Deutsches Zentrum
DLR für Luft- und Raumfahrt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg
Campus Friedrichshafen

Clock Domain Crossing Analysis of a SpaceWire IP-Core

Bachelor Thesis

Degree course Aerospace Engineering

Field of study Aerospace Elektronics

Duale Hochschule Baden-Württemberg Ravensburg, Campus Friedrichshafen

by

Johann Lossin

Date:	September 23, 2024
Working Period:	01.07.2024 - 23.09.2024
Matriculationnumber:	5113350
Course:	TLE
Partner Company:	DLR e.V.
Referent Company:	Dr. Kai Borchers
Referent University:	Prof. Philipp Krämer

Erklärung

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 25.07.2018.

Ich versichere hiermit, dass ich meine Bachelorarbeit (bzw. Projektarbeit oder Studienarbeit bzw. Hausarbeit) mit dem Thema:

Clock Domain Crossing Analysis of a SpaceWire IP-Core -

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Bremen, den September 23, 2024

Johann Lossin

Abstract

In recent years, the use of multiple clocks inside a single IC has grown rapidly to boost performance. This change comes with unique challenges regarding the data transfer from one clock domain to another, due to different frequencies colliding. Therefore, it is necessary to ensure that data crosses the clock domain boundary correctly.

The aim of this paper is to find and if possible correct clock domain crossing (CDC) violation inducing parts of an open source SpaceWire IP-core, that is used for developing projects in FPGAs. The IP-core has shown anomalies in a code review, that could cause CDC errors.

The paper first introduces the concept of CDC and describes several synchronizers used to solve the problem. After looking at detection methods, the SpaceWire protocol is described. With this basis, we take a look at the analyzed SpaceWire IP-core and its synchronizers, as well as the Questa CDC tool. To find the CDC error inducing parts, a static CDC analysis is performed with Questa CDC, which is part of the Siemens verification suite. The tool found six violations, which were analyzed manually. Finally, corrections were made to the IP-core where necessary, and one usage restriction was found.

Contents

List of Figures v

List of Tables vii

Acronyms viii

1 Introduction 1

2 Theoretical basis 3

2.1 Field Programmable Gate Array . . . 3
2.1.1 Digital Processing . . . 4
2.1.2 FPGA Structure . . . 6
2.1.3 Example VHDL & Signal Time . . . 8
2.2 Clock Domain Crossing . . . 11
2.2.1 Metastability . . . 11
2.2.2 Synchronizers . . . 14
2.2.2.1 2 Flip-Flop Synchronizer . . . 14
2.2.2.2 Gray code Synchronizer . . . 16
2.2.2.3 Asynchronous FIFO . . . 17
2.2.2.4 Single Stage Synchronizer . . . 18
2.2.3 CDC Detection . . . 19
2.2.3.1 Static Verification . . . 19
2.2.3.2 Simulation Based Verification . . . 20
2.3 Space Wire . . . 21
2.3.1 SpaceWire Network . . . 22
2.3.2 Structure . . . 24

3 Used IP-Core & Tool **26**
3.1 The SpaceWire ip-Core . . . 26
3.1.1 2FF IP-core Synchronizer . . . 28
3.1.2 FIFO IP-core Synchronizer . . . 29
3.2 Questa CDC Tool . . . 31

4 CDC Violations & Recovery **33**
4.1 Setup . . . 34
4.2 Single Source Reconvergence of synchronizers . . . 36
4.2.1 FIFO Violation . . . 37
4.2.2 Reset Violation . . . 39
4.2.3 Pointer Crossing bitcnt & headptr . . . 41
4.3 Multi-bit signal across clock domain boundary . . . 43
4.4 FIFO pointer mismatch . . . 44
4.5 Verification . . . 45

5 Discussion **47**

6 Conclusion **49**
6.1 Summary . . . 49
6.2 Outlook . . . 50

Bibliography **51**

Appendix **54**
A.1 Full VHDL example code . . . 54
A.2 Relevant IP-core codes . . . 57
A.3 makefile and filelist . . . 59
A.4 Overview of the used AI-based Tools . . . 62

List of Figures

2.1 Computing in Time . . . 4
2.2 Computing in Space . . . 4
2.3 Basic FPGA Architecture [3] . . . 6
2.4 VHDL example Signals . . . 10
2.5 Types of Signal changes from low to high . . . 12
2.6 Setup Time Violation Waveform . . . 13
2.7 Hold Time Violation Waveform . . . 13
2.8 2FF-Synchronizer . . . 14
2.9 2FF signal timing . . . 15
2.10 First in First out (FIFO) Synchronizer . . . 17
2.11 Single stage synchronizer receiver clock dynamically changing [12] . . . 18
2.12 SpaceWire differential Signals and clock corollation . . . 21
2.13 SpaceWire Packet Format [16] . . . 22
2.14 SpaceWire Path Addressing [16] . . . 23
2.15 SpaceWire Logical Addressing [16] . . . 24
2.16 SpaceWire Token [16] . . . 25
2.17 Nominal SpaceWire Clock Domains [17] . . . 25

3.1 Block Diagram of the SpaceWire ip-Core [18] . . . 27

4.1 Detected Errors . . . 35
4.2 Simple Reconvergence Structure . . . 36
4.3 Reconvergence Signals . . . 36
4.4 First Violation's Logic Circuit . . . 38
4.5 SpaceWire Null Message[16] . . . 39
4.6 Second Violation/ reset violation . . . 40
4.7 Pointer Error Diagram . . . 41

4.8 FIFO violation 44
4.9 Induced Violation 46

List of Tables

2.1 Comparison of digital Technologies [3] 5
2.2 FPGA Types [3] [6] 8
2.3 Gray code and binary counting 16

4.1 Gray code Counter Incrementation 42

A.1 Usage of AI-Tools 62

Acronyms

ASIC .... Application Specific Integrated Circuit
CDC .... Clock Domain Crossing
CIS .... Computing in Space
CIT .... Computing in Time
DLR .... Deutsches Zentrum für Luft- & Raumfahrt e.V.
EMC .... Electromagnetic compatibility
ESA .... European Space Agency
FF .... Flip Flop
FIFO .... First in First out
FPGA .... Field Programmabel Gate Array
GUI .... Graphical User Interface
HDL .... Hardware Discribtion Language
IC .... Integrated Circuit
IP-Core .... Intelectual Property-Core
JAXA .... Japan Aerospace Exploration Agency
Mbps .... Mega bits per second
MTBF .... Mean Time Between Failure
NASA .... National Aeronautics and Space Administration
PCB .... Printed Circuit Board
RTL .... Register Transfer Level
SoC .... System-on-a-Chip

VHDL .... Very High Speed Hardware Description Language

1 Introduction

As the demand for computational capability in non-terrestrial applications rises, the need to implement efficient and reliable solutions has to be met. Especially in the German Aerospace Agency(DLR), which is at the forefront of new space technologies. One option to improve the efficiency is to use multiple clocks inside an integrated circuit (IC) (typically a field programmable gate array (FPGA) for custom logic in low volume production or prototyping). This is a standard practice for highly integrated circuits like smartphones, but not as common for space applications. The different clocks control semiconductor areas in the IC, so-called clock domains.

The use of multiple clocks domains comes with a set of unique challenges, foremost the problem of meta stability where a signal crosses over from one to another clock domain and is not able to reach the defined low or high states in time. This can cause timing issues, lost data and undefined signal states throughout the logical circuits. These issues can occur irregular, making it very difficult to detect them reliably in hardware.

Reliable communication is an important part of most systems. An example is SpaceWire which is a well established interface and bus protocol standard, used by many universities, space agencies, and industry for a multitude of projects. It is defined by the european comity for space standardization in the standard ECSS-E-ST-50-12C.

SpaceWire is also used by DLR, in this case in the form of an IP-core, a form of digital logic that is relatively easy to implement in an FPGA. The IP-core has shown signs of irregularities in the past, that could be caused by clock domain crossing (CDC) issues. Therefore, it should be analyzed and corrections be implemented, if possible. This can reduce data errors, that appear during development of new systems and technologies.

The Analysis is performed with the Questa CDC tool from Siemens, that is able to find potential CDC violations. The violations are then manually checked. If they pose no CDC risk they can safely be ignored or be fixed when they are valid.

The Aim of this bachelor thesis is to analyse the SpaceWire IP-core in regard to CDC related problems and correct them if possible. With the end goal of advising if is can be used reliably.

We begin by understanding the functionality of digital logic and VHDL, the language used by the IP core. We also look at the basics of CDC and metastability, as well as the SpaceWire protocol. We continue with the specific SpaceWire IP-core and its synchronizers, which are used to transmit signals across the clock domain boundary. The next step is to examine at the tool. Finally, we complete the analysis by performing a static CDC analysis on the IP-core. This includes setup, running the tool, manually checking the results, and finally implementing a correction in the IP-core if necessary.

2 Theoretical basis

In the realm of modern digital systems, Field-Programmable Gate Arrays (FPGAs) and robust communication protocols like SpaceWire are fundamental to advancing technological capabilities across various applications. This chapter provides an overview of the theoretical basics essential for understanding and leveraging these technologies. It begins with an exploration of digital processing principles, setting the stage for an examination of the structure and working principles of FPGAs. Further, it delves into critical concepts such as clock domain crossing and metastability, discussing the challenges they present and the synchronizer techniques used to mitigate these issues. The chapter culminates with an introduction to SpaceWire, a high-speed communication protocol designed for space applications, highlighting its key features and operating principle. Through this, readers will gain a foundation in both FPGA technology and the specialized communication protocol SpaceWire.

