TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Andrei Maalberg

SpaceWire to SPI Bridge in VHDL for Microsemi ProASIC3E FPGA

Master's Thesis

Supervisor(s): Prof. René Beuchat (EPFL),

Prof. Eduard Petlenkov (TTÜ),

Dr. Jean-Luc Josset (Space-X),

Dr. Mitko Tanevski (Space-X)

Tallinn 2018

Declaration: I hereby declare that this Master's thesis, my original investigation and achievement, submitted for the Master's degree at Tallinn University of Technology, has not been submitted for any degree or examination.

Deklareerin, et käesolev magistritöö, mis on minu iseseisva töö tulemus, on esitatud Tallinna Tehnikaülikooli magistrikraadi taotlemiseks ja selle alusel ei ole varem taotletud akadeemilist kraadi.

Andrei Maalberg

Date: ....

Signature: ....

Contents

Abstract

SpaceWire to SPI Bridge in VHDL for Microsemi ProASIC3E FPGA

Even though digital designs incorporating a system-on-a-chip (SoC) architecture are becoming increasingly prevalent nowadays, there are still certain applications, e.g. space industry projects, where the rise of the system complexity caused by the use of SoC architectures may not be always justified. This work addresses this issue by designing a lean hardware-only solution for its final goal—a SpaceWire to SPI bridge developed in cooperation with the Space Exploration Institute (Space-X). By analyzing the possible design architecture options, including the one based on SoC, this work provides critical arguments regarding the choice of the hardware-only architecture. Additionally, while targeting ProASIC3E FPGA from Microsemi this work gives the required overview of both the educational and technical resources provided by this FPGA vendor. Based on the positive results of the final design implementation, this work has shown that a hardware-only design approach is indeed a viable architecture solution even if it may take more time to be developed compared to SoC design. Finally, the design solution proposed in this work could be especially attractive for the other projects that require this kind of communication bridge functionality while facing similar design architecture constraints.

List of Figures

List of Tables

Nomenclature

CPHA	Clock PHAse
CPOL	Clock POLarity
CRC	Cyclic Redundancy Check
DDR	Double Data Rate
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GRLIB	Gaisler Research LIBrary
I/O	Input/Output
IP	Intellectual Property
LEWIS	Low Energy Wireless Imaging System
LSB	Least Significant Byte
LSb	Least Significant bit
LVTTL	Low Voltage Transistor Transistor Logic
MISO	Master In Slave Out
MOSI	Master Out Slave In
MSB	Most Significant Byte
MSb	Most Significant bit
NoC	Network-on-Chip
PLL	Phase Locked Loop
RAM	Random Access Memory
RMAP	Remote Memory Access Protocol
Rx	Receive

SNUG	Synopsys Users Group
SoC	System-on-a-Chip
SPI	Serial Peripheral Interface
SpW	SpaceWire
TC	TeleCommand
TM	TeleMetry
Tx	Transmit
VHDL	VHSIC Hardware Description Language

Chapter 1

Introduction

1.1 Context

The current thesis work is performed in the context of LEWIS—a project in the frame of the Technology Research Programme (TRP) [2] of the European Space Agency (ESA). Being established in direct negotiation between ESA and the Space Exploration Institute (Space-X) [3] LEWIS aims to evaluate the feasibility and reliability of using low-energy wireless cameras for space applications. The rationale behind this endeavour is that there are certain use cases in space, e.g. long-reach robotic arms, where wireless connectivity could prove superior to a wired one. Figure 1.1 illustrates one such use case where a spacecraft is missioned to collect samples from the surface of a space object, e.g. an asteroid. In order to allow better control of the sampling process a camera is placed close to the sampling site. As seen in the illustration, the moving parts of the long-reach robotic arm can make it problematic to use a wired connection to communicate with the camera. Instead, by applying a wireless solution this routing problem can be avoided altogether.

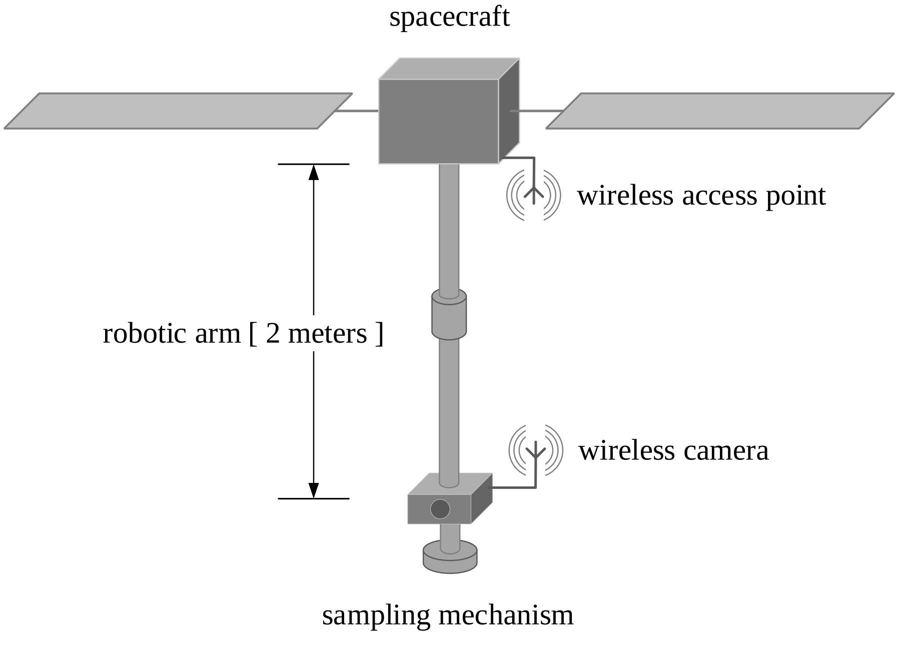


Figure 1.1: LEWIS use case example

1.2 Scope

Figure 1.2 illustrates the scope of the current thesis work while using a color code to emphasize this scope in the context of the general goal of the LEWIS project carried out in the Space Exploration Institute (Space-X).

Starting from the contextual perspective the presented illustration depicts two entities wishing to communicate with each other, i.e. the on-board computer (OBC) of a spacecraft on the right, and the microcontroller (MCU) of a wireless access point on the left. The illustration intentionally underscores that these entities use different communication interfaces, i.e. SPI vs. SpaceWire RMAP. Consequently, there has to be an entity in-between that could bridge this communication gap between them. This communication gap is finally filled by SpaceWire to SPI bridge designed in VHDL and implemented on a Microsemi ProASIC3E FPGA board, thus transitioning the perspective to the scope of this thesis work.

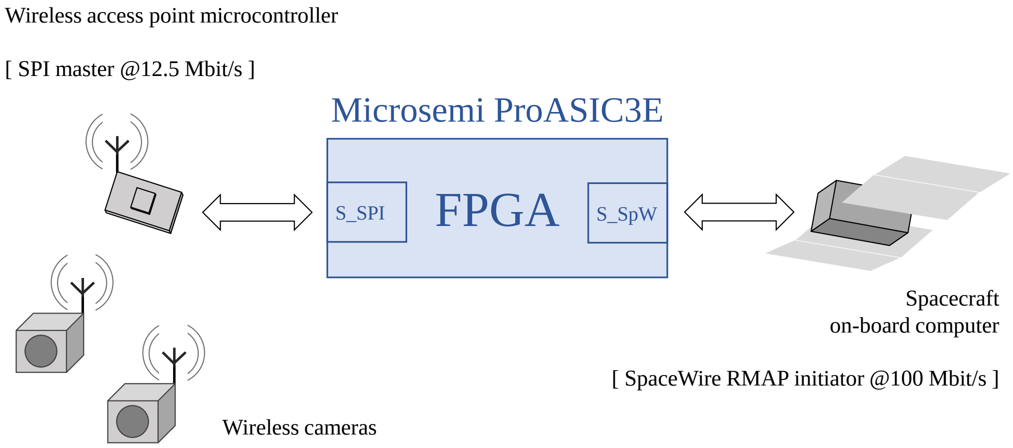

Wireless access point microcontroller
[SPI master @12.5 Mbit/s]
Microsemi ProASIC3E
S_SPI
FPGA
S_SpW
Spacecraft
on-board computer
[SpaceWire RMAP initiator @100 Mbit/s]
Wireless cameras

Figure 1.2: General goal of the project in the Space Exploration Institute (Space-X)

As highlighted by the presented illustration there are two major objectives in the scope of this work:

1. To provide SPI slave connectivity;
2. To provide SpaceWire RMAP target connectivity.

It is important to understand, however, that these objectives are not equal in complexity. SPI is a very simple communication protocol, whereas SpaceWire is a far more elaborated communication technology standardized by the European Cooperation for Space Standardization (ECSS) [4]. This difference in complexities should be kept in mind during the analysis of the candidate design architectures.

1.3 State of the Art

1.3.1 Space Project Regulations

In general space projects are strictly regulated by the rules imposed by such organizations as the European Space Agency. Therefore, any development process carried out in terms of such projects needs to comply with the corresponding standards, e.g. [5] [6]. While following these guidelines a significant amount of documentation needs to be created up front. As such, this may seriously slow down the current thesis work.

1.3.2 SpaceWire RMAP IP Core Availability

According to the European Space Agency (ESA) [7] the SpaceWire RMAP IP cores that are available [8] [9] require a license to be used. Moreover, these IP cores are meant to be used with AMBA bus interconnect typical for a system-on-a-chip system design. Even though one such licensed IP core is provided to Space-X, it may still be required to have a good in-depth understanding of the respective implementation methods in case there will be a need for some customization. With single exceptions [10] the SpaceWire standard document [4] remains the only source containing detailed description of this technology.

1.3.3 Microsemi Community

The requirement imposed by Space-X to target a Microsemi FPGA adds a certain portion of challenge to the current work. The reason lies in the following—since Microsemi owns only a small share of the FPGA market compared to Xilinx and Intel [11], the corresponding community is equally small. This means that the availability of important resources, such as technical documentation and various tutorials, is rather limited which consequently increases the potential learning curve.

1.4 Thesis Outline

Chapter 2 defines the requirements for this work. Every requirement is followed by a brief rationale behind it.

Chapter 3 starts with an overview of the design under analysis. After that, two possible design architecture options are presented with their respective analysis and resolution.

Chapter 4 presents the design proposed for the SpaceWire to SPI bridge module. The chapter goes into detail regarding the memory structure of the bridge and demonstrates the block diagrams of its main submodules.

Chapter 5 first gives a brief overview of SPI protocol. After that, it defines SPI command interface of the designed SPI slave controller. Finally, this controller along with its submodules is demonstrated using block diagrams with their respective explanations.

In the beginning of Chapter 6 the reader learns the essential bits of information regarding SpaceWire followed by the protocol implementation details in this particular work. Afterwards, the chapter guides the reader through the proposed design of the SpaceWire codec.

Chapter 7 starts with an overview of SpaceWire RMAP given from the perspective of the current work. Thereafter, a command interface based on RMAP is defined and the corresponding usage examples are given. Finally, the proposed design of the RMAP target controller is presented.

Chapter 8 describes the prototyping process on the given Microsemi ProASIC3E FPGA board. In the end, the results of this work are presented.

Chapter 2

Requirements

2.1 SPI Slave Connectivity

The design shall provide SPI slave connectivity operating with the serial clock frequency of 12.5 MHz

SPI is quite an obvious choice for establishing a connection between a microcontroller and a peripheral. This serial protocol is simple and sufficiently fast. Based on the capabilities of the microcontroller used in Space-X for the current project, it was defined that the SPI slave shall operate with the serial clock frequency of 12.5 MHz.

2.2 SpaceWire RMAP Target Connectivity

The design shall provide SpaceWire RMAP target connectivity operating with the possible frequency range of 10 to 200 MHz

Since one of the communication sides is an on-board computer of a spacecraft, the requirement to provide SpaceWire RMAP connectivity is quite natural. As this on-board computer acts as the master, or initiator, of the communication, the designed system, therefore, is required to take the role of the RMAP target. What regards the frequency of the SpaceWire communication then there is a certain degree of freedom allowed as it may depend on the achievable implementation results.

2.3 Microsemi ProASIC3E Starter Kit FPGA

The design shall be implemented on a Microsemi ProASIC3E Starter Kit FPGA

This requirement was imposed by Space-X due to the fact that the current work is part of a space project, and thus radiation tolerance is a critical aspect. The given Starter Kit is a commercial equivalent of a radiation tolerant Microsemi RT ProASIC3 FPGA device [12]. Since the latter is sold only as a chip, it was quite natural for Space-X to select an equivalent device, i.e. ProASIC3E, which is conveniently sold as a kit, i.e. a complete FPGA board ready for prototyping.

Chapter 3

Design Under Analysis

To help analyzing the scope of the current work from a schematic perspective Figure 3.1 displays the global block diagram of the design under analysis. The separation of responsibilities in this block diagram can be applied as follows:

- The bridge module is responsible for initialization, or configuration, of the other modules in the system plus it controls the authorization of data access requests coming from the link controllers;
- The link controllers, i.e. the SPI slave and the SpaceWire RMAP target, manage their corresponding communication links;
- The memory controller in its turn manages the memory operations, such as reading and writing;
- The memory is an on-chip memory.

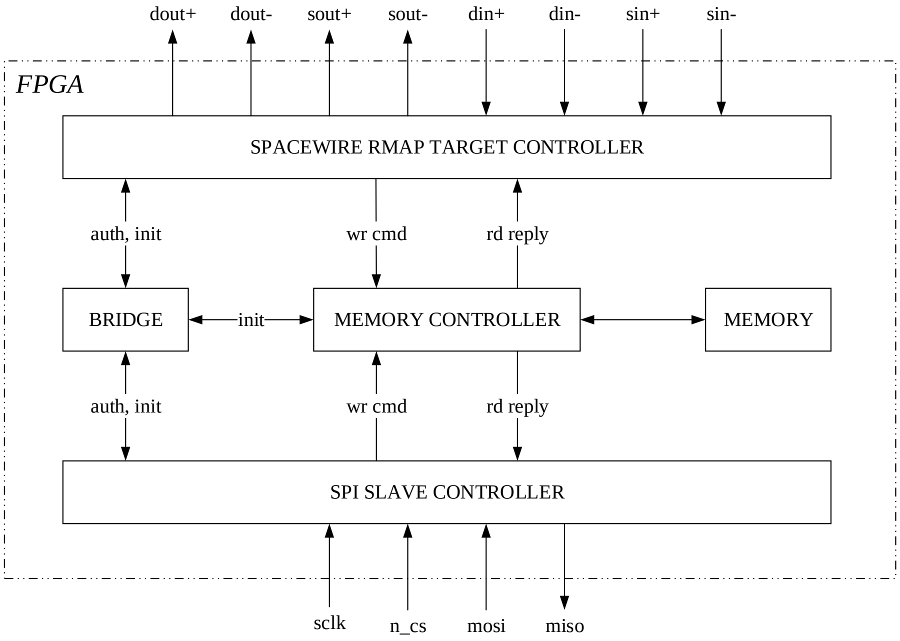


Figure 3.1: Global block diagram of design under analysis

3.1 Double Slave Feature

The global block diagram presents the designed system as a slave on both sides, i.e. the slave on the SPI side and the target on the SpaceWire RMAP side. Indeed, by making the system a double slave the design and implementation of such system becomes significantly simpler. For instance, in terms of SpaceWire RMAP it is sufficient to implement only a limited portion of the target functionality to meet the system functional requirements. Moreover, the double slave design fits more properly into its context as the SPI communication of the MCU can run at a greater frequency (12.5 MHz vs. 4.168 MHz) when the MCU acts as the master [13, "Feature Summary"]. Figure 3.2 depicts a sequence diagram that demonstrates a communication example of the double slave system design. It is seen from this diagram that both the MCU and OBC periodically read the status of the SpW2SPI bridge to find out when the latter is ready to accept or provide new data.

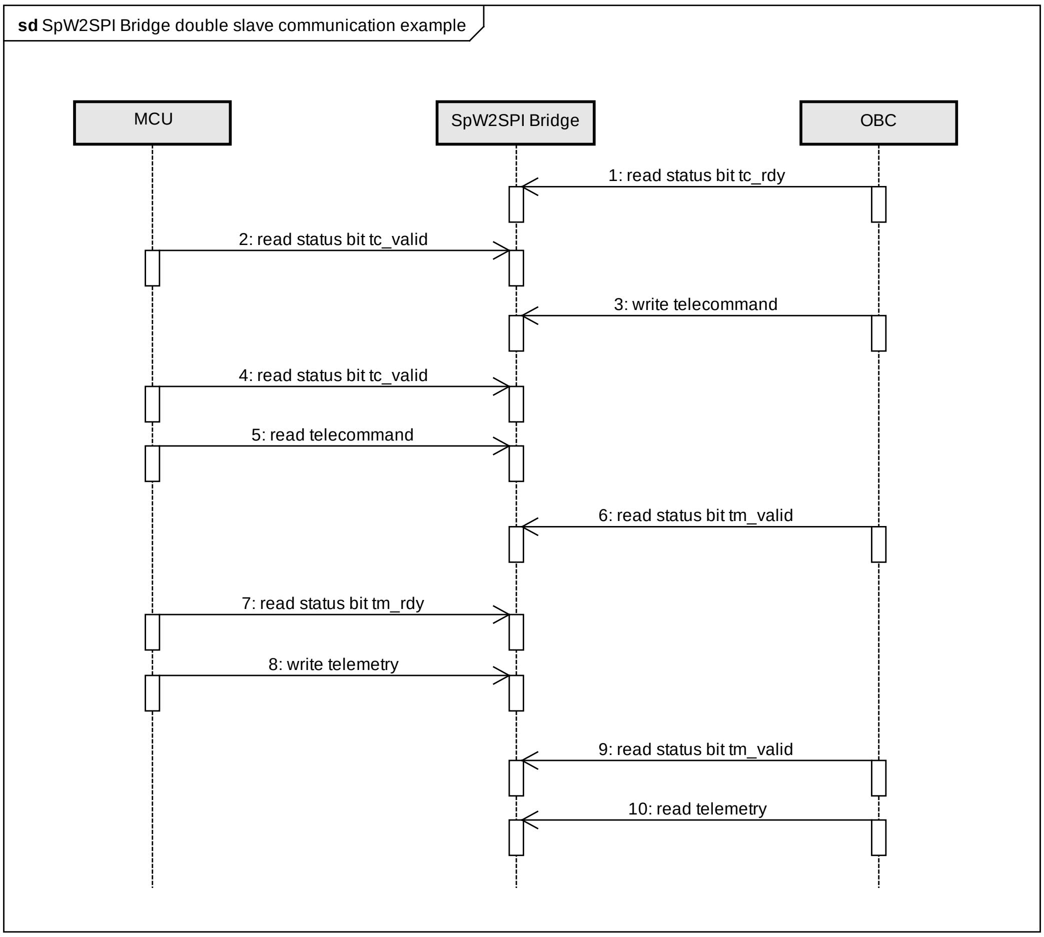

sd SpW2SPI Bridge double slave communication example
MCU
SpW2SPI Bridge
OBC
1: read status bit tc_rdy
2: read status bit tc_valid
3: write telecommand
4: read status bit tc_valid
5: read telecommand
6: read status bit tm_valid
7: read status bit tm_rdy
8: write telemetry
9: read status bit tm_valid
10: read telemetry

Figure 3.2: Sequence diagram of design under analysis

3.2 System-on-a-Chip Architecture

While having the kind of global block diagram as illustrated in Figure 3.1, it may seem like a good choice to use system-on-a-chip architecture for this design. However, a number of important aspects must be first examined before choosing this type of architecture.

3.2.1 Overview

The definition of a system-on-a-chip architecture in the current context is presented in Figure 3.3. As can be seen this architecture has a processor, a system bus and the required modules connected to that bus.

