher and decision which implementation is to be used in desired system.

3.1 Serial ATA protocol analysis

Serial ATA is high speed serial interface operating at 1.5Gb/s for SATA generation one and at 3.0Gb/s for SATA generation 2. The SATA specification is developed and maintained by the Serial ATA International Organization. The specificiation can be downloaded from [11].

Serial ATA protocol is divided into 5 layers, starting from the bottom Physical layer, Link layer, Transport layer, Command layer and Application layer, as illustrated in following figure 3.1.

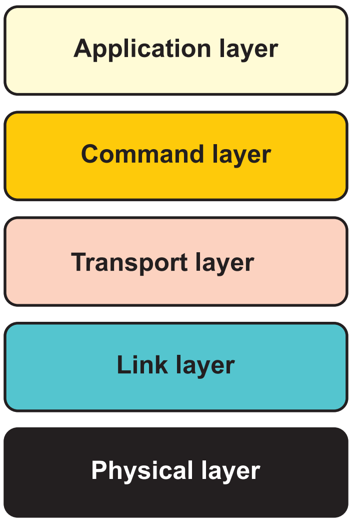


Figure 3.1: serial ATA layers

SATA main features:

- Serial Interface: The serial interface consists of two differential pairs for bi-directional signaling. It may seem that the interface is full-duplex, but it is not, while on one differential pair is transmitted data the other pair is used for status signaling.
- ATA Commands: The SATA protocol supports existing ATA commands, so on Command layer it is backward compatible with its predcessor Parallel ATA.
- Low Voltage Signaling: Signaling is performed at low differential voltage: 500mV peak-to-peak, which is huge advantage when implemented in nowdays nanometer technologies.
- Hot Plug: Cables and connectors are designed to enable hot plug – when inserting cable, first connected are ground pins, then power pins and at last differential signaling pairs.
- Native Command Queuing: Native Command Queuing permits drives to queue 32 commands and then execute them in the order that will result in best overall performance.

3.1.1 Physical layer

This section describes the physical layer of Serial ATA. Serial ATA Physical layer is responsible for parallel to serial conversion and vice versa, for detecting and transmitting OOB signal and for 8b/10b coding and decoding. Signal are transmitted/received on differential pairs using NRZ coding.

8b/10b coding is standard that trades-off two bits off overhead for improved reliability. For SATA it also provides means of transmitting/receiving special characters called K characters, that are used for control functions.

OOB signals (Out Of Band) are special sequences of data bursts and idles, each of which is differential signal level below 50 mV (peak to peak). They are used for startup link speed negotiation before the data transfer occurs. In SATA specification, there are defined 3 type of OOB signal: COMINIT (sent by device), COMRESET (sent by host) and COMWAKE (send by both host and device. They differ in time of idles, time off data bursts is always the same. Following figure 3.3 illustrates these signals.

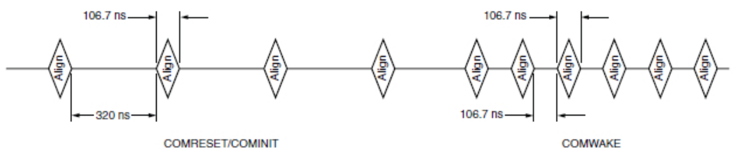


Figure 3.2: OOB signals - taken from [1]

3.1.1.1 Startup link speed negotiation

Startup link negotiation is accomplished through OOB signaling. The sequence is as follows:

1. Host signals COMRESET.
2. Device detects completion of COMRESET .
3. Device signals COMINIT.
4. Host detects completion of COMINIT.
5. Host calibrates and then signals COMWAKE.
6. Device calibrates and then signals COMWAKE.
7. Device transmits continuous stream of ALIGN primitives at the device's highest speed.
8. Host detect ALIGN primitives and in response start transmission of continuous ALIGN primitives.

If the device doesn't detect ALIGN primitives from host for 54.6us, it assumes that host cannot communicate at that speed and tries sending ALIGN primitives at next lower speed. This step continues until all speed are exhausted, if no response, device enters error state.

After the ALIGN primitives are successfully exchanged, host and device are synchronous, and they exchange SYNC primitives entering link idle state. Following figure 3.3 depicts Serial ATA speed negotiation startup sequence.

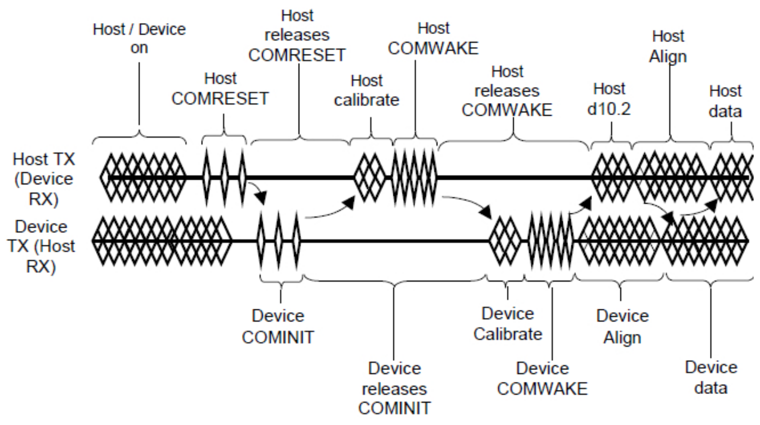


Figure 3.3: OOB startup link speed negotiation - taken from [6]

3.1.1.2 Primitives

Serial ATA protocol is double word (32-bits) oriented. All transfers across serial ATA link are aligned to double word boundary. Primitives are special double words that carry control information, they consist of normal characters and special K character. Serial ATA specification defines 18 primitives, in scope of this thesis are relevant only 6 of them, which are listed in following table 3.1.1.2. K character is always byte 0 of the primitive double word, they are underlined in the hexadecimal representation in the primitive table 3.1.1.2.

Primitive	**Value**	**Name**	**Description**
ALIGN	7B4A4ABCh	Physical layer control	This primitive is always sent in pairs. Using this primitive, receiver detects double word boundary in transmission. SATA specification defines, that it is necesary to retransmit ALIGN primitives pair every 256 double words are transmitted across the link for keeping receiver and transmitter synchronized.
CONT	9999AA7Ch	Continue repeating previous primitive	This primitive is used for repeated primitive suppression, more details in section 3.1.2.5
EOF	D5D5B57Ch	End of frame	This primitive marks the end of frame.
HOLD	D5D5AA7Ch	Hold data transmission	This primitive is used for the payload data flow control. It means, that the transmission buffer of transmitting node is empty. It is also transmitted by receiving node when receiving node buffer is full.
HOLDA	9595AA7Ch	Hold acknowledge	This primitive is used for the payload data flow control. It is transmitted by transmitting node to acknowledge, that receiving node's buffer is full.
R_ERR	5656B57Ch	reception error	This primitive informs transmitting node, that receiving node received frame with error.
R_IP	5555B57Ch	Reception in progress	This primitive informs transmitting node, that receiving node is receiving the payload data.
R_OK	3535B57Ch	reception with no error	This primitive informs transmitting node, that receiving node received frame with no error.
R_RDY	4A4A957Ch	Receiver ready	This primitive informs transmitting node, that receiving node is ready to receive frame.
SOF	3737B57Ch	Start of frame	This primitive marks the start of frame.
SYNC	B5B5957Ch	Synchronization	This primitive is used to signalize idle bus state. It is also used for termination current reception/transmission.
WTRM	5858B57Ch	Wait for frame termination	After transmission of EOF primitive, the transmitting node transmits this primitive, while waiting reception status from receiving node.
X_RDY	5757B57Ch	Transmission data ready	This primitive informs receiving node, that transmitting node is ready to send frame.

Table 3.1: Primitives

3.1.2 Link layer

The Link layer manages the frame transmission protocol. A major part of this protocol is generating and decoding primitives and also data flow control.

When requested by the Transport layer to transmit a frame, the Link layer does following:

- Receives data from the Transport layer
- Calculates CRC and inserts calculated CRC value at the end of a frame before EOF primitive
- Insert primitive SOF and EOF, to mark frame boundaries
- Provides frame flow control using HOLD and HOLDA primitives
- Scramblers data
- Reports status of frame transmission

When data is received from Physical layer, the Link layer does following:

- Acknowledges peer node that it is ready to receive frame
- Receives data from Physical layer
- Recognizes received primitives and removes them
- Unscrambles data
- Calculates CRC and compares to CRC received in frame
- Reports status of frame reception

When both host and device want to transmit a frame, the device has higher priority.

3.1.2.1 Frame transmission sequence

When requested by upper layer to transmit a frame, the sequence is following:

1. Transmitting node starts transmitting X_RDY primitive, indicating that it is ready to transmit frame
2. Receiving node detects X_RDY primitive and if it is ready to receive frame, starts transmitting R_RDY primitive, indicating that it is ready to receive frame
3. Transmitting node detects R_RDY primitive and starts data. One SOF primitive is transmitted first followed by payload data burst.
4. Receiving node detects SOF primitive, starts receiving data and in response transmits R_IP primitive, indicating that receive is in progress.

5. Transmitting node transmits all frame data, followed by CRC value and EOF primitive and immediately after that starts transmitting WTRM primitive, indicating that it is waiting for frame reception status.
6. Receiving node receives payload data until EOF primitive is detected. After detecting EOF primitive, receiving node checks received CRC value, which is the last double word before EOF primitive, with calculated CRC value. If the received CRC value is equal to calculated CRC value, receiving node starts transmitting R_OK primitive indicating that it has received frame without errors. If the received CRC value is not equal to calculated CRC value, receiving node starts transmitting R_ERR primitive indicating that received frame is corrupted.
7. Transmitting node detects R_OK or R_ERR primitive, reports frame reception status to upper layer and starts transmitting SYNC primitive.
8. Receiving node detects SYNC primitive and starts transmitting SYNC primitive in response, placing the link in idle state.
9. Link is now in idle state and next tranSmission can start.

This transmission sequence is the same for device and host. If the frame carried control information and the was received corrupted, the Link layer automatically retransmits the frame. If the frame carried data to be written to disk or data read from disk it is up to transport layer, what happens next.

3.1.2.2 Flow control

During the frame transmission, if the transmitting node's transmit buffer is empty, the transmitting transmits HOLD primitive instead of the data. After the transmitting node's buffer has some data to transmit, start transmitting the data instead of HOLD primitive.

During frame reception, if the receiving node's receive buffer is nearly full, it starts transmitting HOLD primitive. If the transmitting node detects HOLD primitive, it understands that receiving node's buffer is nearly full and starts transmitting HOLDA primitive instead of the data. When the receiving node's buffer has enough space for receptions, the receiving node stops transmitting HOLD primitive, transmitting node detects it at continues transmitting the data instead of HOLDA primitive. Serial ATA specification defines, that transmitting node has 20 double word cycles to start transmitting HOLDA primitive upon reception HOLD primitive. This is called round trip delay time.

3.1.2.3 CRC

The Serial ATA protocol uses 32 bit Cyclic Redundancy Code for verifying frame integrity. The calculated CRC value is inserted at the end of frame payload data right before EOF primitive. The CRC is calculated on double word quantities and is calculated according to following 32-bitgenerator polynomial:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

The CRC value is initialized with a value 52325232h

3.1.2.4 Data scrambling

All data transferred across the SATA link must be scramble due to EMI emissions. Scrambling and unscrambling is accomplished by XORing the data, to be transmitted/received with an output of a linear feedback shift register (LSFR). The shift register shall implement the following polynomial:

$$G(x) = x^{16} + x^{15} + x^{13} + x^4 + 1$$

The LSFR is initialized with value FFFFh. Primitives are not scrambled. CRC value is scrambled.

3.1.2.5 Repeated primitives suppression

Because repeated primitives can cause high EMI emissions, Serial ATA protocol defines manner how to suppress these emissions. After transmission of two identical primitives, CONT primitive follows and the output is generated from scrambling LSFR. Scrambling LSFR is the same LSFR used for data scrambling, so there need to be 2 scramblers in the design, one for data scrambling and another for scrambling output after CONT primitive. Example sequence with CONT primitive illustrates following figure 3.4.

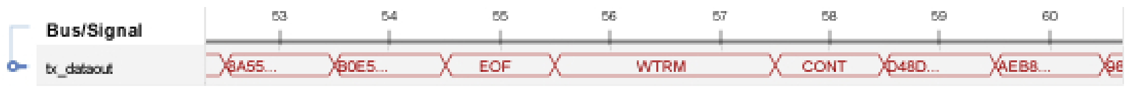


Figure 3.4: Repeated primitive suppression example

3.1.3 Transport layer

Requests to send may come from Command layer or Application layer. In either case, the Transport layer is responsible for creating Frame Information Structures (FIS) and manages transmissions of these Frame Information Structures by controlling Link layer. Serial ATA specification defines several types of them, the following table 3.1.3 displays their name, hexadecimal type value, direction and size. FIS type specifies FIS length.

In scope of this thesis are only important Register, PIO Setup and Data Frame Information Structures.

Serial ATA protocol is on Command layer backward compatible with parallel ATA. Parallel ATA uses a set of registers for control and data exchange. In Serial ATA a copy of the legacy ATA registers is called "shadow registers" and they are kept in Host Bus Adapter. Application layer software access these shadow registers and Serial ATA Transport layer then transfers content of these register using Frame Information Structures to hard drive where the real registers take place.

FIS name	Type value	Direction	Size
Register FIS - Host to Device	27h	Host to Device	5 double words
Register FIS - Device to Host	34h	Device to Host	5 double words
Set device Bits	A1h	Device to Host	2 double words
PIO Setup FIS	5Fh	Device to Host	5 double words
DMA Activate FIS	39h	Device to Host	1 double word
DMA Setup FIS	41h	Bidirectional	7 double words
Data FIS	46h	Bidirectional	max 2049 double words
BIST Activate FIS	58h	Bidirectional	3 double words

Table 3.2: Frame Information Structure types

Double word	Byte 3	Byte 2	Byte 1	Byte 0
0	Features	Command	CRR	FIS Type
1	Device	LBA high	LBA mid	LBA low
2	Features (exp)	LBA high (exp)	LBA mid (exp)	LBA low (exp)
3	Control	Reserved	Sector Count (exp)	Sector Count
4	Reserved	Reserved	Reserved	Reserved

Table 3.3: Register FIS - Host to Device format

3.1.3.1 Register FIS - Host to Device

Register FIS - Host to Device transfers Host Bus Adapters shadow registers to registers in device. Table 3.1.3.1 depicts format of this FIS.

Brief description:

- Command filed specifies the ATA command to be executed.
- Bit 7 of CRR register specifies if the FIS transfer is triggered by change of Command register (bit is set) or Control register (bit is cleared).
- LBA registers are used for 48-bit addressing scheme of disk sectors.
- Sector count registers specify how many sector are to be transferred or received.

Meaning of these registers vary according to which ATA command is used. ATA specification [12] explains meaning of these registers for each command.

3.1.3.2 Register FIS - Device to Host

Register FIS - Device to Host transfers content of hard drive registers to Host Bus Adapters shadow registers. Table 3.1.3.1 depicts format of this FIS.

Meaning of these registers vary according to which ATA command is used. ATA specification [12] explains meaning of these registers for each command.

Double word	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0	Error	Status	RIR	FIS Type
1	Device	LBA high	LBA mid	LBA low
2	Reserved	LBA high (exp)	LBA mid (exp)	LBA low (exp)
3	Reserved	Reserved	Sector count (exp)	Sector Count
4	Reserved	Reserved	Reserved	Reserved

Table 3.4: Register FIS - Device to Host format

3.1.3.3 PIO Setup FIS

This FIS is delivered from a SATA device to host during execution of Programmed IO (PIO) command. Table 3.1.3.3 depicts format of this FIS.

Double word	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0	Error	Status	RIDR	FIS Type
1	Device	LBA high	LBA mid	LBA low
2	Reserved	LBA high (exp)	LBA mid (exp)	LBA low (exp)
3	E_Status	Reserved	Sector count (exp)	Sector Count
4	Reserved	Reserved	Transfer Count (exp)	Transfer Count

Table 3.5: PIO Setup FIS format

3.1.3.4 DATA FIS

This FIS is delivered either from Host to Device or from Device to Host. It carries the data to be read from disc or the data to be written to disc. Minimum size of data payload is one double word, maximum size 2048 double words. Usually the data payload size in bytes is multiply of 512 bytes (default sector size). Table 3.1.3.4 depicts format of this FIS.

Double word	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0	Reserved	Reserved	Reserved	FIS Type
1 ... (N-1)	N double words of data (minimum is one double word, maximum is 2048 double words)			

Table 3.6: Data FIS format

3.1.4 Command layer

As I mentioned in previous section, Serial ATA protocol is on Command layer backward compatible with parallel ATA. Specification ATA/ATAPI-7 [12] – the seventh revision of the ATA standard released in 2003 defines these commands. Meaning of control and status registers vary according to which commands are used. In this thesis, I use these commands:

- IDENTIFY DEVICE - in response to this command, device returns 512 bytes of information describing device capabilities, additional protocol support, device serial number string, manufacture string and etc.
- READ NATIVE MAX ADDRESS EXT - in response to this command, device returns maximum sector address using 48-bit addressing scheme.
- READ SECTOR(S) EXT - this command is used for writing one or more sectors using 48-bit addressing scheme.
- WRITE SECTOR(S) EXT - this command is used for reading one or more sectors using 48-bit addressing scheme.

These commands except READ NATIVE MAX ADDRESS EXT follow PIO command data protocol. READ NATIVE MAX ADDRESS EXT command follows non-data command protocol.

3.1.4.1 Non-data command protocol

This command protocol is used for commands, where no data is being transferred. This command follows this sequence:

- Host transmits command using Register FIS - Host to Device. If during reception an illegal transition occurs, the device send SYNC primitive to abort the transfer.
- Device process command.
- Device transmits Register FIS - Device to Host containing command results.

Figure 3.5 depicts this protocol.

3.1.4.2 PIO command data protocol

This command protocol differs if the payload data are transferred to device or from device:

- PIO Data-IN - data is transferred from device to host
- PIO Data-OUT - data is transferred from device to host

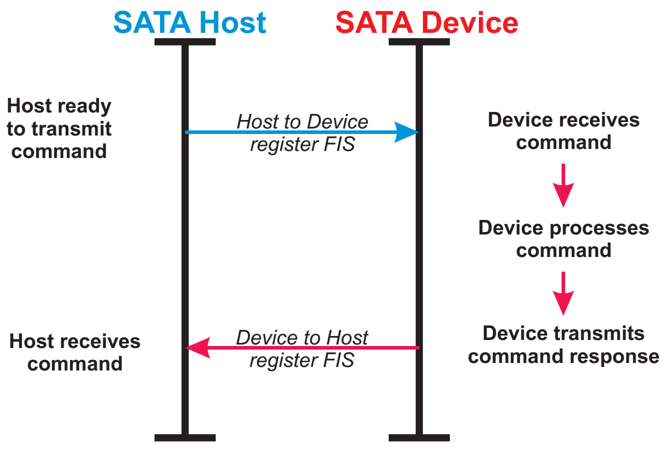

SATA Host
SATA Device
Host ready to transmit command
Host to Device register FIS
Device receives command
Device processes command
Device transmits command response
Device to Host register FIS
Host receives command

Figure 3.5: Non-data command protocol

PIO Data-IN protocol

This command protocol follows this sequence:

- Host transmits Register FIS - Host to Device.
- Device reads data from disc.
- Device transmits PIO Setup FIS with initial and end status.
- Data is transferred from device to host using Data FIS.
- If the device detects an error, it reports the error to the Host using Register FIS - Device to Host.

Figure 3.6 depicts this protocol.

PIO Data-OUT protocol

This command protocol follows this sequence:

- Host transmits Register FIS - Host to Device.
- Device seeks write location on disk.
- Device is ready to accept data and transmits PIO Setup FIS.
- Data is transferred from host to device using Data FIS.
- Drive reports write status to host using Register FIS - Device to Host

Figure 3.7 depicts this protocol.