2.1 Field Programmable Gate Array

Field-Programmable Gate Arrays (FPGAs) are powerful and versatile digital processing devices that offer a unique blend of hardware performance and flexibility. Resulting in a wide field of applications, from telecommunications to automotive systems, consumer electronics, and aerospace. This chapter introduces the core concepts and practical applications of FPGAs, beginning with digital processing followed by the structure of FPGAs and their technology. Finally, a short VHDL example is shown to illustrate how FPGA designs can be defined.

2.1.1 Digital Processing

For any kind of data processing, some sort of computational hardware is needed, with a certain processing structure. Those can be broadly categorized in computing in time (CIT) and computing in space (CIS). Computing in time is commonly used in generic computer processors and microcomputers, in which operations take place one after the other, as illustrated in Figure 2.1, with every operation requiring a fixed amount of time. Therefore, to perform complex calculations, the key resource is the time it requires. But this approach enables a highly flexible and cost-effective solution, due to the relatively few semiconductor components required to build those. Although they are highly flexible, there exist specialized processors with corresponding optimized arithmetic logic unit, depending on the task. For example, digital signal processors for high performance processing or microprocessors for low power applications. [1]

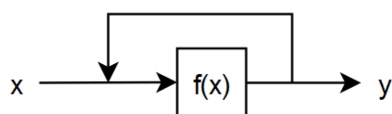

Figure 2.1: Computing in Time

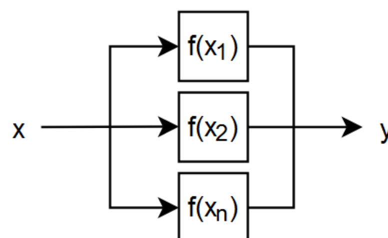

Figure 2.2: Computing in Space

Computing in space on the other hand utilizes more semiconductor structures to perform multiple calculations simultaneously, as depicted in Figure 2.2, while being not as adaptable to rapidly changing tasks. This simultaneous computation is done by using parallel logic or computing structures. These can often only perform a single function. Common example technologies for simultaneous computing are ASICs (Application Specific Integrated Circuits) and FPGAs (Field Programmable Gate Array).

ASICs are logic units that are tailored for a specific problem, where the function is directly manufactured in silicone. Therefore the functionality can not be changed after production, as it is the case for processors where only the software has to be exchanged. These application specific solutions allow for a very fast and energy efficient operation. But it is expensive in production and has a long development time, due to the high development effort and specific semiconductor manufacturing process where a unique

lithography mask is necessary. This makes them optimal for high quantity production. [2]

FPGAs, on the other hand, retain the capacity for simultaneous computation while also exhibiting reconfigurability, rendering them readily adaptable to a diverse array of applications. This feature renders FPGAs particularly suited to small-scale production and prototyping. However, they exhibit a higher power consumption and a limited number of logic units in comparison to ASICs. The main attributes of each technology are illustrated in Table 2.1 for comparison.

	Strength	**Weaknesses**
ASIC	fully custom high performance low power	design effort inflexibility high cost
FPGA	flexibility half custom cost	power limited resources
Processor	low design effort high flexibility cost	fixed architecture lower performance power

Table 2.1: Comparison of digital Technologies [3]

It is important to note that a variety of combinations of both CIS and CIT exist. Notable examples include multiprocessors, system on a chip (SoC), and vector processors. A multiprocessor is a system that employs multiple CIT processors in parallel to perform computations. The use of parallel processing allows for an increase in processing capacity and a reduction in overall computation time, depending upon the parallelization capabilities of the software. In contrast, SoCs employ one or multiple processors in conjunction with an FPGA structure on a single IC. This dual utilization enables the FPGA structure to be utilized for time consuming calculations, while the processor is reserved for sequential and control operations. Additionally, vector processors are employed when the same calculation is performed on a vast quantity of data. This vector processor employs a conventional processor architecture with a single control structure, but with multiple processing structures that perform the same operation parallel on multiple data sets. [4]

It has to be noted that it is possible to implement a processing core in an FPGA, resulting in a so called softcore. Although this softcore works like a regular processing unit, due to the not optimized silicone structure, it is often slower and more energy inefficient compared to a regular processor.

2.1.2 FPGA Structure

As we discussed in Section 2.1.1, an FPGA is a digital processing device, that is reconfigurable and is classified as CIS, with a unique structure as shown in Figure 2.3.

Its fundamental components include logic blocks and switching cells. While switching cells serve the function of a customizable routing network, they can also be interpreted as an intersection or switches, in that they connect the building blocks. In order to form a customizable routing network. They form together with the logic blocks the basis for the reconfigurability. The logic blocks are the basis for the FPGA, by being reconfigurable or field programmable. The blocks consist of logic gates, which is why the FPGA designs are called gateware, rather than software. Lastly, the blocks are arranged in an array as shown in Figure 2.3.

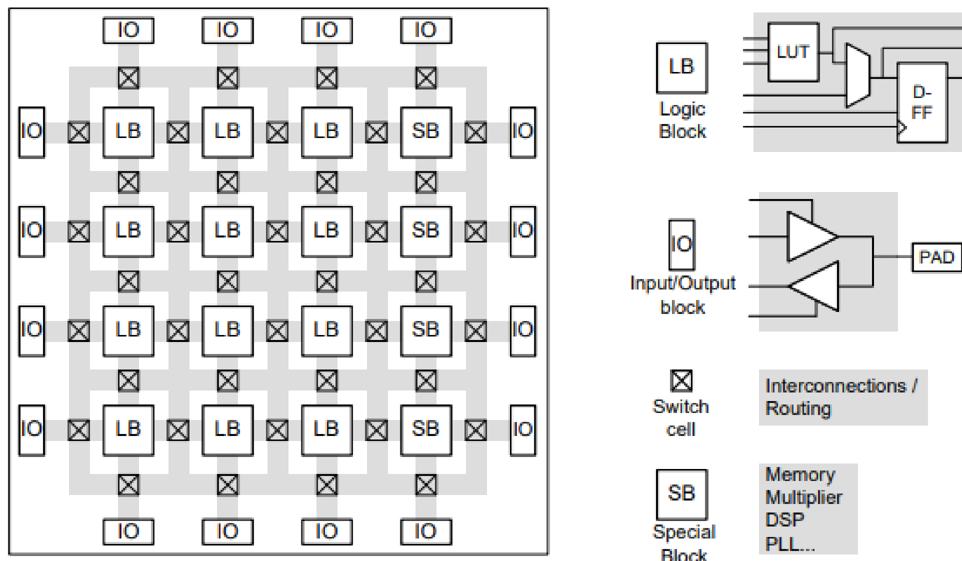

Figure 2.3: Basic FPGA Architecture [3]

The logic blocks consist of a D-flip-lop (FF) and programmable memory within a look-up table (LUT), allowing the FPGA to be configured with any desired binary logic. [5] The I/O-blocks are the connection between the internal FPGA structure and the pins on the package to the circuit board. The final blocks, are special blocks that provide a fixed function in silicone, with a specific commonly used function to enhance efficiency. For instance, they include memory registers, multiplication accumulation, addition, and numerous other operations.[5] This method allows to save space and semiconductor elements.

To understand how an FPGA operates, it is important to know the difference between combinatorial and sequential processes. In combinatorial processes only basic binary logic is used to perform computations where the time necessary directly correlates with the amount of LUTs the signal has to travel through and their lead time. Sequential logic utilizes memory and can therefore store information. Thus, it needs clocks to synchronize the access, prevent insufficient set time for the memory elements and to avoid race conditions, that cause inconsistent result. Those memory elements include not just register but flip-flops also serve as memory blocks, which save a signal state. This saving allows developers to use elements like if- and else-statements, counters, shift registers, edge detection and many more. With that a sequential operation can be introduced, to build control sequences, while still fully using the parallel computing. [4]

Gateware employs its own programming language, a hardware description language (HDL), which is utilized to describe the functionality of silicon gates or the configuration at the register transfer level (RTL). The most commonly used languages are VHDL and Verilog. A specific function can be described as an IP-core. This IP-core can be conceptualized as an IC on a circuit board, wherein the circuit board serves as the FPGA. The IP-core comprises of input signals, internal logic, and output signals. For FPGAs, the process of converting the code and loading it to the device differs from that of software. The VHDL code is synthesized to a netlist, which is then routed for the specific FPGA. Finally, the FPGA is configured by loading a resulting bitfile that contains all required information.. [1]

There are three main FPGA technologies that are differentiated from the corresponding configuration memory technology. These are Antifuse, where a small fuse is melted

	Antifuse	**SRAM**	**Flash**
Reprogrammable	No	Yes	Yes
Programming Voltage	High	Vcc	High
Volatile	No	Yes	No
Total Ionizing Dose Tolerant	Immune	High	Low
Single Event Upset Tolerant	Immune	Low	Medium
Power Consumption	Low	Medium	Medium

Table 2.2: FPGA Types [3] [6]

to form a permanent memory entry, and two semiconductor memory types, SRAM and Flash. The different technologies have their advantages and disadvantages, especially in the space environment. The technologies are listed in Table 2.2 with their key attributes in comparison. At first glance, antifuse is the obvious choice due to low power and radiation tolerance. However, it has a detrimental weakness. It is only possible to program the device once, which is suboptimal for prototyping and impossible for in-orbit changes or updates. In contrast, SRAM has the advantage of only requiring a supply voltage to accomplish this. [1]

2.1.3 Example VHDL & Signal Time

In order to illustrate the functionality and signal timeline, a brief example of a full adder will be employed. As the name implies, the full adder performs the operation of adding up the inputs. The example comprises of three inputs, a carry (cary) and two one-bit signals (X(0) and X(1)), as well as one two-bit long output (Y), which represents the result of the sum of the three inputs in binary code.