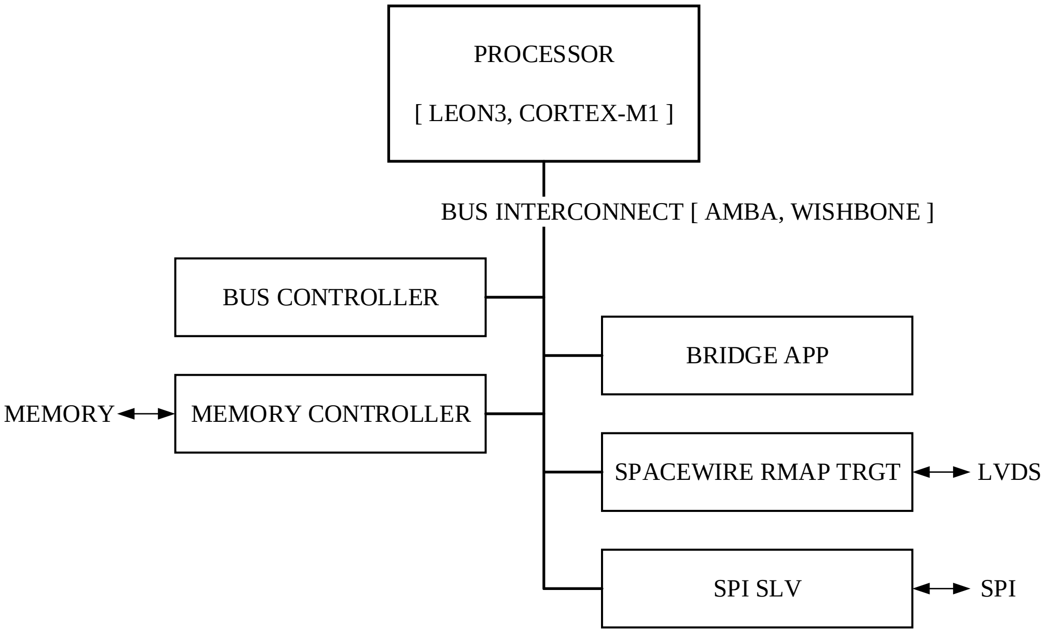


Figure 3.3: System-on-a-chip architecture of design under analysis

The notes regarding the presented block diagram are the following:

- SpW2SPI bridge application could be done in software which would reside somewhere in the memory and be executed by the processor. Provided that all the other necessary modules, i.e. the link controllers, are available as third party IP cores, the development process could allow rapid prototyping of the required functionality.
- The processor could be a space-certified LEON3 soft-core processor from the Gaisler Research Library [14] or a Cortex-M1 soft-core processor to simply make a fast prototype.
- The most obvious choice for the bus interconnect in this case would be AMBA [15].

3.2.2 Applicability

Even though the analyzed SoC architecture could facilitate the scalability and maintainability of the design, the application of this architecture to the current design has a few serious drawbacks.

3.2.2.1 Design Redundancy

Having software parts running on a processor may be useful to speed up the development process and facilitate scalability. However, this kind of design architecture is completely redundant for the current project. The scope of this work is simply to translate, or adapt, messages as they pass from one communication interface to the other, hence a ’full blown’ processor, such as LEON3, cannot be justified for such a trivial task.

3.2.2.2 Support by FPGA Tools

The Libero SoC development tools [16] provided by Microsemi do not offer any system-on-a-chip work flow support when the selected FPGA device is ProASIC3E (the required FPGA device for this project). That is, all the required tools, such as the System Builder, become unavailable in the development environment. Therefore, without such support it becomes very impractical to use this type of architecture for the current design.

3.2.3 Resolution

The current analysis has shown that the power of rapid prototyping inherent to system-on-a-chip designs relies heavily on the support by the tools. The lack of such support, as in this case, can make the development process as complicated as developing everything from scratch. Moreover, there are actually cases where the possibility to deliver some working result fast may not be the best solution. As such, this type of architecture was deemed inapplicable to this design.

3.3 Non System-on-a-Chip Architecture

3.3.1 Overview

Figure 3.4 displays the block diagram of a purely hardware architecture. It can be noticed that there are no extra modules on this diagram, only the ones that are directly required to provide the requested functionality of the system.

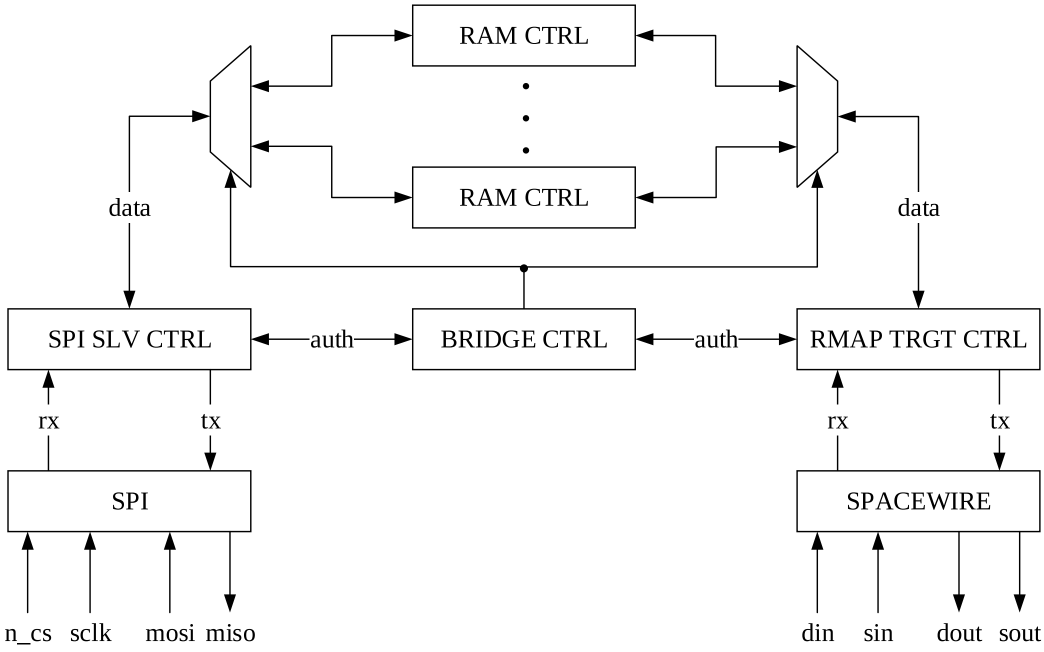


Figure 3.4: Non system-on-a-chip architecture of design under analysis

3.3.2 Applicability

Keeping in mind what was said in Section 3.2.2, there are generally no limitations for a purely hardware design in terms of this project. However, some important design decisions still need to be made.

3.3.2.1 SpaceWire RMAP IP Core

Since the SpaceWire RMAP IP core provided to Space-X by ESA relies on a bus interconnect, it is not directly applicable to this type of architecture. Even though the source code of the IP core was available, its examination showed that stripping the AMBA interface from the core would not be an easy task as the RMAP part of the core lacked the desired modularity. Therefore, it was decided that a new SpaceWire RMAP IP core should be developed specifically for the current project.

3.3.3 Resolution

The non system-on-a-chip, or purely hardware, architecture was chosen as it satisfies the needs of this project and does not have any serious drawbacks.

Chapter 4

SpaceWire to SPI Bridge Design

4.1 Overview

Having chosen the architecture of the design in Section 3.3, the block diagram of the SpaceWire to SPI, or SpW2SPI, bridge is easy to derive—see Figure 4.1.

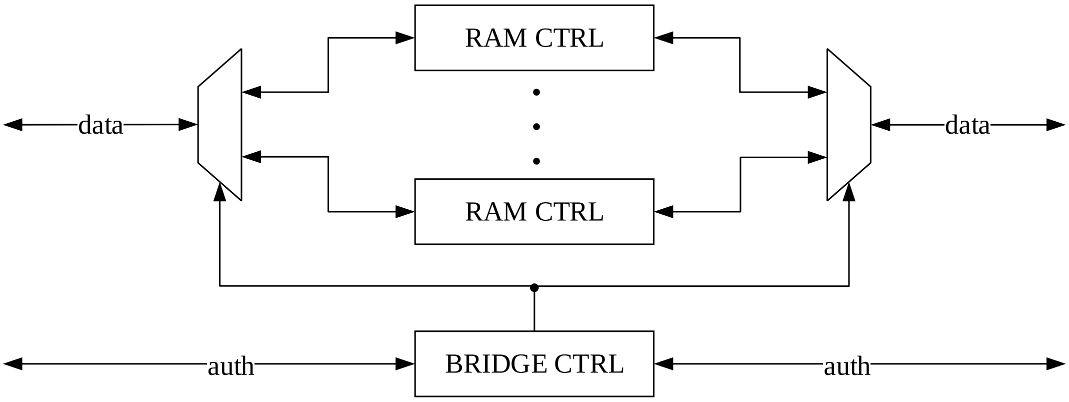


Figure 4.1: SpW2SPI bridge block diagram

The presented block diagram is quite self-explanatory on the top level—the bridge controller authorizes data access requests from both links, while the RAM controllers handle the subsequent data memory operations.

The SpW2SPI bridge memory, which is not shown on the presented block diagram, nevertheless plays an important role in the functionality of the design. The structure of this memory, or its mapping, requires a thorough explanation.

4.2 Memory Mapping

One of the main responsibilities of the SpW2SPI bridge is to temporarily store data as it passes from one client to the other. Therefore, a memory structure has to be defined which shall 1) temporarily store the passed data and 2) allow the clients to access this data by addressing the corresponding regions of the defined memory structure. This section focuses on the second aspect, i.e. the elaboration of the memory structure through the process of memory mapping.

The memory mapping of the SpW2SPI bridge is divided into two classes:

1. Registers;
2. Mailboxes.

The rationale behind this kind of division—different responsibilities of the mapped regions. These responsibilities are explained in Sections 4.2.1 and 4.2.2.

4.2.1 Registers

The registers are mapped into the memory in a straightforward manner which is illustrated in Table 4.1. By looking at this memory mapping there are two things that become evident. The first one is that the registers serve as various status indicators, e.g. a one-byte status register which holds information relevant for the SpaceWire side communication. Hence the responsibility of this memory mapping class—status indication.

Table 4.1: SpW2SPI bridge register memory mapping

Byte #	0	1	2	3 4	5 30
Name	spw_comstat	spi_comstat	tc_size[1]	tm_size[1]	lewis_features[1]

[1] These registers are placeholders reserved for future use.

The second one is the register addressing scheme. The memory in the SpW2SPI bridge is byte addressable. Hence, a single register shall occupy one byte at the least even if the status information it holds has only a couple of meaningful bits. The extra space in this case is reserved for future use. Tables 4.2 and 4.3 present the definition of the two most important status registers.

Table 4.2: SpW2SPI bridge SpaceWire communication status register flags

Bit #	7 2	1	0
Name	reserved	tc_rdy	tm_valid
Access	R	R	R
@Reset	0	0	0

`tm_valid` This flag is set when the telemetry mailbox is valid to be read from the SpaceWire side, it is reset otherwise.

`tc_rdy` This flag is set when the telecommand mailbox is ready to be written from the SpaceWire side, it is reset otherwise.

Table 4.3: SpW2SPI bridge SPI communication status register flags

Bit #	7 2	1	0
Name	reserved	tm_rdy	tc_valid
Access	R	R	R
@Reset	0	0	0

`tc_valid` This flag is set when the telecommand mailbox is valid to be read from the SPI side, it is reset otherwise.

`tm_rdy` This flag is set when the telemetry mailbox is ready to be written from the SPI side, it is reset otherwise.

Finally, Table 4.4 summarizes the registers defined in the Spw2SPI bridge.

Table 4.4: SpW2SPI bridge registers

Address	Name	No. bytes	SPI access	SpW access	Description
0	spw_comstat	1	NA	R	SpaceWire communication status
1	spi_comstat	1	R	NA	SPI communication status
2	tc_size	1	R	NA	TC mailbox data size in bytes
3	tm_size	2	NA	R	TM mailbox data size in bytes
5	lewis_features	24	W	R	LEWIS features

4.2.2 Mailboxes

There are two mailboxes in the proposed design of the SpW2SPI bridge, i.e. one for the telecommand data and the other one for the telemetry data. The address space of these mailboxes is differentiated from the normal directly accessible memory space. That is, each mailbox has an internal base pointer which determines where in memory the mailbox memory is mapped. This pointer is of no concern to the client, and the latter uses the mailbox address to address a concrete mailbox. Figure 4.2 illustrates this idea.

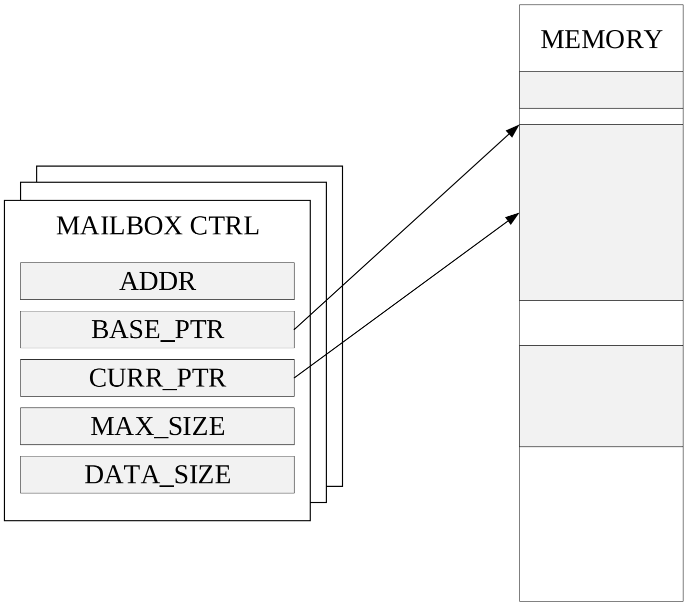


Figure 4.2: SpW2SPI bridge mailbox memory mapping

The fields of the mailbox as illustrated in Figure 4.2 are defined as follows:

`addr` The address of the mailbox used by the client to address this mailbox. Note that this address does not directly point to any memory space, this address serves as an identifier.

`base_ptr` The internal base pointer that determines where in memory the mailbox memory is mapped.

`curr_ptr` The current pointer to the mailbox memory used internally while performing read or write operations.

`max_size` The maximum size of the mailbox as defined in its corresponding specification (see Table 4.5).

`data_size` The size of the data currently held in the mailbox. This field is queried when starting a read operation and updated internally after a successful write operation.

The encapsulation of the mailbox memory mapping gives certain flexibility—any future changes to where and how the mailbox memory is mapped will not affect the client. Thus, this approach leaves the client with very few things to worry about—the address and the maximum size. Provided that these are correct, the client is free to transfer a chunk of data, or a mail, to the other side of the communication channel. Hence the responsibility of this memory mapping class—transfer of data chunks, or mails.

Finally, Table 4.5 summarizes the mailboxes defined in the SpW2SPI bridge.

Table 4.5: SpW2SPI bridge mailboxes

Address	Name	Max no. bytes	SPI access	SpW access	Description
0	tc_data	32	R	W	TC mailbox
1	tm_data	2048	W	R	TM mailbox

4.3 Bridge Controller

The logic of the SpW2SPI bridge controller as seen in Figure 4.1 is quite straightforward. The controller authorizes requests from both link sides to access the memory data. Once the request from one of the sides is authorized the controller configures the corresponding RAM controller for the data access. Additionally, the controller configures the (de)multiplexing logic so that the right data could flow to the right place. The controller may reject an authorization request for a number of reasons, including wrong memory access addresses and unavailable data resource.

4.4 RAM Controller

There are in fact two types of RAM controllers—one for each type of the memory mapping:

1. Mailbox RAM controller;
2. Register RAM controller;

4.4.1 Mailbox RAM Controller

Figure 4.3 shows the block diagram of the mailbox RAM controller. The first thing that becomes apparent here is the 2-slot memory buffering. Whenever new data is successfully written, or received, into one of the slots while the other slot is empty, the slots are switched. This way the mailbox is immediately ready to receive new data. Provided that the data is read from the mailbox at the same rate or faster, the mailbox potentially gains a channel of 100% write throughput [17, "Full-Bandwidth 2-Slot Elastic Buffer"]. Once again, this holds on one condition—the data must be read from the mailbox at the same rate it is written or faster. This scenario can be applied to the case in which the MCU writes many packets of telemetry through a slow SPI connection, while the OBC quickly reads these packets out through a faster SpaceWire connection.

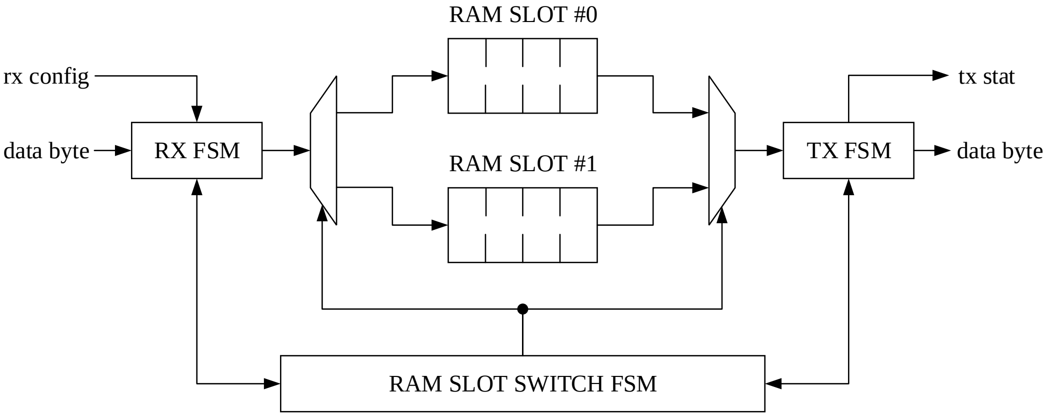


Figure 4.3: Spw2SPI bridge mailbox RAM controller

4.4.2 Register RAM Controller

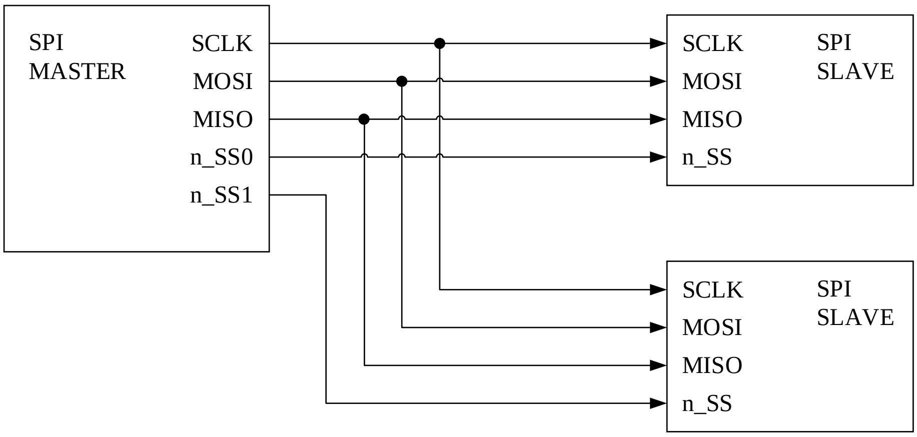
The register RAM controller is in fact very primitive. The register simply reflects data, e.g. a status, that originates from some other entity, such as a mailbox RAM controller.

Chapter 5

SPI Slave Controller Design

5.1 Protocol Description

Serial Peripheral Interface (SPI) is a synchronous full-duplex serial data protocol often used by microcontrollers to communicate with one or more peripheral devices over short distances. In SPI the microcontroller acts as the communication master which controls the peripheral devices, i.e. the slaves. The protocol uses separate clock and data wires, along with a select wire to let the SPI master communicate with a particular slave device. This wiring scheme allows making the clock and data wires common to all the devices while keeping only the select wires specific to each peripheral device. This concept is illustrated in Figure 5.1.



Signal notes:

➢ SCLK	a serial clock signal sent from the master to all the slaves
➢ MOSI	a data line from the master to all the slaves, named Master Out Slave In
➢ MISO	a data line from the slaves to the master, named Master In Slave Out
➢ n_SS	an active low slave select signal for each slave

Figure 5.1: SPI master slave wiring scheme

The fact that in SPI the clock signal gets delivered from the master to the slave makes the communication synchronous as this clock signal effectively keeps both sides in sync. By having this kind of synchronization both the master and slave are able to know exactly when to serialize and sample bits on the data lines. However, since SPI is a rather loose communication protocol, there are certain variations regarding this aspect. Namely, there two parameters—clock polarity and clock phase [18]—that give four possible combinations to configure the timing of the data lines respective to the serial clock.

SPI does not define any communication speed limits, and various implementations often go over 10 Mbit/s. However, the speed usually depends on the capabilities of a concrete slave device, and these capabilities must be checked in the datasheet of the device in order to configure the SPI master properly.

Finally, the fact that SPI protocol is both simple and loose allows every peripheral device to specify its own command interface that suits this device best. Again, the corresponding datasheets must be consulted prior to interfacing such devices.

5.2 Protocol Usage Considerations

5.2.1 Clock Polarity and Phase

The designed SPI slave uses the following parameters for the serial clock polarity and phase:

CPOL	0
CPHA	0

The corresponding timing diagram for the selected parameters' settings can be observed in Figure 5.2. The timing diagram shows the following:

- CPOL parameter set to 0 assigns the serial clock level when this clock is not active to 0 as well. In its turn this means that the leading edge of the serial clock will be the rising edge (note the arrows on the serial clock wave).