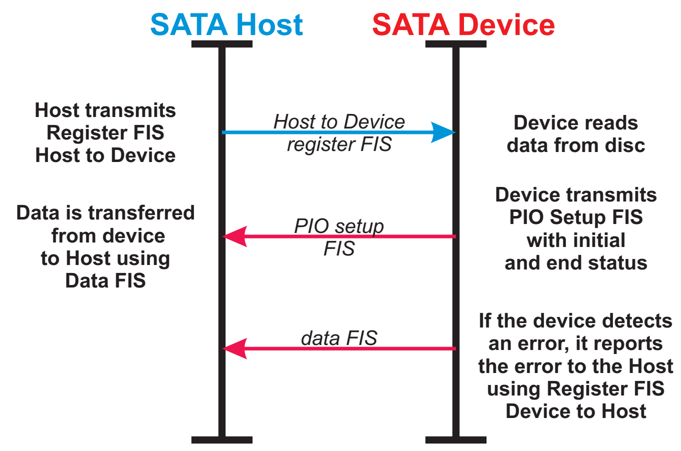

SATA Host
SATA Device
Host transmits Register FIS Host to Device
Host to Device register FIS
Device reads data from disc
Device transmits PIO Setup FIS with initial and end status
Data is transferred from device to Host using Data FIS
PIO setup FIS
data FIS
If the device detects an error, it reports the error to the Host using Register FIS Device to Host

Figure 3.6: PIO Data-IN protocol

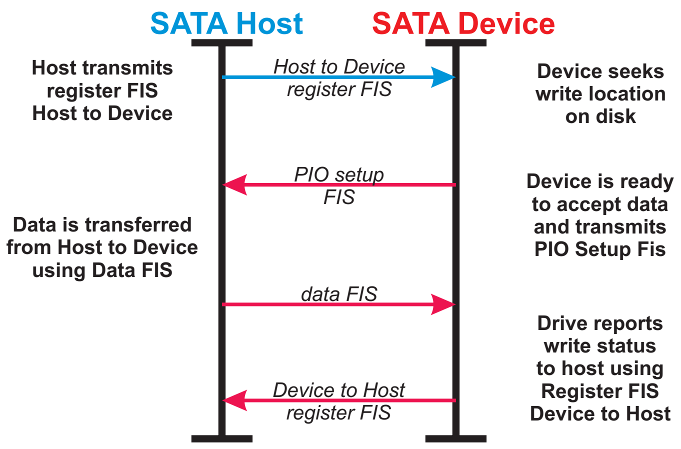

SATA Host
SATA Device
Host transmits register FIS Host to Device
Host to Device register FIS
Device seeks write location on disk
PIO setup FIS
Device is ready to accept data and transmits PIO Setup Fis
Data is transferred from Host to Device using Data FIS
data FIS
Drive reports write status to host using Register FIS Device to Host
Device to Host register FIS

Figure 3.7: PIO Data-OUT protocol

3.2 Analysis of cryptographic cores

As I mentioned in previous chapters, I decided to use symmetric block cipher AES with 128 bits key length, and not to implement AES from scratch, but to use an implementation freely available. In this chapter I describe the reasons for this decision and I also describe implementations of AES that I found, and decision which one I use.

The first question is: Symmetric block cipher or symmetric stream cipher? While the payload data are written/read on hard drive where it is stored in blocks called sectors which are usually 512 bytes length (in ATAPI-7 [12] there is command for changing sector length), it seems that block cipher is the right choice. Stream cipher could be theoretically also used, when each sector would represent one stream. Ciphering whole disk with stream cipher as one stream is impossible, because operation system reads/writes sectors from random addresses, stream cipher would need to read all preceding sectors to have right context for ciphering particular sector. Another issue is, that stream ciphers are designed to cipher stream, when we use one sector as a stream and same encryption key for each sector, stream ciphers would be very vulnerable to attack, because the cracker would have in his hands a lot of small streams, knowing that each stream has the same ciphering key. These are the reasons, why I decided to use symmetric block cipher.

Another question is: Why AES? According to source [15]

- AES – (Advanced Encryption Standard) – Created by US government for ciphering their documents. Block length is 128 bits and key length 128, 192 or 256 bits. Has not been cracked yet.
- DES – It was developed in 70 and is considered as insecure because the key is only 56 bits length. It is possible to crack this cipher in one day by brute-force attack using today's personal computer.
- Triple DES – Successor of DES, key length is 168 bits, it is more secure than DES but is very slow.
- IDEA - Block length is 64 bit and key length is 128 bits. It is generally considered as the most secure. Patented till 2010 – 2011.
- RC2 - (Rivest Cipher 2) Block is 64-bit length, key has variable length. Secret of RSA, SDI.

I decide to use AES because it is fast and secure. Another reason is that the standardization process of AES cipher took 5-year, fifteen competing designs were presented and evaluated before Rijnndel was selected as the most suitable. AES was announced as standard by National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197 (FIPS 197) on November 26.

Very detailed comparison of symmetric ciphers can reader find at [16].

3.2.1 Comparison of available implementations

3.2.1.1 AES (Rijndael) IP Core

Source [13]:

Description from authors:

- 16 byte block size
- 16 byte key size
- separate cipher (encrypt) block
- separate inverted cipher (decrypt) block
- incorporated key expansion module
- written in verilog
- Authors believe that this core is fully complies to FIPS-197.

My observation:

- +fast - takes 10 cycles for encryption/decryption
- +encryption key is entered same tame as data
- +128 bit interface
- -only 128 bit key length supported

3.2.1.2 128/192 AES

Source [2]:

Description from authors:

- SystemC and Verilog code provided
- Verified using TLM(Transaction Level Modeling Style)
- Encoder and decoder in the same block
- 128 bits low area implementation, takes about 500 cycles to encrypt/decrypt a block.
- 192 bits low area implementation, takes about 280 cycles to encrypt/decrypt a block.
- They don't use memories to store the S-box and have many other architectural improvements to reduce the area consumption. Implements the encoder and decoder in the same block. The cores were written in SystemC RTL, and verified using TLM(Transaction Level Modeling Style). Verilog synthesizable code is also provided.

My observation:

- very slow
- no documentation at all provided

3.2.1.3 AES core modules

Source [3]:

Description from authors:

- AES encoder 128/192/256 bit
- AES decoder 128/192/256 bit

My observation:

- 8 bit interface
- very bad documentation
- not clear how fast encoder and decoder works

3.2.1.4 AES128

Source [4]:

Description from authors:

- Input and key length of 128 bits.
- Operation in ECB mode.
- Performance adheres to FIPS-197.
- Core with high speed and low latency.
- RTL and TB in VHDL.

My observation:

- same module for encoder and decoder
- not very good documentation
- needs 13 cycles for crypt

When I was choosing which implementation to use my first choice was AES core modules because it supports all possible AES key sizes. In later research, I changed my opinion, because the core has 8 bit interface, the documentation is very bad, there is not cellar, how fast does the core work. I decided to use AES (Rijndael) IP Core, because it has very good documentation, 128-bit interface, has separate encryption and decryption block and it takes 10 cycles for encryption/decryption. Another reason was, that the core has very clear implementation, which I consider very good for the future development. The reason why 128-bit interface suits me better is that the interface to Ethernet core is 8 bits wide (can be changed to 10) and operates at 125Mhz frequency, so while converting data from 8-bit interface to 128-bit interface, which takes 16 clock cycles, the previous block of data could be encrypted in 10 cycles and in the result, there would be no pause in input stream.

3.3 Reference design for Serial ATA Physical layer

Xilinx company provides reference design, how to operate RocketIO GTP core for serial tat functions. Reference's design block level schematic depicts figure 3.8. This design contains two major blocks, OOB control block and Speed Negotiation Control (SNC) block. OOB Control block handles control of OOB signals during startup speed negotiation sequence a signals to SNC block, that the host and device have locked. Speed Negotiation Control block dynamically reconfigures GTP phase locked loop and by that changes speed between SATA generation 1 and 2.

This design uses Virtex-5 LXT FPGA while I chose Virtex-5 FXT FPGA for my disposal. The LXT FPGA contains RocketIO GTP core, while FXT FPGA contains RocketIO GTX core. The GTX version of RocketIO is upgraded version of GTP RocketIO core. Maximum datapath width for RocketIO GTP core is 16 bits and for Rocket GTX core is 32 bits. The internal datapath widths also differ, while GTP version's width is 10 bits, GTX version's width is 20 bits. Due to these differences the clocking also differs.

I used 32 bit wide datapath from RocketIO GTX core in my design so I had to completely rewrite OOB control block. I also had to change DCM configuration and clock paths. I use the Speed Negotiation Control block without modifications.

3.4 Example design for TEMAC core

Xilinx company provides for its Tri-Mode Ethernet MAC core (TEMAC) an example design, which is very easy to understand and to use. The example design is ended by address swap module which receives Ethernet frame from receive FIFO then swaps destination and source MAC addresses and then sends the modified frame to transmit FIFO. The design needs two clock sources, 200Mhz clock for IDELAY primitive and 125Mhz for rest of the design. All that is needed for making the example design to work is modifying the user constrains file. However that is not all. There is not brought-out reset signal for physical interface. Without this signal, the example design does not work. Figurin this out took me quite a long while.

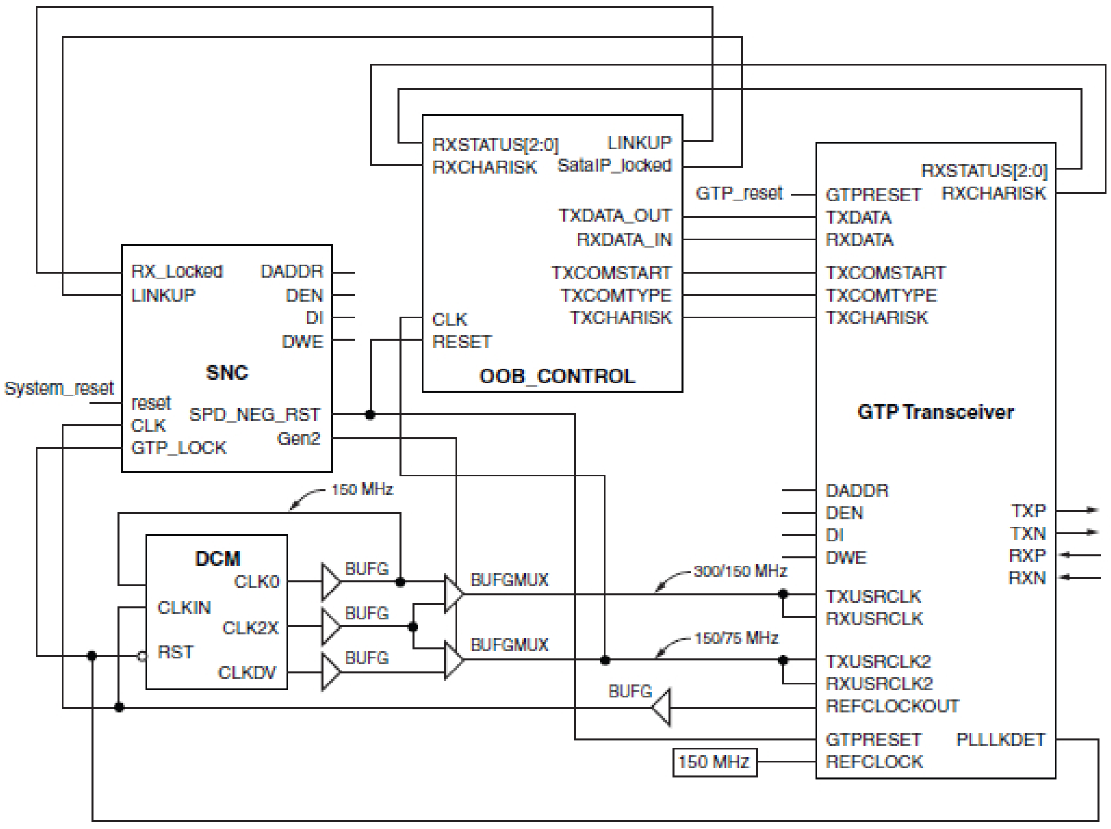


Figure 3.8: Serial ATA Physical Link Initialization with the GTP Transceiver of Virtex-5 LXT FPGAs block schematic - taken from [1].

Chapter 4

Design proposal

In this chapter I describe the overall system design. The most important blocks of this design are described in detail in next chapter. At the end of this chapter, reader can find which development tools I used.

4.1 Overall system design

The system consists of three major parts that are sata core, eth core and sata_to_eth_mem entity. The following figure 4.1 depicts the top level entity. The sata core entity manages Serial ATA operations, eth core manages Ethernet operations and sata_to_eth_mem entity is the junction point between them. Both sata core and eth core have their own clock domain. Eth core runs on 125Mhz clock and sata core runs either on 37.5Mhz clock for Serial ATA generation 1 speed or on 75Mhz clock for Serial ATA generation 2 speed. The main datapath width for eth core is 8 bits and for sata core is 32 bits.

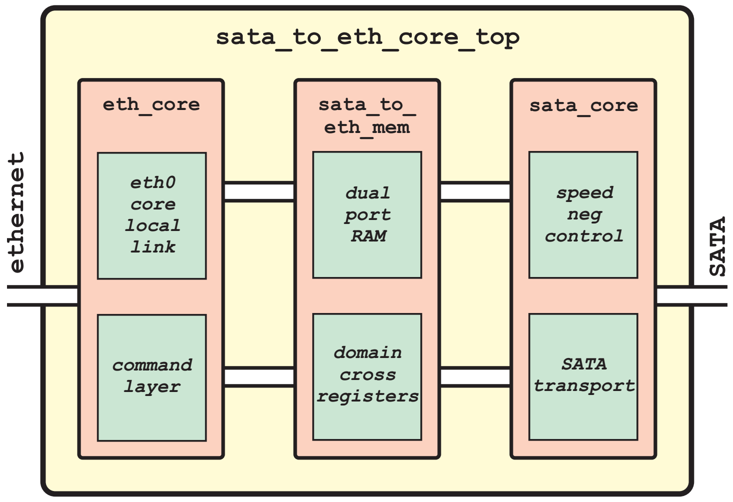


Figure 4.1: Top level block diagram

Sata_to_eth_mem entity consists of 4 dual port random access memories and of flip-flops for clock domain crossing. These random access memories act as buffers for data transfer. Two buffers are devoted to command transfer, one for command to be transmitted on sata interface (Register FIS - Host to Device) and the other one for command response from SATA interface (Register FIS – Device to Host, PIO Setup FIS). Another two much larger buffers are devoted to payload data transfer, one for data to be transmitted on SATA interface and the other for data received on SATA interface. These buffer pairs can be joined so there will be only command buffer and data buffer, however I find solution with 4 buffer better for future core development and debugging.

4.1.1 ETH core

Eth core consist of two major components, that are command_layer entity and eth0_core-_locallink entity. Eth0_core_locallink entity is the component generated by Core Generator that instantiates Tri-Mode Ethernet MAC core. I describe Core Generator configuration for generating TEMAC core used in this thesis in appendix B. This entity is ended by receive and transmit FIFO.

Command_layer entity on side communicates with SATA core through sata_to_eth_mem entity and on the others side it receives and transmit frames to sata_to_eth_mem entity through sata_to_eth_mem entity's receive and transmit FIFO. It extracts from received frame control command and data to be executed in SATA core, stores them in buffers, starts SATA core FSM, wait until the desired action is executed and then transmits the results in form of Ethernet frame. This entity is also equipped with timeout for SATA operations and due to higher clock frequency than in SATA core, this is where encryption and decryption of payload data takes place.

4.1.2 SATA core

Sata core consists of two major components that are sata_transport entity and speed-_neg_control_tile entity. Speed_neg_control_tile entity contains startup link speed negotiation FSM and also OOB control FSM. Sata_transport entity is the implementations of SATA Link and Transport layers. It consists of sata_core_cntrl, sata_rxtransport and sata_txtransport entities. Sata_rxtransport entity handles frame reception, sata_txtransport handles frame transmission and sata_core_controller control controls both of them. This is described in more detail in next chapter.

4.1.3 Ethernet communication protocol

This section describes system's communication interface with Command and Apllication layers of Serial ATA protocol. The system communicates with outer world through Etherenet frames. Table 4.1.3 describes format of frames that the system accepts. Table 4.1.3 defines format of Ethernet frames transmitted by the system to outer world. Commands that are accepted and executed by the system are described in table 4.1.3, statuses of commands execution are described in table 4.1.3. If the requested command does not operate with command buffer and data buffer, there must not be command buffer size field and command buffer content and data buffer content. If the requested command does not operate with command buffer and operates with data buffer, there must not be command buffer size field and command buffer content, the received frame contains after requested command value content of data buffer. If the requested command operates with command buffer and does not operate with data buffer, there must not be command buffer content in received frame. Same rules apply to frames, that are transmitted from the system.

Byte	**Description**
0 - 5	Destination MAC address (Standard Ethernet frame head).
6 - 11	Source MAC address (Standard Ethernet frame head).
12 - 13	Protocol ID value. I use value FFFFh. (Standard Ethernet frame head)
14	Requested command value, defined in table 4.1.3.
15	CMDBS Size of command buffer in double words. Minimum size is 1 and maximum size is 31.
16 - (CMDBS * 4 - 1)	Command buffer content
(16 + CMDBS * 4) - (16 + CMDBS * 4 + 516)	Data buffer content. This is 4 bytes of Data FIS head and 512 bytes of data to be written to hard drive(1 sector)

Table 4.1: Received Ethernet frame format

4.2 Development tools

During development of this system I used following programs and tools:

- Xilinx ISE Design Suite 10.1
- Mentor Graphics Modelsim SE 6.5a
- Wireshark 1.0.8
- Colasoft Packet Builder 1.0
- GCC + GLIBC on linux operation

Byte	**Description**
0 - 5	Destination MAC address (Standard Ethernet frame head).
6 - 11	Source MAC address (Standard Ethernet frame head).
12 - 13	Protocol ID value. I use value FFFFh. (Standard Ethernet frame head)
14	Executed command status value, defined in table 4.1.3.
15	CMDBS Size of command buffer in double words, that has been received on SATA interface. Minimum size is 1 and maximum size is 31.
16 - (CMDBS * 4 - 1)	Command buffer content received on SATA interface
(16 + CMDBS * 4) - (16 + CMDBS * 4 + 516)	Data buffer content. This is 4 bytes of Data FIS head and 512 bytes of data that were read from hard drive(1 sector)

Table 4.2: Transmitted Ethernet frame format

Command name	**Value**	**Description**
RESET	1h	This command resets the whole Serial ATA part of the system.
WrKEY	2h	This command writes the 128-bit length AES encryption/decryption key. After receiving this command, the system remembers the key until this command is received again or the system is reseted.
RdCmdDataNEX	Ah	This command just transmits on Ethernet interface content of receive command and data buffers. No SATA action is executed. I used this command for debugging, because it contains responses of previously executed SATA commands.
WrCmdRdCmd	Bh	This command realizes SATA Non-data command protocol. It transmits content of command buffer that has been extracted from received Ethernet frame on SATA interface, waits until command response is received on SATA interface, and then transmits the response on Ethernet interface.
WrCmd	3h	This command transmits content of command buffer that has been extracted from received Ethernet frame on SATA interface and waits until transmitted.
RdCmd	8h	This command waits until PIO Setup FIS or Register FIS - Device to Host is received on SATA interface and then transmits its content on SATA interface.
WrData	Ch	This command extracts the payload data from received Ethernet frame and then transmits them on SATA interface through Data FIS.
WrDataCrypt	Dh	This command encrypts the data extracted from Ethernet frame and then transmits them on SATA interface through Data FIS.
RrData	Eh	This command waits for reception of Data FIS and then transmits its content on Ethernet interface.
RdDataDecrypt	Fh	This command waits for reception of Data FIS, decrypts its content and then transmits decrypted data on Ethernet interface.

Table 4.3: Commands

Command response value	**Description**
1h	Command executed normally.
Fh	Command execution failed. Main timeout expired.

Table 4.4: Command response values

Chapter 5

Implementation

5.1 Command layer

Receiving and transmitting Ethernet frames handles command_layer entity. The following figure 5.1 illustrates command_layer entity and its components.