In order to continue the analogy of electronics or an IC for VHDL code, it is necessary to divide the code into two distinct parts. The entity declaration and, consequently, the ports in lines one to nine of the VHDL code 2.1 correspond to the pads of a package and are utilized to connect VHDL constructs and to facilitate the transfer of signals in and out of the FPGA.

It is possible to let multiple VHDL constructs work together to build a larger functionality, like multiple ICs and other components build a complex circuit. How those constructs are connected is defined at the beginning of the architecture portion. The

architectural Section incorporates a specific design, as well as any internal signals. Additionally, the top of the code is where the used packages and other VHDL constructs are being declared, that are to be utilized.

VHDL is a low-level language that primarily employs binary operations, were registers and signals are declared. The summation is therefore performed via bit logic, as illustrated in VHDL-code 2.1. In this context, safe(0) represents the least significant bit and has the numerical value of one. This safe(0) is high, if the number of input signals that are high is uneven. In contrast, safe(1) numerical value is two and is high when at least two input signals are high. If both are high, the result would be a binary three (2b11).

Code 2.1: VHDL:example Full-adder

```
entity example is --FUll_adder
  port(
    cary    : in  std_ulogic;
    X       : in  std_ulogic_vector (1 downto 0);
    Y       : out std_ulogic_vector (1 downto 0);
    clk_i   : in  std_ulogic;
    rst_i   : in  std_ulogic
  );
end entity example;

architecture rtl1 of example is
  signal safe : std_ulogic_vector (1 downto 0);
begin
  p1: process(clk_i)  begin
    if(rising_edge(clk_i)) then
      if(rst_i = '1') then
        Y   <= "00";
        safe<= "00";
      else
        safe(0)<= cary xor (X(0) xor X(1));
        safe(1)<= (X(1) and X(0)) or ((X(0) xor X(1)) and cary);
        Y<=safe;
      end if;
    end if;
  end process p1;
end architecture rtl1;
```

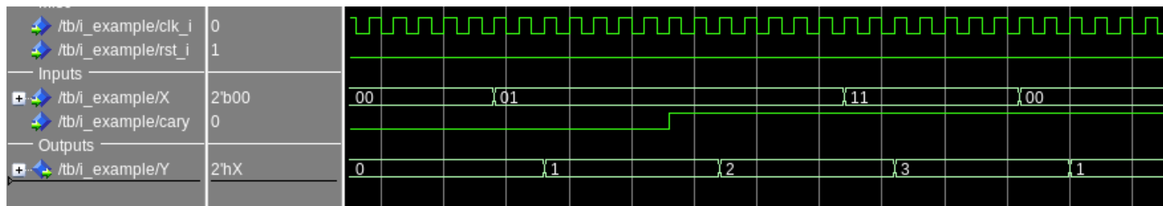

Figure 2.4: VHDL example Signals

The signal "safe" is employed to illustrate the delay in signal propagation due to the flip-flops and the dependency on the clock signal. It could be replaced with Y for efficiency. This delay can be observed in Figure 2.4, where two clock cycles are required for completion of the process. One cycle for safe to change and one additional one for the change of Y. This phenomenon is typical of FPGAs and must be taken into account during design, especially for functions that depend on one another. Although delays and, consequently, computing in time is present to a certain degree, CIS greatly surpasses this, as evidenced by the use of multiple logic operations in a single clock cycle. Lines twenty, and twenty one are performed quasi-simultaneously, and each line has multiple operations. If this were done via a simple microprocessor, each operation would be performed consecutively.

2.2 Clock Domain Crossing

Clocks are widely used in digital computing. FPGAs use them as well to indicate a simultaneous signal transfer or flip-flop change, to perform sequential calculations. In recent years, the amount of clocks inside a FPGA-design has increased to accommodate the increased desire for energy efficiency and speed. Currently, the average design uses three to four clocks, with some even going up to fifty. Each of these clocks form a clock domain where they actively control the logic.[7]

These individual domains inside a chip, need to interchange data to work together, as one system. These clocks often use different frequencies. Furthermore, uncertainties like clock-jitter or asynchronous clocks with a phase discrepancy influence the ability to exchange data. [8] While crossing the clock domain boundary with a mismatched clock edge, the signal can lose its integrity and becomes metastable, resulting in lost data or erroneous behavior that propagates through the logic circuits.

2.2.1 Metastability

In the digital environment one tends to think only in binary values (0 & 1) but in reality those values only represent a certain voltage potential inside semiconductor components. The voltage range between two potentials can not be handled properly by the components and can cause invalid or faulty signals that propagate through the system. Additionally, it is easy to assume signal changes happening instantly, even though in reality that is not the case. Signal lines have a certain impedance due to charging and recharging the magnetic and electric field. The transistors who build the logic have a certain capacity which stores energy and therefore delays the change of the voltage potential. These factors delay the voltage switch. [4] These characteristic views are illustrated by the signals in the Figure 2.5. The metastable error is the worst case scenario where the receiving flip-flop was not able to charge or discharge long enough in order to switch to the other potential. This scenario can happen if the clock frequency is too high or two or more clock domains interact with each other.

If the different clock domains are not properly aligned during a signal change, it is possible that metastability occurs. Where the signals don't reach the required

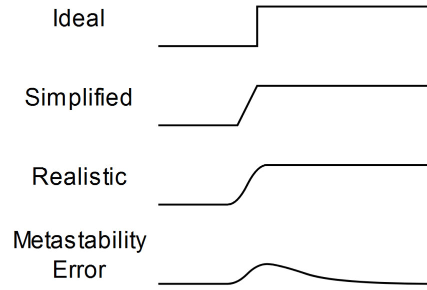

Figure 2.5: Types of Signal changes from low to high

potential for a high value before sampling by the receiver clock, what can be seen in Figure 2.5. Therefore, the signal can hover in an undefined state before settling randomly to either high or low. [8]

Mean time between failure (MTBF) is an indicator of system reliability. If the value exceeds the expected runtime of the system, it is unlikely that an error will occur during operation. To determine the occurence of metastability, the Formula 2.1 can be used.

$$MTBF = \frac{\exp\left(t_{res}/K1\right)}{K2 * f_{clk} * f_{data}} \tag{2.1}$$

The equation above suggests that the receiver clock frequency (f_{clk}) and input signal frequency (f_{data}) have a significant impact. The resolution time (t_{res}), or the time since a clock edge, also has a significant impact as it indicates the time the signal has to stabilize. The type of FPGA used is also important due to the specifics of how fast the signal can change depending on the structure and material used, represented by $K1$ and $K2$. [9] This means the only practical values to change are the clock frequencies.

Metastability errors can be classified as either setup or hold time violation. Depending on the relative phases of both clock edges. Signals can change their values at a rising clock edge.[10] If the sampling rising clock edge (Clk2) is too close to the transmitting clock edge (Clk1), it is possible that the transmitted signal (S1) wasn't able to change in time. Thereby, the received signal (S2) does not correspond to S1. This constitutes a setup violation.[9] This correlation is visualized in Figure 2.6.

Generally, signals are either set for longer duration or change rapidly, i.e. indicating a state, or rapidly changing due to calculations or bus transmissions. From this, two

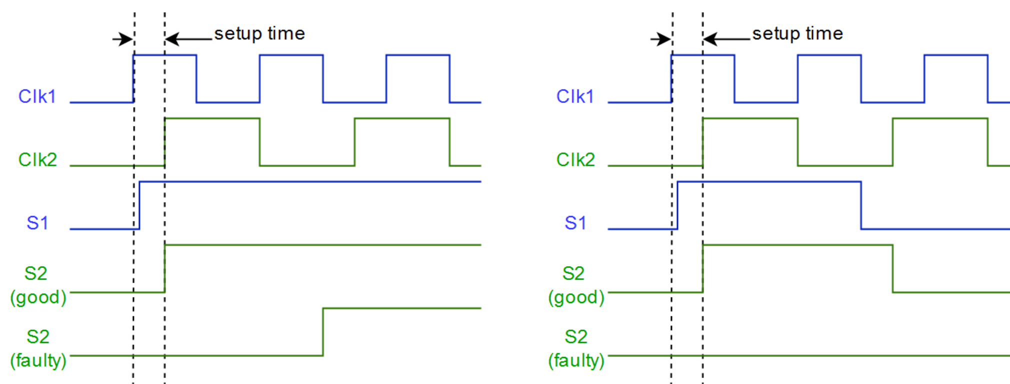

Figure 2.6: Setup Time Violation Waveform

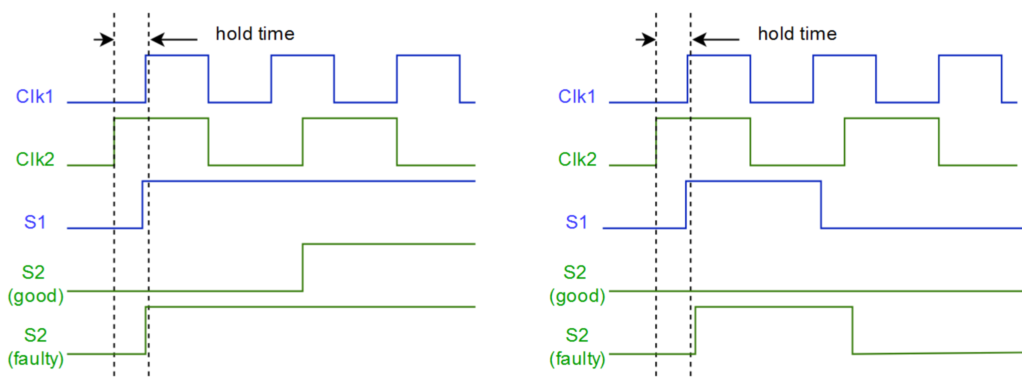

Figure 2.7: Hold Time Violation Waveform

distinct error patterns arise, as depicted in the figures 2.6. The left one depicts a longer duration of signal hold this is relatively stable, because the signals settles at the next clock edge. [10] Nevertheless, this brings uncertainties for developers and can be especially problematic if the signal is combined with others. The right Figure shows a rapidly changing signal transmission, in which whole bits can be lost and therefore can cause reliability issues or even functional failure.