- CPHA parameter represents the shifting of the data capturing phase. By setting CPHA to 0 the data is captured on the leading edge of the serial clock. Since the leading edge is already defined to be the rising edge by CPOL parameter, the data will be captured on the rising edge of the serial clock. This naturally means that the data must be propagated on the falling edge of the serial clock in order to be stable during the next capturing phase.

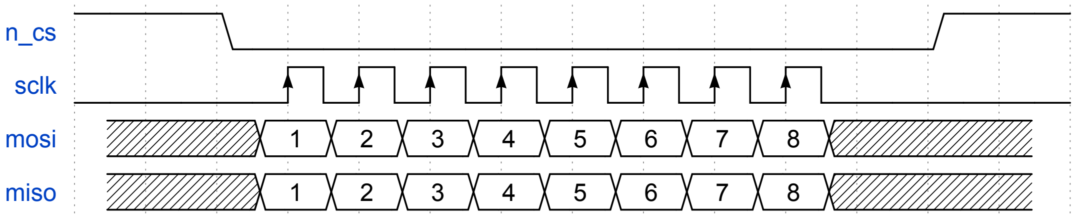


Figure 5.2: CPOL and CPHA parameters used by SPI slave controller

5.2.2 Serial Clock Frequency

As defined by the requirements in Section 2.1, the designed SPI slave controller shall support serial clock frequency of 12.5 MHz. Figure 5.3 illustrates the timing criteria corresponding to the required serial clock frequency.

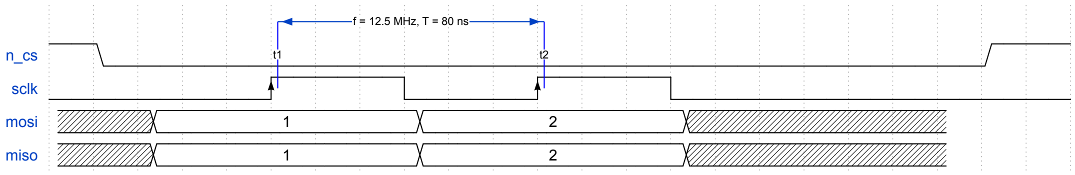


Figure 5.3: Serial clock frequency supported by SPI slave controller

5.2.3 MSb first vs. LSb first

Since SPI protocol allows sending data with either the most significant bit first or the least significant bit first, the designed slave controller shall send and receive the data with the most significant bit first. Figure 5.4 illustrates this definition by showing the corresponding bit sequence when sending a byte of data.

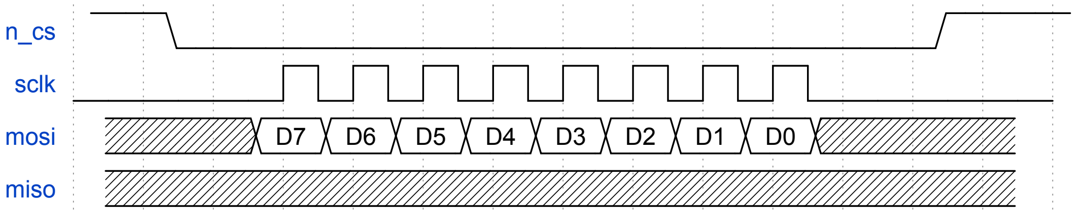


Figure 5.4: Most significant bit sent first during byte transmission over SPI

5.3 Command Interface

Having defined the memory structure of the SpW2SPI bridge in Section 4.2, a corresponding SPI command interface can now be defined as well. The command interface shall support two types of operations:

1. Read operation;
2. Write operation.

The type together with the address defines a concrete command. These two pieces of information are encoded in the command byte—the first byte sent by the SPI master. Table 5.1 presents the bit structure of the command byte.

Table 5.1: Bit structure of command byte for SPI command interface

MSb			LSb
7 5	4	3	2 0
reserved	wr	addr_ext	addr

`reserved` The purpose of this reserved three-bit field is to support the reset signal practice used in the current work (see Appendix A.1).

wr This one-bit field defines the type of the operation. When it is set it defines a write operation, otherwise it defines a read operation.

addr_ext This bit allows differentiating between addressing the mailboxes and the registers in the SpW2SPI bridge. Setting this bit allows addressing the mailbox memory region in the SpW2SPI bridge. Resetting this bit allows addressing the registers in the SpW2SPI bridge.

addr The three bits in this field form an address capable of addressing all the required memory data structures as defined in Section 4.2. Depending on the value of addr_ext bit the addressed memories can be either the five registers specified in Table 4.4 or the two mailboxes specified in Table 4.5.

5.3.1 Read Command Interface

The way the read command interface is designed is illustrated in Figure 5.5.

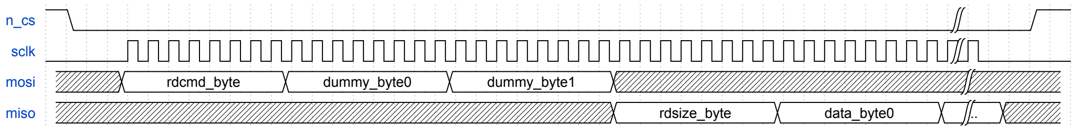


Figure 5.5: Data read from SpW2SPI bridge through SPI

This timing diagram shows that when the SPI master performs the reading then the read command must be followed by two dummy bytes instead of the read size. There is a couple of reasons behind this design decision. Firstly, there is an assumption that the read size is known beforehand, e.g. the maximum size is known from the specification presented in Section 4.2, while the current size can be read from the corresponding register during run-time. Secondly, the dummy bytes are important to give the slave application enough time to perform the required data fetching and its subsequent synchronization into the serial clock domain.

Additionally, the presented timing diagram shows that the first byte transmitted by the slave over the MISO line is the size of the returned data. This becomes very useful when the master requests the maximum number of bytes and then simply looks at the returned size to determine the size of the true data.

5.3.2 Write Command Interface

The operation of the write command interface is illustrated in Figure 5.6.

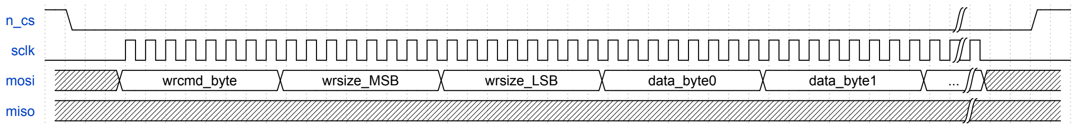


Figure 5.6: Data write to SpW2SPI bridge through SPI

The most important observation taken from the presented timing diagram is that the SPI master must specify the size along with the data when performing the write command. This kind of design was chosen as it is more concise compared to another option in which the master performs the write operation in two steps: 1) writes to the corresponding size register and 2) writes the according amount of data. The latter design option was found to be redundant. Finally, the sent size allows the slave controller to immediately check that there is enough space to accept the written data.

5.3.3 Examples of Command Interface Usage

Figure 5.7 demonstrates how the SPI master writes two bytes of telemetry—bytes 0xCA and 0xFE—to the SPI slave by using the defined command interface. The decomposition of this timing diagram is as follows:

- The value 0x19 of the command byte is decoded the following way:
  - `wr` bit is set (1) to indicate a write operation;
  - `addr_ext` bit is set (1) to address the mailbox memory region of the SpW2SPI bridge;
  - `addr` bit field is set to 001b to address the TM mailbox (see Table 4.5).
- The size of the telemetry, i.e. value 2 in this case, is transmitted as two bytes with the most significant byte, byte 0x00, being transmitted first and the least significant byte, byte 0x02, being transmitted second.
- The data bytes, bytes 0xCA and 0xFE, are transmitted after the two bytes of size.

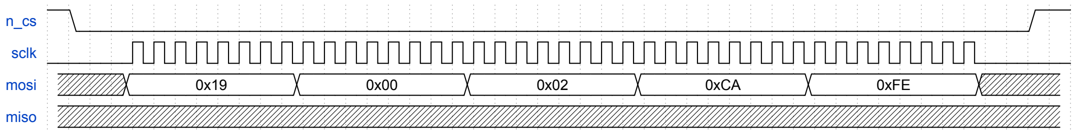


Figure 5.7: Write telemetry using SPI command interface

The command interface allows reading the SPI communication status register as demonstrated in Figure 5.8. The key points of this example are:

- The value 0x01 of the command byte is decoded as follows:
  - `wr` bit is cleared (0) to indicate a read operation;
  - `addr_ext` bit is cleared (0) to address the register memory region of the SpW2SPI bridge;
  - `addr` bit field is set to 001b to address the SPI communication status register (see Table 4.4).
- The value of the dummy bytes is not used, it is set to 0x00 in this example.
- The first byte transmitted by the SPI slave through the MISO line is the size of the status register. Since the latter is a one-byte register, the value of the transmitted byte is 0x01.
- The last byte transmitted by the SPI slave represents the contents of the status register. In this particular example it is assumed that this value is 0x02.

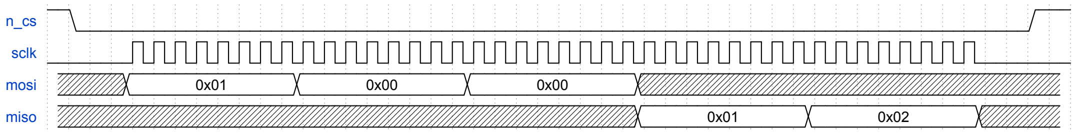


Figure 5.8: Read SPI communication status using SPI command interface

5.3.4 Take-Away Points

When writing to the SPI slave using the command interface, as exemplified in Section 5.3.3, the number of the transmitted data bytes must correspond to the transmitted size value. Any extra data bytes will be discarded by the SPI slave. Moreover, the maximum number of the transmitted data bytes must adhere to the specification presented in Section 4.2, otherwise the whole data sequence transmitted by the SPI master will be discarded.

When reading from the SPI slave, as demonstrated in Section 5.3.3, it is crucial that the SPI master is aware of the maximum size of the data it wants to read, as this ensures that the master clocks the slave the proper amount of times. This awareness can be achieved by looking at the specification presented in Section 4.2.

Finally, Table 5.2 summarizes the important command bytes.

Table 5.2: Command byte constants for SPI command interface

Command name	Byte
Write telemetry	0x19
Read telecommand	0x08
Read SPI communication status	0x01

5.4 Controller Block Diagram

The block diagram of the designed SPI slave controller is presented in Figure 5.9. While examining the presented block diagram it is important to note the design uses an asynchronous region for serial communication. The rationale behind this design choice is that the FPGA clock is only 1.6 times faster than the serial clock. Therefore, there would be significant delays if the oversampling approach was used. These delays are not that crucial when the SPI master only writes data to the slave. However, when the master reads data from the slave the delays would seriously impact the correct operation of the communication.

The presented block diagram can be split into three main parts:

MCUCOM_SPI	Capture and propagation of SPI signals;
FIFO	Clock domain crossing as well as elastic data buffering;
MCUCOM	Encoding and decoding the SPI command interface.

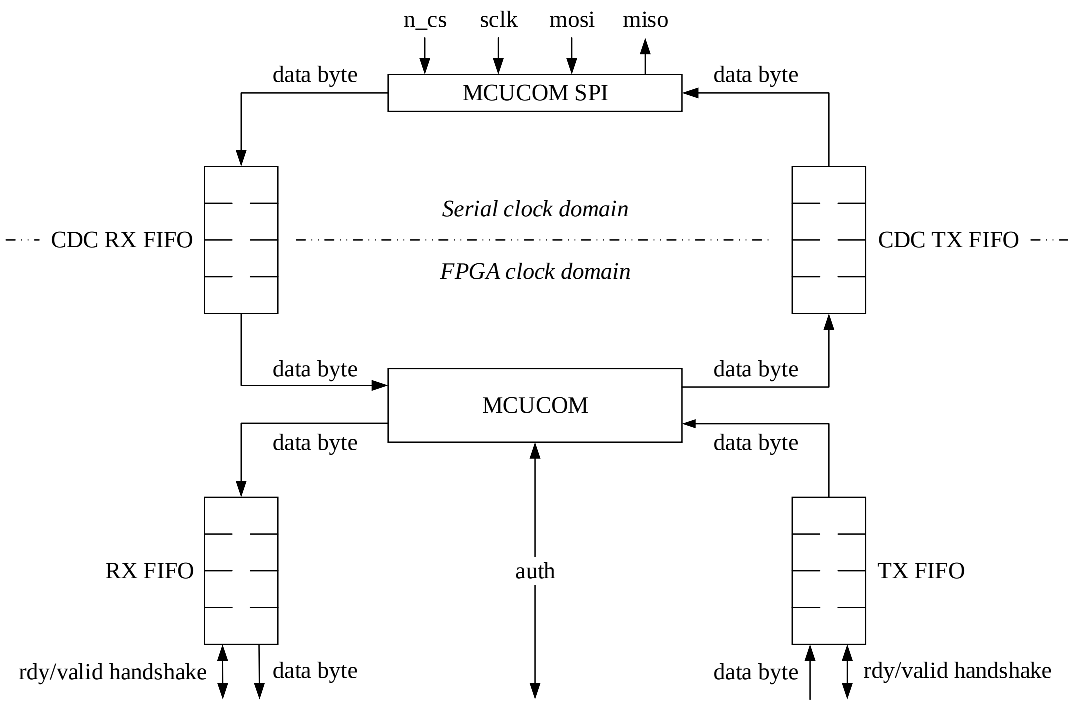


Figure 5.9: SPI slave controller block diagram

5.4.1 Capture and Propagation of SPI Signals

Figures 5.10 and 5.11 depict the capture and propagation of SPI signals, respectively. There are a few important notes about these circuits, namely:

- Even though the capturing circuit uses a common way of sampling the MOSI input using a shift register, the least significant bit of the output data byte is taken directly from the MOSI line. The reason is that the last serial clock cycle must be accommodated to register the output data byte into the CDC FIFO. Therefore, the least significant bit does not go into the shift register, but directly into the FIFO.
- The propagation circuit uses a single falling edge flip-flop to synchronize the MISO output following the SPI parameters' settings defined in Section 5.2.1. Minimizing the number of falling edge flip-flops is the recommended practice as guided by Reuse Methodology Manual [19, "Avoid Mixed Clock Edges"].

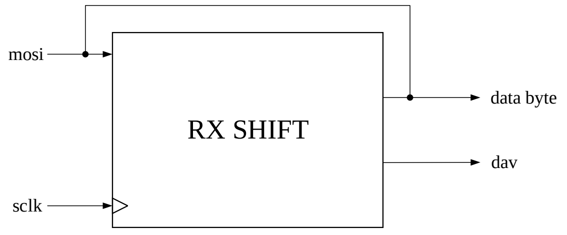


Figure 5.10: SPI signal capture circuit

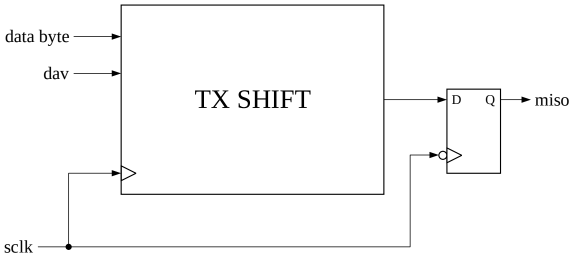


Figure 5.11: SPI signal propagation circuit

5.4.2 Clock Domain Crossing

The clock domain crossing was accomplished using an asynchronous FIFO design presented in SNUG San Jose 2002 paper [20]. The block diagram of the proposed asynchronous FIFO can be seen in Figure 5.12. Additionally, in order to highlight the important timing features of the FIFO a timing diagram is depicted in Figure 5.13. This timing diagram demonstrates a simple scenario in which a 1-slot FIFO is first written from the write clock domain and then read from the read clock domain. From the perspective of the client code using this FIFO it is important to note that there is a delay of two clock cycles after deasserting the write or read enable signal and before being able to observe the properly updated value of the full or empty flag, respectively. To overcome this timing issue and improve the usability of the module one could implement the almost full and almost empty flags to augment or even replace the ordinary full and empty flags. This way, by putting the almost full offset to, say, 4 words, the writer would be able to write exactly this amount of data first and after that immediately observe the properly updated almost full flag value.

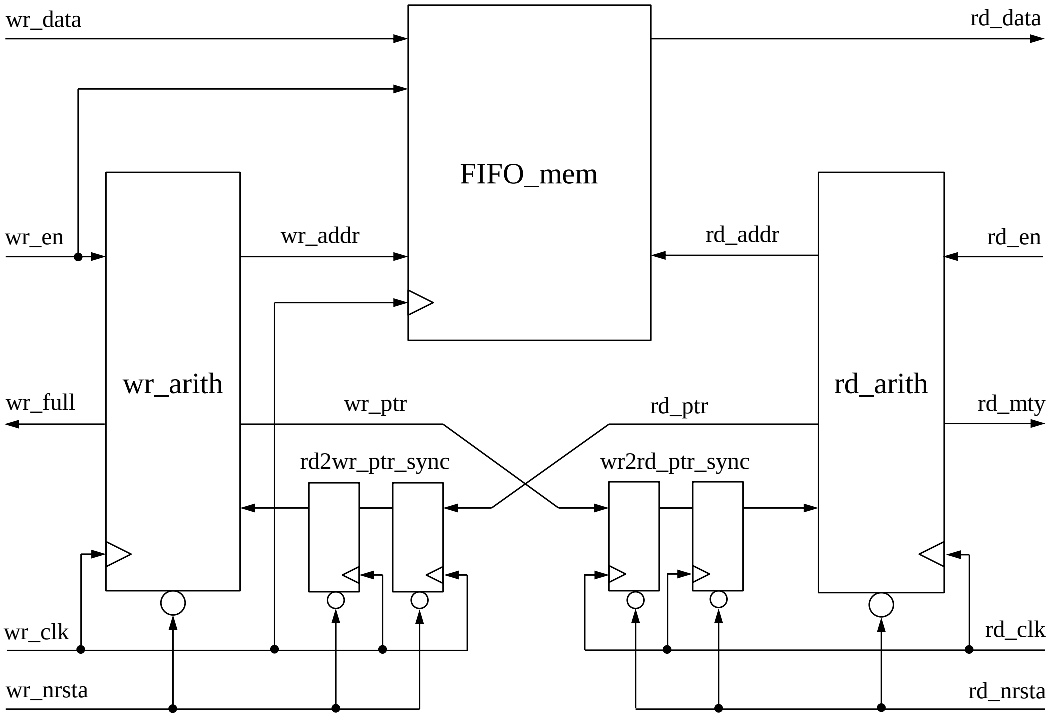


Figure 5.12: Asynchronous FIFO block diagram for SPI controller CDC

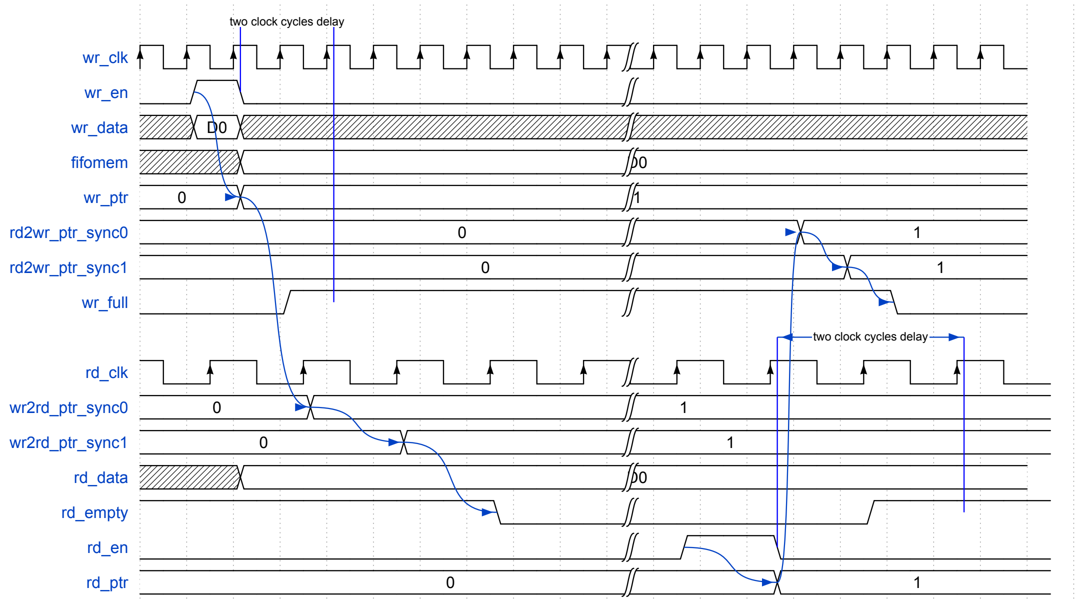


Figure 5.13: Asynchronous FIFO timing diagram for SPI controller CDC

Finally, since the FIFO code found in the paper was written in Verilog, this code had to be rewritten in VHDL to comply with the current work.