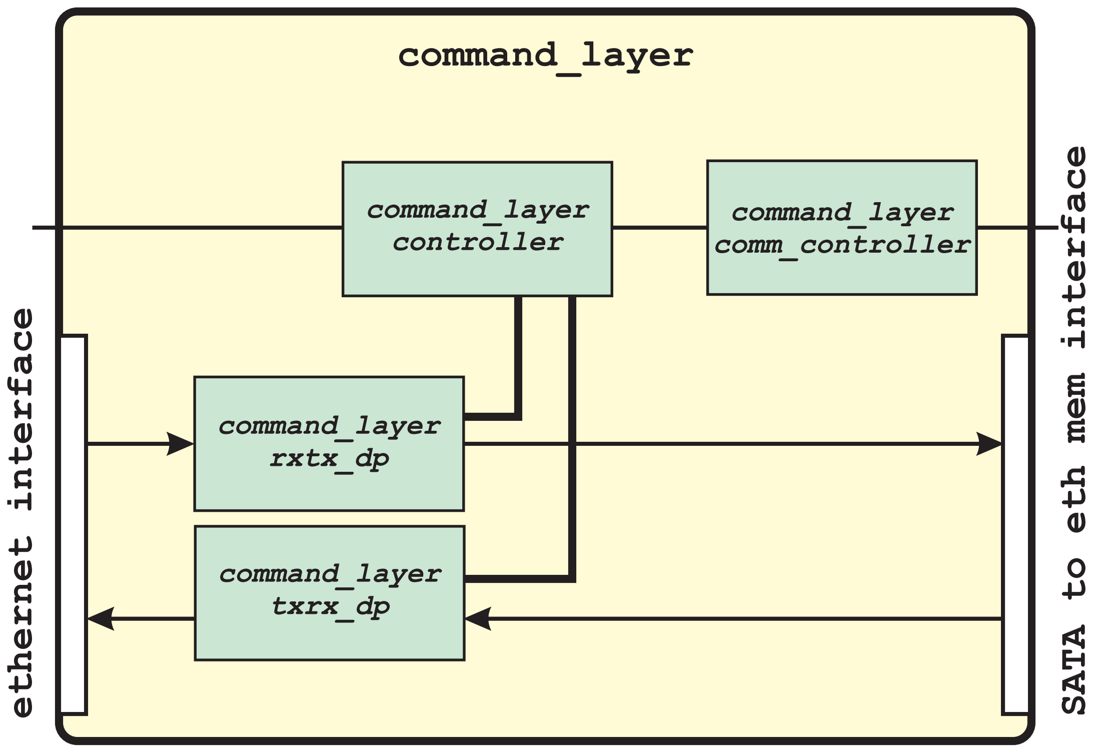


Figure 5.1: Command layer

The command_layer_controller entity is a FSM that controls command_layer_rxtx_dp (receive from Ethernet, transmit on SATA datapath) and command_layer_txrx_dp (receive from SATA, transmit on Ethernet datapath) entities. These entities are depicted in figures 5.2 and 5.3.

The command_layer_comm_controller entity is a FSM that is responsible for handshake across clock domain. It start SATA FSM and waits until the requested operation is completed. It also contains timeout counter, so if the SATA operation does not finish until the timeout expires, the status value FFh is reported in response Ethernet frame.

When the frame is received, following happens:

- Frame destination MAC address is extracted and stored in register.
- Frame source MAC address is extracted and stored in register.
- Protocol ID values are checked, if they don't match value FFFFh the operations are aborted and the state machine waits for another frame.
- Requested command value is extracted and stored in register.
- If the requested command includes transfer of command buffer size and command buffer data, these data are extracted from received frame and stored in RAM. The same happens if the requested command includes data buffer content. If the requested command also involves data buffer encryption, the data are transferred to RAM through AES encryption core. The encryption core accepts 128bit length block and produces encrypted block in same length. The encryption process takes 10 clock cycles. Because 8 bits of data are received from FIFO at a time, the shift register needs 16 clock cycles for building 128 bit length block, so while the shift register is building the 128 bit length block, previous block of data is being encrypted. This means that the difference in time between receiving the payload data and just storing in RAM and receiving the payload data, encrypting them a then storing in RAM is insignificant.
- The SATA operation is started through command_layer_comm_controller entity and the FSM waits until it finishes or timeout expires.

When SATA operation has completed, the following happens:

- Received frame source MAC address is sent as destination MAC address of transmitted frame.
- Received frame destination MAC address is sent as source MAC address of transmitted frame.
- Protocol ID value FFFFh is sent.
- Executed SATA operation status value is sent.
- If the requested command includes transfer of command buffer size and command buffer data that were received on SATA interface, these data are transferred from RAM to Ethernet transmit FIFO. The same happens if the requested command includes transfer of data buffer received on SATA interface. If the requested command also involves data

buffer decryption, the data are transferred from RAM to AES decryption core. The decryption core accepts 128bit length block of encrypted data and produces decrypted block in same length. The decryption process takes 10 clock cycles. Because Ethernet transmit FIFO accepts 8 bits of data at a time, the shift register needs 16 clock cycles for shifting 128 bit length block to 8 bit length data, so while the shift register is shifting, the next 128 bit length block is being decrypted. This means that the difference in time between data movement from RAM to Ethernet transmit FIFO and data movement from RAM to Ethernet transmit FIFO with decryption process is insignificant.

- If the requested command does not include in response any data, pad bytes and EOF are sent, else just EOF is sent.

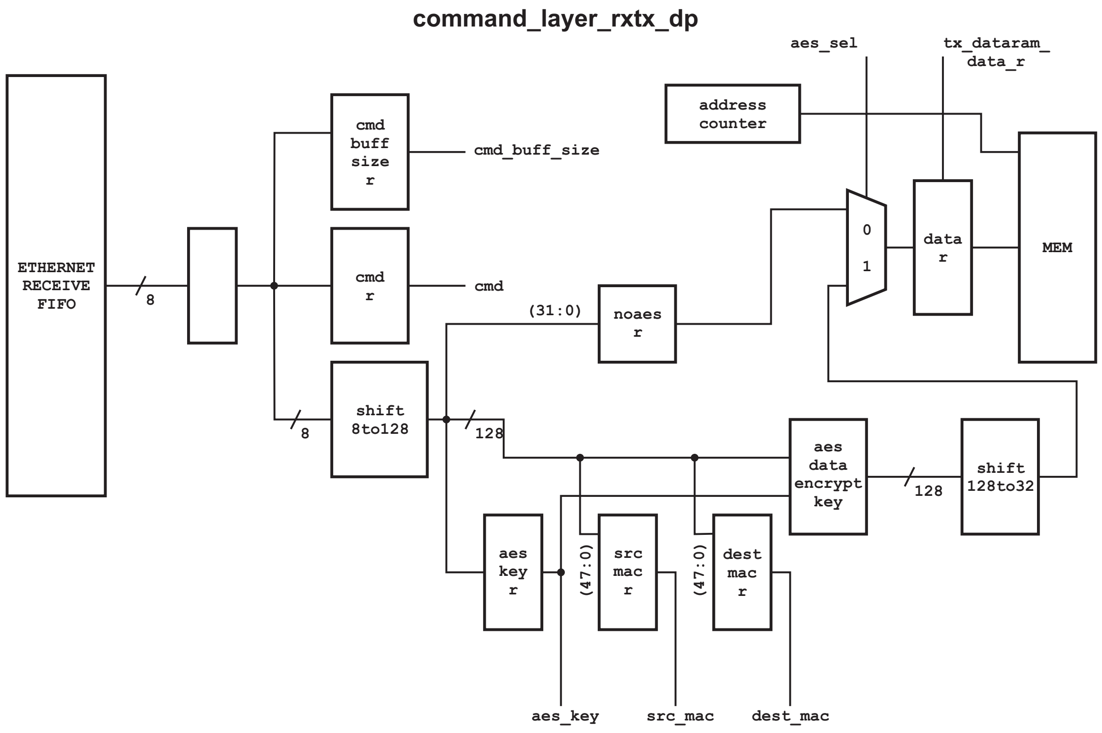


Figure 5.2: Command layer - receive from Ethernet, transmit on SATA datapath

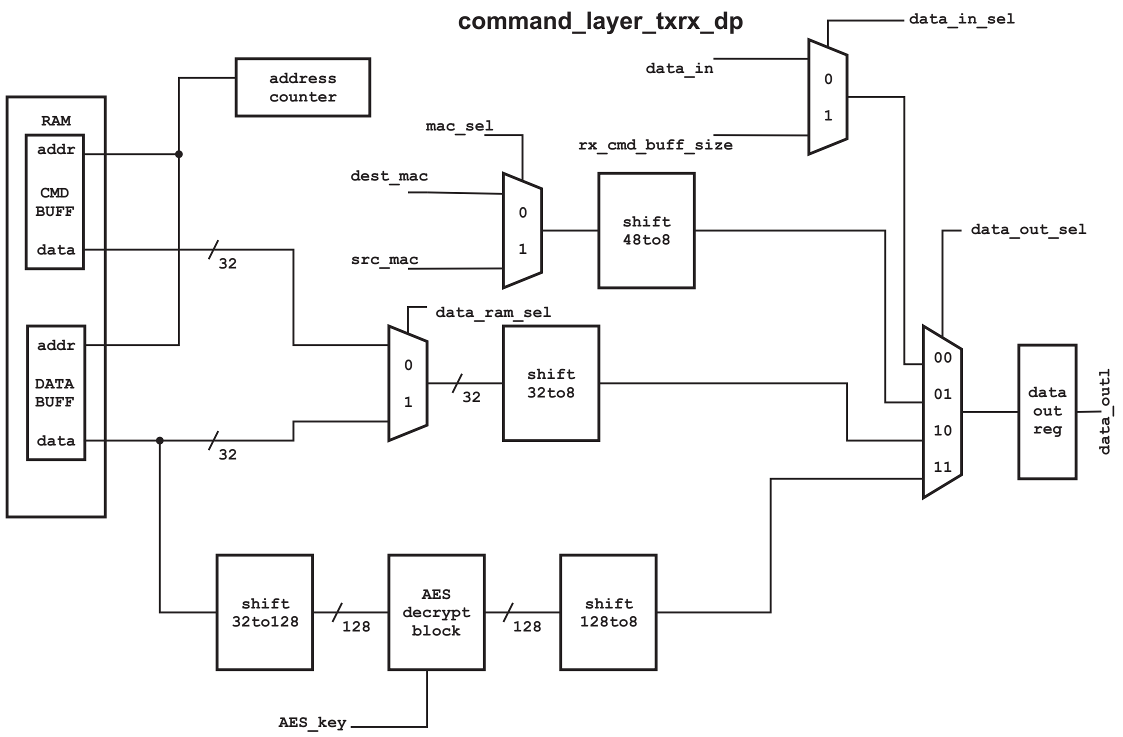


Figure 5.3: Command layer - transmit on Ethernet, receive from SATA datapath

5.2 SATA Link and Transport layers

Entity sata_transport and its components implement Link and Transport layers of Serial ATA protocol. Figure 5.4 depicts block diagram of this entity. This entity also contains ChipScope Pro Integrated Logic Analyzer core and ChipScope Pro Integrated Controller. More details about debugging and verification using ChipScope Pro can reader find in next chapter.

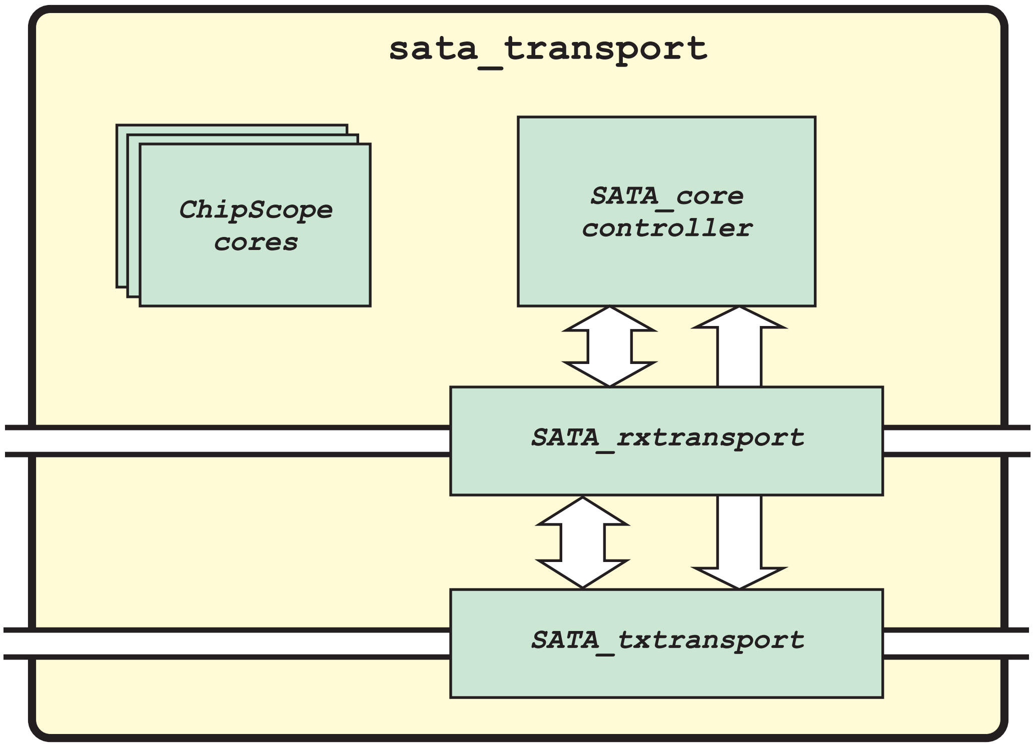


Figure 5.4: SATA Transport and Link layers

Sata_core_contoller is the main FSM of the Serial ATA part of the design. It receives requests from command_layer_comm_controller entity that is placed in the Ethernet part of the design. According to reqeusted command (command definitions are in table 4.1.3) the FSM controls sata_rx_transport a sata_tx_transport entities. Following figure 5.5 illustrates the block level diagram of sata_rx_transport entity and figure 5.7 illustartes the block level diagram of sata_tx_transport. Graphical representation of sata_core_contoller FSM can reader find in document sata_core_contoller.pdf on appended CD.

After reset the sata_core_contoller's FSM waits for link to become ready. OOB control FSM after successful startup link speed negotiation signalizes that link is ready. When the link becomes ready, sata_core_contoller waits for reception of first Register FIS - Device to Host, which contains device signature. Then the FSM signalizes that Transport layer is ready meaning that it is ready to process commands.

Sata_rx_transport entity consists of rx_transport_controller, rx_transport_dp and sata_rxtransport_WE components. Rx_transport_controller is a FSM, that is responsible for frame reception. Graphical representation of rx_transport_controller FSM can reader find in document sata_rxtransport_controller.pdf on appended CD. Scheme of rx-_transport_dp is depicted in figure 5.6. Entity sata_rxtransport_WE is responsible for recognizing if the received frame is Data FIS or non Data FIS. It sets accordingly write enable signals.

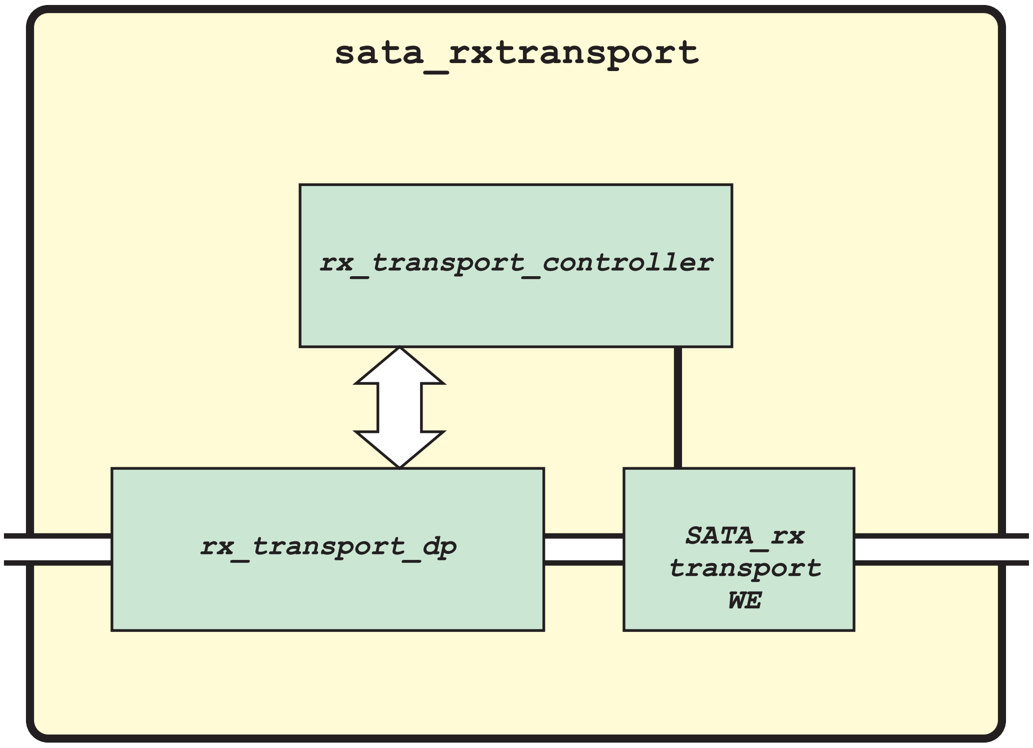


Figure 5.5: SATA rxtransport

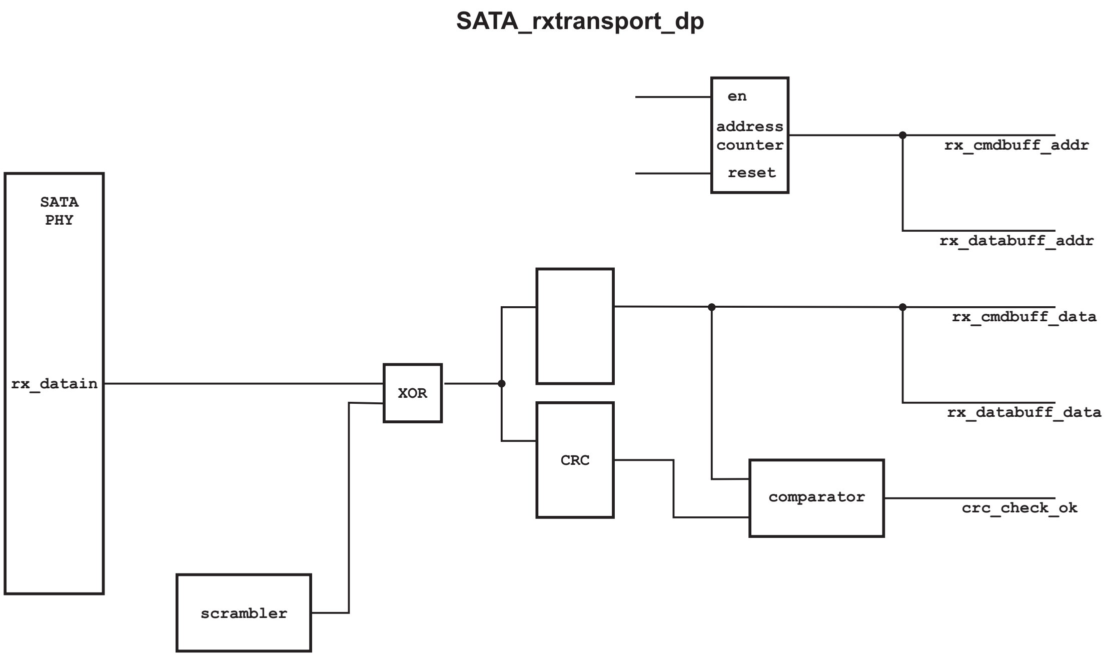


Figure 5.6: SATA rxtransport datapath

Sata_tx_transport entity consists of tx_transport_controller, tx_transport_dp and sata_align_controller components. Entity tx_transport_controller is a FSM that is responsible for frame transmission. Graphical representation of tx_transport_controller FSM can reader find in document sata_rxtransport_controller.pdf in appended CD. Scheme of tx_transport_dp is depicted in figure 5.8. Entity sata_align_controller informs the FSM that it is time to transmit ALIGN primitive pair. It is just simple counter modulo 256, when overflow happens, appropriate signal is activated and it is activated until the FSM acknowledges ALIGN primitive pair transmission.

The state machines tx_transport_controller and rx_transport_controller communicate with each other. When rx_transport_controller FSM is receiving frame, the tx_transport_controller FSM is sending primitives according to rx_transport_controller requests. When tx_transport_controller FSM is transmitting frame, it receives primitives through rx_transport_controller FSM. Rx_transport_controller FSM implements repeated primitive suppression technique.