Hold time violations appear when the transmitting signal holds too long high and the receiver incorrectly samples the signal.[10] The arising problems are visualized in Figure 2.7. Unlike setup violations, the sample value is unexpected high instead of low. Even though, it seems less error inducing if the signal is switching too early, it still can cause errors with timing, which is detrimental for communication and real time applications.

2.2.2 Synchronizers

To prevent CDC-errors, synchronizers are used to remove metastability, by creating a crossing where the output has a defined value at all times. This removes undefined values between high and low, that would propagate through the logic and cause further problems in the logic. Although that removes part of the concerns, timing uncertainties still remain. Depending on the use case, different synchronizers can be implemented with their respected advantages and disadvantages.

2.2.2.1 2 Flip-Flop Synchronizer

As mentioned before, a crucial part of an FPGA are flip-flops (FF), that serve as memory and beginning and endpoints of signals. By introducing a second receiver flip-flop in series, the risk of metastability is greatly reduced, creating a two flip-flop synchronizer (2FF). [9] While this is a common synchronizer, it still has the ability to go metastable [9]. Therefore, it is used for low data rates.[8]

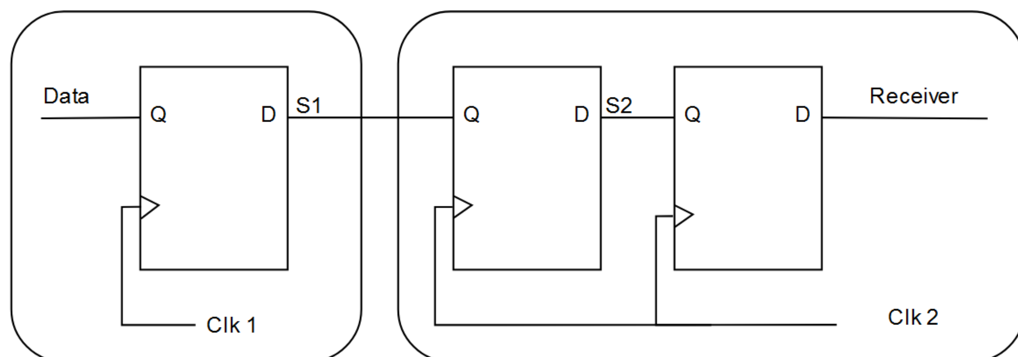

Figure 2.8: 2FF-Synchronizer

After the data signal changes and crosses the clock domain boundary after exiting the first flip-flop, the first receiver flip-flop, samples it with a high metastability change, but the third flip-flop or the second receiver flip-flop, provides a safe output signal. Meanwhile the metastable flip-flop has a full clock cycle to become stable. While the third flip-flop delays the signal. This approach reduces the metastability rate greatly with very low resource consumption.

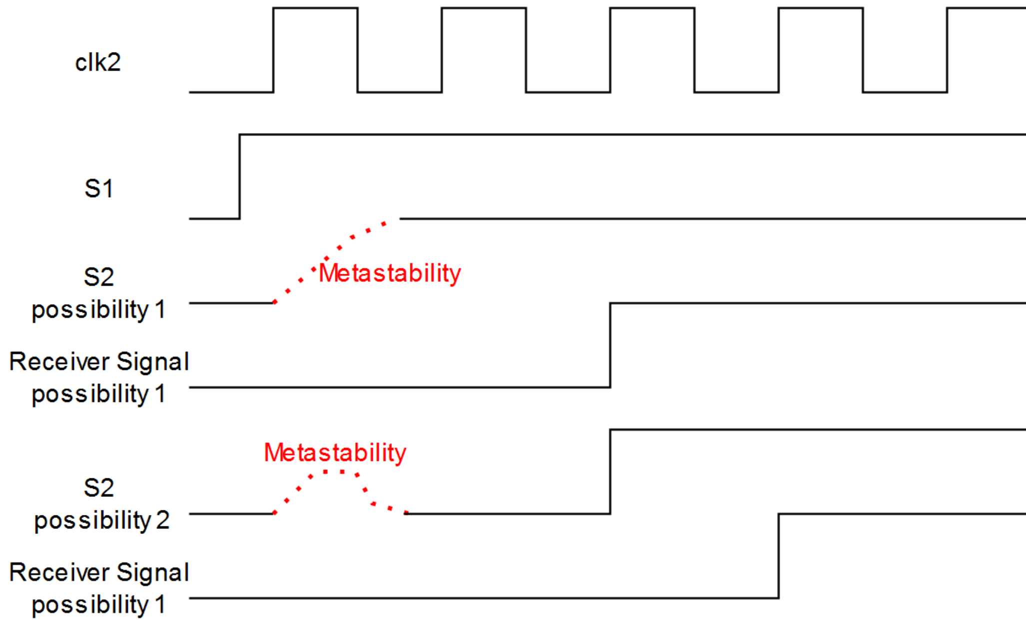

Figure 2.9: 2FF signal timing

To visualize this principle, Figure 2.9 is used. In this Figure, the data signal changes from low to high at the output of the first flip-flop, which is controlled by the first clock. The second flip-flop (2FF) samples the signal (S1), but metastability occurs. Resulting in an undefined state, shown in red. But due to the characteristics of the synchronizer, the state of the outgoing signal is always defined, as illustrated as Receiver Signal. To note is that depending on the type of occurring metastability, the time required to move through the synchronizer varies. As shown through possibility one and two, this attribute also brings some uncertainties in the transition and resulting in an effect that data can not be reliably passed through the clock domain boundary with this type of synchronizer. Therefore this type of synchronizer is only recommended for single bit signals.

It is possible to make the flip-flop synchronizer even more reliable by using a cascade of flip-flops. While the MTBF directly correlates with the number of flip-flops (N) used, see adjusted MTBF equation 2.2. While increasing the occurring signal delay. [11]

$$MTBF = \frac{\exp((N-1) * t_{res}/K1)}{K2 * f_{clk} * f_{data}} \tag{2.2}$$

2.2.2.2 Gray code Synchronizer

If we use a 2FF synchronizer for a multi bit signal, numerous problems arise. For serial transmission the timing of received bits can not be presumed as correct, due to setup and hold time violations, that could change the signal timing at the receiver. If the signals are transmitted in parallel, the question arises as to when all the bits are correct before the logic samples the signals. Or rather, after the different signal paths have reconverged, and when do they change again. [9]

To circumvent this issue while still using 2FF a synchronizer to reduce metastability, is the use of gray code. Gray code is a special way of changing a multi-bit signal. The parallel signals can only change one bit at a time (Hamming distance = 1) with this technique a changed signal can be positively identified after exactly one bit is different compared to the previous signal. Therefore, signal integrity can be assured.

Gray code

$$00 \rightarrow 0X \rightarrow 01 \rightarrow X1 \rightarrow 11 \rightarrow 1X \rightarrow 10 \rightarrow X0 \rightarrow 00$$

Binary

$$00 \rightarrow 0X \rightarrow 01 \rightarrow XX \rightarrow 10 \rightarrow 1X \rightarrow 11 \rightarrow XX \rightarrow 00$$

Table 2.3: Gray code and binary counting

To illustrate the principle of gray code, the Table 2.3 is used. A two bit parallel signal counts upwards until an overflow occurs. The changing bit (X) of the gray code never exceeds the specified one bit, while still allowing all possible values to be reached. The obvious disadvantage of gray code is the long transition time between signals, that have a large amount of different bits (high Hemming distance) and the necessary use of an additional signal to indicate when it should be read, either as an individual signal or as a bit in the gray code multi-bit signal.

2.2.2.3 Asynchronous FIFO

Even though gray code synchronizers are a reliable solution for multi-bit signals, their throughput is rather limited. A common solution is an asynchronous First in First out (FIFO) synchronizer, when a large data rate is needed.[8]

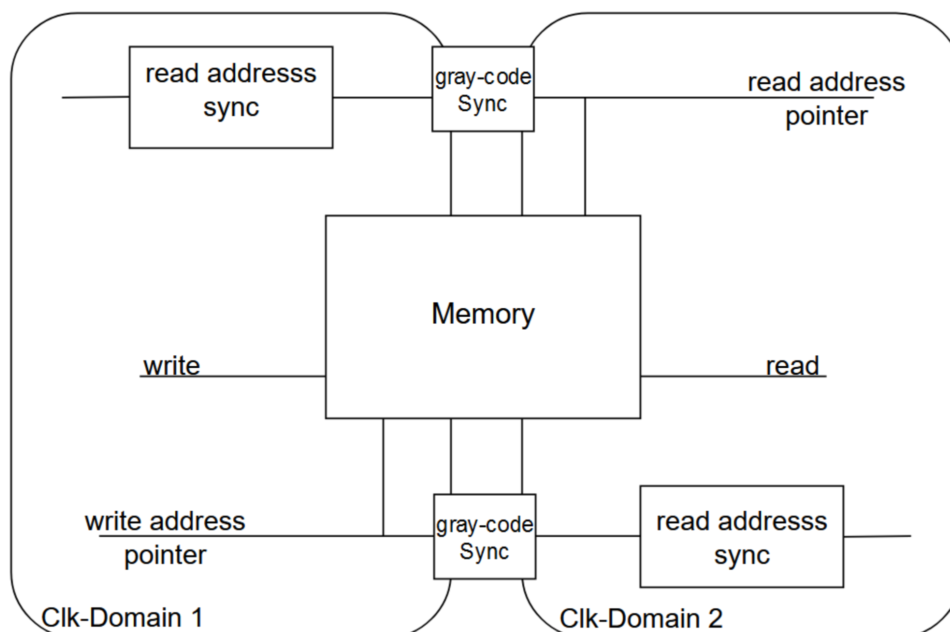

Figure 2.10: First in First out (FIFO) Synchronizer

To achieve the high transfer rate and reliability, both clock domains share a common memory block. A read and write signal and address pointers are used to manage the memory. While the read and write signal is used to control memory operations, the more important part is the address pointers, that indicate which register is being used by the corresponding clock domain to prevent simultaneous access and thus data corruption. During a read or write operation, the affected register is in the corresponding clock domains jurisdiction and is therefore no longer affected by metastability. However, this requires the address pointer to cross the clock domains. This crossing is typically done by using a gray code synchronizer, ensuring the integrity of the address pointer. Large differences in the access pointers can be circumvented by intelligent register usage, like only accessing close registers that the gray code pointer is able to quickly point to. Thus enabling fast communication. [9]

2.2.2.4 Single Stage Synchronizer

The aforementioned synchronizer types have one thing in common, they define signal paths and structures to prevent metastability. An alternative approach would be to detect the relative distance of both clock edges and with that predict the metastability risk and mitigate it.