5.4.3 Command Interface Codec

The important aspect of the MCUCOM module, i.e. the SPI command interface codec, is the timing constraint of the read operation. Due to the fact that SPI protocol in general does not have any flow control, the slave controller must make sure that the data to be read is put on the MISO line at the right moment. Figure 5.14 depicts the sequence of events that happen during the read operation. As can be seen from the presented sequence diagram the time interval between the bytes received from the SPI link is approximately 640 nanoseconds at the serial clock frequency of 12.5 MHz. Therefore, the current logic of the codec is designed in such a way that the reception of the read command byte serves as the trigger to start the authorization process. In its turn the reception of the first dummy byte serves as the trigger to already transmit the read data size byte. Consequently, the authorization process must be finished before the first dummy byte is received in order to have all the required read data ready in advance. This leaves the authorization process with 640 ns / 25 ns $\approx$ 25 FPGA clock cycles which is more than enough at the moment. Nevertheless, this may be one of the limiting factors to increase the SPI frequency above 12.5 MHz, and thus must be paid attention to.

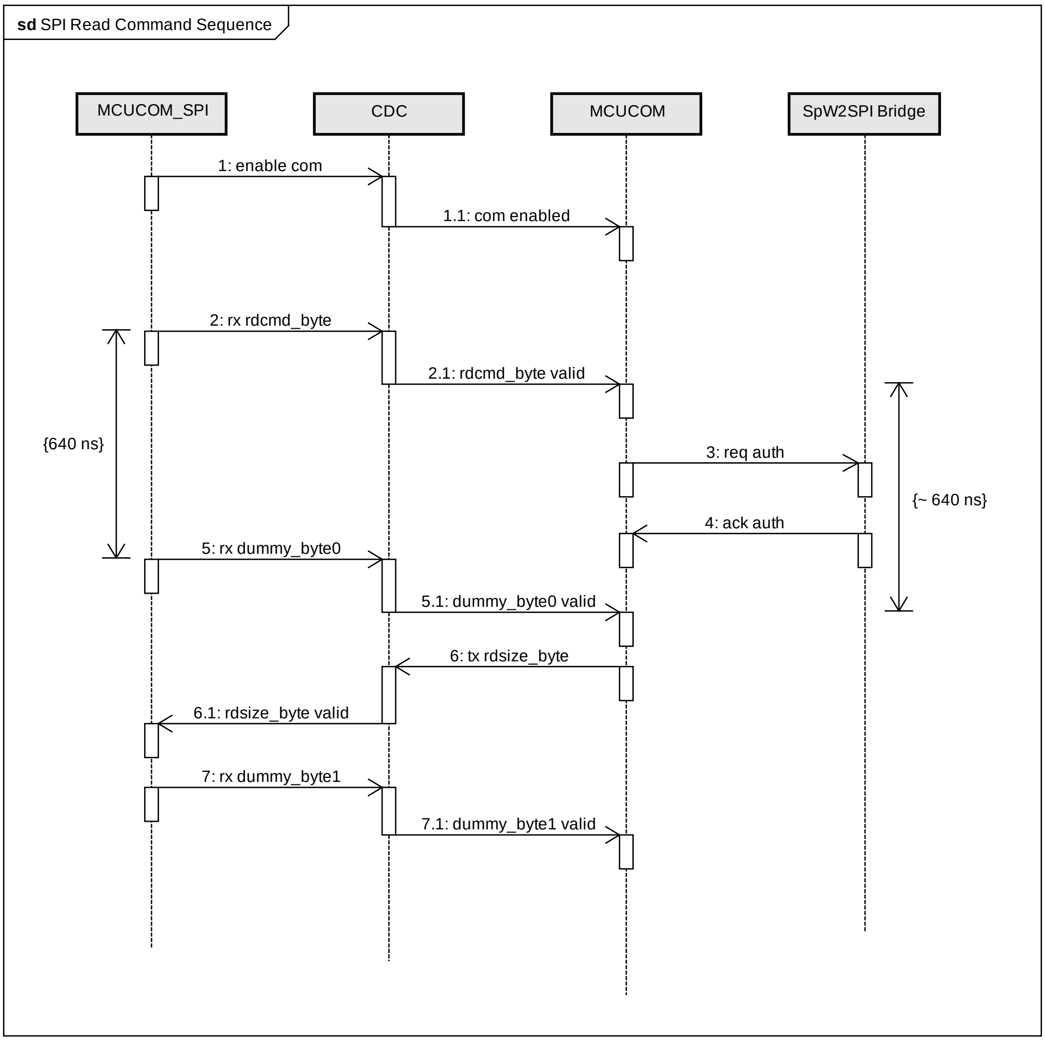


Figure 5.14: SPI slave controller timing constraint during read operation

Chapter 6

SpaceWire Codec Design

6.1 Protocol Description

SpaceWire is a serial communication technology defined by the European Cooperation for Space Standardization (ECSS) Standard ECSS-E-ST-50-12C [4]. According to Microsemi [21, "SpaceWire Coding and Signaling Overview"] this communication technology “provides a unified, high-speed data-handling infrastructure for connecting sensors, processing elements, mass memory units, down-link telemetry subsystems, and electrical ground support equipment (EGSE)”. The mentioned infrastructure nodes are interconnected using short distance point-to-point links that operate in full-duplex mode with a data rate from 2 Mbit/s to 400 Mbit/s.

SpaceWire uses a synchronous data communication which means that a clock signal must be passed from the transmitter to the receiver. However, instead of having the clock signal itself as a separate wire, e.g. like in SPI, SpaceWire uses data/strobe encoding to recover the clock from the received signals. This encoding technique, therefore, dictates that SpaceWire has two main signal wires (omitting the LVDS details for now)—the data and the strobe. While the data signal directly follows the data bit-stream, i.e. the data signal is high when the data bit is 1 and low when the data bit is 0, the strobe signal changes its state whenever the data remains constant. This allows recovering the clock signal from these two wires alone using a simple XOR operation. Figure 6.1 illustrates this clock recovery technique. As noted by P. Walker and B. Cook [22, "Data-Strobe encoding"] this technique “is one of the contributing factors in SpaceWire being a simple, digital, circuit, without needing analog electronics”.

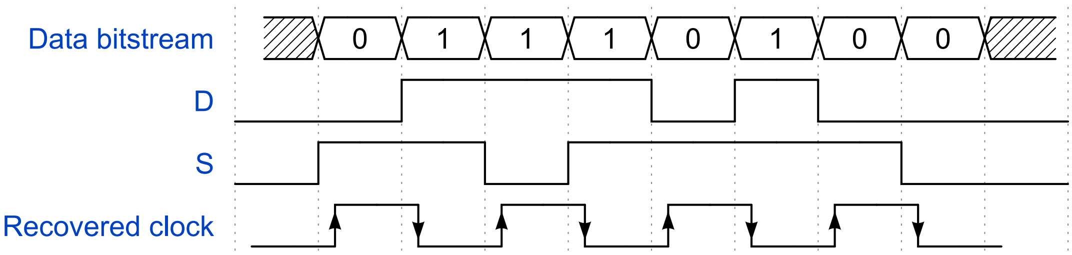


Figure 6.1: SpaceWire DS encoding technique

As briefly mentioned above, SpaceWire uses low voltage differential signaling (LVDS) [23] for the data and strobe signals at the physical layer which gives SpaceWire the fault tolerant properties of LVDS [24]. Figure 6.2 illustrates SpaceWire LVDS signaling schematic.

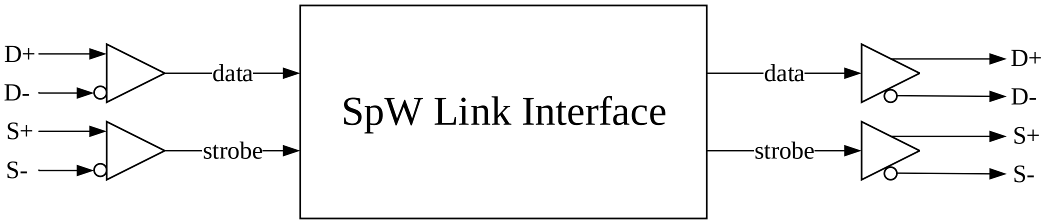


Figure 6.2: SpaceWire LVDS signaling

On the exchange level SpaceWire defines a number of link- and normal-characters [4, "Link-characters and normal-characters"] which have their specific responsibilities—to keep the link alive and to carry user data, respectively. Therefore, beside its main purpose to encode and decode user data characters as they are being transmitted from one SpaceWire node to the other, a SpaceWire codec dedicates a large portion of logic to keeping the corresponding SpaceWire link up and running. The latter involves the link initialization sequence, flow control, error detection and recovery logic.

6.2 Protocol Usage Considerations

6.2.1 SDR vs. DDR

There are SpaceWire codec implementations [25, "Single Data Rate (SDR) and Double (or Dual) Data Rate (DDR) bit-stream encoding"] that “allow[] the user to select SDR or DDR encoding for the transmitted bit-stream”. The designed codec, however, shall support only DDR configuration for its transmitted bit-stream.

6.2.2 Data Signaling Rate

The maximum data signaling rate supported by the designed SpaceWire codec shall be set to 100 Mbit/s. Since the data transfer rate has already been defined to be DDR (see Section 6.2.1), a 50MHz SpaceWire clock will be able to provide the necessary data signaling rate.

6.2.3 Transmitter Clock Generation

The transmission clock of the designed SpaceWire codec shall be a phase locked loop multiple of the local FPGA clock. The advantage of using a PLL is that the latter provides a more robust clock source for the SpaceWire transmitter circuit. From the perspective of the design integration, however, this decision forces the corresponding transmitter clock generation component to be external to the main codec design which can be viewed on the global block diagram presented in Section 6.3.

6.2.4 Receiver Clock Recovery

The receiver clock shall be recovered by simply XOR'ing the data and strobe signals. No additional filter circuits shall be designed to ensure glitch-free recovered clock [21, "RTG4

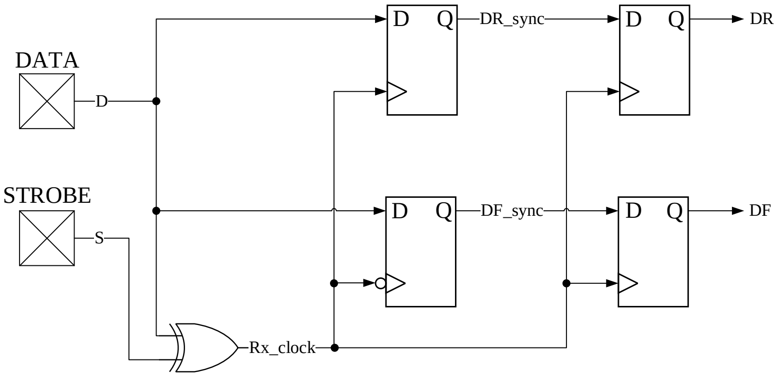


Figure 6.3: SpaceWire receiver clock and data recovery logic

SpaceWire Clock Recovery Block Overview"]. Figure 6.3 illustrates a basic way to design the SpaceWire clock and data recovery circuitry. Even though simple this circuitry can pose serious challenges related to the placement of these components inside the FPGA chip. However, since the design in this work uses a relatively slow SpaceWire clock defined in Section 6.2.2, it is possible to achieve adequate clock recovery results by properly constraining the placement and routing process.

6.2.5 Time-Code Support

The Time-Code as defined by the SpaceWire standard [4, "Control characters and control codes"] shall not be supported. The reason is that this functionality was not needed for the current project.

6.3 Global Block Diagram

Figure 6.4 depicts the global block diagram of the designed SpaceWire codec. The necessity to present the design from a global point of view is related to the specific partitioning inherent to this design—certain components, e.g. the receiver clock recovery as well as the transmitter clock generation, have been made external to the core of the codec. There is a couple of reasons behind this design decision. Firstly, it is usually more preferable to implement such components like DDR registers by using of the primitives provided by the concrete FPGA. Secondly, using a PLL to create the required SpaceWire transmitter clock is again a more preferable solution as in this case the synthesis tool makes sure that the generated clock signal resides on the dedicated high speed global bus. Finally, there is a certain flexibility with this kind of partitioning approach. Namely, the SpaceWire receiver clock recovery is currently implemented using a simple XOR operation (see Section 6.2.4). However, later this simple circuit could be substituted by dedicated clock recovery blocks like the ones provided by high-end FPGA boards [21, "Using RTG4 SpaceWire Clock Recovery Block"]. Therefore, delegating these responsibilities outside the core of the codec has its rightful reasons.

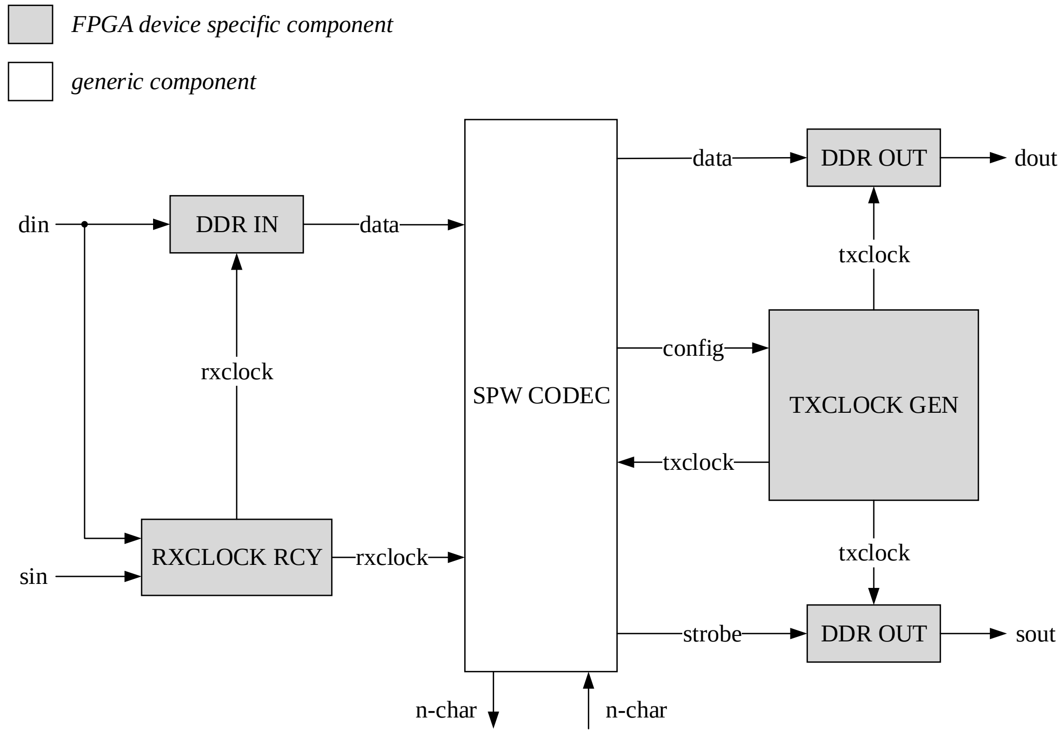


Figure 6.4: SpaceWire codec global block diagram

6.4 Core Block Diagram

The core block diagram of the designed SpaceWire codec is presented in Figure 6.5. The examination of the presented diagram reveals that the proposed design differs from the one suggested by the SpaceWire standard [4, "Encoder-decoder block diagram"] by a number of important aspects. The most notable of them are the following:

- The main algorithm of the codec resides totally in the synchronous FPGA clock region. Therefore, Rx and Tx asynchronous regions can be made very small with no complex logic inside. This kind of design not only facilitates better synthesis results, but also allows the developed component to keep the power consumption of the FPGA low.
- The host interface is fully synchronous to the FPGA clock domain. Consequently, no additional synchronization is needed outside the codec component.
- Transmitter and receiver FIFO's are placed inside the codec. This way the host is freed from the responsibility to maintain these buffers.

The proposed design solution is similar to the one presented in a Master's thesis completed in Chalmers University of Technology [10]. There are, however, certain differences. The most important of them being the clock domain crossing technique. As seen in Figure 6.5, the solution developed in this work uses asynchronous FIFO's to synchronize data between the FPGA clock domain and both SpaceWire clock domains. This kind of synchronization scheme was chosen, because it helps keeping the logic simple. Instead of having numerous complicated synchronization channels it is possible to have just one which operates in a straightforward way. True, there are constraints related to using a uniform FIFO approach. For example, a FIFO forces to use a fixed and often the maximum width of the data being synchronized which may not seem

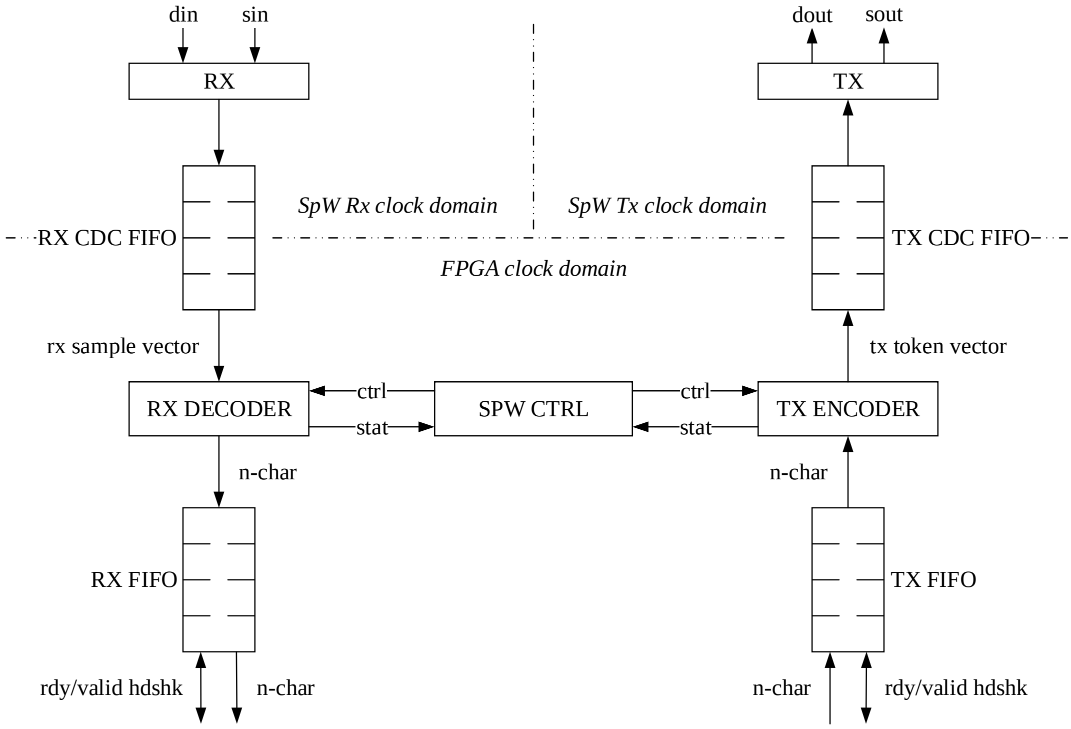


Figure 6.5: SpaceWire codec core block diagram

ideal from the perspective of the FPGA resource usage. It may also be difficult to urgently send something like a Time-Code if the FIFO is already filled with other data. Nevertheless, the simplicity of the designed logic was deemed superior for the current work, and therefore the asynchronous FIFO approach prevailed.

There are also other possible ways to partition a SpaceWire codec design, e.g. to make the whole design synchronous to the FPGA clock [26]. The fully synchronous solution seems interesting, but it was not studied thoroughly during this work due to the limited time.

6.5 Rx Pipeline

This section will provide the most essential notes regarding the Rx pipeline as seen in Figure 6.5 starting from the data/strobe input signals and following the Rx data flow down to the host interface.

6.5.1 Deserializer

Compared to the receiver circuit proposed in the other Master's thesis [10, "Receiver"], the design in the current work adheres to the principle of keeping the Rx block as simple as possible. Therefore, this design can be illustrated with a simple block diagram presented in Figure 6.6. The sample vector output observed on the presented diagram has a generic width, i.e. the number of 2-bit samples that are shifted into the Rx shift register can be easily specified in the VHDL code. Furthermore, this generic sample vector in its turn determines the data that the Rx decoder will have to decode in one FPGA clock cycle. All together this generic behavior allows quick adjustments to the Rx pipeline in order to test its support for various SpaceWire Rx clocks frequencies.