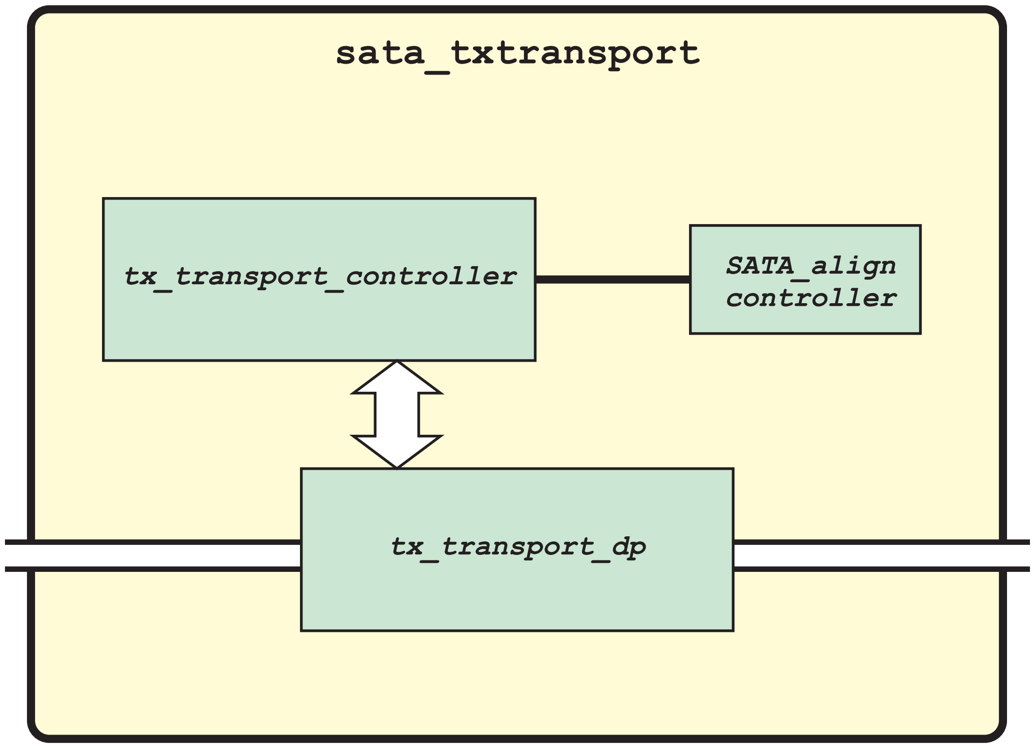


Figure 5.7: SATA txtransport

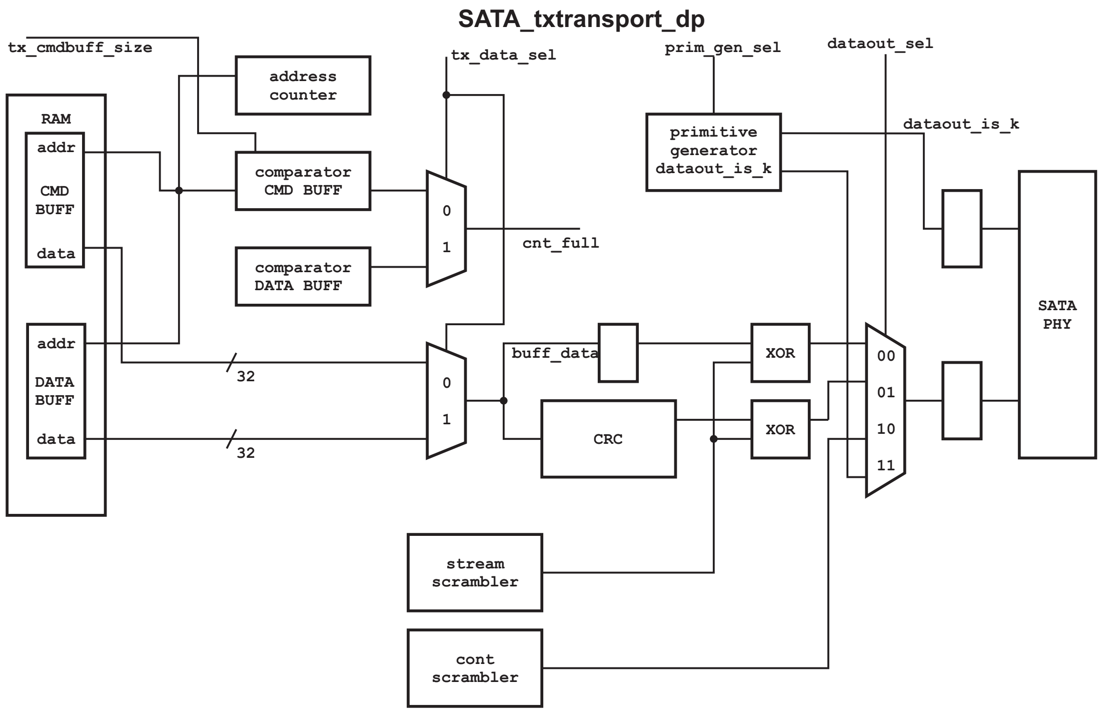


Figure 5.8: SATA txtransport datapath

5.2.1 CRC and Scrambler implementation

The Serial ATA specification provides example implementations of CRC and Scrambler in C source code. I implemented CRC and Scrambler entities according to these source codes. I also tried to use Xilinx CRC core and was unsuccessful. The Xilinx CRC core provides inverted and byte reversed output. The startup value of CRC was all right but when first data came the output was wrong. In my opinion, there may be possible bug in Xilinx CRC implementation because in other older FPGA circuits the core was soft and in Virtex-5 FPGA the core is hard.

5.3 Clock domain crossing

As I mentioned in previous chapter, both sata_core and eth_core entities have their own clock domain. Eth_core runs on 125Mhz clock and sata_core runs either on 37.5Mhz clock for Serial ATA generation 1 speed or on 75Mhz clock for Serial ATA generation 2 speed. Clock domain crossing is implemented in entity Sata_to_eth_mem, which consists of 4 dual port random access memories and of flip-flops for control signal transfer. These flip-flops are implemented as described in figure 5.9.

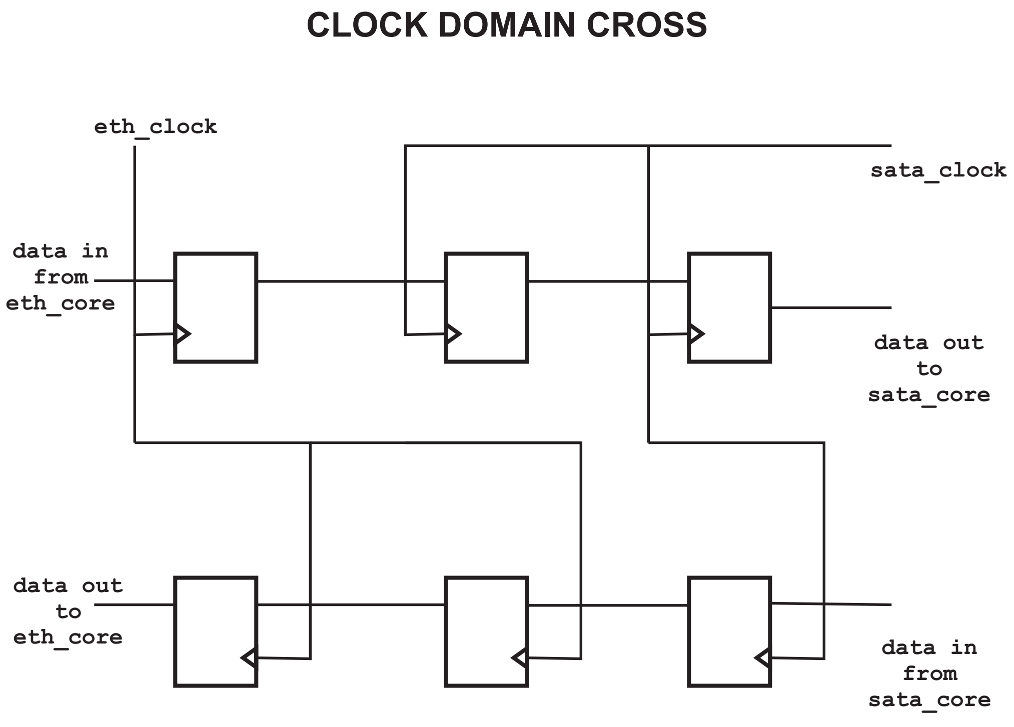


Figure 5.9: Clock domain crossing

5.4 Implementation problems and blind ways

I experienced lot of hard times making this design to work. First trouble I encountered with RocketIO GTX core configuration. Xilinx company provides in its Core generator template configuration of RocketIO GTX core for Serial ATA operations. While the core supports many other protocols except SATA, there are many parameters. In the provided template for SATA, there is set the value of SATA TX COM Sequence Bursts to 15. This value is wrong, it is supposed to be 5. This value is correctly set in the reference design [1], however it is folded in mist of other parameters, that the template sets correctly, so it is overlooked very easily. This mistake took me really lot of time.

Another trouble I encountered was making the TEMAC core to work. It looks easy, you just have to modify the User Constraints File and the example design that comes with TEMAC core should work. I don't know why but the provided example design does not bring-out a reset signal for physical interface. Without this signal, the example design does not work.

The Serial ATA specification suggests when ALIGN primitive pair is retransmitted, the suppressed primitive that is currently on the link, should be brought out and then again CONT primitive followed by output of scrambler, so the state of the bus will be visible for logical analyzer. When I implemented this feature I did a huge mistake. When the sata_rx_transport entity is receiving a frame, the sata_tx_transport is just transmitting primitives like R_RDY, R_IP, but the tx_transport's FSM is in state, when SYNC primitive was transmitted last, followed by CONT primitive and then the output of scrambler, so when was a time to retransmit ALIGN primitive pair, the FSM transmitted ALIGN primitive pair, followed by two SYNC primitives and CONT primitive and this aborted frame reception.

I also made some mistakes with my initial designs. My original thoughts were that the frame reception would be always enabled, and the PIO data protocol would be fully implemented in hardware. As I realized later, it was not possible in the scope of this thesis because I needed to examine the ATA command protocol in real. The ATA protocol is quite old and is designed to work with parallel interface. There is a bit mess in it and some relations between ATA protocol and Serial ATA protocol were not completely clear to me. I tried to implement PIO data transfer protocol fully in hardware, while the PIO Setup FIS would not be visible to the user, I realized later that there can status in PIO Setup FIS that will cause abortion of the transfer. This would require a lot of additional work that is in my opinion out of scope of this thesis.

5.5 FPGA resources used

Virtex-5 FXT utilization summary with ChipScope cores:

```
Number of BSCANs                          1 out of 4      25%
 Number of BUFDSs                          1 out of 8      12%
 Number of BUFGs                           8 out of 32     25%
    Number of LOCed BUFGs                  1 out of 8      12%

 Number of BUFGCTRLs                       2 out of 32      6%
 Number of DCM_ADVs                        1 out of 12      8%
 Number of GTX_DUALs                       1 out of 8      12%
    Number of LOCed GTX_DUALs              1 out of 1     100%

 Number of IDELAYCTRLs                     2 out of 22      9%
 Number of ILOGICs                        10 out of 800     1%
 Number of External IOBs                  76 out of 640    11%
    Number of LOCed IOBs                  37 out of 76     48%

 Number of IODELAYs                       11 out of 800     1%
 Number of External IPADs                  4 out of 690     1%
    Number of LOCed IPADs                  4 out of 4     100%

 Number of OLOGICs                        16 out of 800     2%
 Number of External OPADs                  2 out of 32      6%
    Number of LOCed OPADs                  2 out of 2     100%

 Number of RAMB18X2s                      11 out of 148     7%
 Number of RAMB18X2SDPs                    2 out of 148     1%
 Number of RAMB36_EXPs                    17 out of 148    11%
 Number of TEMACs                          1 out of 2      50%
 Number of Slice Registers              4287 out of 44800   9%
    Number used as Flip Flops           4287
    Number used as Latches                 0
    Number used as LatchThrus              0

 Number of Slice LUTS                   4855 out of 44800  10%
 Number of Slice LUT-Flip Flop pairs    6576 out of 44800  14%
```

Virtex-5 FXT utilization summary without ChipScope cores:

```
Device Utilization Summary:

   Number of BUFDSs                          1 out of 8       12%
   Number of BUFGs                           7 out of 32      21%
      Number of LOCed BUFGs                  1 out of 7       14%

   Number of BUFGCTRLs                       2 out of 32       6%
   Number of DCM_ADVs                        1 out of 12       8%
   Number of GTX_DUALs                       1 out of 8       12%
      Number of LOCed GTX_DUALs              1 out of 1      100%

   Number of IDELAYCTRLs                     2 out of 22       9%
   Number of ILOGICs                        10 out of 800      1%
   Number of External IOBs                  76 out of 640     11%
      Number of LOCed IOBs                  37 out of 76      48%

   Number of IODELAYs                       11 out of 800      1%
   Number of External IPADs                  4 out of 690      1%
      Number of LOCed IPADs                  4 out of 4      100%

   Number of OLOGICs                        16 out of 800      2%
   Number of External OPADs                  2 out of 32       6%
      Number of LOCed OPADs                  2 out of 2      100%

   Number of RAMB18X2s                      11 out of 148      7%
   Number of RAMB18X2SDPs                    2 out of 148      1%
   Number of RAMB36_EXPs                     4 out of 148      2%
   Number of TEMACs                          1 out of 2       50%
   Number of Slice Registers              3081 out of 44800    6%
      Number used as Flip Flops           3081
      Number used as Latches                 0
      Number used as LatchThrus              0

   Number of Slice LUTS                   3998 out of 44800    8%
   Number of Slice LUT-Flip Flop pairs    5038 out of 44800   11%
```

Chapter 6

Testing

6.1 Testbenchs

I performed numerous behavioral and timing simulations, however I did not simulate the system as one unit while I was unsuccessful obtaining hard drive model. I simulated separately Serial ATA part of the system and Ethernet part of the system. My testbenches provide only stimuli, I verified the results by viewing content of random access memories. I did not simulate the entities that were provided in Xilinx reference designs. The testbench files are provided on appended CD.

6.1.1 Simulation of command_layer entity and its components

Testbench command_layer_tb.vhd tests command_layer entity and its components. It provides stimuli in form how Ethernet receive FIFO does. Sata_to_eth_mem entity is replaced by entity command_layer_memloop which merges receive and transmit buffers (command and data) into loop buffer (command and data). Stimuli are commented in testbench file. The following figure 6.1 depicts fragment of simulation.

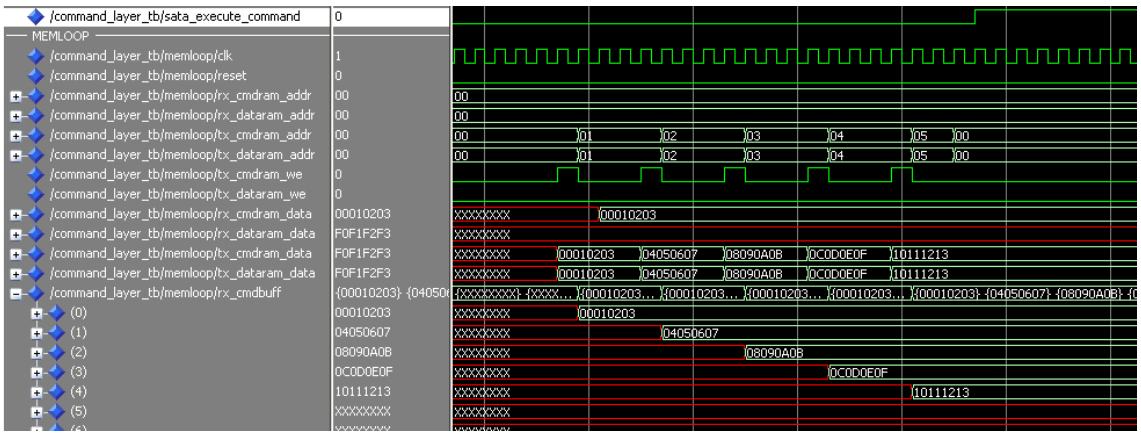

/command_layer_tb/sata_execute_command 0
MEMLOOP
/command_layer_tb/memloop/clk 1
/command_layer_tb/memloop/reset 0
/command_layer_tb/memloop/rx_cmdram_addr 00
/command_layer_tb/memloop/rx_dataram_addr 00
/command_layer_tb/memloop/tx_cmdram_addr 00
/command_layer_tb/memloop/tx_dataram_addr 00
/command_layer_tb/memloop/tx_cmdram_we 0
/command_layer_tb/memloop/tx_dataram_we 0
/command_layer_tb/memloop/rx_cmdram_data 00010203
/command_layer_tb/memloop/rx_dataram_data F0F1F2F3
/command_layer_tb/memloop/tx_cmdram_data F0F1F2F3
/command_layer_tb/memloop/tx_dataram_data F0F1F2F3
/command_layer_tb/memloop/rx_cmdbuff {00010203} {04050
(0) 00010203
(1) 04050607
(2) 08090A0B
(3) 0C0D0E0F
(4) 10111213
(5) XXXXXXXX

Figure 6.1: Fragment of command_layer entity simulation

6.1.2 Simulation of sata_transport entity and its components

Testbench sata_transport_tb.vhd tests stat_transport entity and its components. The incoming data from SATA Physical layer are simulated using entity sata_test_data. This entity contains 4 SATA frames. One of them is response from hard drive to ATA command IDENTIFY DEVICE captured by ChipScope. Sata_to_eth_mem entity is replaced by entity sata_test_mem, which is the same as entity command_layer_memloop that is described in previous section.

I also created many other testbenchs that tested smaller parts of the design, but they are now obsolete, because these parts rapidly changed during system development. These testbenchs are also included on appended CD due to future design development.

6.2 Chipscope testing

I placed ChipScope Pro Integrated Logic Analyzer core and ChipScope Pro Integrated Controller core in the sata_transport. The Integrated Logic Analyzer core monitors almost all inputs, outputs and significant internal signal of sata_transport entity. States of sata_core_contoller, tx_transport_controller and rx_transport_controller state machines are monitored as well. The following figure 6.2 views captured frame transmission captured by ChipScope Pro software. ChipScope project files are included on appended CD in directory chipscope.

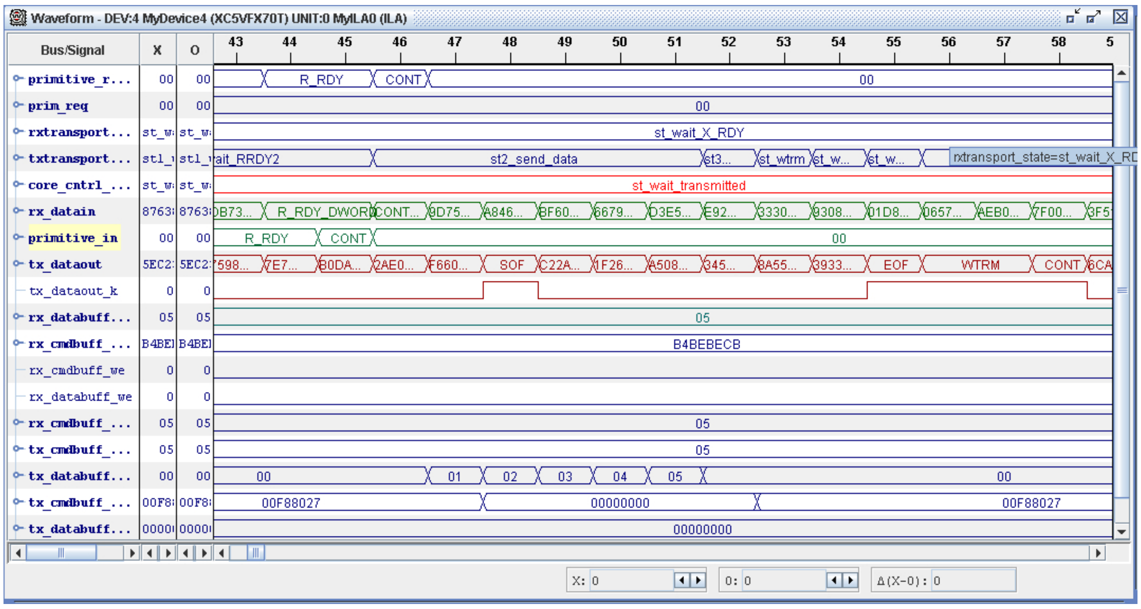


Figure 6.2: Debugging and protocol analysis with ChipScope Pro software

6.3 Test programs

I created a small library and few test programs, that are:

- reset - This program issues command that resets the whole Serial ATA part of the system.
- writekey - This program writes 128 bit length AES encryption/decryption key. The encryption key is located in file sata_to_eth.h on appended CD.
- identifydevice - This program issues ATA command IDENTIFY DEVICE and displays the output
- read_native_max_address_ext - This program issues ATA command READ NATIVE MAX ADDRESS EXT and displays the output.
- read_sector_ext - This program issues ATA command READ SECTOR EXT and displays the output.
- read_sector_ext_crypt - This program issues ATA command READ SECTOR EXT, instructs the hardware to decrypt the data and displays the output.
- write_sector_ext - This program issues ATA command WRITE SECTOR EXT.
- write_sector_ext_crypt - This program issues ATA command WRITE SECTOR EXT and instructs the hardware to encrypt the data.