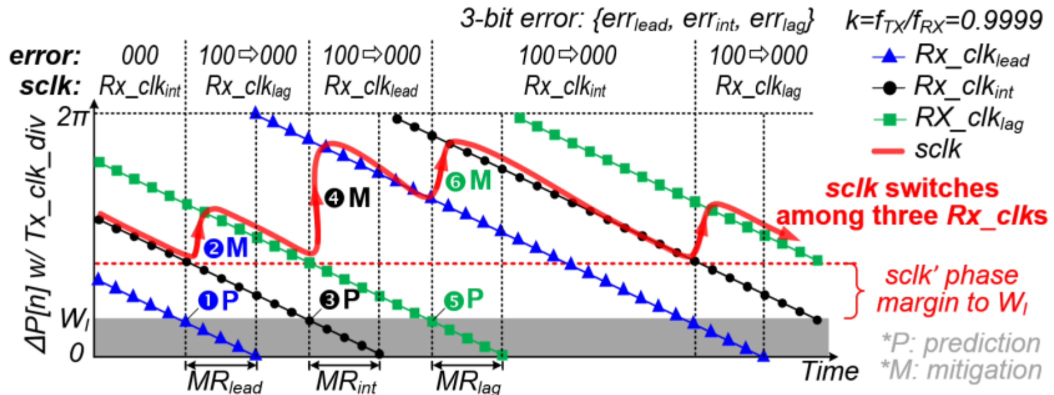

Figure 2.11: Single stage synchronizer receiver clock dynamically changing [12]

[12] proposes to that end the use of 3 different phases shifted receiver clocks as well as in situ error detection to predict future metastability. This information is used to choose a clock which is not affected by metastability and therefore minimize potential errors. As can be seen by following the red path in Figure 2.11. By doing so, the use of a single receiver flip-flop as a receiver is possible. Thus, enabling a lower data transfer latency and power consumption.[12] This approach requires additional external resources with possible repercussions on hardware designs, such as the phase shifted clocks. Furthermore, it is a very complex design.

2.2.3 CDC Detection

The detection of CDC issues is a challenging task, primarily due to the inconsistency in the error characteristics exhibited by a hardware system. As previously stated, the statistical error referenced in equation 2.1 is characterized by an occurrence probability that renders its identification challenging.

For this reason, and to circumvent the production of multiple ASIC units or debug embedded systems, it is preferable to identify structures that could potentially induce violations at an early stage of the design process. The detection is therfore typically conducted prior to synthesis, after all functions were implemented.

It is common practice to employ software tools such as Questa from Siemens, Spyglass from Synopsys, or the Conformal CDC checker from Cadence for detecting CDC violations. These tools support the common languages VHDL and Verilog. The detection methods employed by those tools often work in tandem and can be differentiated between static and dynamic methods. Static methods check the crossing and its structure to identify problematic areas, whereas dynamic simulations run, as the name suggests, a simulation where the resulting signal timeline is checked for violations. [11]

2.2.3.1 Static Verification

Static verification can be divided into two distinct parts. The initial check of signals that cross the clock domain boundary, including the assessment of whether synchronizers are employed there, is one such part. Another part is the evaluation of whether signals reconverge after crossing the clock domain boundary. [11] Secondly, assertion or formal methods are employed. These methods utilise the properties and mathematical methods of Linear Temporal Logic to ascertain whether the property is untrue at any point. If the property is false, this would indicate an error. [13] In other words, the property represents the intended behavior. [14] [15]

Assertion are capable of verifying boolean properties over explicit time domains, encompassing synchronous and asynchronous clocks, as well as delays. This enables the assessment of the functionality of the developer in comparison to a specific behavior,

indicating that it is not solely utilized for CDC but also for general verification. The limitation of assertion is that it is designed to handle only Boolean values, i.e., 1 and 0. While this is sufficient for typical applications, it is unable to model metastability, where the value lies between those two values, as described in Section 2.2.1. This limitation on the otherwise powerful tool's capabilities with regard to CDC violations is noteworthy. [14]

The primary advantage of assertion-based verification is the capacity to validate the functionality as intended of the IP-core in the presence of potential random delays, which may be attributed to CDC-related issues that can cause signal delays. This can even occur when synchronizers are employed, as illustrated in Section 2.2.2.1. To integrate these considerations into the analysis, assertion-based techniques utilize pseudorandom delays that induce delays at the crossing. These delays can occur sporadically to emulate CDC behavior.

2.2.3.2 Simulation Based Verification

Simulations are designed to emulate the behavior of the IP-core, though they may be executed at varying levels, depending on the test bench. Some may simulate the signal timings and ascertain whether the sampling clock edges are closer than a previously defined threshold, which would indicate a CDC violation. Other simulators may perform a Spice simulation at the flip-flop level to identify metastable phenomena. Such an undertaking would necessitate a profound understanding of the specific silicon technology employed. [11]

Nevertheless, simulations can accurately detect issues that may be overlooked by static methods due to the fact that they do not go through the same operations. One disadvantage is the discrepancy between reality and simulation due to a multitude of unpredictable parameters, including induced power due to EMC, manufacturing tolerances, and available power supply due to other function blocks on the integrated circuit (IC). These factors alter the precise switching time and probability set up and hold time violations. Additionally, simulations necessitate a prolonged runtime due to the extensive computation of each step and signal. Furthermore, a test bench is required to interact with the IP-core and checks all possible inputs. [11]

2.3 Space Wire

SpaceWire is a high-speed, low-power, and standardized serial communication protocol designed specifically for use in spacecraft and satellite systems. The European Space Agency (ESA) developed SpaceWire in collaboration with academic institutions and international partners. It facilitates efficient and reliable data transfers between various on-board components, including sensors, instruments, processors, downlink telemetry, and mass-memory units. It is employed by numerous international space agencies, including NASA, ESA, JAXA, and Roskosmos, for a multitude of missions, such as Bepi-Colombo and the James Webb Space Telescope. [16]

SpaceWire employs a bidirectional, full-duplex data link with a peer-to-peer connection and the option of utilizing routing switches to establish a communication network. With a high data rate of 2 to 200 Mbps in both directions, and the potential for data rates of up to 400 Mbps through the use of matched impedance connectors. The signal is transmitted for each direction, over a twisted pair cable with differential signals. Those differential signals are a data signal and a strobe signal. The strobe signal changes states when ever the data signal does not. This approach allows for the recovery of clock information and the implementation of a redundant transmission in the event that the clock frequency is known. The clock information can be obtained by XOR-ing the data and strobe signal, as illustrated in Figure 2.12. [16]

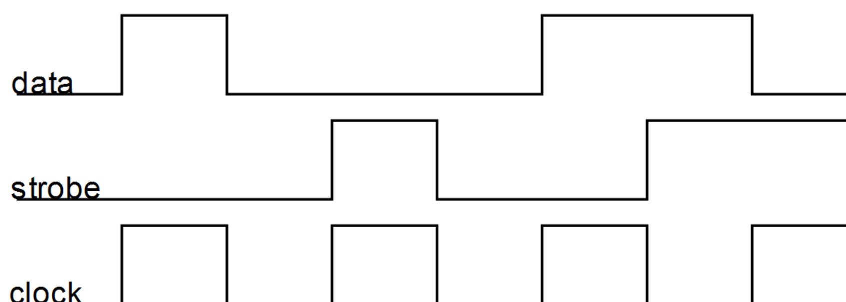

Figure 2.12: SpaceWire differential Signals and clock corollation

SpaceWire data is transmitted in packages, with a fixed format depicted in Figure 2.13. In the event that a SpaceWire network with routers is in operation, the package commences with the destination address. In the case of a point-to-point connection, this address is not a required component. In a network context, the package either

contains the recipient port identity (or node) or the path information the package must traverse, since the network and all participants are known. The subsequent element of the package is the cargo, which represents the transmitted data. The cargo frame can contain any number of bytes, as long as the receiver is able to buffer and process them. If this is not the case, a flow control process is initiated where the sender stops transmitting the cargo. In consequence, SpaceWire is not subject to size limitations and permits uninterrupted data transfer. The final frame marks the conclusion of the package and is the end-of-package marker (EOP/EEP). Dependent if an error was detected during transit (EEP) or not (EOP). [16]

Figure 2.13: SpaceWire Packet Format [16]

2.3.1 SpaceWire Network

The process of establishing a link between two nodes is referred to as link initialization. The link initialization procedure is first performed in order to establish a connection. A state machine controls the progression through the following states: The ErrorReset state, the initial state, it is also the state to which the controller returns after an error has occurred. The ErrorWait state enables the Rx, which then switches to ready after a set time of 12.8 µs. The Started state is then archived, where a Null message is sent. The Connecting state is then reached, where control tokens can also be sent. After the necessary tokens have been exchanged, the Run state is reached, where the nominal data exchange takes place.