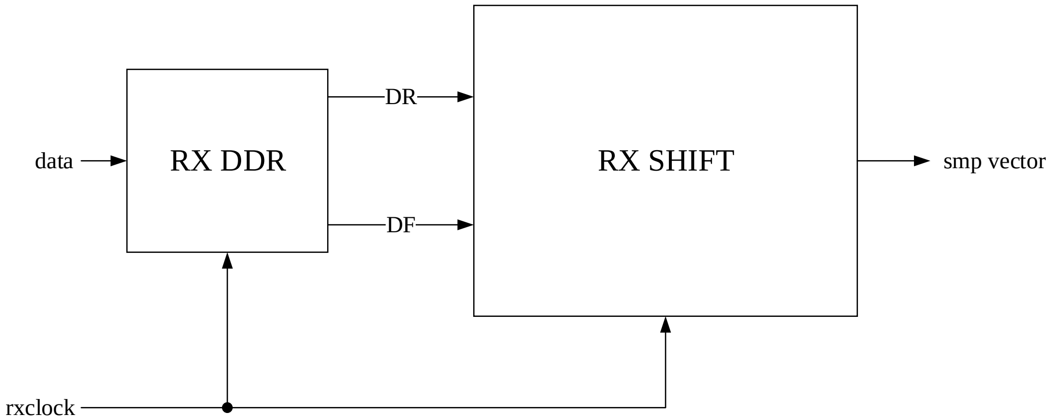


Figure 6.6: SpaceWire codec Rx deserializer block diagram

6.5.2 Clock Domain Crossing

The clock domain crossing was done using the same approach as explained in Section 5.4.2. The sample vector—the output of the Rx deserializer—was flattened, or serialized, into a bit string before being pushed into the asynchronous FIFO. Once synchronized to the other side of the clock domain boundary the bit string was popped from the FIFO and then deserialized again into the sample vector—now the input to the Rx decoder.

6.5.3 Decoder

The idea was to design a decoder which could handle a generic width of the Rx data input. Such design could later prove useful for testing the ability of the codec to support various SpaceWire Rx clock frequencies. However, such design also proved to be rather challenging both during the design and implementation phases. Undoubtedly, most of the challenge was caused by a natural requirement in this case that the generic data input had to be processed in one clock cycle. As the data kept constantly coming from the receiver, failure to decode it on time would result in losing this data. Moreover, due to the nature of the SpaceWire character decoding the pipeline approach did not seem applicable as it would suffer from data hazards on a constant basis without the possibility to stall the decoding pipeline. Finally, the decoding algorithm had to be easily scalable to properly support the generic data input.

Figure 6.7 illustrates the state diagram of the decoder FSM that was easily scalable and functionally correct. However, the most important observation here is that each state in this FSM operated on a 2-bit sample. Any attempts to design an equally scalable and correct FSM that operates on wider samples failed. This circumstance in its turn posed an implementation challenge, because the algorithm was supposed to be implemented into hardware, not software. Too long combinational paths could dramatically slow down the timing of the final hardware.

The solution that offered satisfactory synthesis results was to replicate the above-mentioned FSM using a VHDL for loop construct by a number of times which corresponded to the generic number of 2-bit samples contained in an input data vector. Further testing both in simulation and in hardware showed that the proposed solution was indeed viable.

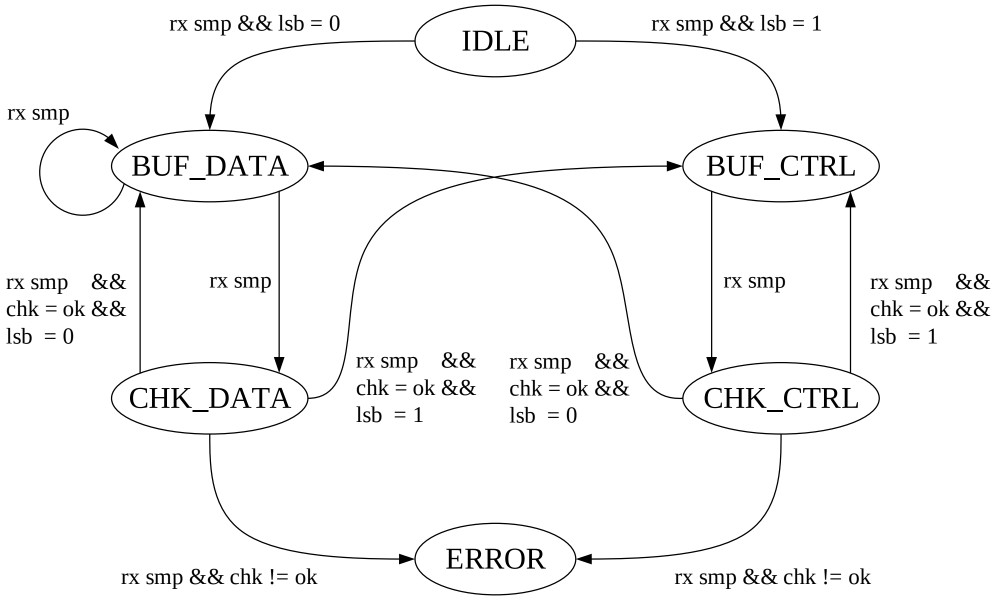


Figure 6.7: SpaceWire Rx decoder FSM operating on 2-bit samples (smp)

6.5.4 Host FIFO

As described in Section 6.4 the Rx FIFO that receives the normal characters from the Rx decoder was placed inside the SpW codec, and thus inside the Rx pipeline. Such encapsulation allows hiding from the client the details of the credit flow handling inherent to SpaceWire link technology [4, "Flow control (normative)"]. The read interface of the FIFO is exposed to the client, thus forming the Rx interface of the codec. This interface follows the ready/valid handshake protocol [17, "Link-Level Flow Control and Buffering"].

6.6 Tx Pipeline

The transmission pipeline as seen on the right side in Figure 6.5 shall be described in this section. The description will start from the host interface and continue following the Tx data flow up to the data/strobe output signals.

6.6.1 Host FIFO

As noted in Section 6.4 the Tx FIFO that receives the normal characters from the host was placed inside the SpW codec. Moreover, the corresponding rationale and the interface details explained and defined in Section 6.5.4 hold for the Tx FIFO as well.

6.6.2 Encoder

Figure 6.8 illustrates the block diagram of the Tx encoder.

The Tx encoder is responsible for the encoding of all the information that is going to be transmitted over the link. This includes the normal-characters as well as the link-characters. Even

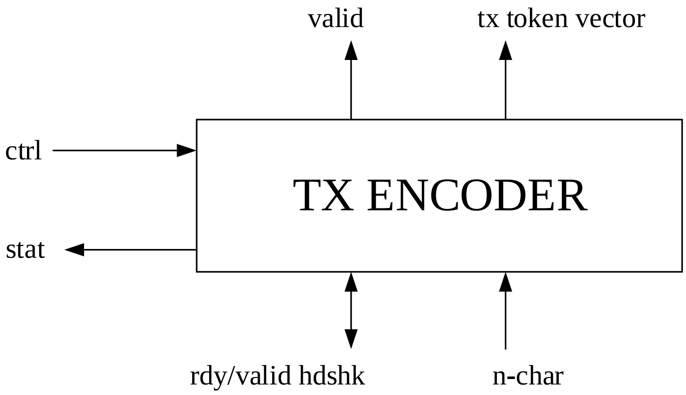


Figure 6.8: SpaceWire Tx encoder block diagram

though various SpaceWire characters can have different transmission priorities as defined by the standard [4, "Flow control"], the Tx encoder is completely separated from this responsibility. The corresponding character prioritization is done in another place, in the dedicated control logic as seen in the center of Figure 6.5.

Since the encoded data will be further pushed into an asynchronous FIFO for the clock domain crossing purposes, the data is encoded in a specific, but at the same time uniform way. That is, the size of the output token vector will always be the same no matter which character is encoded in that vector. Clearly, such a uniform vector must be able to accommodate the maximum character to be sent—the 14-bit Time-Code. Hence, the size of this vector is set accordingly. Moreover, since the size of the payload in the vector can vary, this vector needs to carry an extra control information. Table 6.1 defines the structure of the vector. It is seen that each token includes two encoded bits of data and a flag indicating whether this particular token contains valid data or not. Consequently, the fixed size of the vector is calculated as 14 bits / 2-bit pair = 7 tokens.

Table 6.1: SpaceWire Tx encoder output token vector structure

Token #	6		5		4		...		0	
Data	MSb	LSb	MSb	LSb	MSb	LSb	MSb	LSb	MSb	LSb
Data valid	Boolean		Boolean		Boolean		Boolean		Boolean	

Table 6.2 demonstrates the token vector containing an encoded NULL character. It is noteworthy that all the tokens with valid data are grouped in the left-most part of the vector—this is the defined encoding behavior of the Tx encoder.

Table 6.2: SpaceWire Tx encoder output token vector for NULL character

Token #	6		5		4		3		2		1		0	
Data	0	1	1	1	0	1	0	0	0	0	0	0	0	0
Data valid	true		true		true		true		false		false		false	

The final important encoding feature worth mentioning here is the encoding of FCT characters. When a FCT character is encoded, a NULL character is automatically appended after the FCT.

The reason is related to the asynchronous nature of the Tx serializer. In the run mode this serializer will be working with a higher frequency, so in order to keep it constantly sending data without unnecessary breaks it was decided to artificially make the encoded FCT character 'wider'. The latter has no negative side effects in terms of the standard protocol. Table 6.3 demonstrates the token vector containing an encoded FCT character.

Table 6.3: SpaceWire Tx encoder output token vector for FCT character

Token #	6		5		4		3		2		1		0	
Data	0	1	0	0	0	1	1	1	0	1	0	0	0	0
Data valid	true		true		true		true		true		true		false	

6.6.3 Clock Domain Crossing

The clock domain crossing was done using the same approach as explained in Section 5.4.2. The token vector—the output of the Tx encoder—was first flattened, or serialized, into a bit string before being pushed into the asynchronous FIFO. Once synchronized to the other side of the clock domain boundary the bit string was popped from the FIFO and then deserialized again into the token vector—now the input to the Tx serializer.

6.6.4 Serializer

The Tx serializer implements a parallel-in-serial-out circuit that shifts the token vectors encoded by the Tx encoder to output the corresponding DDR data every Tx clock cycle.

6.6.5 Strobe Generator

Listing 6.1 demonstrates the algorithm for strobe generation used in this work. Although the algorithm is written in pseudocode, the comments should provide clear description in order to derive the corresponding VHDL code. It can be noticed from the code that the corresponding Tx strobe module does not use a reset signal, but rather an active high enable signal. The latter comes from the Tx serializer module. Consequently, this enable signal will be high when there is data to transmit and low when there is no such data.

Listing 6.1 SpaceWire Tx strobe generation algorithm in pseudocode

```
-----------------------------------------
-- Pseudocode for SpW strobe generation
-----------------------------------------

-----------------------------------------
-- Register input data
-----------------------------------------
data_in := inp.data_in;

-----------------------------------------
-- Propagate data to output
-----------------------------------------
if    en = ’0’ then
      data_out.dr := ’0’;
      data_out.df := ’0’;
else
      data_out    := data_in;
end if;

-----------------------------------------
-- Generate strobe
-----------------------------------------
if    en = ’0’ then
      safe_dr_rst := data_out.df and strb_out.df;
      strb_out.dr := safe_dr_rst;
      strb_out.df := ’0’;
else
      dr          := not (data_out.df xor data_in.dr xor strb_out.df );
      strb_out.dr := dr;
      strb_out.df := not (data_in.dr  xor data_in.df xor dr          );
end if;
```

Chapter 7

SpaceWire RMAP Target Controller Design

7.1 Protocol Description

The remote memory access protocol (RMAP) [27] allows a SpaceWire node to write to or read from the memory inside another SpaceWire node. The protocol is intended to standardize the way SpaceWire nodes are configured and how they exchange data between each other. For instance, RMAP can be used to configure a remote camera to take a picture by writing to a specific memory region inside the camera device. The camera may then write the captured image data into a known allocated memory region inside a mass memory device. Later the initiator of the initial request can read this image data from this memory device using RMAP.

There are two types of RMAP commands that are useful for the current work:

1. Read command;
2. Write command.

7.1.1 Read Command

Table 7.1 illustrates the read command format as defined by the RMAP standard [27, "Read Command"].

7.1.2 Write Command

Table 7.2 illustrates the write command format as defined by the RMAP standard [27, "Write Command"].

7.2 Protocol Usage Considerations

Based on the requirements of the current work the designed RMAP target controller shall only provide partial RMAP implementation as defined by the standard [27, "Partial Implementations of RMAP"]. Therefore, the following list applies:

Table 7.1: SpaceWire RMAP read command format

First byte transmitted

	Target SpW Address	...	Target SpW Address
Target Logical Address	Protocol Identifier	Instruction	Key
Reply Address	Reply Address	Reply Address	Reply Address
Reply Address	Reply Address	Reply Address	Reply Address
Reply Address	Reply Address	Reply Address	Reply Address
Initiator Logical Address	Transaction Identifier (MSB)	Transaction Identifier (LSB)	Extended Address
Address (MSB)	Address	Address	Address (LSB)
Data Length (MSB)	Data Length	Data Length (LSB)	Header CRC
EOP			

Last byte transmitted

Bits in Instruction field

MSb						LSb
reserved = 0	command = 1	read = 0	verify data = 0	reply = 1	increment = 1 no inc = 0	reply addr len
packet type		command				reply addr len

- Initiator functionality shall not be supported as the designed system solution is a target only device;
- Read-modify-write command shall not be supported as this functionality is not needed for the current work;
- SpaceWire addressing (previously called path addressing) shall not be supported, only the logical addressing;
- The write command shall only be supported in the non-acknowledged, non-verified mode as defined by the standard [27, "Write commands"];
- The increment of the read/write address shall not be supported, because it is not needed for the current work.

7.3 Command Interface

The command interface describes how the RMAP packet header fields must be used by the RMAP initiator in order to:

1. Respect the partial RMAP implementation defined in Section 7.2;
2. Comply with the memory structure of the SpW2SPI bridge as defined in Section 4.2.

As the RMAP standard inherently supports read and write commands the command interface description is a relatively straightforward process.

Table 7.2: SpaceWire RMAP write command format

First byte transmitted

	Target SpW Address	...	Target SpW Address
Target Logical Address	Protocol Identifier	Instruction	Key
Reply Address	Reply Address	Reply Address	Reply Address
Reply Address	Reply Address	Reply Address	Reply Address
Reply Address	Reply Address	Reply Address	Reply Address
Initiator Logical Address	Transaction Identifier (MSB)	Transaction Identifier (LSB)	Extended Address
Address (MSB)	Address	Address	Address (LSB)
Data Length (MSB)	Data Length	Data Length (LSB)	Header CRC
Data	Data	Data	Data
Data	...	...	Data
Data	Data CRC	EOP	

Last byte transmitted

Bits in Instruction field

MSb						LSb
reserved = 0	command = 1	write = 1	verify data = 1 don't verify = 0	reply = 1 no reply = 0	increment = 1 no inc = 0	reply addr len
packet type		command				reply addr len

7.3.1 Command Field

7.3.1.1 Read

When the RMAP initiator, i.e. OBC, issues a read command to the target the Instruction field of the corresponding RMAP packet header must be set to value 0x48. This value is decoded as follows (refer to Table 7.1):

- The Packet Type field is set to value 01b, that is:
  - The Reserved bit is cleared (0) as defined by the standard [27, "Packet type field"];
  - The Command bit is set (1) to indicate a command.
- The Command field is set to value 0010b, that is:
  - The Write/Read bit is cleared (0) to indicate a read;
  - The Verify-Data-Before-Write bit is cleared (0), because this bit is not relevant in this case;
  - The Reply bit is set (1), because the read data is sent back to the initiator as a reply;
  - The Increment/No Increment bit is cleared (0), because this feature is not supported.
- The Reply Address length field is set to 00b, because there is no support for this feature.

7.3.1.2 Write

When the RMAP initiator, i.e. OBC, issues a write command to the target the Instruction field of the corresponding RMAP packet header must be set to value 0x60. This value is decoded as follows (refer to Table 7.2):

- The Packet Type field is set to value 01b, that is:
  - The Reserved bit is cleared (0) as defined by the standard [27, "Packet type field"];
  - The Command bit is set (1) to indicate a command.
- The Command field is set to value 1000b, that is:
  - The Write/Read bit is set (1) to indicate a write;
  - The Verify-Data-Before-Write bit is cleared (0) due to the non-acknowledged, non-verified mode used by this design;
  - The Reply bit is cleared (0) again due to the non-acknowledged, non-verified mode used by this design;
  - The Increment/No Increment bit is cleared (0), because this feature is not supported.
- The Reply Address length field is set to 00b, because there is no support for this feature.

7.3.2 Address Fields

Both the Extended Address and Address fields of the RMAP packet header are used to address the memory structure of the SpW2SPI bridge as specified in Section 4.2. The addressing scheme based on the RMAP address fields is defined as follows:

- The Extended Address field is used to differentiate between addressing the mailboxes and the registers in the SpW2SPI bridge. Setting this byte to value 0x01 allows addressing the mailbox memory region in the SpW2SPI bridge. Alternatively, by setting this byte to 0x00 allows addressing the registers in the SpW2SPI bridge.
- The Address field represents a four-byte address which is used to address all the required memory data structures as specified in Sections 4.2.1 and 4.2.2.

7.3.3 Redundant Fields

Due to the specific nature of the current application some fields of the RMAP packet header do not matter, i.e. they can be any value. The following list enumerates these redundant fields:

- Target Logical Address;
- Key.

7.3.4 Examples of Command Interface Usage

Table 7.3 demonstrates how the RMAP initiator, i.e. OBC, writes a telecommand to the RMAP target using the defined command interface. The notable key-points of this example are as follows:

- The Instruction field is set to 0x60 to indicate a write command (see Section 7.3.1.2);
- The Extended Address field is set to 0x01 to select the mailbox memory region (see Section 7.3.2);
- The Address field is set to 0x00000000 to select the telecommand mailbox (see Table 4.5).

Table 7.3: Write telecommand mailbox using RMAP command interface

Field		No. bytes	Value
Target SpW Address		0	-
Target Logical Address		1	0x54
Protocol Identifier		1	0x01
Instruction	Packet Type	1	01b
	Command		1000b
	Reply Address Length		00b
Key		1	0x88
Reply Address		0	-
Initiator Logical Address		1	0x76
Transaction Identifier		2	0x00 0x00
Extended Address		1	0x01
Address		4	0x00 0x00 0x00 0x00
Data Length		3	0x00 0x00 0x09
Header CRC		1	0xE7
Data		9	0x80 0x01 ... 0x04
Data CRC		1	0xC5
	TOTAL	26	

Table 7.4 demonstrates how the RMAP initiator, i.e. OBC, reads the OBC status register from the RMAP target using the defined command interface. The essential key-points of this example are as follows:

- The Instruction field is set to 0x48 to indicate a read command (see Section 7.3.1.1);
- The Extended Address field is set to 0x00 to select the register memory region (see Section 7.3.2);
- The Address field is set to 0x00000000 to select the OBC status register (see Table 4.4).

Table 7.4: Read OBC status register using RMAP command interface

Field		No. bytes	Value
Target SpW Address		0	-
Target Logical Address		1	0x54
Protocol Identifier		1	0x01
Instruction	Packet Type	1	01b
	Command		0010b
	Reply Address Length		00b
Key		1	0x88
Reply Address		0	-
Initiator Logical Address		1	0x76
Transaction Identifier		2	0x00 0x07
Extended Address		1	0x00
Address		4	0x00 0x00 0x00 0x00
Data Length		3	0x00 0x00 0x01
Header CRC		1	0xF4
	TOTAL	16	

7.4 Controller Block Diagram

7.4.1 Overview

Figure 7.1 presents the block diagram of the SpaceWire RMAP controller.