These test computer programm are intended for testing the hardware. Parameters are stored in source code file sata_to_eth.h. Programs write_sector_ext and write_sector_ext-_crypt use as a sector content increment sequence starting from 0 and ending with 255. This sequence is repeated twice. These programs are included on appended CD in directory testapp.

I also created a script test.sh that issues these programs in following sequence:

```
reset
writekey
identifydevice
read_native_max_address_ext
write_sector_ext
read_sector_ext
read_sector_ext_crypt
write_sector_ext_crypt
read_sector_ext
read_sector_ext_crypt
```

The output of this sequence is also included on appended CD in file test.log.

Example output of read_native_max_address_ext test program

```
Frame recived Lenght: 84

Frame status: OK  status: 1

Register FIS - Device To Host

        Error: 0x0
        Status: 0x50 BSY: 0 DRDY: 1 DF: 0 ERR:0
                Control register update, rir: 0x40
        FIS type: 0x34
        Device: 0x0
        LBA high: 0x93
        LBA mid: 0x90
        LBA low: 0xaf
        LBA high extended: 0x0
        LBA mid extended: 0x0
        LBA low extended: 0x2e
        Sector count extended: 0x0
        Sector count: 0x0
```

Example output of identifydevice test program

```
Frame recived Lenght: 531

Frame status: OK  status: 1
DATABUFF
, fis: 0x0 0x0 0x0 0x46

Addr: 0                 0x3f 0xff 0x42 0x7a
Addr: 1                 0x0 0x10 0xc8 0x37
Addr: 2                 0x0 0x0 0x0 0x0
Addr: 3                 0x0 0x0 0x0 0x3f
Addr: 4                 0x0 0x0 0x0 0x0
Addr: 5                 0x20 0x20 0x20 0x20
Addr: 6                 0x44 0x2d 0x20 0x57
Addr: 7                 0x41 0x4e 0x57 0x43
Addr: 8                 0x35 0x34 0x55 0x31
Addr: 9                 0x37 0x31 0x32 0x31
Addr: 10                0x80 0x0 0x0 0x0
Addr: 11                0x30 0x37 0x0 0x32
Addr: 12                0x32 0x45 0x2e 0x30
Addr: 13                0x57 0x44 0x30 0x37
Addr: 14                0x57 0x44 0x43 0x20
Addr: 15                0x30 0x30 0x34 0x30
```

```
Addr: 16                0x2d 0x30 0x4b 0x53
Addr: 17                0x4e 0x42 0x30 0x4d
Addr: 18                0x20 0x20 0x30 0x20
Addr: 19                0x20 0x20 0x20 0x20
Addr: 20                0x20 0x20 0x20 0x20

!! OUTPUT OMITTED !!

Addr: 126               0x0 0x0 0x0 0x0
Addr: 127               0x1f 0xa5 0x0 0x0


Serial number:          "     WD-WCANU1542171"
Firmware revision:      "07.02E07"
Model number:           "WDC WD4000KS-00MNB0                    "
```

Chapter 7

Conclusion

I managed to fulfill the Master thesis statement completely.

The result of this thesis is a working system that on one hand receives and transmits Ethernet frames and on the other hand reads and writes data on hard drive. The system contain AES cryptographic core that makes possible to encrypt or decrypt the data. The system has been tested with real hard drive.

The Ethernet side of this system was not contained in the Master Thesis statement, but it was necessary for testing and debugging. I had several possibilities how to generate and collect testing data, while Serial RS232 interface is too slow, EDK generated system is quite common in other Master thesis. I found Ethernet interface the most interesting and the system can be cut in the half and then after few modifications used in EDK generated system.

I managed to implement Serial ATA Link layer fully in the hardware, while part of the Transport layer is implemented in the software. This is also big advantage because lot of other Serial ATA protocol features that are out of scope of this thesis can be observed and tested by just making few modifications to software instead of writing complicated hardware. I also needed to observe some of protocol's behavior while it was not very clear to me just by reading the Serial ATA specification.

I used one of freely available implementations of AES cipher instead of making one from scratch due to in my opinion I would not make a better one and also development and testing the other parts of the system was very time consuming.

Other result of this work is Serial ATA protocol analyzer using Xilinx ChipScope Pro software.

There may be many possibilities for future development and usage of results of this Master thesis. Transport and Command layers can be fully implemented in hardware. The system can be incorporated in EDK generated system. The junction point of this system, random access memory, can be replaced by FIFO. The error management could be improved.

Bibliography

[1] Xapp870 - serial ata physical link initialization with the gtp transceiver of virtex-5 lxt fpgas.
`http://www.xilinx.com/`.

[2] 128/192 aes.
`http://opencores.org/?do=project&who=systemcaes`.

[3] Aes core modules.
`http://opencores.org/?do=project&who=aes_128_192_256`.

[4] Aes128.
`http://opencores.org/?do=project&who=aes_crypto_core`.

[5] Virtex-5 fpga rocketio gtp transceiver user guide.
`http://www.xilinx.com/`.

[6] Serial ata specification rev. 2.5.
`http://sata-io.org/`.

[7] Sata ip market survey.
`http://www.design-reuse.com/research/sata/`.

[8] Ml507 board documentation.
`http://www.xilinx.com/products/boards/ml507/docs.htm`.

[9] Synopsys.
`http://www.synopsys.com/`.

[10] Nuvation.
`http://www.nuvation.com/`.

[11] Serial ata international organization.
`http://sata-io.org/`.

[12] Information technology - at attachment with packet interface - 7.
`http://www.t10.org/`.

[13] Aes (rijndael) ip core.
`http://opencores.org/?do=project&who=aes_core`.

[14] D. Anderson. *SATA Storage Technology.* MindShare, Inc., 1st edition, 2007.

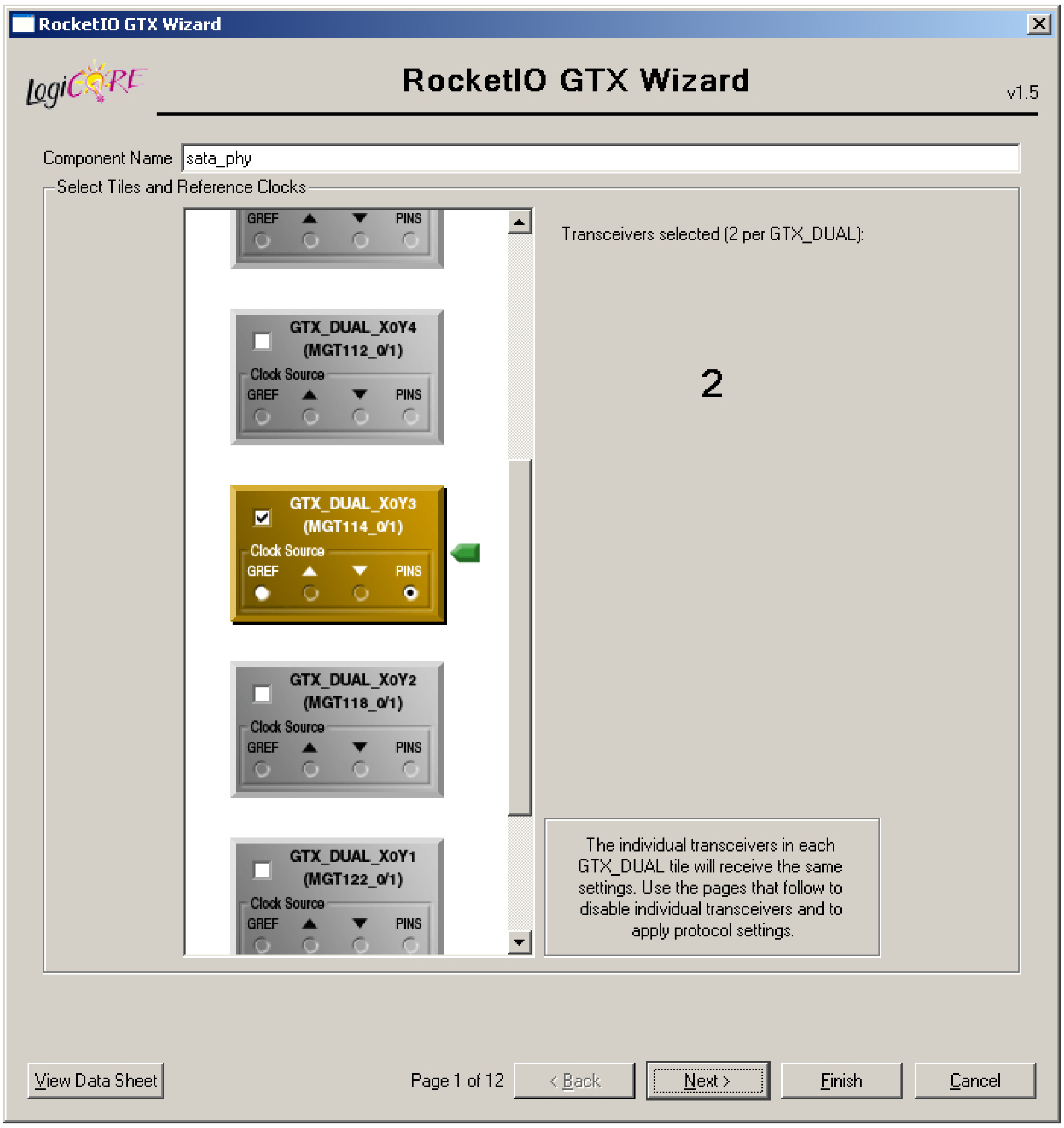
[15] O. Baar. Šifrování , symetrické šifrování, 2007.
`http://www.owebu.cz/bezpecnost/vypis.php?clanek=1217`.

[16] M. Henricksen. Design, implementation and cryptanalysis of modern symmetric ciphers, 2005. `http://eprints.qut.edu.au/16055/`.

Appendix A

Configuring RocketIO GTX for SATA functions

The following figures describe Xilinx Core Generator configuration for generating RocketIO GTX core used in this design.


RocketIO GTX Wizard

RocketIO GTX Wizard v1.5

Component Name: sata_phy

Select Tiles and Reference Clocks

GTX_DUAL_X0Y4 (MGT112_0/1) — Clock Source: GREF ▲ ▼ PINS

[✓] GTX_DUAL_X0Y3 (MGT114_0/1) — Clock Source: [● GREF] ▲ ▼ [● PINS]

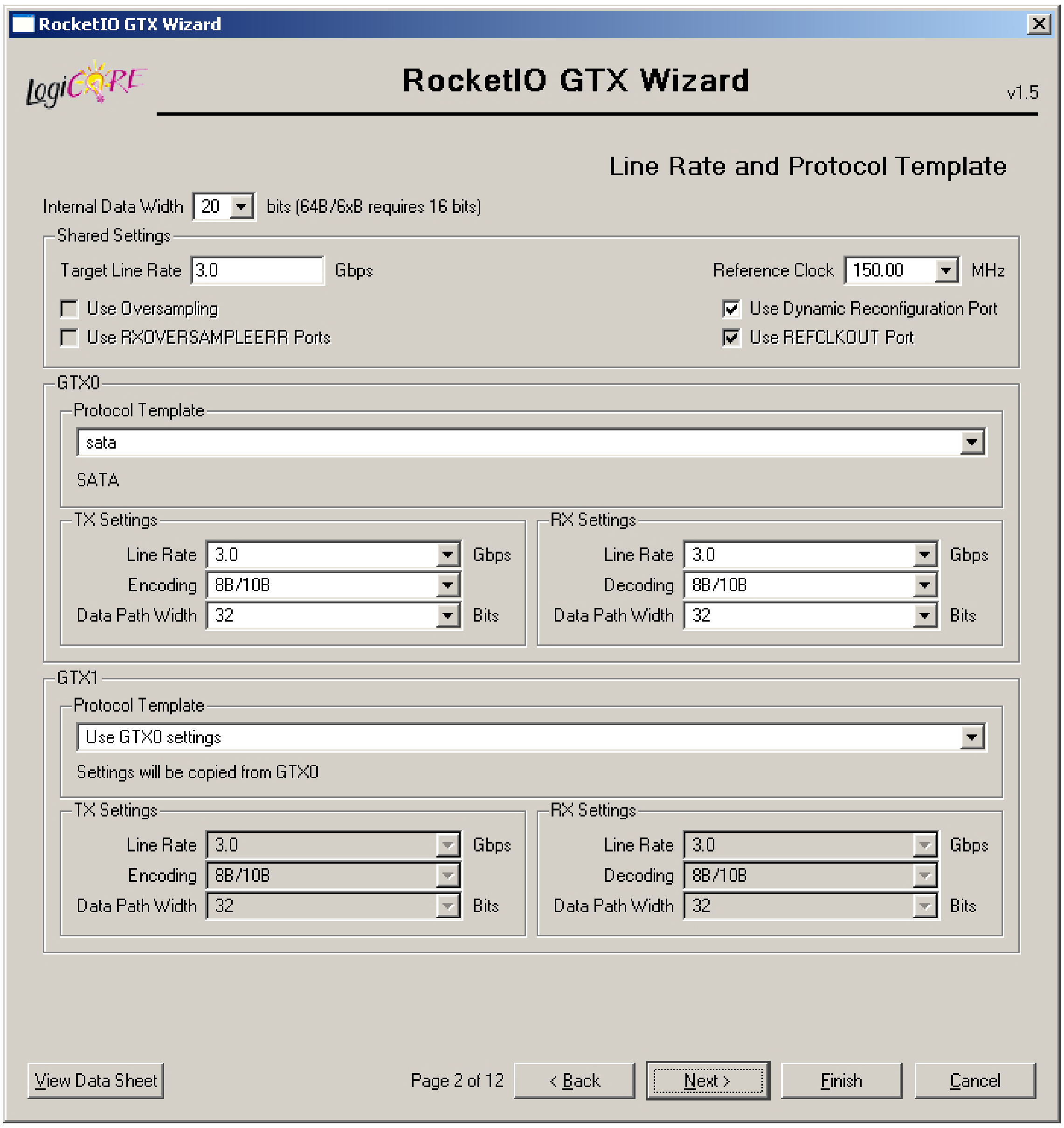
GTX_DUAL_X0Y2 (MGT118_0/1) — Clock Source: GREF ▲ ▼ PINS

GTX_DUAL_X0Y1 (MGT122_0/1) — Clock Source: GREF ▲ ▼ PINS

Transceivers selected (2 per GTX_DUAL): 2

The individual transceivers in each GTX_DUAL tile will receive the same settings. Use the pages that follow to disable individual transceivers and to apply protocol settings.

Figure A.1: Configuring RocketIO GTX for SATA functions - Step 1



Internal Data Width: 20 bits (64B/6xB requires 16 bits)

Shared Settings

Target Line Rate: 3.0 Gbps

Reference Clock: 150.00 MHz

☐ Use Oversampling

☐ Use RXOVERSAMPLEERR Ports

☑ Use Dynamic Reconfiguration Port

☑ Use REFCLKOUT Port

GTX0

Protocol Template: sata

SATA

TX Settings			RX Settings		
Line Rate	3.0	Gbps	Line Rate	3.0	Gbps
Encoding	8B/10B		Decoding	8B/10B	
Data Path Width	32	Bits	Data Path Width	32	Bits

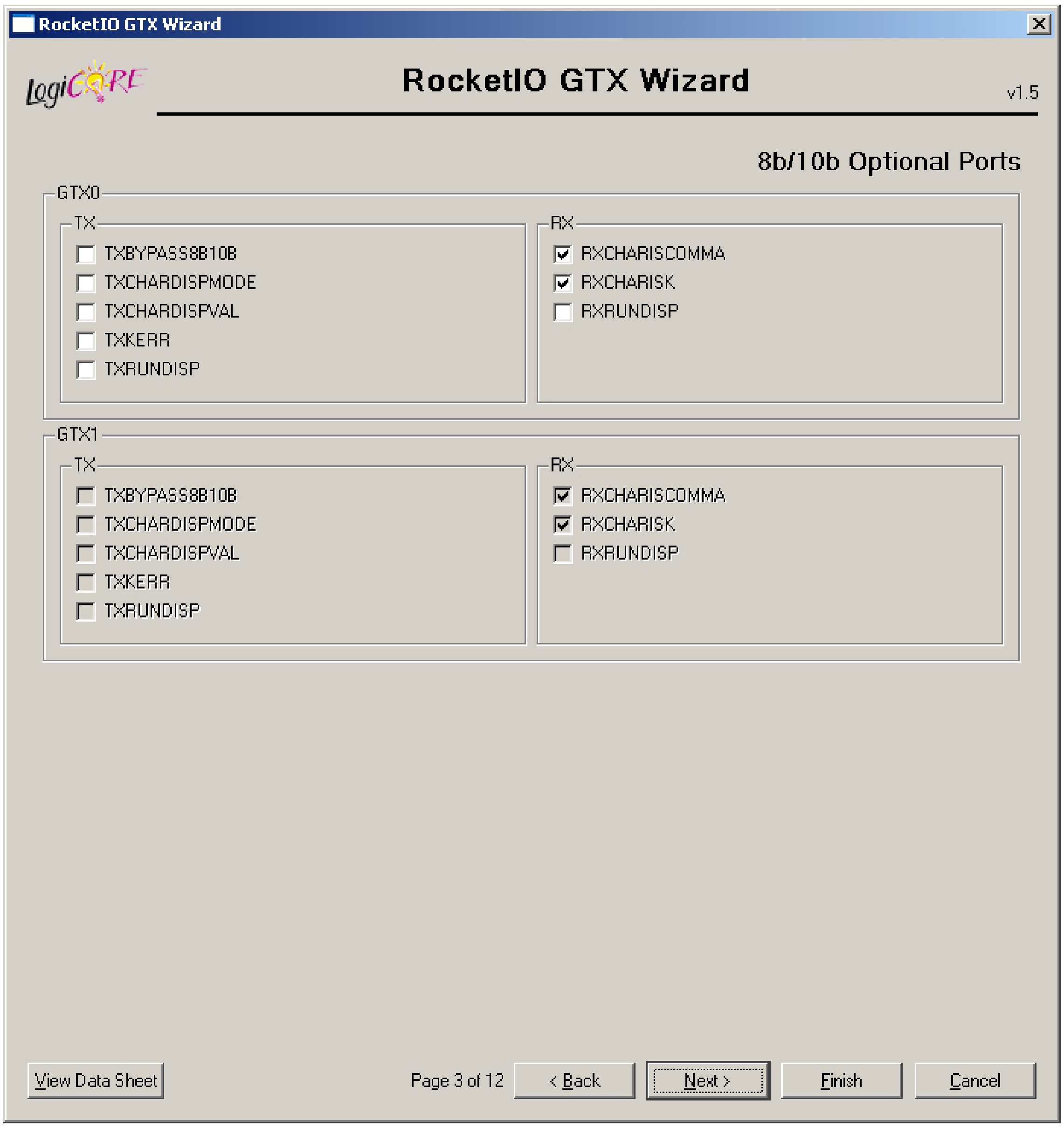
GTX1

Protocol Template: Use GTX0 settings

Settings will be copied from GTX0

TX Settings			RX Settings		
Line Rate	3.0	Gbps	Line Rate	3.0	Gbps
Encoding	8B/10B		Decoding	8B/10B	
Data Path Width	32	Bits	Data Path Width	32	Bits