Once the link is established, data is sent as packets across the network. The packets are transmitted asynchronously, meaning they are not constrained by a global clock. Instead, each link operates independently at its own negotiated speed. This flexibility allows nodes with different processing capabilities to communicate efficiently.

In complex SpaceWire networks with multiple nodes, routers facilitate the flow of data between nodes that are not directly connected. SpaceWire routers forward packets based on the address header information, ensuring that data reaches its intended des-

tination. The address header can either have a path or a logical address, as mentioned before briefly.

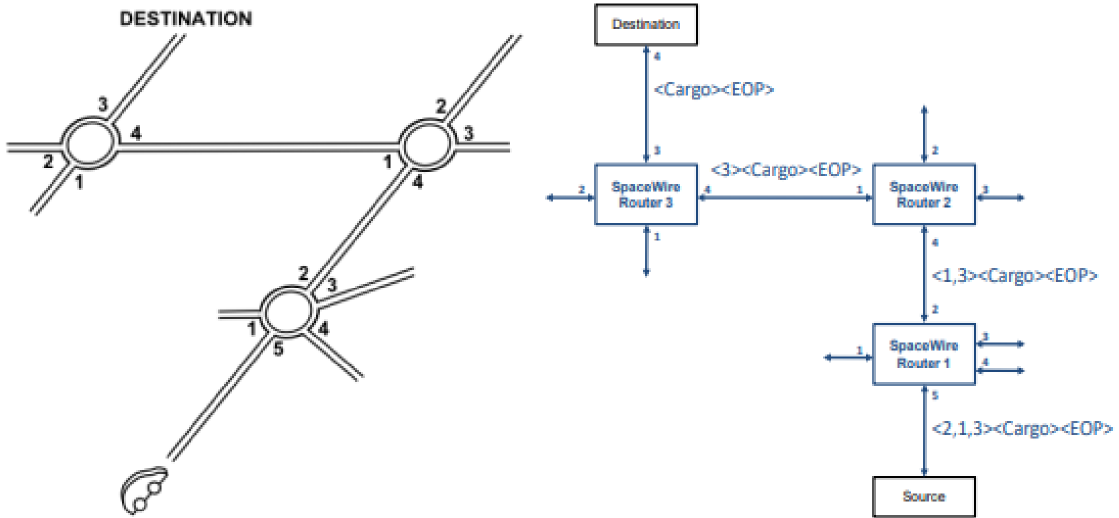

Figure 2.14: SpaceWire Path Addressing [16]

The most effective method for comprehending the path addressing system is through the use of an analogy that compares it to providing directions to a person navigating a network of roundabouts. Where an instruction is embedded for each roundabout or router, the instruction indicates which exit or port the package must take. The exit or path instructions are listed within the package. Upon selecting the exit, the corresponding instruction is discarded by the router, as it is no longer required. This concept is visualized in Figure 2.14. This is possible due to the fact that the network topology is known to the designer. An exit or port may be assigned an address within the range of 0 to 31, as a SpaceWire router is capable of supporting a maximum of 31 ports. [16]

In contrast, logic addressing employs the destination address to determine the requisite path. This pathfinding is accomplished by embedding routing tables into all routers that are aware of the direction toward the destination. If we revert to the roundabout analogy, it can be argued that at each roundabout (router), multiple signs are displayed indicating the direction to all potential destinations, as depicted in Figure 2.15. This approach offers the benefit of requiring only a single address byte, but it needs the use of routing tables for each router. The logical addresses can span

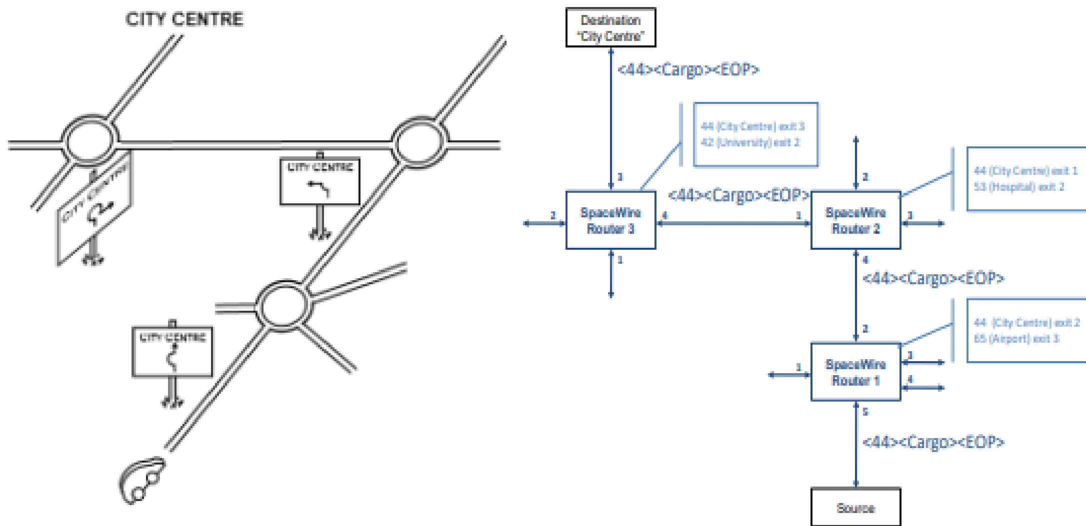

Figure 2.15: SpaceWire Logical Addressing [16]

a range of 32 to 255, ensuring that both addressing types are clearly distinguishable. [16]

2.3.2 Structure

SpaceWire can be divided into several layers. The first layer, which is responsible for the mechanical and electrical connection, has clear specifications regarding PCB tracks cables, shielding and their impedance, to ensure low EMC, high data rate and no error operations.

The encoding and decoding layer pulls the bus high and low, as well as serializing and deserializing the 10 bit token, of which a package is consisting of. The token carries 8 bit of data as well as a parity bit, to validate the message integrity and a data-control flag that indicates that the data slot has a control message to be interpreted by the receiver. The token is depicted in the Figure 2.16. [16]

The packages are build by the data layer, which also establishes the connection, by controlling the flow of information, error handling and broadcasts.

A typical SpaceWire IP-core is divided into multiple clock domains to offer the ability to transmit and receive at different data rates. The Figure 2.17 is a typical structure

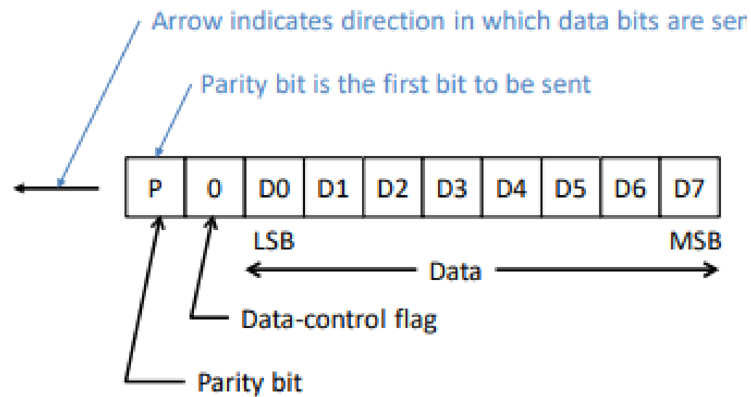

Figure 2.16: SpaceWire Token [16]

for this. While the encoding layer is typically part of the transmitting and receiving clock domain, in the Figure clk2 and clk3. The data layer is part of the computational module, making it easy for other functions to access it. This specific architecture recovers the receiver clock, from the received strobe and data signal. In doing so, insuring that the frequency and phase match at all time. [17] This is one possibility, another is to have a separate clock that controls the receiver domain, creating an independence from the transmitted clock signal that is more susceptible to glitches, e.g. EMC induced noise.

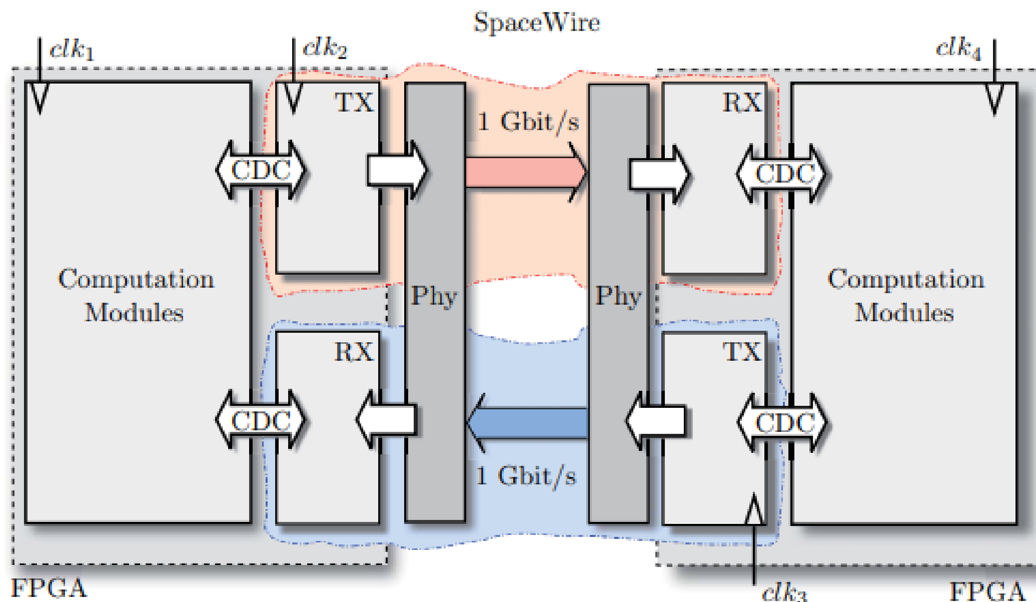

Figure 2.17: Nominal SpaceWire Clock Domains [17]

3 Used IP-Core & Tool

Now that we have reviewed the basics of SpaceWire and CDC issues and mitigation options, such as synchronizers and detection capabilities, it is time to put this knowledge to work. To do this, it is necessary to examine the IP-core that has suspected CDC problems and the capabilities of the CDC detection tool that will be used for CDC analysis.