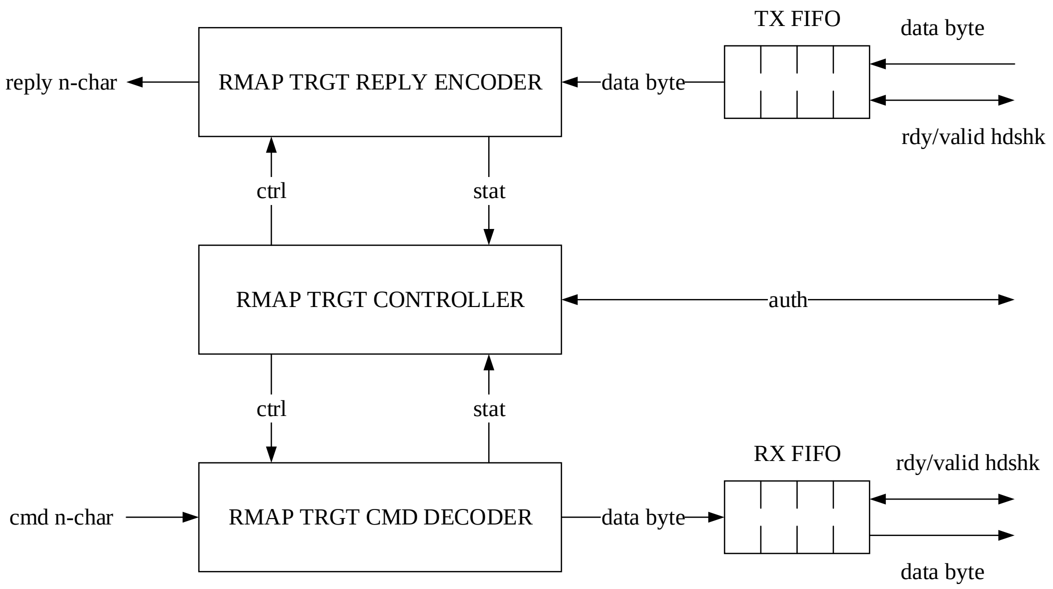


Figure 7.1: SpaceWire RMAP target controller block diagram

7.4.2 Command Decoder

The target command decoder is responsible for decoding RMAP command packets.The headers of these RMAP packets are first checked for validity using the header CRC and then the authorization parameters are passed to the target controller to be authorized by the host.

7.4.3 Reply Encoder

The responsibility of the target reply encoder is to send RMAP reply packets with the data from a read command.

7.4.4 Target Controller

The target controller controls the reception, authorization and reply of an RMAP target transaction.

7.4.5 Global Timing Diagrams for RMAP Write and Read Commands

As it was specified in Section 7.1 there are two RMAP commands that are required for the current work, i.e. the write and the read commands. As such, Figures 7.2 and 7.3 demonstrate the global timing diagrams of the designed RMAP target controller for the write and the read commands, respectively. These diagrams reveal the operation steps made inside the designed controller when processing the supported RMAP commands.

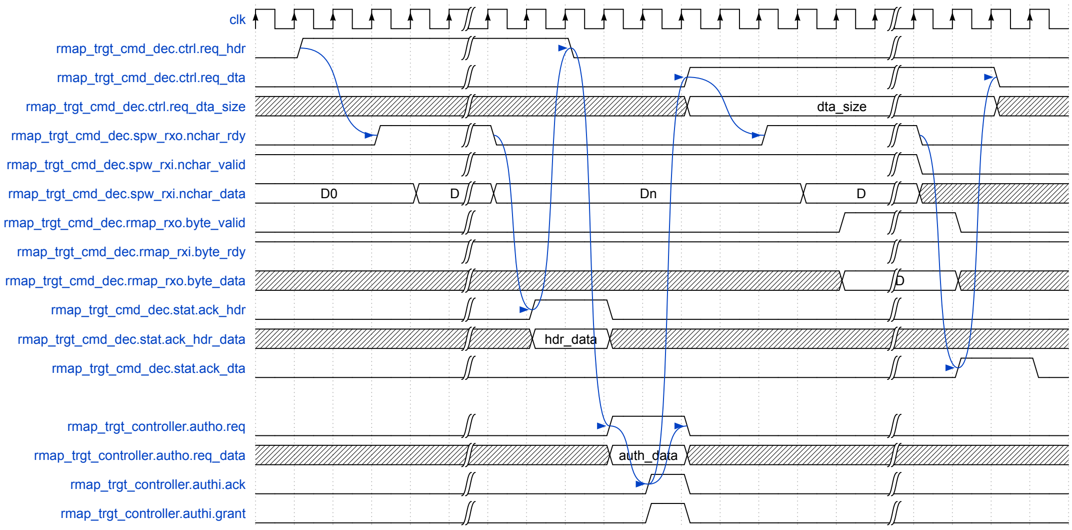

clk
rmap_trgt_cmd_dec.ctrl.req_hdr
rmap_trgt_cmd_dec.ctrl.req_dta
rmap_trgt_cmd_dec.ctrl.req_dta_size
dta_size
rmap_trgt_cmd_dec.spw_rxo.nchar_rdy
rmap_trgt_cmd_dec.spw_rxi.nchar_valid
rmap_trgt_cmd_dec.spw_rxi.nchar_data
D0
D
Dn
D
rmap_trgt_cmd_dec.rmap_rxo.byte_valid
rmap_trgt_cmd_dec.rmap_rxi.byte_rdy
rmap_trgt_cmd_dec.rmap_rxo.byte_data
D
rmap_trgt_cmd_dec.stat.ack_hdr
rmap_trgt_cmd_dec.stat.ack_hdr_data
hdr_data
rmap_trgt_cmd_dec.stat.ack_dta
rmap_trgt_controller.autho.req
rmap_trgt_controller.autho.req_data
auth_data
rmap_trgt_controller.authi.ack
rmap_trgt_controller.authi.grant

Figure 7.2: SpaceWire RMAP target global timing diagram for RMAP write command

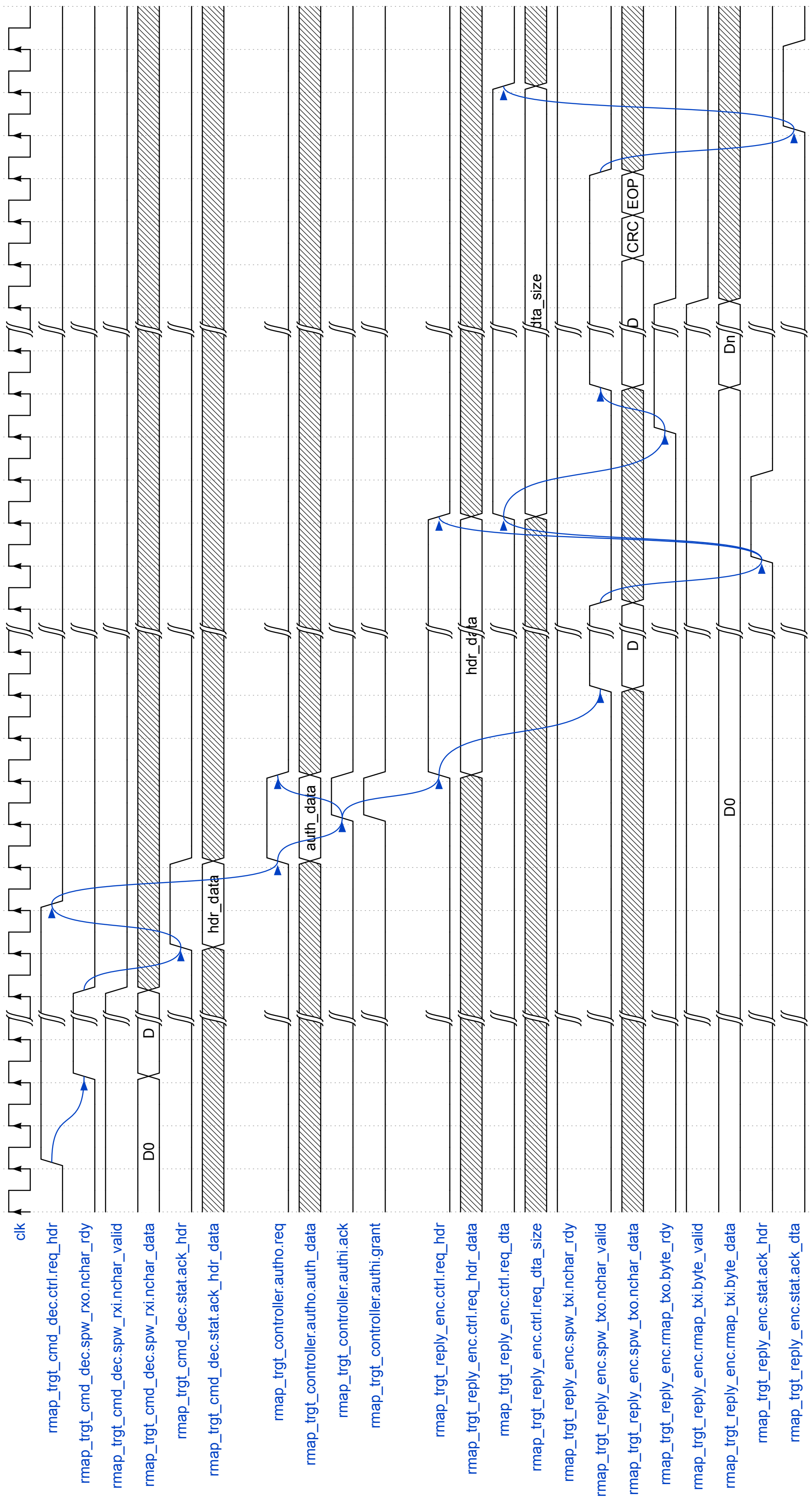

clk
rmap_trgt_cmd_dec.ctrl.req_hdr
rmap_trgt_cmd_dec.spw_rxo.nchar_rdy
rmap_trgt_cmd_dec.spw_rxi.nchar_valid
rmap_trgt_cmd_dec.spw_rxi.nchar_data
rmap_trgt_cmd_dec.stat.ack_hdr
rmap_trgt_cmd_dec.stat.ack_hdr_data
rmap_trgt_controller.autho.req
rmap_trgt_controller.autho.auth_data
rmap_trgt_controller.authi.ack
rmap_trgt_controller.authi.grant
rmap_trgt_reply_enc.ctrl.req_hdr
rmap_trgt_reply_enc.ctrl.req_hdr_data
rmap_trgt_reply_enc.ctrl.req_dta
rmap_trgt_reply_enc.ctrl.req_dta_size
rmap_trgt_reply_enc.spw_txi.nchar_rdy
rmap_trgt_reply_enc.spw_txo.nchar_valid
rmap_trgt_reply_enc.spw_txo.nchar_data
rmap_trgt_reply_enc.rmap_txo.byte_rdy
rmap_trgt_reply_enc.rmap_txi.byte_valid
rmap_trgt_reply_enc.rmap_txi.byte_data
rmap_trgt_reply_enc.stat.ack_hdr
rmap_trgt_reply_enc.stat.ack_dta
D0
D
hdr_data
auth_data
hdr_data
dta_size
D
D
CRC
EOP
D0
Dn

Figure 7.3: SpaceWire RMAP target global timing diagram for RMAP read command

Chapter 8

Prototyping the Design on Microsemi ProASIC3E Starter Kit FPGA

8.1 Overview

When prototyping the designed system on Microsemi ProASIC3E Starter Kit FPGA board (demonstrated in Figure 8.1) there are two main sources of information regarding the hardware:

1. ProASIC3E Flash Family FPGAs with Optional Soft ARM Support [28];
2. ProASIC3E Starter Kit User Guide [29].

Here it is very important to understand that the first document describes the general features of ProASIC3E FPGA device, while the second document defines a concrete application of the described device in terms of the Starter Kit FPGA board. The configuration of the FPGA I/O banks can be taken as an example of why this difference is so important (see Section 8.2).

Additionally, some clarification should be given regarding the optional ARM support mentioned in the name of the first document (see Section 8.3).

There were also certain challenges during the prototyping process. Even though one of them—the inference of RAM—is more related to Synopsys synthesis tools which are part of Libero SoC software, it is still worth describing this issue in the current prototyping context (see Section 8.4).

Since the current work closely deals with SpaceWire, the details of constraining the SpaceWire Rx clock recovery must be specified (see Section 8.5).

8.2 I/O Banks Configuration

The documentation of Microsemi ProASIC3E FPGA family A3PE1500 device states [28, "Table 1-1 : ProASIC3E Product Family"] that there are 8 I/O banks available on the FPGA board. These banks can be seen in Figure 8.1 as four pin headers around the FPGA chip. Each I/O bank can be configured to use the I/O standards supported by the FPGA device, including LVTTL and LVDS. During the selection of the required I/O standards in the MultiView Navigator (MVN)

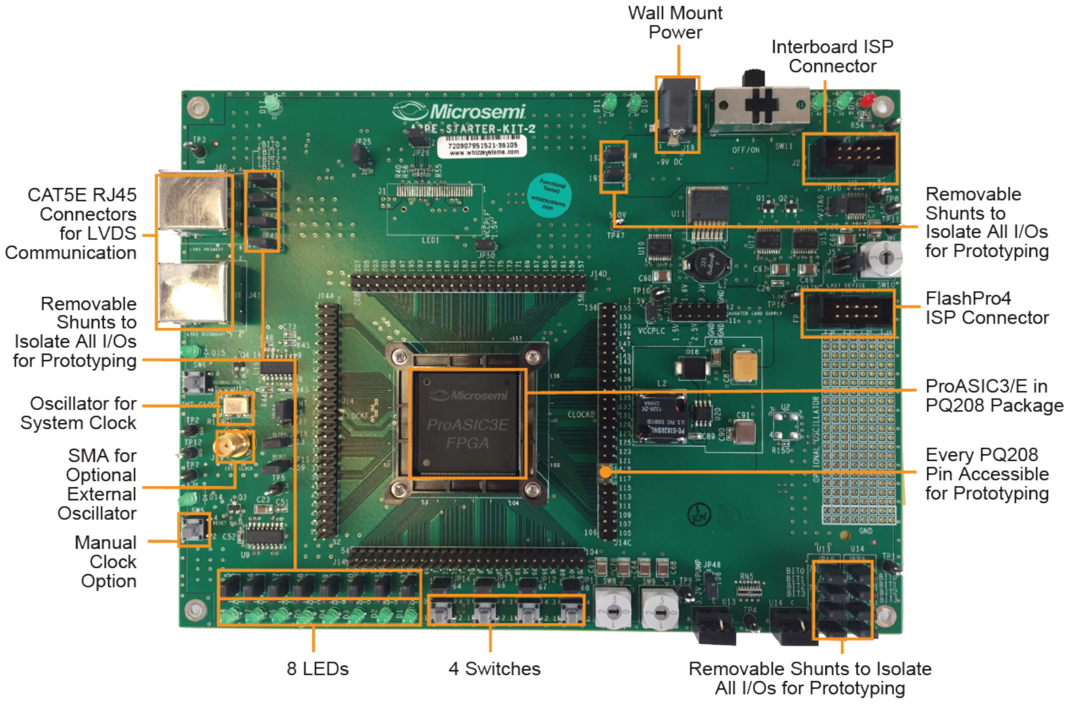

Wall Mount Power
Interboard ISP Connector
CAT5E RJ45 Connectors for LVDS Communication
Removable Shunts to Isolate All I/Os for Prototyping
Oscillator for System Clock
SMA for Optional External Oscillator
Manual Clock Option
Removable Shunts to Isolate All I/Os for Prototyping
FlashPro4 ISP Connector
ProASIC3/E in PQ208 Package
Every PQ208 Pin Accessible for Prototyping
8 LEDs
4 Switches
Removable Shunts to Isolate All I/Os for Prototyping

Figure 8.1: Microsemi ProASIC3E Starter Kit [1]

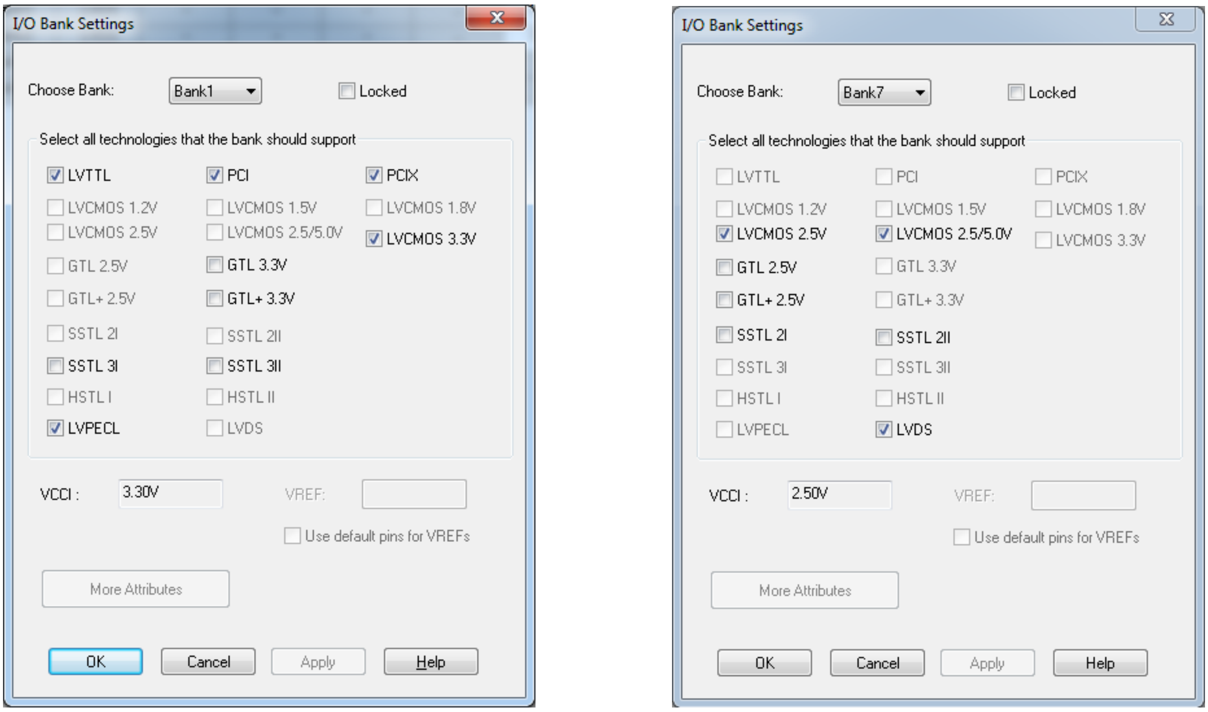
tool [30] in Libero SoC the proper voltage levels are automatically applied to the corresponding I/O banks (see Figure 8.2).

Even though the MVN tool allows applying various voltages to the I/O bank, the actual hardware, i.e. the starter kit FPGA board, may not support certain voltages on certain I/O banks. This is where the difference between the general specification and the actual implementation comes into play. As depicted in one of the board schematics for ProASIC3E Starter Kit FPGA [29, "Figure 13, FPGA Headers and Expansion Bus"] only two I/O banks—banks 4 and 5—can be configured to provide various voltage levels, all the other banks have their certain voltage levels fixed. As an additional note Figure 8.3 shows how switches SW9 and SW8 can be used to configure the voltage levels of banks 4 and 5, respectively. The necessity to provide such detail here is that the schematics provided by Microsemi are rather ambiguous in this regard.

The final important note here is that the voltage level for LVDS I/O standard supported by ProASIC3E FPGA family is exactly 2.5 volts, not 3.3 volts [28, "Table 2-78, LVDS Minimum and Maximum DC Input and Output Levels"]. Failure to know this detail in advance may seriously impact the prototyping process and thus the overall design time schedule.

8.3 Soft ARM Support

Even though mentioned in the first document presented in Section 8.1, the given Starter Kit does not have the soft ARM support. To use this feature one has to acquire another development kit from Microsemi which is Cortex-M1 enabled [31].



I/O Bank Settings

Choose Bank: Bank1 ☐ Locked

Select all technologies that the bank should support

☑ LVTTL ☑ PCI ☑ PCIX
☐ LVCMOS 1.2V ☐ LVCMOS 1.5V ☐ LVCMOS 1.8V
☐ LVCMOS 2.5V ☐ LVCMOS 2.5/5.0V ☑ LVCMOS 3.3V
☐ GTL 2.5V ☐ GTL 3.3V
☐ GTL+ 2.5V ☐ GTL+ 3.3V
☐ SSTL 2I ☐ SSTL 2II
☐ SSTL 3I ☐ SSTL 3II
☐ HSTL I ☐ HSTL II
☑ LVPECL ☐ LVDS

VCCI : 3.30V VREF:

☐ Use default pins for VREFs

(a)

I/O Bank Settings

Choose Bank: Bank7 ☐ Locked

Select all technologies that the bank should support

☐ LVTTL ☐ PCI ☐ PCIX
☐ LVCMOS 1.2V ☐ LVCMOS 1.5V ☐ LVCMOS 1.8V
☑ LVCMOS 2.5V ☑ LVCMOS 2.5/5.0V ☐ LVCMOS 3.3V
☐ GTL 2.5V ☐ GTL 3.3V
☐ GTL+ 2.5V ☐ GTL+ 3.3V
☐ SSTL 2I ☐ SSTL 2II
☐ SSTL 3I ☐ SSTL 3II
☐ HSTL I ☐ HSTL II
☐ LVPECL ☑ LVDS

VCCI : 2.50V VREF:

☐ Use default pins for VREFs

(b)

Figure 8.2: Microsemi ProASIC3E I/O banks configuration

8.4 RAM Inference

According to documentation [28, "Table 1-1 : ProASIC3E Product Family"] Microsemi ProASIC3E FPGA family A3PE1500 device has 270 Kbits (1024 bit) of RAM. This was more than enough for the needs of the current work. However, the inference of this RAM proved to be rather challenging.