Figure A.2: Configuring RocketIO GTX for SATA functions - Step 2



RocketIO GTX Wizard

8b/10b Optional Ports

GTX0

TX

- [ ] TXBYPASS8B10B
- [ ] TXCHARDISPMODE
- [ ] TXCHARDISPVAL
- [ ] TXKERR
- [ ] TXRUNDISP

RX

- [✓] RXCHARISCOMMA
- [✓] RXCHARISK
- [ ] RXRUNDISP

GTX1

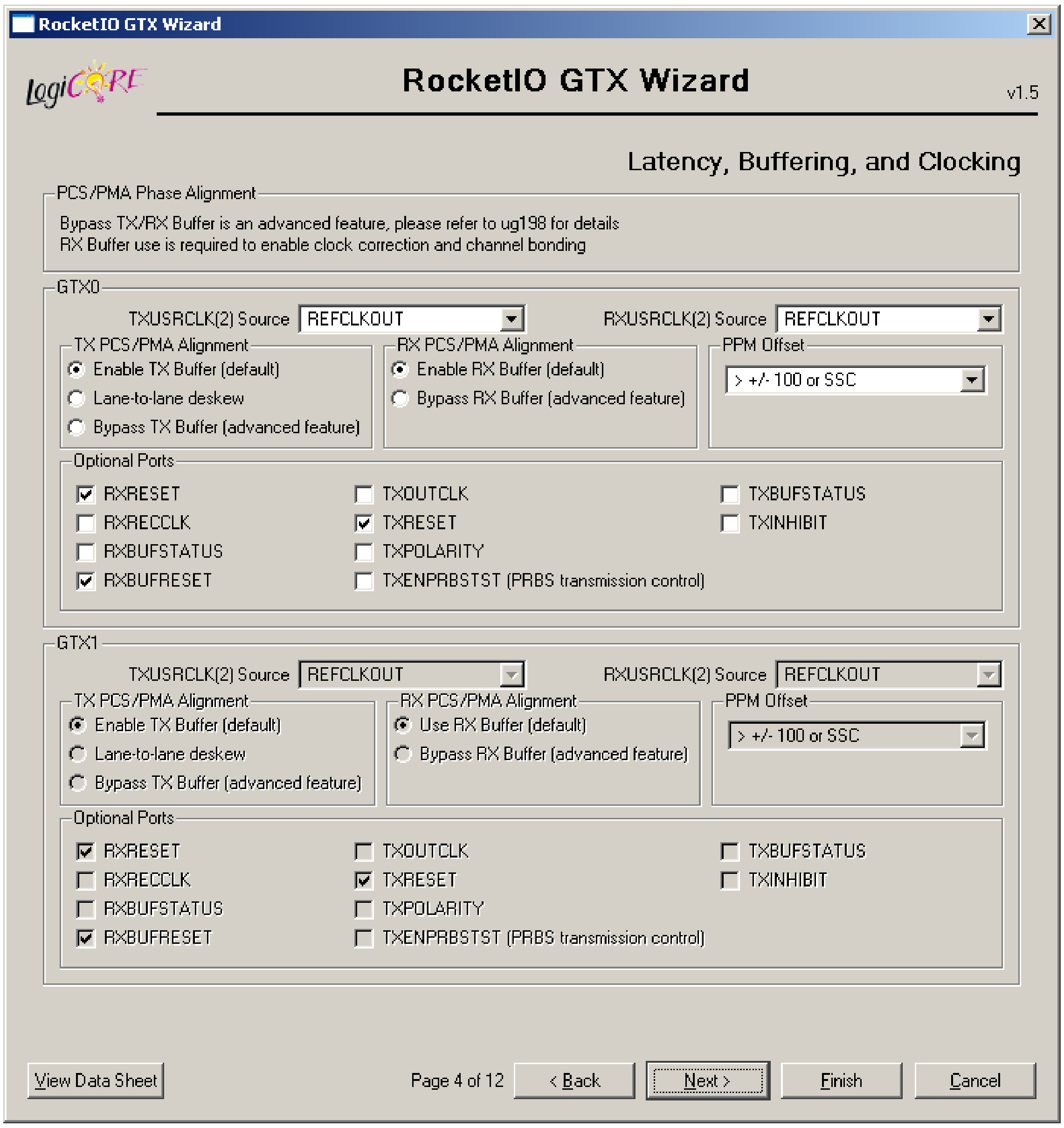
TX

- [ ] TXBYPASS8B10B
- [ ] TXCHARDISPMODE
- [ ] TXCHARDISPVAL
- [ ] TXKERR
- [ ] TXRUNDISP

RX

- [✓] RXCHARISCOMMA
- [✓] RXCHARISK
- [ ] RXRUNDISP

Figure A.3: Configuring RocketIO GTX for SATA functions - Step 3


RocketIO GTX Wizard

RocketIO GTX Wizard v1.5

Latency, Buffering, and Clocking

PCS/PMA Phase Alignment

Bypass TX/RX Buffer is an advanced feature, please refer to ug198 for details
RX Buffer use is required to enable clock correction and channel bonding

GTX0

TXUSRCLK(2) Source: REFCLKOUT
RXUSRCLK(2) Source: REFCLKOUT

TX PCS/PMA Alignment: [✓ Enable TX Buffer (default)] Lane-to-lane deskew; Bypass TX Buffer (advanced feature)

RX PCS/PMA Alignment: [✓ Enable RX Buffer (default)] Bypass RX Buffer (advanced feature)

PPM Offset: > +/- 100 or SSC

Optional Ports

✓ RXRESET	TXOUTCLK	TXBUFSTATUS
RXRECCLK	✓ TXRESET	TXINHIBIT
RXBUFSTATUS	TXPOLARITY	
✓ RXBUFRESET	TXENPRBSTST (PRBS transmission control)	

GTX1

TXUSRCLK(2) Source: REFCLKOUT
RXUSRCLK(2) Source: REFCLKOUT

TX PCS/PMA Alignment: [✓ Enable TX Buffer (default)] Lane-to-lane deskew; Bypass TX Buffer (advanced feature)

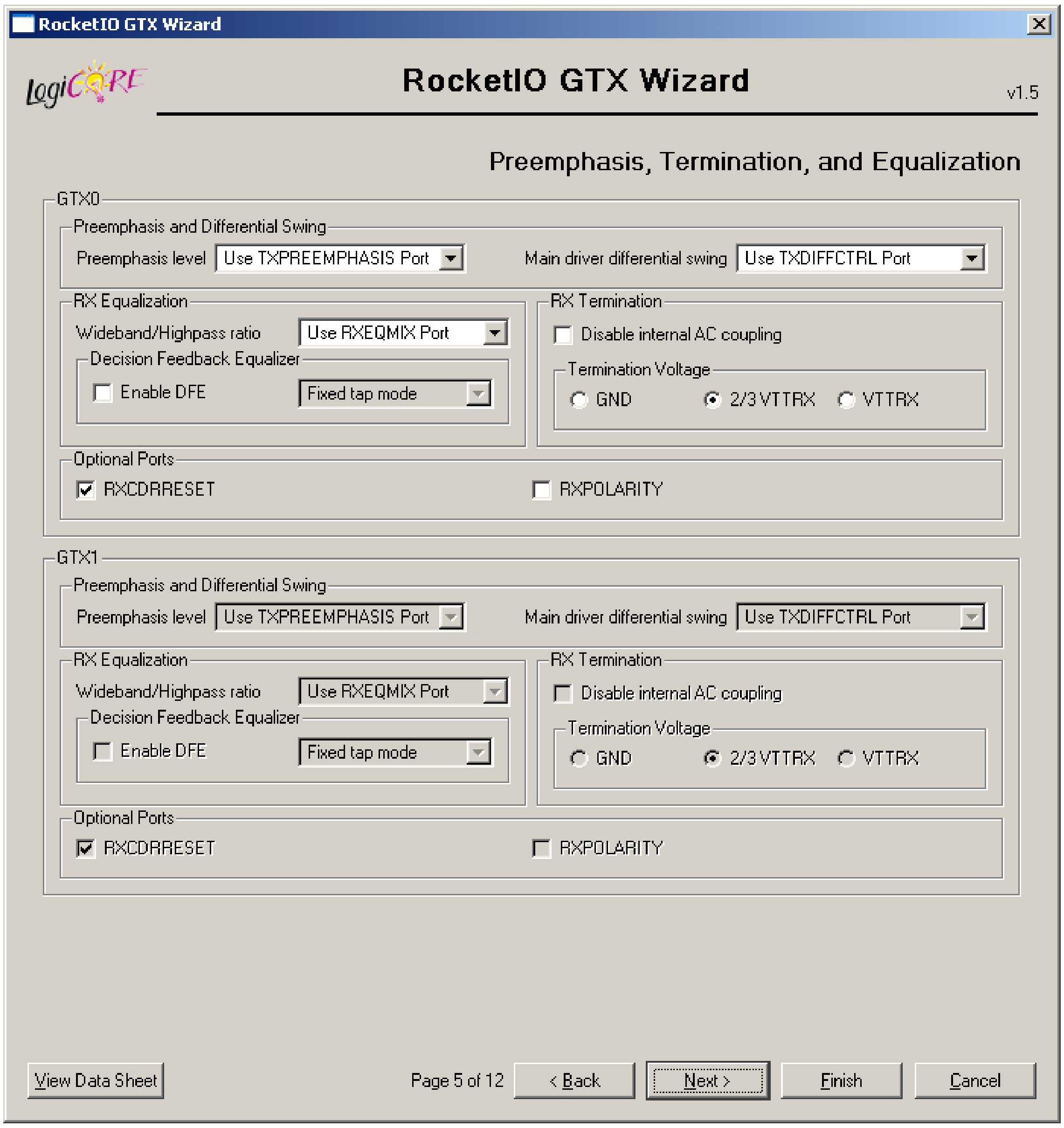
RX PCS/PMA Alignment: [✓ Use RX Buffer (default)] Bypass RX Buffer (advanced feature)

PPM Offset: > +/- 100 or SSC

Optional Ports

✓ RXRESET	TXOUTCLK	TXBUFSTATUS
RXRECCLK	✓ TXRESET	TXINHIBIT
RXBUFSTATUS	TXPOLARITY	
✓ RXBUFRESET	TXENPRBSTST (PRBS transmission control)	

Figure A.4: Configuring RocketIO GTX for SATA functions - Step 4



Preemphasis, Termination, and Equalization

GTX0

Preemphasis and Differential Swing

Preemphasis level: Use TXPREEMPHASIS Port
Main driver differential swing: Use TXDIFFCTRL Port

RX Equalization

Wideband/Highpass ratio: Use RXEQMIX Port

Decision Feedback Equalizer

☐ Enable DFE — Fixed tap mode

RX Termination

☐ Disable internal AC coupling

Termination Voltage: GND [✓ 2/3 VTTRX] VTTRX

Optional Ports

☑ RXCDRRESET
☐ RXPOLARITY

GTX1

Preemphasis and Differential Swing

Preemphasis level: Use TXPREEMPHASIS Port
Main driver differential swing: Use TXDIFFCTRL Port

RX Equalization

Wideband/Highpass ratio: Use RXEQMIX Port

Decision Feedback Equalizer

☐ Enable DFE — Fixed tap mode

RX Termination

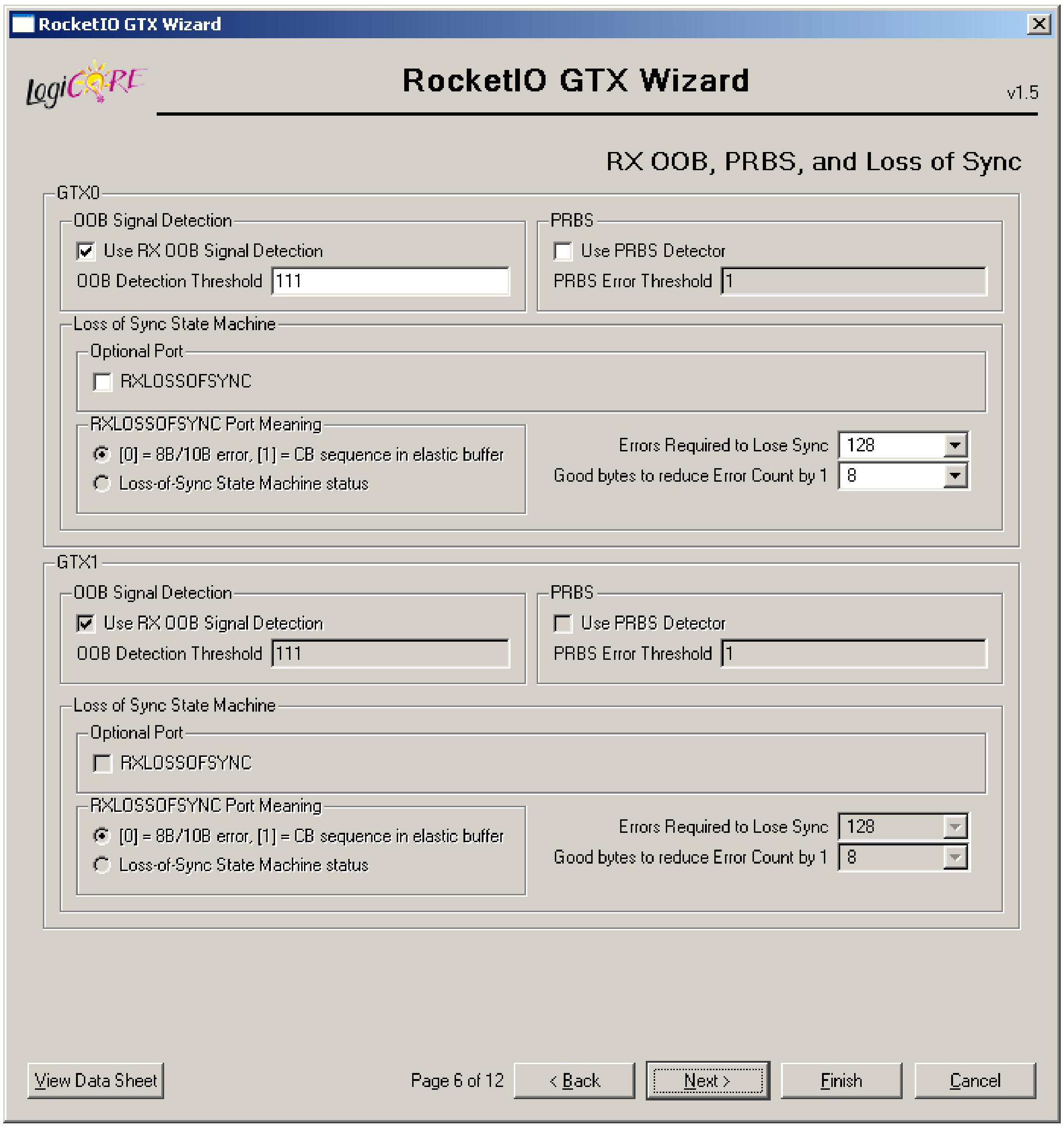
☐ Disable internal AC coupling

Termination Voltage: GND [✓ 2/3 VTTRX] VTTRX

Optional Ports

☑ RXCDRRESET
☐ RXPOLARITY

Figure A.5: Configuring RocketIO GTX for SATA functions - Step 5



RocketIO GTX Wizard

RX OOB, PRBS, and Loss of Sync

GTX0

OOB Signal Detection

[✓] Use RX OOB Signal Detection

OOB Detection Threshold 111

PRBS

[ ] Use PRBS Detector

PRBS Error Threshold 1

Loss of Sync State Machine

Optional Port

[ ] RXLOSSOFSYNC

RXLOSSOFSYNC Port Meaning

[✓ [0] = 8B/10B error, [1] = CB sequence in elastic buffer]

Loss-of-Sync State Machine status

Errors Required to Lose Sync 128

Good bytes to reduce Error Count by 1 8

GTX1

OOB Signal Detection

[✓] Use RX OOB Signal Detection

OOB Detection Threshold 111

PRBS

[ ] Use PRBS Detector

PRBS Error Threshold 1

Loss of Sync State Machine

Optional Port

[ ] RXLOSSOFSYNC

RXLOSSOFSYNC Port Meaning

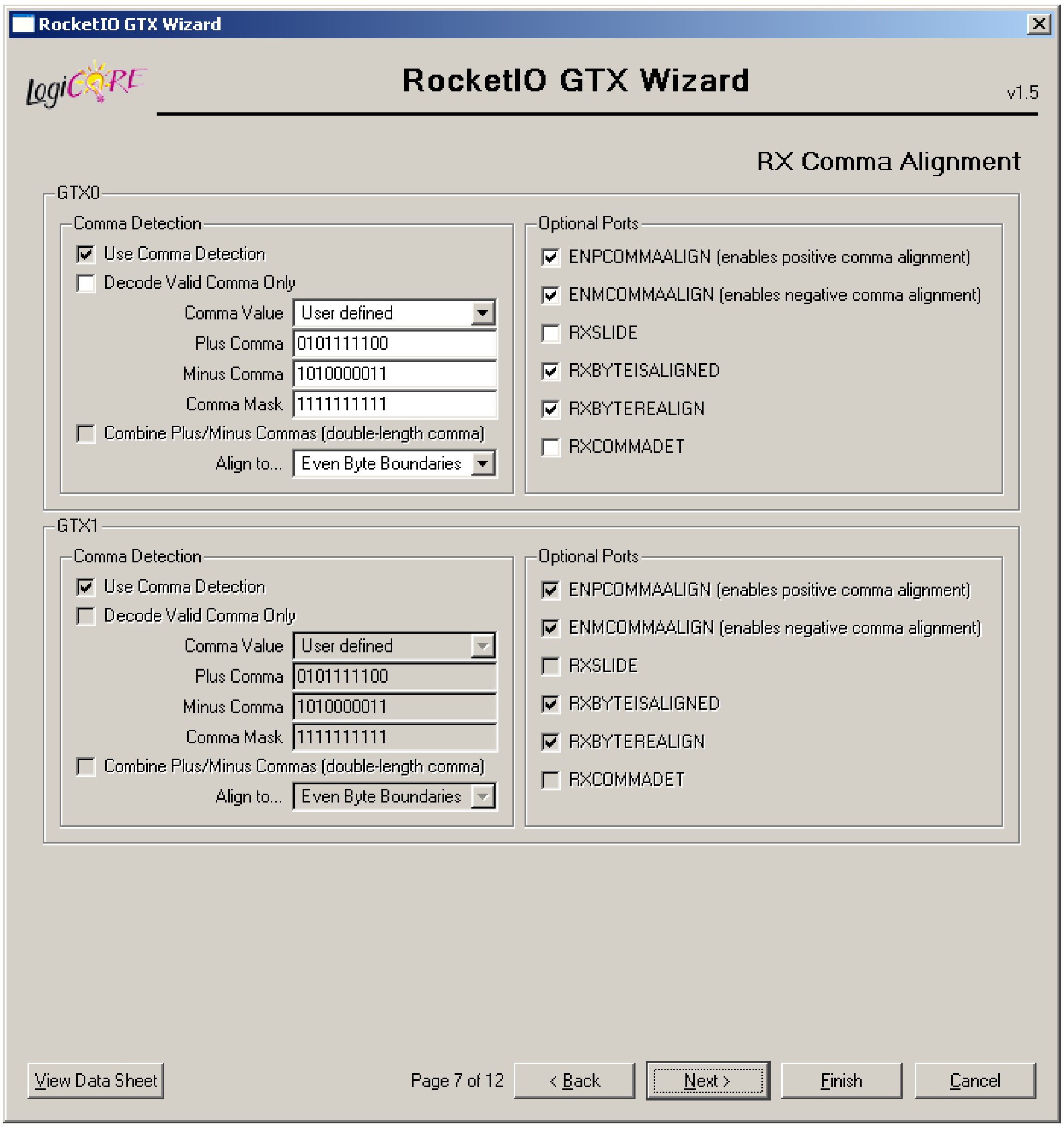
[✓ [0] = 8B/10B error, [1] = CB sequence in elastic buffer]

Loss-of-Sync State Machine status

Errors Required to Lose Sync 128

Good bytes to reduce Error Count by 1 8

Figure A.6: Configuring RocketIO GTX for SATA functions - Step 6



RX Comma Alignment

GTX0

Comma Detection

- [✓] Use Comma Detection
- [ ] Decode Valid Comma Only

Comma Value	User defined
Plus Comma	0101111100
Minus Comma	1010000011
Comma Mask	1111111111

- [ ] Combine Plus/Minus Commas (double-length comma)

Align to... Even Byte Boundaries

Optional Ports

- [✓] ENPCOMMAALIGN (enables positive comma alignment)
- [✓] ENMCOMMAALIGN (enables negative comma alignment)
- [ ] RXSLIDE
- [✓] RXBYTEISALIGNED
- [✓] RXBYTEREALIGN
- [ ] RXCOMMADET

GTX1

Comma Detection

- [✓] Use Comma Detection
- [ ] Decode Valid Comma Only

Comma Value	User defined
Plus Comma	0101111100
Minus Comma	1010000011
Comma Mask	1111111111