3.1 The SpaceWire ip-Core

The SpaceWire IP-core has been employed in a number of development projects, during a code review some anomalies have been identified. Although the IP-core works initially, such anomalies can manifest randomly, which would be consistent with the occurrence of CDC errors and should be analysed. The aforementioned IP-core is open-source gateware, which is available to the public.

The SpaceWire IP-core incorporates both a fast and generic mode, enabling the attainment of a higher transmission rate through the utilization of the fast mode. For a CDC analysis, the fast mode is the only relevant component, as it employs separate clock domains to enhance the data rates, consequently increasing the respective clock speeds at the receiving and transmitting ports of the IP-core.

The IP-core is divided into multiple smaller blocks, each of which performs a specific function. This approach reduces the overall complexity of the design, while also facilitating better maintainability through the enablement of reuse. The various blocks are illustrated in Figure 3.1. Furthermore, this methodology facilitates the development process by enabling the simulation and testing of specific components, thereby improving the debugging process.

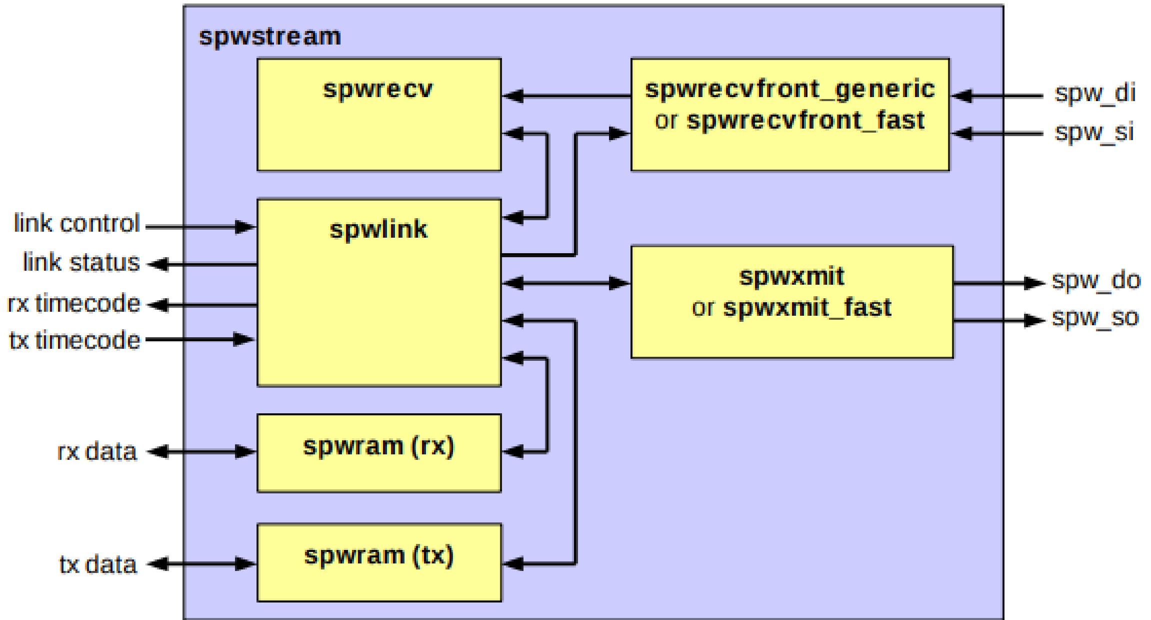

Figure 3.1: Block Diagram of the SpaceWire ip-Core [18]

The spwstream represents the top-level entity within the IP-core, wherein all subordinate entities are instantiated and the connecting ports (in and outgoing) of the IP-core are defined. A fundamental component is the spwlink, which serves as a finite state machine for controlling the connection. The data is conveyed in a FIFO scheme and stored in the spwram blocks for receiving (RX) and transmitting (TX) data. The transmitted data is converted into a SpaceWire-specific bit stream, as described in Section 2.3, and sent by the spwxmit and spwxmit_fast blocks. The receiver comprises two blocks: the spwrecv and the spwrecfront_generic or spwrecfront_fast. The spwrecfront_fast is primarily responsible for decoding the incoming bit stream into interpretable SpaceWire tokens, the tokens are then interpreted by the spwrecv into data and subjected to an integrity check, with the corresponding parity bit. [18]

The fast mode of the IP-core utilizes three distinct clocks, the main clock or system clock (sysclk) and the two faster ones for the receiving (rxclk) and transmitting domain (txclk). Those are used by spwrecfront_fast and spwxmit_fast respectively, additionally to the sysclk. To synchronize those clock domains, the IP-core employs two synchronizing structures, that are explained in the following subsections.

3.1.1 2FF IP-core Synchronizer

The SpaceWire IP-core uses a 2FF synchronizer for a large portion of signals, namely status and ... signals, which is a common practice for single signals. It is a separate function block, in order to be able to be implemented where needed. This is done by declaring the function block as a component, with the existing ports and assigning the connecting signals. The assignment is done as depicted in VHDL-code 3.1. Where a specific component is created with a corresponding structure, in this case the component is syncrx_reset with the specific structure or function of the synchronizer (syncdff) and the signals are assigned. While the left signals related to the synchronizer component the right ones are the connected signals. For instance, the clk is part of the dff component, this can be seen in the VHDL-code 3.2, and is connected to the rxclk, our clock signal.

Code 3.1: component assignment example

```
1     syncrx_reset: syncdff
2         port map ( clk => rxclk, rst => r.rxdis, di => '1', do =>
    syncrx_rstn );
```

The ip-Core synchronizer follows the core principles as explained in Section 2.2.2.1, by using two flip-flops, as can be seen in VHDL-code 3.2. By using it as a component the use of an assignable clk is needed, therfore the clk signal can be assigned to either of the three clocks sysclk, txclk or rxclk.

Code 3.2: IP-core 2FF synchronizer

```
1  entity syncdff is
2      port (
3          clk:        in  std_logic;
4          rst:        in  std_logic;
5          di:         in  std_logic;
6          do:         out std_logic
7      );
8  end entity syncdff;
9  architecture syncdff_arch of syncdff is
10 begin
11     do <= syncdff_ff2;
12     process (clk, rst) is
```

```
13      begin
14          if rst = '1' then
15              -- asynchronous reset
16              syncdff_ff1 <= '0';
17              syncdff_ff2 <= '0';
18          elsif rising_edge(clk) then
19              -- data synchronization
20              syncdff_ff1 <= di;
21              syncdff_ff2 <= syncdff_ff1;
22          end if;
23      end process;
24  end architecture syncdff_arch;
```

The synchronizer differs slightly from the in Section 2.2.2.1 described, by using a reset signal to pull the internal and outgoing signals to '0'. As well as attribute declaration to remove unwanted or erroneous optimization during synthesis, that can be seen in the full syncdff code in the appendix A.3. These attributes are specific for the Xilink synthesis tool, since the IP-core was developed for a Xilinx FPGA specifically the Spartan-3. [18] For instance, the use of shift-registers is disabled, which often are fixed silicone structures and have different behavior than two flip-flops. Furthermore, register duplication is disabled where additional flip-flops are used by the synthesizing tool, to improve the signal timing by duplicating the function in different parts of the FPGA and thus decreasing the signal path. This additional elements and changed paths, could have adverse effects for the synchronizer, because the structure can be changed. 2.2.2.1

3.1.2 FIFO IP-core Synchronizer

The IP-core employs a FIFO synchronizer to facilitate the transfer of received and transmitted data across the clock domain boundary. Furthermore, the FIFO serves as a buffer, this is also a primary function that is used in the IP-cores generic mode. The generic Receiver does not use the FIFO, because the received data can be directly relayed to the spwrecv, spwlink and lastly the user blocks, without the need to store the messages.

The FIFO component is a scalable solution, due to the use of the generic type abits and dbits, that allow the use of a changeable signal in this case the s_mem is used as a register size, as can be seen in the VHDL-code excerpt 3.3. It employs a package signal type to generate static ram blocks, that can be interpenetrated as such from the synthesizer.

Code 3.3: IP-core FIFO excerpt

```
architecture spwram_arch of spwram is
    type mem_type is array(0 to (2**abits - 1)) of
                     std_logic_vector(dbits-1 downto 0);
    signal s_mem:   mem_type;
begin
    process (rclk) is -- read process
    begin
        if rising_edge(rclk) then
            if ren = '1' then
                rdata <= s_mem(to_integer(unsigned(raddr)));
            end if;
        end if;
    end process;
```

The IP-cores FIFO incorporates both a read and a write portion with potentially separated clock domains as well as an enable signal. The read portion can be seen in the VHDL-code 3.3 and is very similar to the write portion with the only real difference being the signal assignment on enable activation being toward the register. The FIFO component works exactly as described in the Section 2.2.2.3 and is therefore suited for data transmission over clock domain boundaries. Although, the graycode counter is not included in this component to be used more freely in a none CDC manner.