Although there were VHDL coding guidelines provided by Synopsys FPGA Synthesis manuals for RAM inference specifically for Microsemi ProASIC3E devices [32, "VHDL Guidelines for RAM Inference"], the given VHDL code (see Listing 8.1) nevertheless failed to infer RAM during the synthesis process. The requested RAM was implemented using flip-flops instead of the available RAM blocks. This was very undesirable since the design used a decent amount of RAM, and by implementing this RAM using flip-flops the synthesis process quickly ran out of FPGA resources.

The RAM inference failure could be related to the fact that the given reference code did not register the read output. Indeed, the read path involved a large multiplexer that induced a non-negligible delay overhead. Instead, by registering the output the read path could be completely isolated within the RAM entity thus simplifying the synthesis process [19, "Register All Outputs"].

Listing 8.2 presents the VHDL code for RAM inference that successfully inferred all the required RAM blocks in Microsemi ProASIC3E FPGA device. However, when using this code one must pay attention to the one clock cycle delay during the read operation caused by the additional register on the read path.

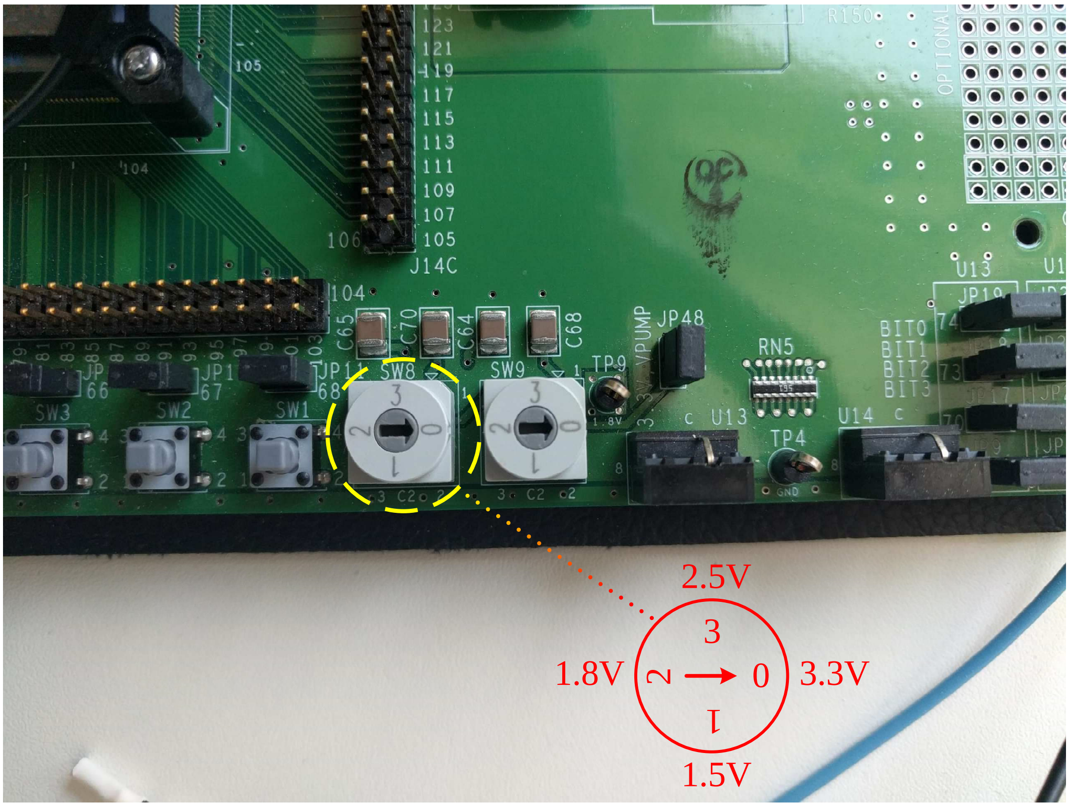


Figure 8.3: Microsemi ProASIC3E Starter Kit voltage level settings for I/O banks 4 and 5

8.5 SpaceWire Rx Clock Recovery Constraints

First of all it should be mentioned that the input and output LVDS buffers were embedded into the top level schematic of the design using the macro blocks `INBUF_LVDS` and `OUTBUF_LVDS` provided in the Libero SoC catalog. Figure 8.4 demonstrates the top level schematic of the design made in Libero SoC Smart Design tool.

Having the LVDS I/O buffers in place the SpaceWire Rx clock recovery was constrained based on the recommendations provided in Microsemi documents [21, "Timing Analysis of RTG4 Data Recovery Block"]. This resulted in the placement of important components, such as the first pair of capturing D flip-flops and the XOR gate, as presented in Figure 8.5.

8.6 Verification and Results

8.6.1 Simulation

The prototype was simulated both partially and completely to check that the overall functionality was correct. Figure 8.6 demonstrates the block diagram of the simulation testbench used in this work. The presented block diagram omits some secondary details, such as the DDR registers and the Tx clock generator of the intermediate SpaceWire codec module. This is done intentionally in order to focus on the important bits. Namely, it can be seen that the testbench is driven by two top-level types of stimuli: 1) transfer RMAP data and 2) transfer SPI data. These

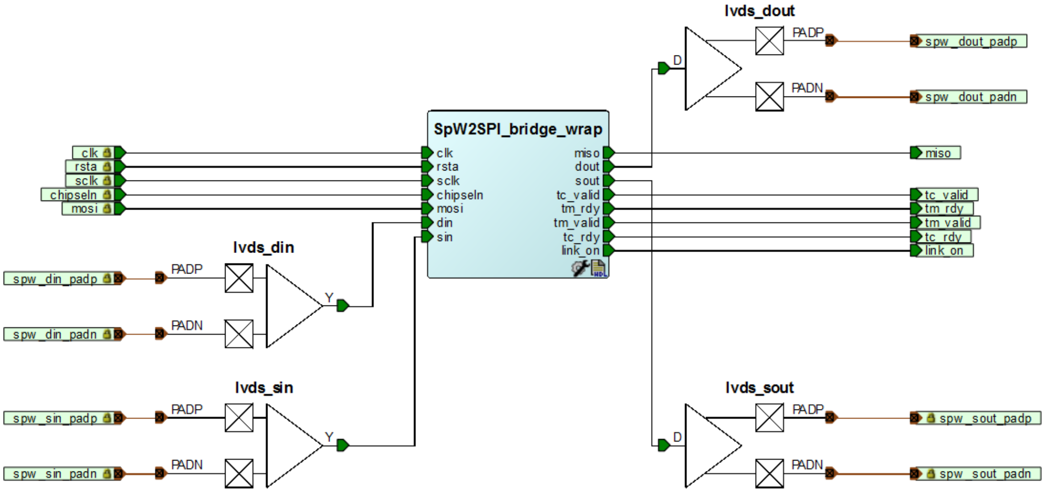


Figure 8.4: Microsemi ProASIC3E Starter Kit prototype top level schematic

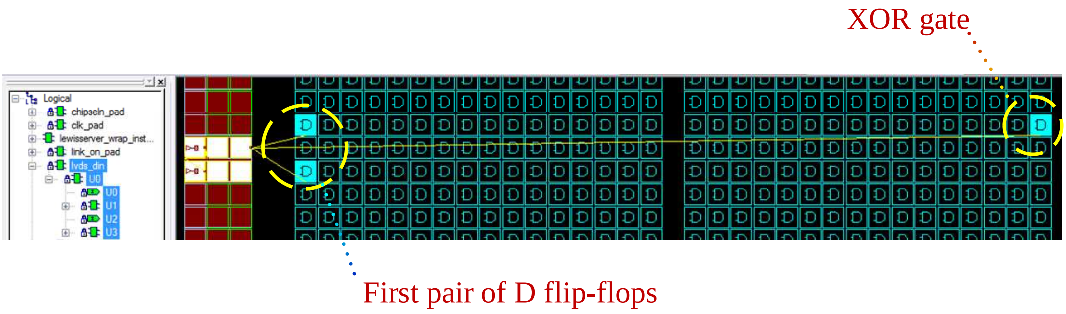


Figure 8.5: Microsemi ProASIC3E Starter Kit prototype SpW Rx clock recovery placement

Listing 8.1 Microsemi ProASIC3E RAM inference VHDL coding guideline by Synopsys

```
----------------------------------------------------------------
-- Synopsys VHDL guideline
-- for RAM inference on Microsemi ProASIC3E FPGA devices
--
-- NOTE      : The code is provided AS IS with no modifications.
--
-- IMPORTANT : std_logic_unsigned is NOT a standard package,
--             thus its use is HIGHLY DISCOURAGED!!
--
----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ram_test is
port (d   : in  std_logic_vector(7 downto 0);
      a   : in  std_logic_vector(6 downto 0);
      we  : in  std_logic;
      clk : in  std_logic;
      q   : out std_logic_vector(7 downto 0) );
end ram_test;

architecture rtl of ram_test is
  type    mem_type is array (127 downto 0) of std_logic_vector(7 downto 0);
  signal  mem : mem_type;
begin
  process (clk)
  begin
    if    rising_edge(clk) then
          if    we = ’1’ then
                mem(conv_integer (a)) <= d;
          end if;
    end if;
  end process;
  q <= mem(conv_integer (a));
end rtl;
```

Table 8.1: Microsemi ProASIC3E Starter Kit prototype area summary

Resource	Used	Total	%
Core cells	6265	38400	16
Block RAMs	12	60	20

stimuli are wrapped into their corresponding procedures (the emulators) to ease the configuration of the test cases. Therefore, by using this testbench it is possible to simulate complete end-to-end communication use cases.

8.6.2 Area Summary

The synthesis of the whole system gave the results presented in Table 8.1. In fact, the usage of the core cells could be even lower provided that more RAM blocks were used for various FIFO's in the design.

8.6.3 Prototype Assembly and Testing

Later, when the prototype was assembled (see Figure 8.7), an end-to-end hardware verification was performed at the Space Exploration Institute (Space-X). The sequence of actions presented in Figure 8.8 was successfully executed on the prototype which showed that the communication through SPI and, most importantly, SpaceWire was working correctly.

Finally, Figures 8.9 and 8.10 demonstrate the operation of the links during run-time captured with logic analyzers.

Listing 8.2 Microsemi ProASIC3E RAM inference VHDL code

```
------------------------------------------------------------
-- VHDL guideline
-- for RAM inference with registered output.
--
-- NOTE : The rd_data output is assigned inside a synchronous
--        process, thus inferring a register at the output.
--
------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity dev_utils_byteram is
generic ( ramsize   : positive                      );
port    ( clk       : in  std_logic;
          addr      : in  natural range ramsize - 1 downto 0;
          rd_data   : out std_logic_vector(7 downto 0);
          wr_data   : in  std_logic_vector(7 downto 0);
          wr_en     : in  std_logic                 );
end entity dev_utils_byteram;

architecture fpga_proasic3e of dev_utils_byteram is
  type    mem_type is array (ramsize - 1 downto 0) of std_logic_vector(7 downto 0);
  signal  mem : mem_type;
begin
  ram_proc : process(clk) is
  begin
    if    rising_edge(clk) then
          rd_data <= mem(addr); -- the read data is registered here
          if    wr_en = ’1’ then
                mem(addr) <= wr_data;
          end if;
    end if;
  end process ram_proc;
end architecture fpga_proasic3e;
```

8.7 Areas for Improvement

As shown in Section 8.6.3 the prototype successfully executed an end-to-end communication test sequence. However, it was nevertheless noted that by changing environmental conditions, e.g. by cooling the prototype FPGA board, certain transmitted bits got flipped thus corrupting the transmitted data. Interestingly, it was discovered that this problem occurred on the SPI communication side, and since the current SPI implementation does not use any data integrity checks on the signal level this data was successfully transmitted further through SpaceWire. The solution to this problem could be to apply very precise FPGA pin constraints. These constraints would define the setup and hold times required for a stable communication between the MCU of the wireless access point and the FPGA.

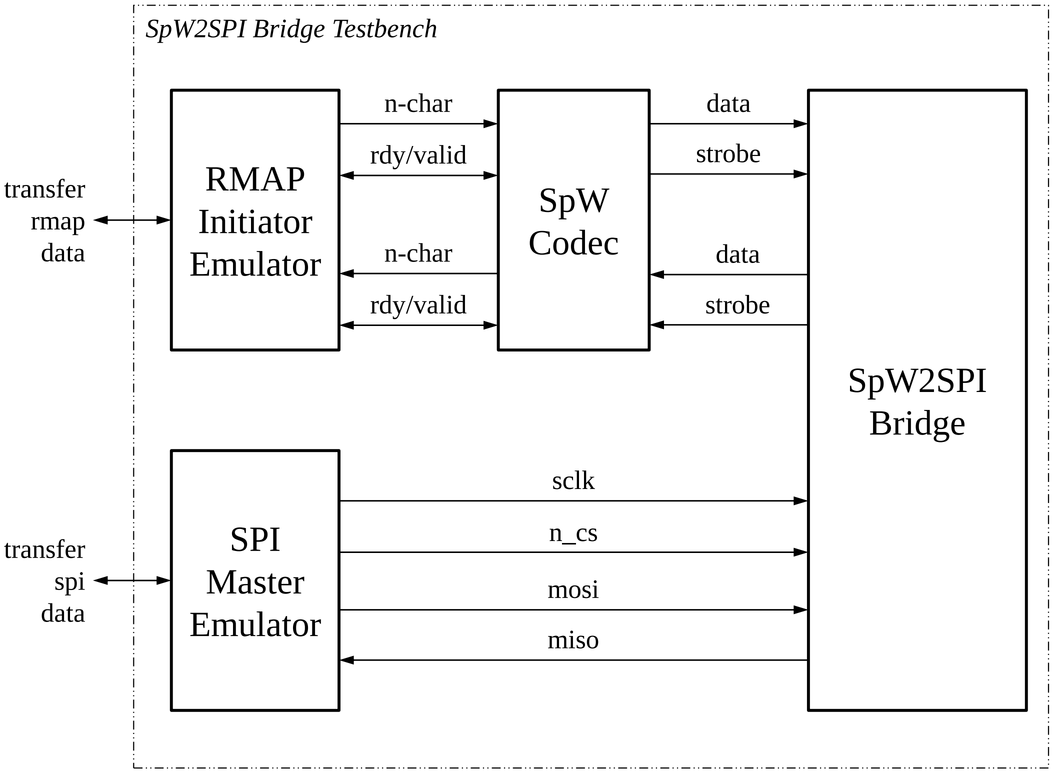


Figure 8.6: SpW2SPI prototype simulation testbench block diagram

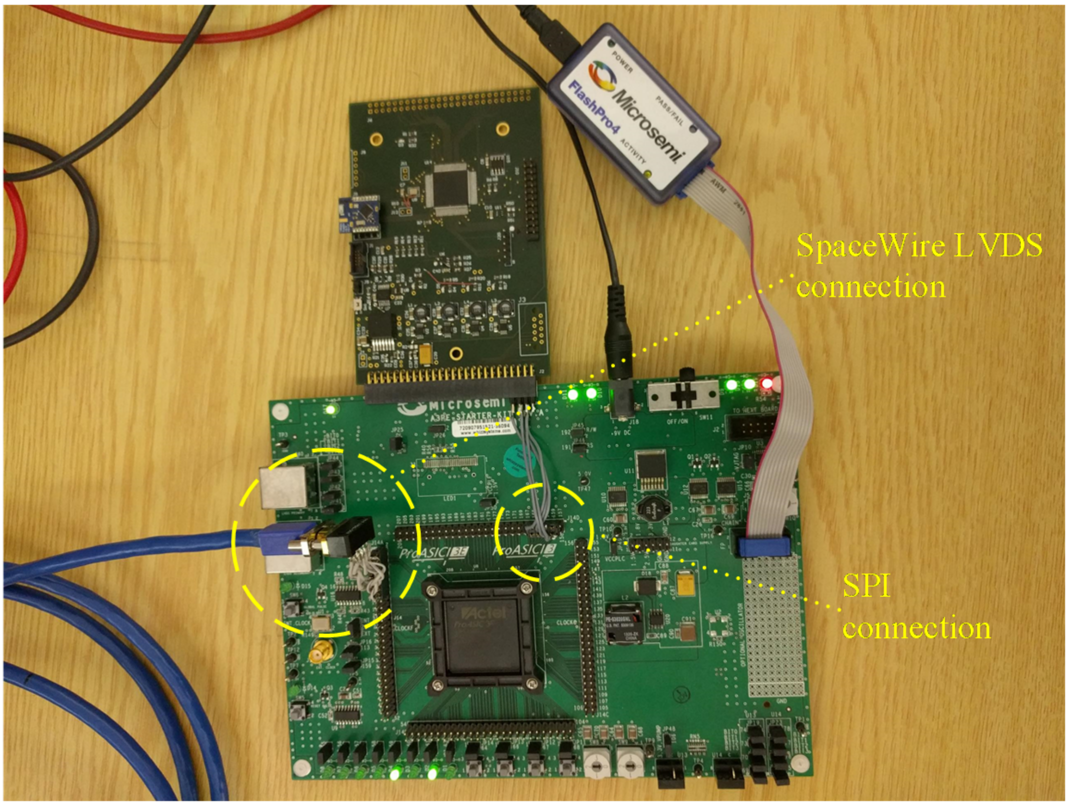


Figure 8.7: Microsemi ProASIC3E Starter Kit prototype in action

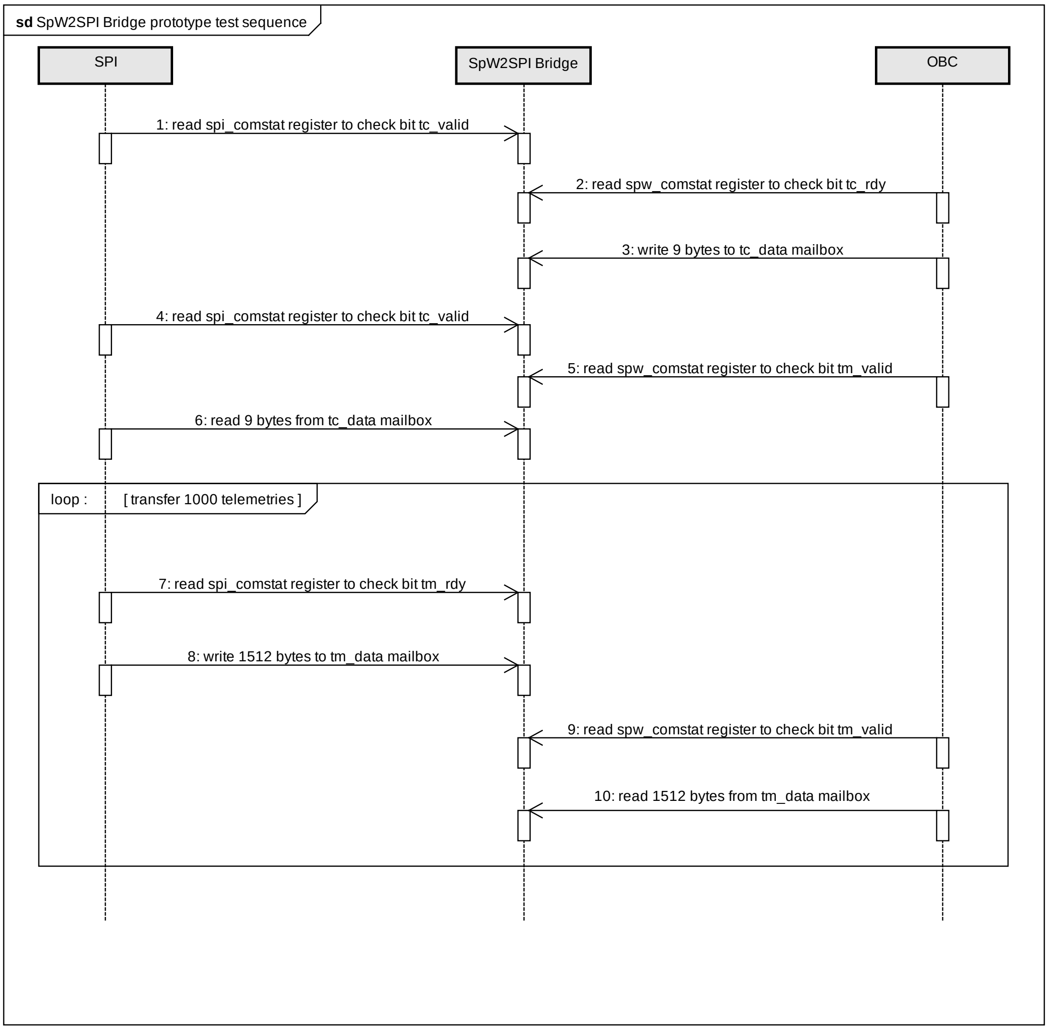
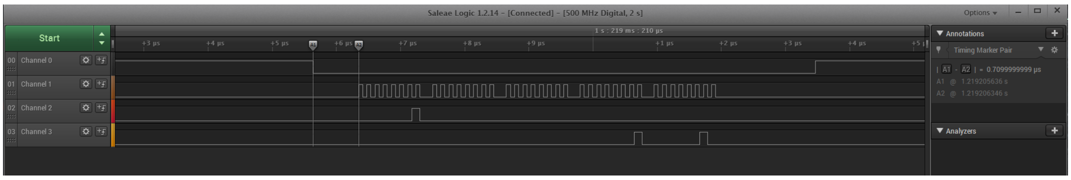

sd SpW2SPI Bridge prototype test sequence
SPI
SpW2SPI Bridge
OBC
1: read spi_comstat register to check bit tc_valid
2: read spw_comstat register to check bit tc_rdy
3: write 9 bytes to tc_data mailbox
4: read spi_comstat register to check bit tc_valid
5: read spw_comstat register to check bit tm_valid
6: read 9 bytes from tc_data mailbox
loop :
[transfer 1000 telemetries]
7: read spi_comstat register to check bit tm_rdy
8: write 1512 bytes to tm_data mailbox
9: read spw_comstat register to check bit tm_valid
10: read 1512 bytes from tm_data mailbox