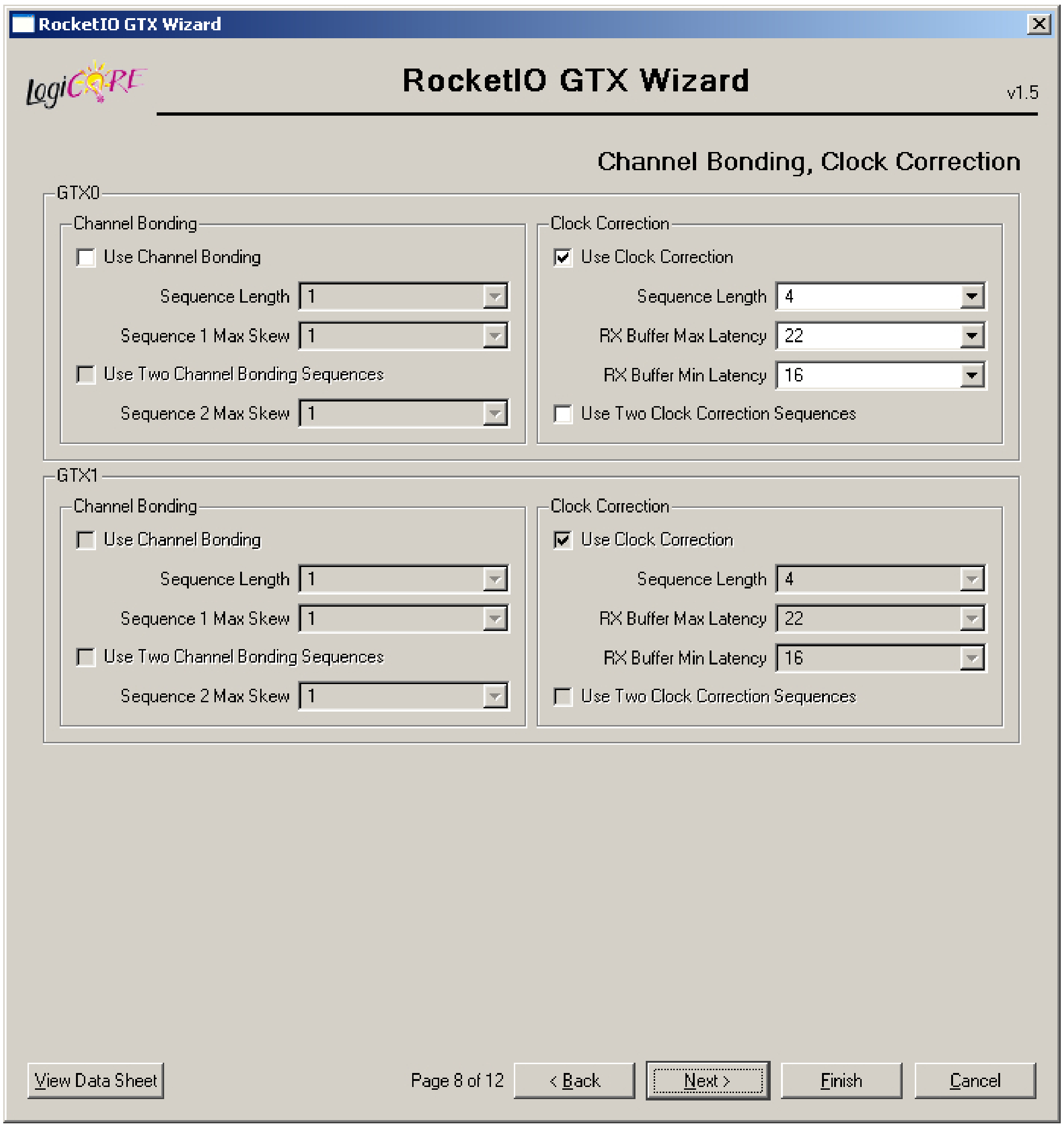
- [ ] Combine Plus/Minus Commas (double-length comma)

Align to... Even Byte Boundaries

Optional Ports

- [✓] ENPCOMMAALIGN (enables positive comma alignment)
- [✓] ENMCOMMAALIGN (enables negative comma alignment)
- [ ] RXSLIDE
- [✓] RXBYTEISALIGNED
- [✓] RXBYTEREALIGN
- [ ] RXCOMMADET

Figure A.7: Configuring RocketIO GTX for SATA functions - Step 7


RocketIO GTX Wizard

RocketIO GTX Wizard v1.5

Channel Bonding, Clock Correction

GTX0

Channel Bonding

- [ ] Use Channel Bonding
- Sequence Length: 1
- Sequence 1 Max Skew: 1
- [ ] Use Two Channel Bonding Sequences
- Sequence 2 Max Skew: 1

Clock Correction

- [✓] Use Clock Correction
- Sequence Length: 4
- RX Buffer Max Latency: 22
- RX Buffer Min Latency: 16
- [ ] Use Two Clock Correction Sequences

GTX1

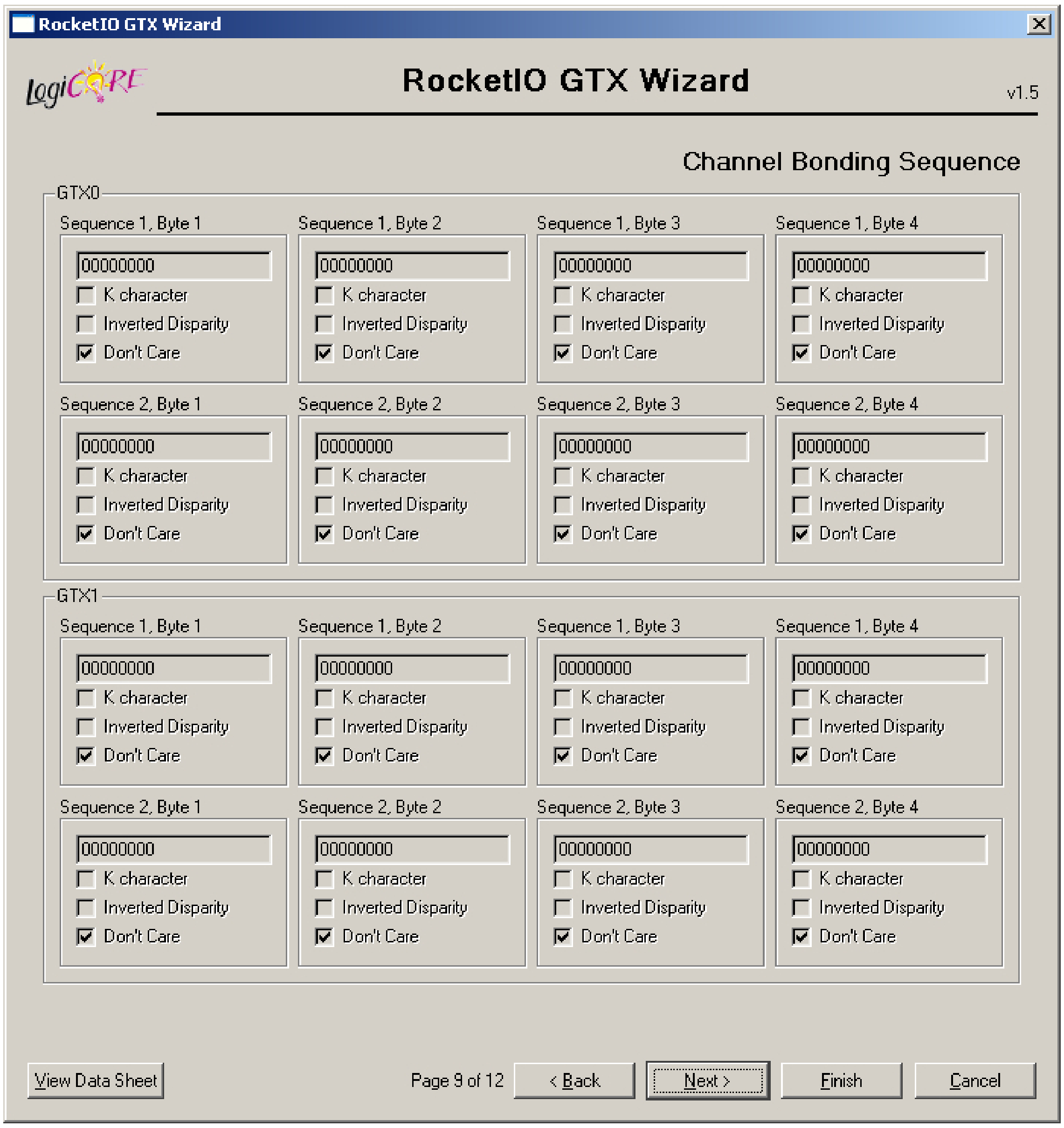
Channel Bonding

- [ ] Use Channel Bonding
- Sequence Length: 1
- Sequence 1 Max Skew: 1
- [ ] Use Two Channel Bonding Sequences
- Sequence 2 Max Skew: 1

Clock Correction

- [✓] Use Clock Correction
- Sequence Length: 4
- RX Buffer Max Latency: 22
- RX Buffer Min Latency: 16
- [ ] Use Two Clock Correction Sequences

Figure A.8: Configuring RocketIO GTX for SATA functions - Step 8



RocketIO GTX Wizard

RocketIO GTX Wizard v1.5

Channel Bonding Sequence

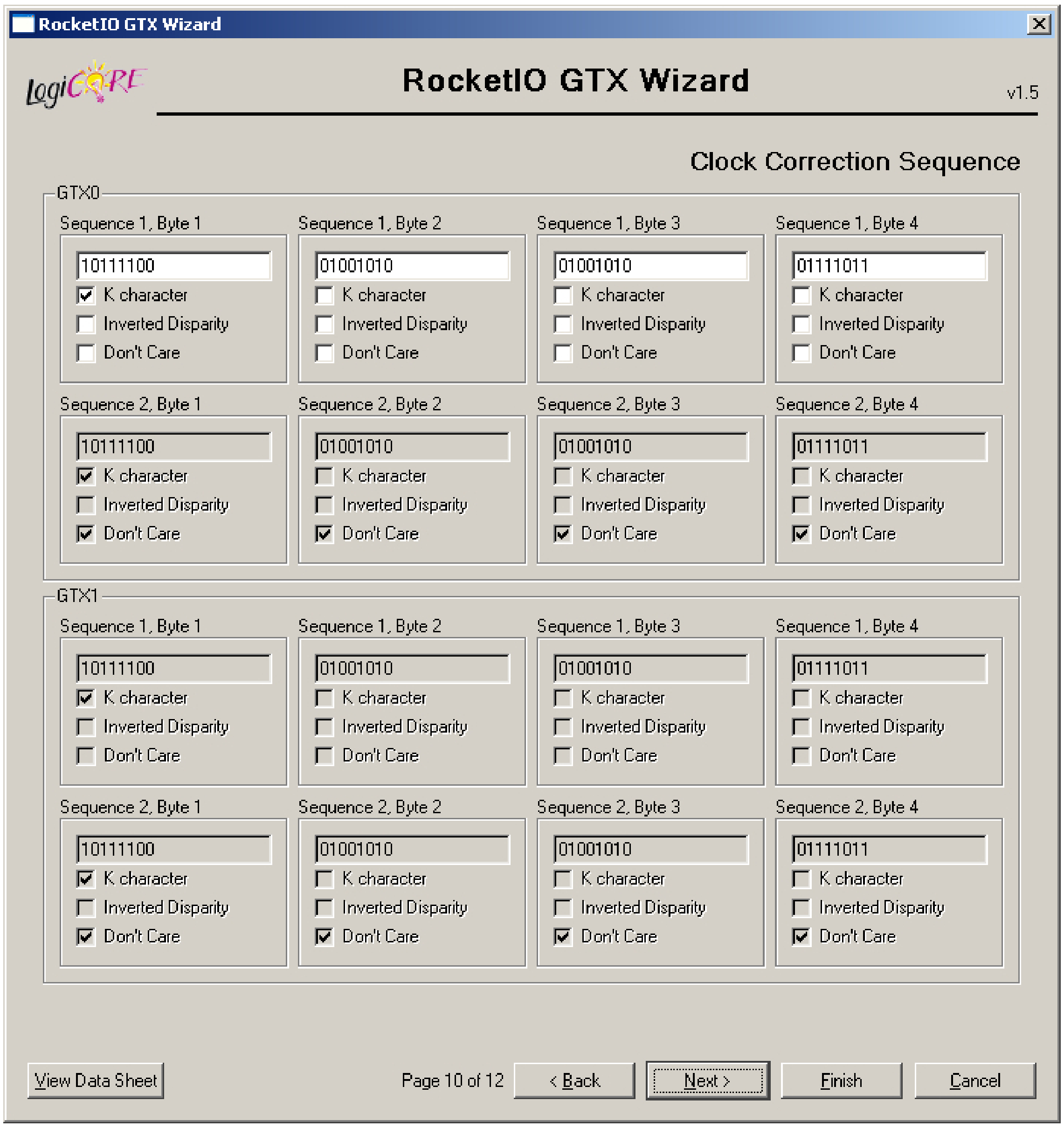
GTX0

	Byte 1	Byte 2	Byte 3	Byte 4
Sequence 1	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care
Sequence 2	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care

GTX1

	Byte 1	Byte 2	Byte 3	Byte 4
Sequence 1	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care
Sequence 2	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care	00000000; ☐ K character; ☐ Inverted Disparity; ☑ Don't Care

Figure A.9: Configuring RocketIO GTX for SATA functions - Step 9



Clock Correction Sequence

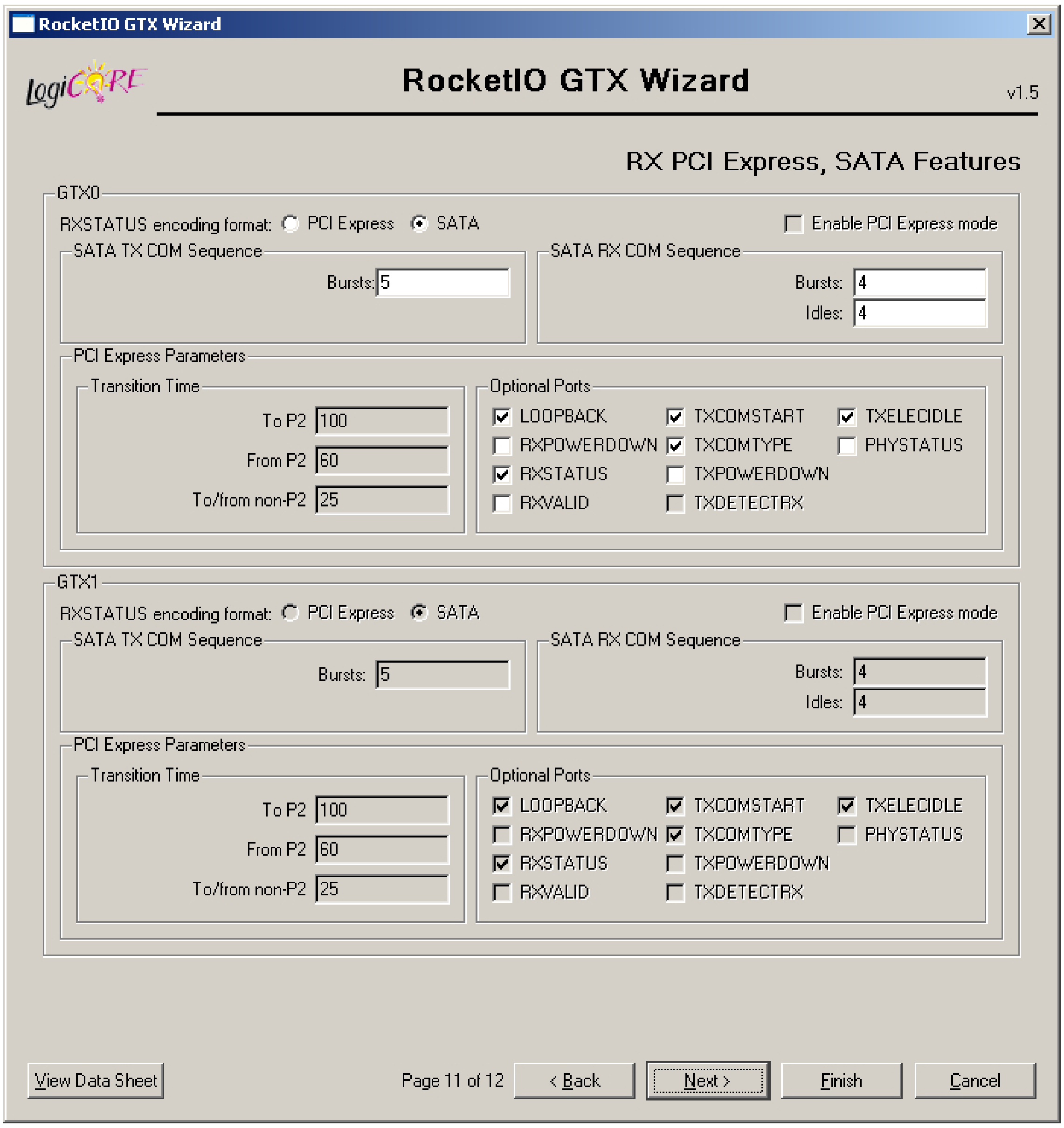
GTX0

	Byte 1	Byte 2	Byte 3	Byte 4
Sequence 1	10111100	01001010	01001010	01111011
K character	☑	☐	☐	☐
Inverted Disparity	☐	☐	☐	☐
Don't Care	☐	☐	☐	☐
Sequence 2	10111100	01001010	01001010	01111011
K character	☑	☐	☐	☐
Inverted Disparity	☐	☐	☐	☐
Don't Care	☑	☑	☑	☑

GTX1

	Byte 1	Byte 2	Byte 3	Byte 4
Sequence 1	10111100	01001010	01001010	01111011
K character	☑	☐	☐	☐
Inverted Disparity	☐	☐	☐	☐
Don't Care	☐	☐	☐	☐
Sequence 2	10111100	01001010	01001010	01111011
K character	☑	☐	☐	☐
Inverted Disparity	☐	☐	☐	☐
Don't Care	☑	☑	☑	☑

Figure A.10: Configuring RocketIO GTX for SATA functions - Step 10



RocketIO GTX Wizard

RX PCI Express, SATA Features

GTX0

RXSTATUS encoding format: PCI Express [✓ SATA]

☐ Enable PCI Express mode

SATA TX COM Sequence — Bursts: 5

SATA RX COM Sequence — Bursts: 4; Idles: 4

PCI Express Parameters

Transition Time — To P2: 100; From P2: 60; To/from non-P2: 25

Optional Ports

☑ LOOPBACK	☑ TXCOMSTART	☑ TXELECIDLE
☐ RXPOWERDOWN	☑ TXCOMTYPE	☐ PHYSTATUS
☑ RXSTATUS	☐ TXPOWERDOWN	
☐ RXVALID	☐ TXDETECTRX	

GTX1

RXSTATUS encoding format: PCI Express [✓ SATA]