Figure 8.8: Microsemi ProASIC3E Starter Kit prototype test sequence



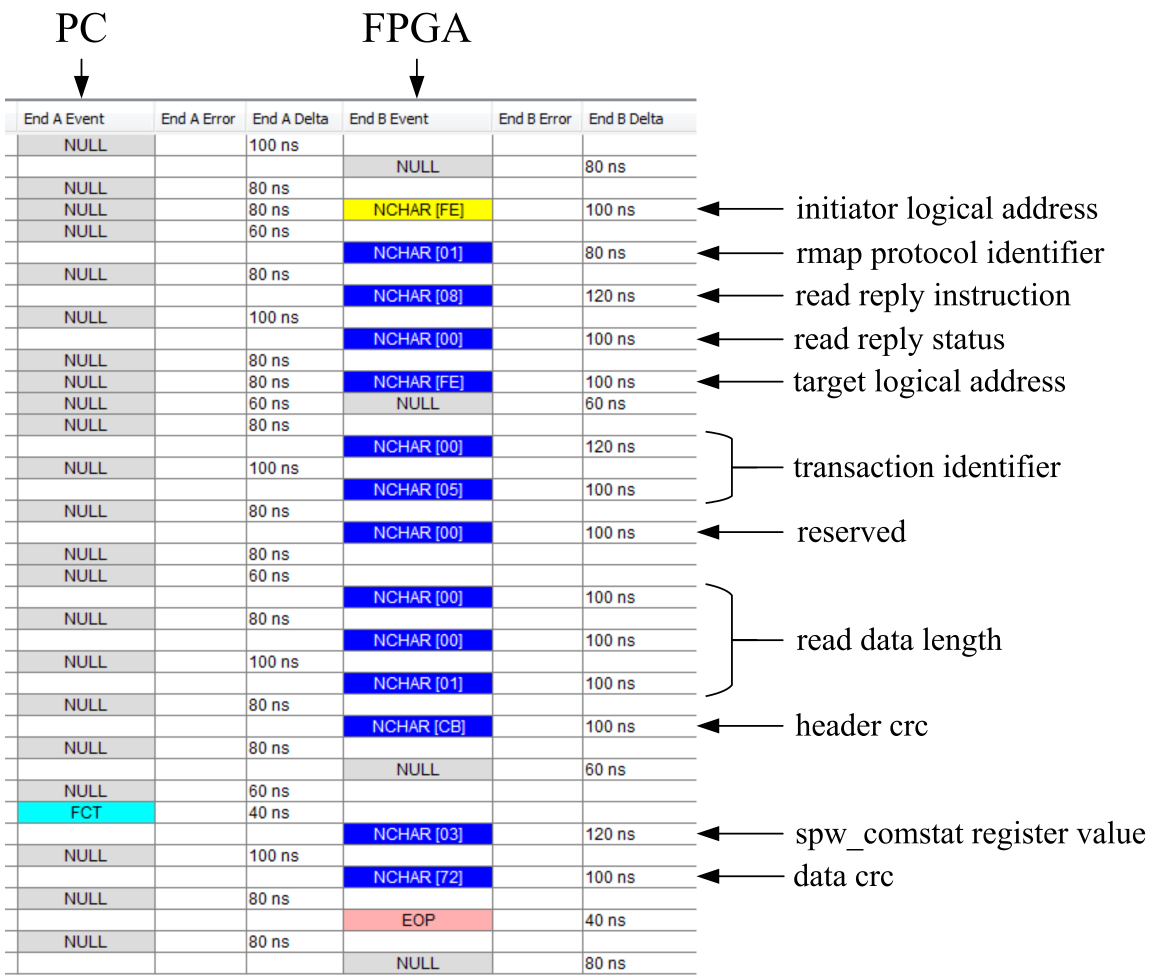
Channel_0 `n_cs`, or an active low chip select signal;

Channel_1 `sclk`, or a serial clock signal;

Channel_2 `mosi`, or a master-out slave-in signal;

Channel_3 `miso`, or a master-in slave-out signal.

Figure 8.9: Microsemi ProASIC3E Starter Kit prototype SPI status register reading



End A Event	End A Error	End A Delta	End B Event	End B Error	End B Delta
NULL		100 ns			
			NULL		80 ns
NULL		80 ns			
NULL		80 ns	NCHAR [FE]		100 ns
NULL		60 ns			
			NCHAR [01]		80 ns
NULL		80 ns			
			NCHAR [08]		120 ns
NULL		100 ns			
			NCHAR [00]		100 ns
NULL		80 ns			
NULL		80 ns	NCHAR [FE]		100 ns
NULL		60 ns	NULL		60 ns
NULL		80 ns			
			NCHAR [00]		120 ns
NULL		100 ns			
			NCHAR [05]		100 ns
NULL		80 ns			
			NCHAR [00]		100 ns
NULL		80 ns			
NULL		60 ns			
			NCHAR [00]		100 ns
NULL		80 ns			
			NCHAR [00]		100 ns
NULL		100 ns			
			NCHAR [01]		100 ns
NULL		80 ns			
			NCHAR [CB]		100 ns
NULL		80 ns			
			NULL		60 ns
NULL		60 ns			
FCT		40 ns			
			NCHAR [03]		120 ns
NULL		100 ns			
			NCHAR [72]		100 ns
NULL		80 ns			
			EOP		40 ns
NULL		80 ns			
			NULL		80 ns

Figure 8.10: Microsemi ProASIC3E Starter Kit prototype SpW RMAP status register reading

Conclusions

The SpaceWire to SPI bridge developed during this thesis involves a number of important contributions specifically designed and implemented for this work, including:

- SPI slave IP core;
- SpaceWire codec IP core;
- SpaceWire RMAP target IP core;
- SpaceWire to SPI bridge IP core;

Despite having multiple contributions, this work as a whole offers a lean design solution that includes only the necessary hardware components to provide the required functionality. As such, no processor involving architecture was used during this work, and thus the delivered solution does not contain any software parts. Even though the usage of software could facilitate rapid prototyping of the requested functionality, it would also increase the complexity of the overall design, hence negatively impacting its reliability. Additionally, the increased complexity would require more documentation, more analysis and more testing and verification. These listed activities are especially critical in terms of a space project. Therefore, the benefits of having software in this kind of project would not have outweighed the complexity it would have brought about.

Based on the discussion regarding the software worthlessness in the scope of this work, it was an interesting finding to realize that a fast solution, even if it is robust and easily scalable, may not be the best solution for the problem at hand. Compared to a university with its academical interests an industry dictates its own rules, and the latter holds especially in case of space industry. Therefore, a solution that seems sub-optimal from one point of view, may actually be absolutely justified from another point of view.

In spite of being quite a challenge, the implementation of a SpaceWire codec from scratch proved to be very rewarding in terms of digital design knowledge and experience. Moreover, there is no better way to learn a communication protocol, such as SpaceWire, than to implement it yourself in hardware. Needless to say that back then, in the beginning of the project, it was a fair risk to take up such a demanding task, but the mentioned positive aspects took priority over the risks.

Due to the thorough documentation of the SpaceWire to SPI bridge interfaces presented in Sections 4.2, 5.3 and 7.3 the outcome of this work can be applied to other projects as well, provided they require this type of communication bridge functionality. Moreover, the fact that there is no such off-the-shelf product on the market makes this work unique for potential applications.

Future perspectives of this thesis could include design options where multiple SpaceWire nodes wish to communicate with multiple microcontrollers, or one microcontroller using multiple SPI links to improve the data throughput. Such options could offer quite an interesting challenge to be implemented again without any software support. One possible way to tackle this problem could be the application of the network-on-chip (NoC) architecture [33].

Appendix A

Essential RTL Design Practices Used

A.1 Reset Synchronizer

Guided by SNUG Boston 2003 paper [34] the current design uses a reset synchronizer circuit. Figure A.1 presents a block diagram to illustrate the concept behind this design practice.

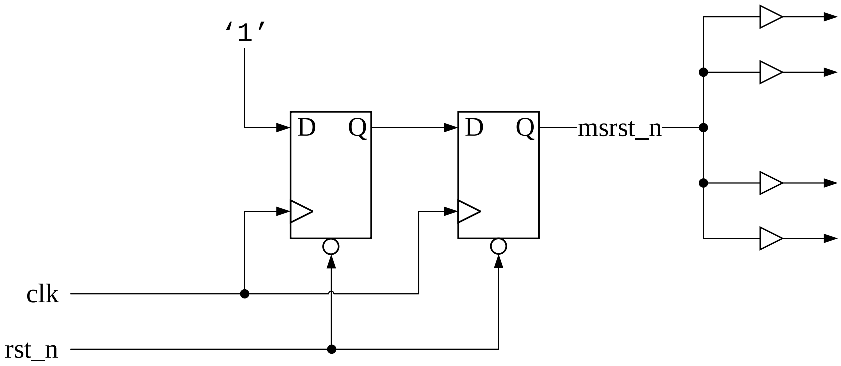


Figure A.1: Reset synchronizer block diagram

This practice instructs the system reset to be asserted asynchronously, but deasserted synchronously. The above-mentioned paper explains the importance of adhering to this reset manipulation sequence. However, from the perspective of the current work it is very important to note that the system reset will be deasserted with a delay of two clock cycles. This feature imposes certain constraints to some parts of the design process, e.g. the definition of the SPI command interface as presented in Section 5.3.

A.2 Two-Process Design Method

In order to improve code readability the two-process method proposed by Jiri Gaisler [35] was first studied and then successfully applied during this work. Indeed, by increasing the abstraction level of the VHDL code it was possible to improve not only the readability of the code, but also its maintainability which in its turn facilitated the overall development process. Listing A.1 demonstrates the application of this method on a simple 8-bit counter. Even though this example is very primitive, the fact that the whole GRLIB, including the space-certified LEON3 soft-core processor, is written using this VHDL coding style shows that the proposed method is indeed very beneficial and viable.

It should be mentioned though that the proposed method may not be quite beginner-friendly as it requires the designer to have a decent experience in digital logic development using VHDL. Even though the resulting code feels much like software, it is still hardware description, and the failure to understand that will eventually lead to bad synthesis results.

Listing A.1 Two-process VHDL design method applied to simple 8-bit counter

```
-----------------------------------------
-- Package for 8-bit counter component
-----------------------------------------
library ieee; use ieee.std_logic_1164.all;

package counter_pkg is
  type counter_in_type is record
    ld_en    : std_logic;
    cnt_en   : std_logic;
    ld_data  : natural range 0 to 255;
  end record counter_in_type;

  type counter_out_type is record
    cnt_data  : natural range 0 to 255;
    zero      : std_logic;
  end record out;

  component counter is port ( clk   : in  std_logic;
                              rstna : in  std_logic; -- active low asynchronous reset
                              inp   : in  counter_in_type;
                              outp  : out counter_out_type );
  end component counter;
end package counter_pkg;

-----------------------------------------
-- 8-bit counter entity
-----------------------------------------
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
use work.counter_pkg.all;

entity counter is port ( clk   : in  std_logic;
                         rstna : in  std_logic; -- active low asynchronous reset
                         inp   : in  counter_in_type;
                         outp  : out counter_out_type );
end entity counter;

architecture two_proc of counter is
  type reg_type is record
    ld_en       : std_logic;
    cnt_en      : std_logic;
    zero        : std_logic;
    cnt_data    : natural range 0 to 255;
  end record reg_type;

  constant REG_RST : reg_type := ( ld_en => ’0’, cnt_en => ’0’, zero => ’0’, cnt_data => 0 );

  signal   r       : reg_type := REG_RST;
  signal   r_nxt   : reg_type;
begin
  --------------------------------------------
  -- Combinational process for counter logic
  --------------------------------------------
  logic_proc : process(r, inp) is
    variable v : reg_type;
  begin
    ---------------------------------
    -- Default assignment
    ---------------------------------
    v := r;

    ---------------------------------
    -- Overriding assignment
    ---------------------------------
    v.ld_en := inp.ld_en; v.cnt_en := inp.cnt_en; v.zero := ’0’;

    ---------------------------------
    -- Main algorithm
    ---------------------------------
    if    r.cnt_en   = ’1’ then v.cnt_data := r.cnt_data + 1; end if;
    if    r.ld_en    = ’1’ then v.cnt_data := inp.ld_data;    end if;
    if    v.cnt_data =  0  then v.zero     := ’1’;            end if;

    ---------------------------------
    -- Drive output
    ---------------------------------
    outp.cnt_data <= r.cnt_data; outp.zero <= r.zero;

    ---------------------------------
    -- Update state register inputs
    ---------------------------------
    r_nxt <= v;
  end process logic_proc;

  ---------------------------------------------
  -- Sequential process for counter state
  ---------------------------------------------
  state_proc : process(clk, rstna) is
  begin
    if    rstna = ’0’ then
          r <= REG_RST;
    elsif rising_edge(clk) then
          r <= r_nxt;
    end if;
  end process state_proc;
end architecture two_proc;
```

Bibliography

[1] ProASIC3 Starter Kit. Microsemi. Retrieved 06.05.2018. [Online]. Available: https://www.microsemi.com/images/soc/products/hardware/ProASIC3_board_2015_3.png

[2] About the Technology Research Programme. European Space Agency. Retrieved 07.05.2018. [Online]. Available: https://www.esa.int/Our_Activities/Space_Engineering_Technology/Shaping_the_Future/About_the_Technology_Research_Programme_TRP

[3] Space Exploration Institute. Space-X. Retrieved 07.05.2018. [Online]. Available: http://www.space-x.ch/

[4] *SpaceWire – Links, nodes, routers and networks*, European Cooperation for Space Standardization Std. ECSS-E-ST-50-12C, jul 2008.

[5] *System engineering general requirements*, European Cooperation for Space Standardization Std. ECSS-E-ST-10C, mar 2009.

[6] *Software*, European Cooperation for Space Standardization Std. ECSS-E-ST-40C, mar 2009.

[7] About ESA IP Cores. European Space Agency. Retrieved 04.05.2018. [Online]. Available: http://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/About_ESA_IP_Cores

[8] SpW-RMAP-Astrium. European Space Agency. Retrieved 04.05.2018. [Online]. Available: https://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/SpW-RMAP-Astrium

[9] SpW-RMAP-Dundee. European Space Agency. Retrieved 04.05.2018. [Online]. Available: https://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/SpW-RMAP-Dundee

[10] D. Juliusson, "Development of a SpaceWire interface in VHDL," Master's thesis, Chalmers University of Technology, 2012.

[11] P. Dillien. And the Winner of Best FPGA of 2016 is... EETimes. Retrieved 04.05.2018. [Online]. Available: https://www.eetimes.com/author.asp?section_id=36&doc_id=1331443

[12] RT ProASIC3. Microsemi. Retrieved 06.05.2018. [Online]. Available: https://www.microsemi.com/product-directory/rad-tolerant-fpgas/1696-rt-proasic3

[13] *VA10820 ARM Cortex-M0 MCU Datasheet*, Vorago Datasheet VA10820, Rev. 1.2.

[14] LEON3 Processor. Cobham Gaisler AB. Retrieved 04.05.2018. [Online]. Available: https://www.gaisler.com/index.php/products/processors/leon3

[15] AMBA Specifications. ARM. Retrieved 04.05.2018. [Online]. Available: https://www.arm.com/products/system-ip/amba-specifications

[16] Libero SoC Design Software. Microsemi. Retrieved 04.05.2018. [Online]. Available: https://www.microsemi.com/product-directory/design-resources/1750-libero-soc

[17] G. Dimitrakopoulos, A. Psarras, and I. Seitanidis, *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*. Springer Science+Business Media, 2015.

[18] *SPI Block Guide*, Motorola, Inc. Document Number S12SPIV3/D, Rev. 03.06, feb 2003.

[19] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*, 3rd ed. Kluwer Academic Publishers, 2002.

[20] C. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," in *Synopsys Users Group Conference*, San Jose, USA, 2002.

[21] *Implementing SpaceWire Clock and Data Recovery in RTG4 FPGAs*, Microsemi Application Note AC444, Rev. 3.0.

[22] P. Walker and B. Cook, "Spacewire: Key principles brought out from 40 year history," in *AIAA/USU Conference on Small Satellites*, North Logan, USA, aug 2006.

[23] S. Huq and J. Goldie, *An Overview of LVDS Technology*, National Semiconductor Application Note 971, jul 1998.

[24] A. Baklezos, C. Nikolopoulos, C. Capsalis, and S. Tsatalas, "Effect of LVDS link speed and pattern length on spectrum measurements of a Spacewire harness," in *2017 International Workshop on Antenna Technology: Small Antennas, Innovative Structures, and Applications (iWAT)*, Athens, Greece, mar 2017.

[25] C. McClements, S. Parkes, and A. Leon, "The Spacewire CODEC," in *International SpaceWire Seminar (ISWS 2003)*, Noordwijk, Netherlands, nov 2003.

[26] B. Cook and P. Walker, "SpaceWire on FPGA - Challenges and Solutions," in *DASIA Conference*, Majorca, Spain, may 2008.

[27] *SpaceWire - Remote memory access protocol*, European Cooperation for Space Standardization Std. ECSS-E-ST-50-52C, feb 2010.

[28] *ProASIC3E Flash Family FPGAs with Optional Soft ARM Support*, Microsemi Datasheet DS0098, Rev. 15.

[29] *ProASIC3/E Starter Kit*, Microsemi User Guide UG0048, Rev. 5.1.

[30] *MultiView Navigator for Libero SoC v11.8*, Microsemi User Guide.

[31] Cortex-M1-enabled ProASIC3L Development Kit. Microsemi. Retrieved 06.05.2018. [Online]. Available: https://www.microsemi.com/existing-parts/parts/143979

[32] *Synplify Pro for Microsemi Edition*, Synopsys FPGA Synthesis, feb 2013.

[33] “A comparison of Network-on-Chip and Busses,” white paper, Arteris, 2005.

[34] C. Cummings, D. Mills, and S. Golson, “Asynchronous & Synchronous Reset Design Techniques - Part Deux,” in *Synopsys Users Group Conference*, Boston, USA, 2003.

[35] J. Gaisler. A structured VHDL design method. Gaisler Research. Retrieved 28.04.2018. [Online]. Available: https://www.gaisler.com/doc/vhdl2proc.pdf