☐ Enable PCI Express mode

SATA TX COM Sequence — Bursts: 5

SATA RX COM Sequence — Bursts: 4; Idles: 4

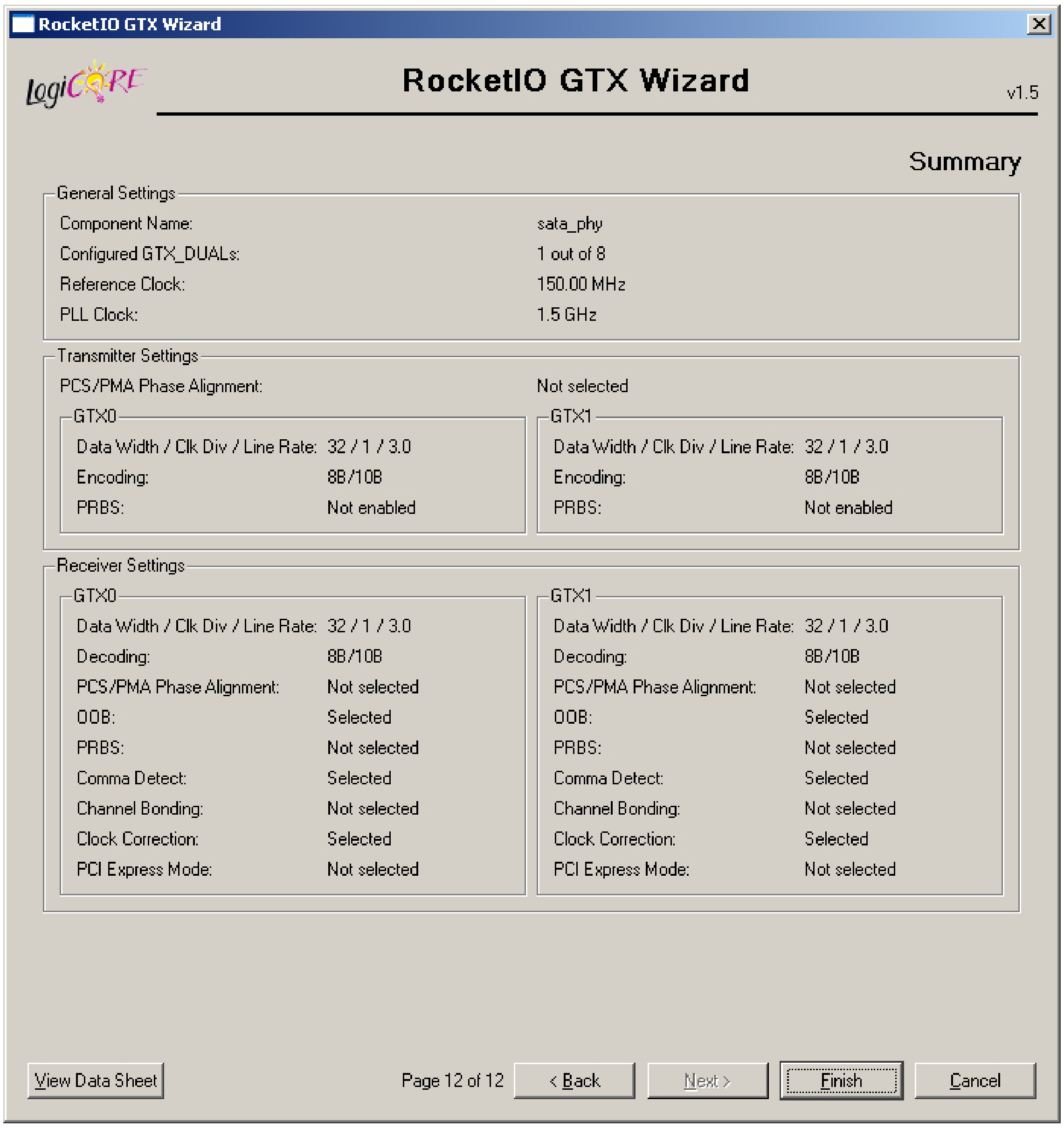
PCI Express Parameters

Transition Time — To P2: 100; From P2: 60; To/from non-P2: 25

Optional Ports

☑ LOOPBACK	☑ TXCOMSTART	☑ TXELECIDLE
☐ RXPOWERDOWN	☑ TXCOMTYPE	☐ PHYSTATUS
☑ RXSTATUS	☐ TXPOWERDOWN	
☐ RXVALID	☐ TXDETECTRX	

Figure A.11: Configuring RocketIO GTX for SATA functions - Step 11



Summary

General Settings

Component Name:	sata_phy
Configured GTX_DUALs:	1 out of 8
Reference Clock:	150.00 MHz
PLL Clock:	1.5 GHz

Transmitter Settings

PCS/PMA Phase Alignment: Not selected

	GTX0		GTX1
Data Width / Clk Div / Line Rate:	32 / 1 / 3.0	Data Width / Clk Div / Line Rate:	32 / 1 / 3.0
Encoding:	8B/10B	Encoding:	8B/10B
PRBS:	Not enabled	PRBS:	Not enabled

Receiver Settings

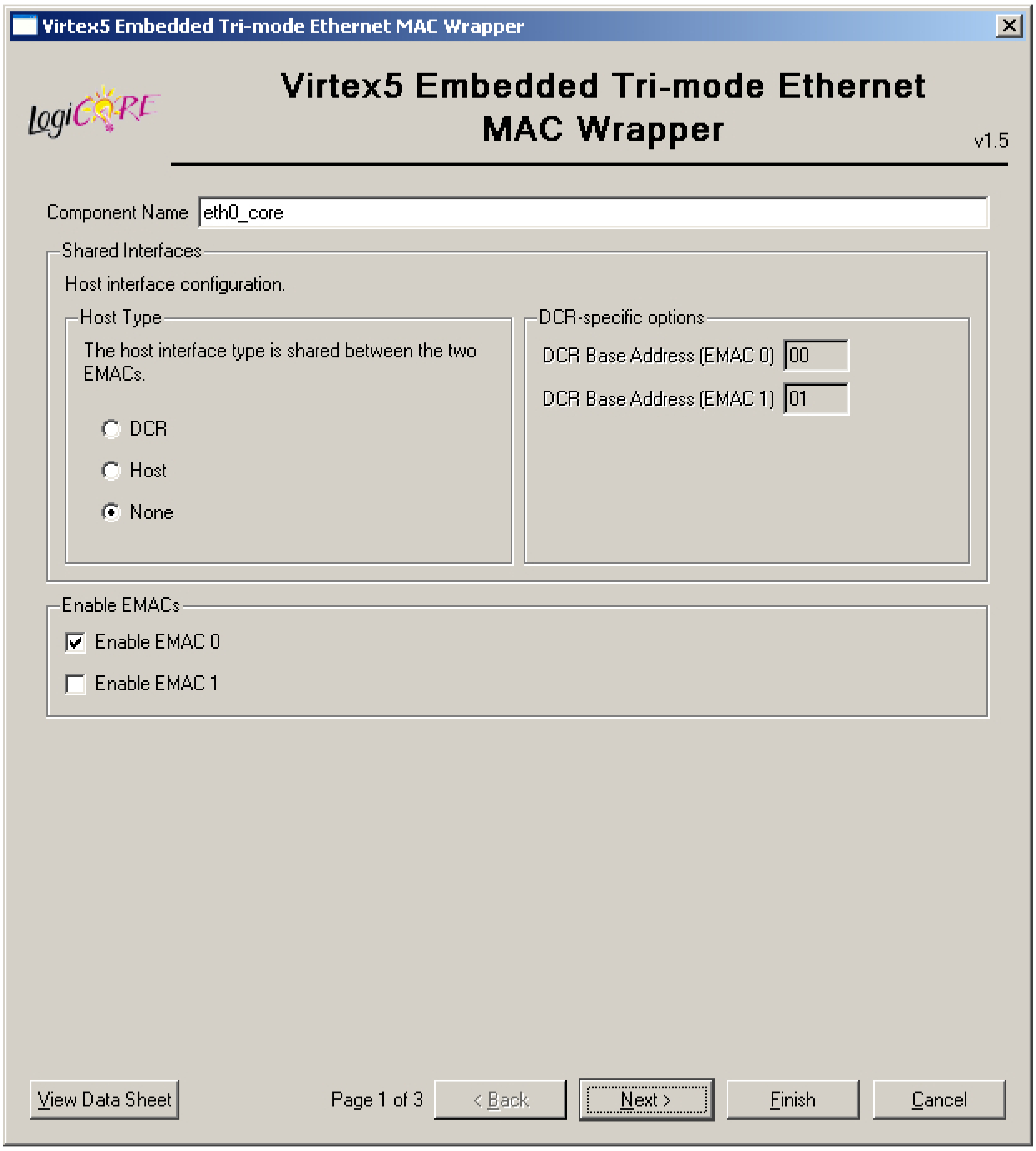
	GTX0		GTX1
Data Width / Clk Div / Line Rate:	32 / 1 / 3.0	Data Width / Clk Div / Line Rate:	32 / 1 / 3.0
Decoding:	8B/10B	Decoding:	8B/10B
PCS/PMA Phase Alignment:	Not selected	PCS/PMA Phase Alignment:	Not selected
OOB:	Selected	OOB:	Selected
PRBS:	Not selected	PRBS:	Not selected
Comma Detect:	Selected	Comma Detect:	Selected
Channel Bonding:	Not selected	Channel Bonding:	Not selected
Clock Correction:	Selected	Clock Correction:	Selected
PCI Express Mode:	Not selected	PCI Express Mode:	Not selected

Figure A.12: Configuring RocketIO GTX for SATA functions - Step 12

Appendix B

Generating Tri-Mode Ethernet MAC core

The following figures describe how to configure Xilinx Core Generator for generating Tri-Mode Ethernet MAC core used in this design.


Virtex5 Embedded Tri-mode Ethernet MAC Wrapper

Virtex5 Embedded Tri-mode Ethernet MAC Wrapper v1.5

Component Name: eth0_core

Shared Interfaces

Host interface configuration.

Host Type

The host interface type is shared between the two EMACs.

DCR [ ] Host [ ] [✓ None]

DCR-specific options

DCR Base Address (EMAC 0): 00

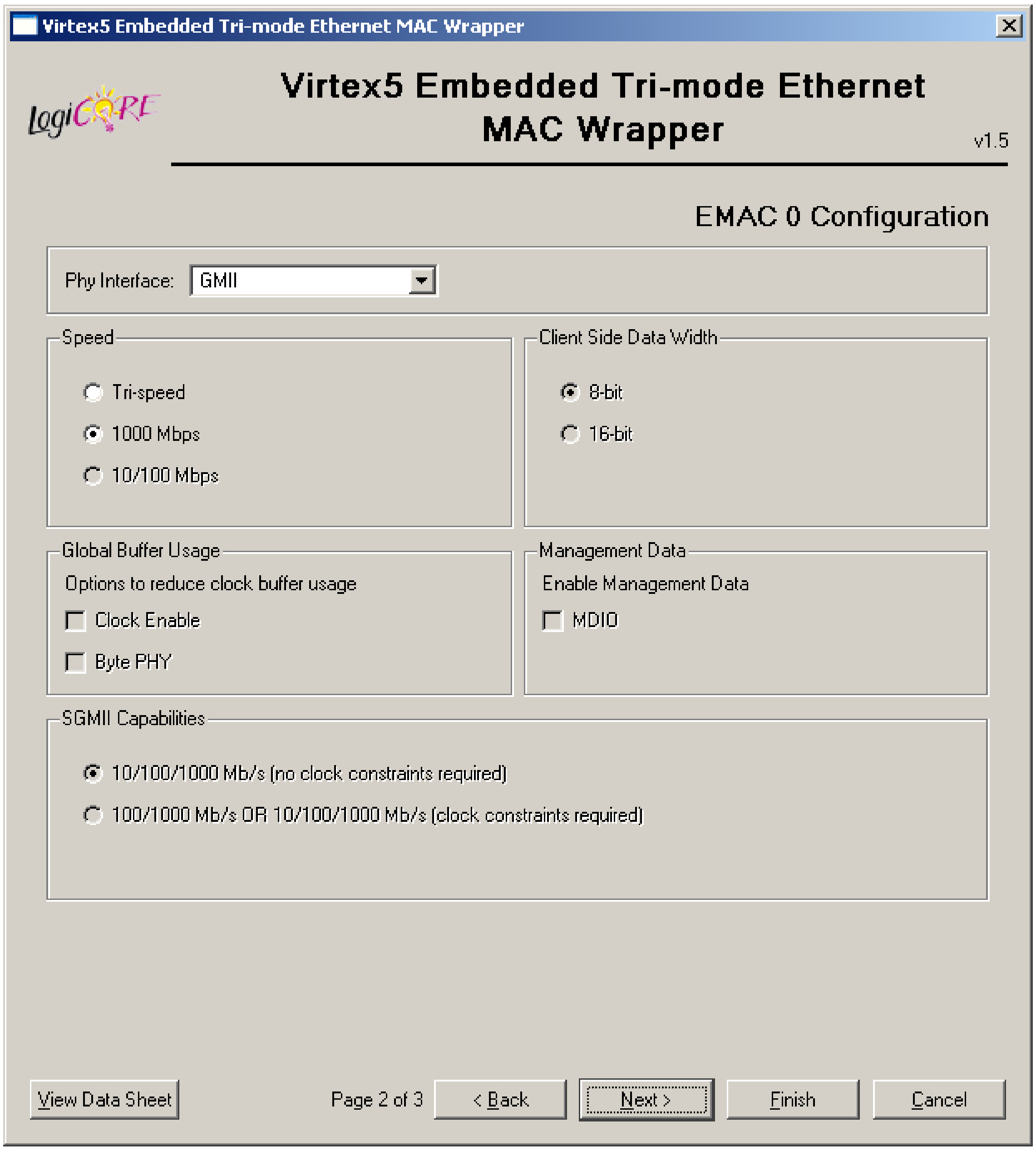
DCR Base Address (EMAC 1): 01

Enable EMACs

[✓] Enable EMAC 0

[ ] Enable EMAC 1

Figure B.1: Generating Tri-Mode Ethernet MAC core - Step 1


Virtex5 Embedded Tri-mode Ethernet MAC Wrapper

Virtex5 Embedded Tri-mode Ethernet MAC Wrapper v1.5

EMAC 0 Configuration

Phy Interface: GMII

Speed
- Tri-speed
- [✓ 1000 Mbps]
- 10/100 Mbps

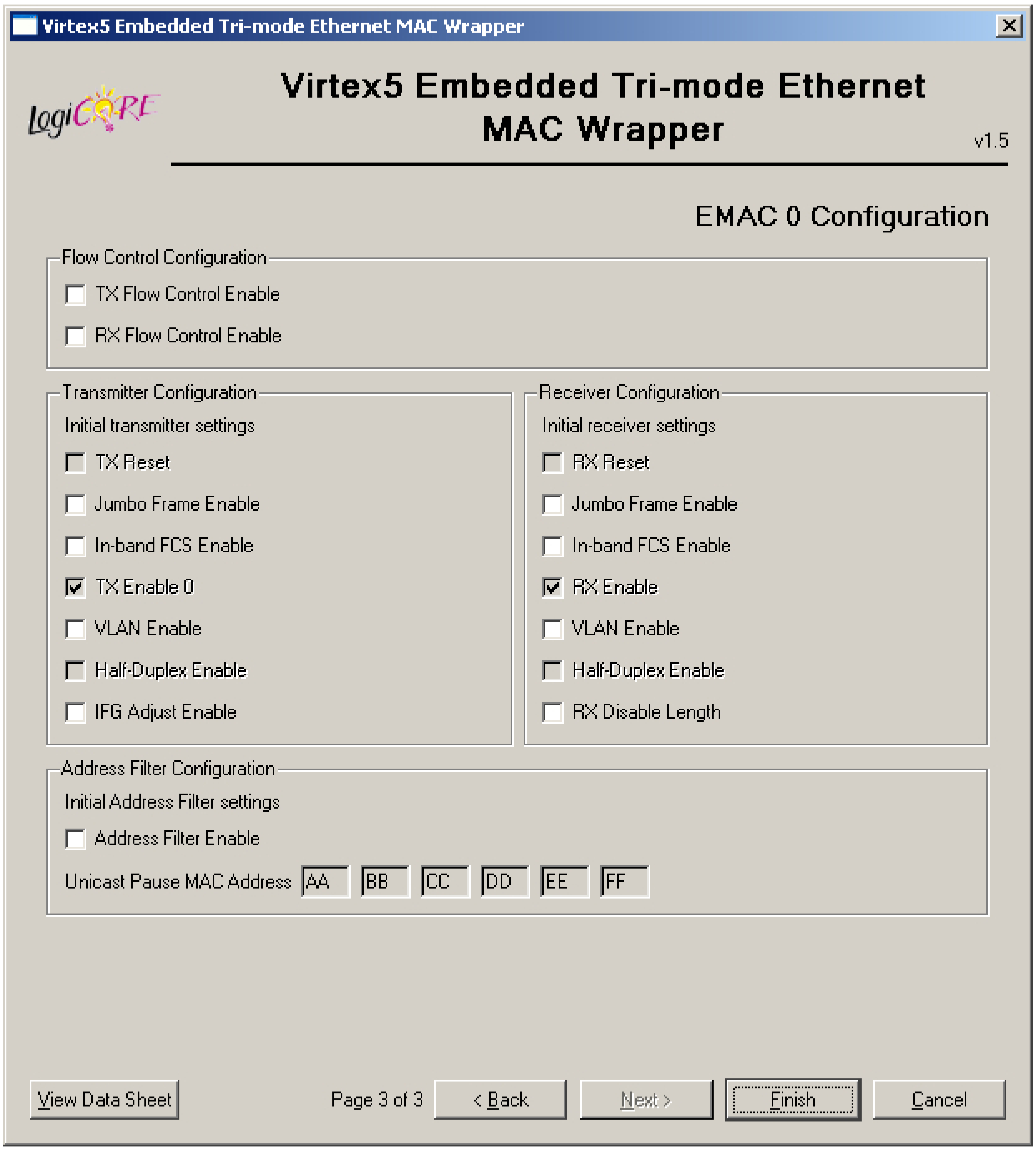
Client Side Data Width
- [✓ 8-bit]
- 16-bit

Global Buffer Usage
Options to reduce clock buffer usage
- ☐ Clock Enable
- ☐ Byte PHY

Management Data
Enable Management Data
- ☐ MDIO

SGMII Capabilities
- [✓ 10/100/1000 Mb/s (no clock constraints required)]
- 100/1000 Mb/s OR 10/100/1000 Mb/s (clock constraints required)

Figure B.2: Generating Tri-Mode Ethernet MAC core - Step 2


Virtex5 Embedded Tri-mode Ethernet MAC Wrapper

LogiCORE

Virtex5 Embedded Tri-mode Ethernet MAC Wrapper v1.5

EMAC 0 Configuration

Flow Control Configuration

- [ ] TX Flow Control Enable
- [ ] RX Flow Control Enable

Transmitter Configuration

Initial transmitter settings

- [ ] TX Reset
- [ ] Jumbo Frame Enable
- [ ] In-band FCS Enable
- [✓] TX Enable 0
- [ ] VLAN Enable
- [ ] Half-Duplex Enable
- [ ] IFG Adjust Enable

Receiver Configuration

Initial receiver settings

- [ ] RX Reset
- [ ] Jumbo Frame Enable
- [ ] In-band FCS Enable
- [✓] RX Enable
- [ ] VLAN Enable
- [ ] Half-Duplex Enable
- [ ] RX Disable Length

Address Filter Configuration

Initial Address Filter settings

- [ ] Address Filter Enable

Unicast Pause MAC Address: AA BB CC DD EE FF

Figure B.3: Generating Tri-Mode Ethernet MAC core - Step 3

Appendix C

ML507 board jumper configuration

In this chapter I describe Xilinx ML507 development board jumper configuration used in this design.

Jumper	**Setting**
J9	Jumper between pins 1 & 2 and between pins 3 & 4
J14	ON
J17	Jumper between pins 1 & 2
J20	Double Jumper between pins 1 & 2
J21	Jumper between pins 1 & 2
J22	Jumper between pins 1 & 2
J23	Jumper between pins 1 & 2
J24	OFF
J54	ON
J56	ON
J62	Jumper between pins 1 & 2
J63	Jumper between pins 1 & 2
J81	Jumper between pins 1 & 2
J82	ON

Table C.1: Xilinx ML507 development board jumper configuration

Appendix D

Operating ML507 board running this design

This chapter describes how to operate Xilinx ML507 development board running this design. Serial ATA hard drive generation 1 or needs to be plugged into SATA HOST 1 connector. Ethernet twisted pair cable category 5E or better needs to by plugged into RJ45 connector of the board. On the other end of the Ethernet cable there should be PC or similar running one of the test applications that are also product of this thesis. The board running the design is controlled by two inputs, CPU RST for reseting the Serial ATA part of the design and DIP SW 8 for reseting the Ethernet part of the design. Ethernet has its own hard wired status light-emitting diodes. Serial ATA core uses GPIO light-emitting diodes for displaying its status, that are described in following table D.

LED	**Description**
GPIO LED 0	RocketIO GTX's PLL locked
GPIO LED 1	Speed negotiation DCM locked
GPIO LED 2	SATA Link layer ready
GPIO LED 3	SATA Transport layer ready
GPIO LED 6	SATA Generation 1 speed
GPIO LED 7	SATA Generation 2 speed

Table D.1: Serial ATA core status light-emitting diodes

Appendix E

Abbreviations list

- AES Advanced Encryption Standard
- FPGA Field-Programmable Gate Array
- ASIC Application-Specific Integrated Circuit
- FSM Finite State Machine
- SATA Serial Advanced Technology Attachment
- FIS Frame Information Structure
- HDL Hardware Description Language
- SOF Start Of Frame
- EOF End Of Frame
- CRC Cyclic Redundancy Check
- EDK Embedded Development Kit
- SAS Serial Attached SCSI

Appendix F

Content of enclosed CD

```
CD_root/
|-- bit
|-- chipscope
|-- doc
|-- src
‘-- testapp
```

- Directory bit contains configuration bitsream for Virtex-5 FXT FPGA.
- Directory chipscope contains Chipscope Pro project files.
- Directory doc contains this thesis in PDF format and also graphical representation of some state machines.
- Directory src contains HDL source code files and testbenchs.
- Directory testapp contains compiled binaries and C source codes of test applications.