3.2 Questa CDC Tool

Questa CDC is a tool used for finding CDC Violations and verifying hardware description level designs (e.g. VHDL or Verilog designs) integrity in this aspect. Developed by Siemens EDA and being part of a verification suite that contains a collection of formal-based functional verification applications. Focusing on automated analyses and providing a graphical user interface (GUI), to enable user friendly debugging of detected CDC issues. The Questa CDC suite contains three applications:

- Questa CDC
- Questa Gate-CDC
- Questa ResetCheck

These applications are specialized for certain use cases, for instance the Gate-CDC specializes in synthesized code, where a gate level netlist was generated. Similar to the ResetCheck, where reset domain crossing is being analyzed. Lastly, the Questa CDC application is the most prominent one, that includes besides a normal CDC analyses, a transfer protocol checker and a method of injecting meta stability effects. The CDC application can further be divided into a static, protocol and FX-metastability injection methods.[19]

The static method, which starts with the report-clocks phase of CDC, is dedicated to the detection of clock trees within the design. During this analysis, the tool identifies all signals that drive clock storage elements and attempts to connect these clocks in order to map out the design's clock trees. It is very important for the CDC static analysis to be conducted accurately, that the identification of clock trees and the grouping together of those associated with the same clock domains is correct. Only those paths that traverse the boundaries of distinct clock domains are identified as such and subjected to further analysis. The examination of these CDC signal path crossings and their corresponding CDC schemes, which encompass the presence or absence of synchronizers, is then conducted. These CDC schemes are subsequently ranked according to their severity, either in accordance with the CDC report scheme directive or by default settings. [19]

CDC protocol verification represents a dynamic extension of static CDC analysis, thereby enhancing the thoroughness of clock domain crossing evaluations. This process comprises two parallel components: simulation using CDC protocol assertions and formal analysis based on CDC protocol properties. The simulation component employs CDC protocol assertions to actively monitor and verify correct behavior during the design's execution, identifying any protocol violations in real time. Meanwhile, the formal analysis component utilizes CDC protocol properties to rigorously prove or disprove the correctness of clock domain crossings through mathematical verification methods. The formal methods are additional to already used ones, that check the synchronizers. The additional ones check the signal functions of the IP-core. [19]

The CDC-FX metastability extension enhances the simulations of the compiled design with metastability injection logic, which emulates the behavior of a hardware implementation experiencing random metastability effects. The introduction of this additional logic results in the simulation of potential metastability events, thereby emulating the unpredictable nature of real-world hardware conditions. While end-to-end tests may pass under standard simulation conditions, they can potentially fail when subjected to these metastability effects unless the design has been appropriately "metastability hardened." This process of hardening is critical to ensure that the design can reliably handle metastability in actual hardware, thereby preventing potential failures in clock domain crossings and improving the overall robustness of the system. [19]

4 CDC Violations & Recovery

After reviewing the basics of digital electronics and the problems of CDC, SpaceWire, and the specific IP-core and its synchronizers, as well as synchronizers in general, it is time to perform the CDC analyses. Beginning with the setup and continuing with the analyses of the detected violations as well as potential changes to the IP-core to correct issues.

In the context of CDC analyses, it is only the Questa CDC application that is being utilized, given that the IP-core is in the register transfer level. Consequently, the Gate-CDC can be excluded from consideration. Moreover, the ResetChecker can be disregarded, as the sole reset incorporated into the IP-core is utilized exclusively for the entire IP-core, rather than during normal operation and the IP-core works. Therefore, it is highly unlikely that the sporadic data errors stem from the reset. The reset primarily sets all registers back to their default values. Given the time constraints, it is not feasible to implement all the possible CDC analysis methods, as they require a significant amount of time for initial setup and learning how to employ them. The differences between the methods are considerable. For example, the protocol verification requires formal verification expertise and a detailed understanding of the IP-core's internal workings to define the necessary properties. In contrast, metastability injection necessitates an entire counterpart for simulation, enabling the IP-core to interact with the surrounding environment. Considering the diverse analytical techniques, the static analysis offers comprehensive coverage with a relatively straightforward setup. Consequently, the static analysis serves as the optimal initial approach, providing good detection results, as demonstrated in the subsequent analyses. Furthermore, it builds a foundation for subsequent methods to be employed at a later time if deemed necessary.

4.1 Setup

To perform a correct analysis, the setup is a crucial step. For an analysis, to work correctly, the setup includes:

- prepare the IP-core
- creating a Makefile
- include all necessary IP-core files
- define clock signals
- define clock domains of signals that the tool is unable to infer

For a formal verification, an additionall constraint file is necessary where all the possible properties and constrains are defined. A simulation would also require a whole counterpart, to and from which the transmissions are sent, as well as a simulated main computational element, where all the received messages originate from and all delivered messages are sent to.

Since the IP-core has two operating modes, fast and generic, it is important to do the proper selection in the IP-core. Furthermore, the system and transmission clock frequencies need to be defined, in order to enable the fixed SpaceWire signal timings, e.g., during link handshake. It is essential for the correct function of the IP-core that the transmission and receiver clock frequency is higher than the system clock. [18]

The makefile (A.5) is used to quickly start the analysis after it is setup, since it holds the necessary commands and executes them in sequence on activation. The commands of the makefile is configured to start by removing any previous results, to avoid outdated results, although this becomes only relevant from the second execution onward. Secondly, the logical VHDL description is mapped to a physical configuration, i.e. forming simulated signal paths, beginning at the device under test which is the top entity. Then it is compiled into a working design. Lastly, the CDC analyses is performed, with the directives specified taken into account. [19]

The next step is the filelist in which all necessary blocks for the IP-core are defined with their paths. This is important for the tool to find the location of the file to be able

to link them together, especially if they are distributed between different directories. The filelist for this project is depicted in the appendix A.6.

Similar to the IP-core, the tool must be aware of all clock signals and their respective period or frequency in order to function correctly. If wanted, one can impose user constrains on the CDC analyses, that are also included in the directives file. The basic setup has been done at this point, although if the analysis is run, multiple errors are detected. These errors are caused by signals, which clock domain can not be inferred by the tool and therefore need to be defined manually in the directives file, this results in the finished directives file (A.7). After this is done, the analyses can be run successfully.

Status-Severity-Check	TX Signal	RX Signal	ID	TX Clock	RX Clock	TX Module	RX Module
? Uninspected (19)							
V Violation (6)							
Single Source Reconvergence of synchronizers (4)							
	xmit_sel1.xmit_fas…	{xmit_sel1.x…	single_sourc…	txclk ,txclk ,txclk	txclk	syncdff ,sync…	spwxmit_fast
	recvfront_sel1.rec…	recvfront_sel…	single_sourc…	clk ,clk ,clk	clk	syncdff ,sync…	spwrecvfront…
	recvfront_sel1.rec…	recv_inst.r.dis…	single_sourc…	clk ,clk ,clk	clk	syncdff ,sync…	spwrecv
	xmit_sel1.xmit_fas…	r.txfifo_rvalid	single_sourc…	clk ,clk	clk	syncdff ,syncdff	spwstream
Multiple-bit signal across clock domain boundary (1)							
	{ xmit_sel1.xmit_f…	{ xmit_sel1.x…	multi_bits_85…	clk	txclk	spwxmit_fast	spwxmit_fast
FIFO pointer mismatch (1)							
	recvfront_sel1.rec…	recvfront_sel…	fifo_memory_…	rxclk	clk	spwram	spwram
E Evaluation (9)							
Single-bit signal synchronized by DFF synchronizer (8)							
Asynchronous reset synchronization (1)							
P Proven (4)							

Figure 4.1: Detected Errors

The CDC check results are illustrated in Figure 4.1. Six initial violations were identified, which were classified into three distinct categories: Single Source Reconvergence of synchronizers, a Multi-bit signal across the clock domain boundary, and a FIFO pointer mismatch. Additionally, to the violations, which are likely to be problematic and require further examination, the tool identifies proven crossings that are deemed safe and crossings that warrant further assessment. These evaluation crossings should be subjected to manual examination, as they may potentially cause CDC errors, although this is unlikely. Some marked evaluations are connected to violations and can help find the CDC error inducing part.

4.2 Single Source Reconvergence of synchronizers

The violation that is detected the most is reconvergence, more specifically Single Source Reconvergence. That refers to a situation where a signals from one clock domain is passed through multiple synchronization paths and then reconverges in another clock domain. Even if each path individually uses proper synchronization techniques, like a 2FF synchronizer, the reconvergence point can introduce timing issues, that are resulting from CDC induced delays and therefore classified as CDC violations.

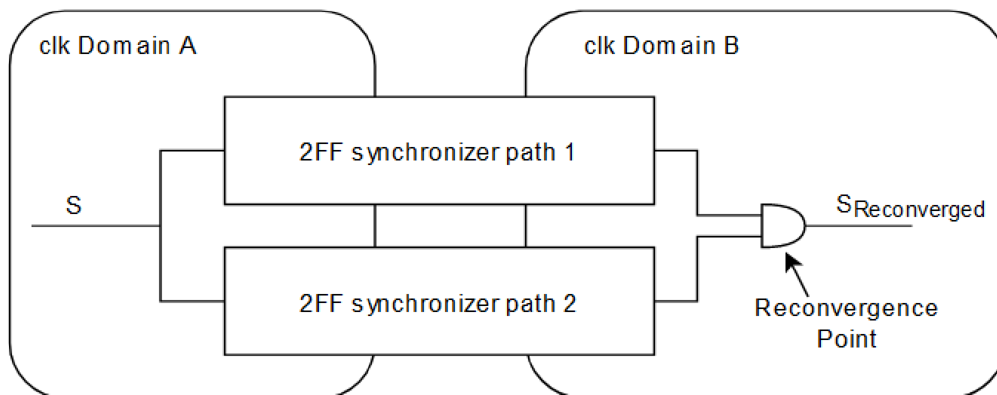

Figure 4.2: Simple Reconvergence Structure

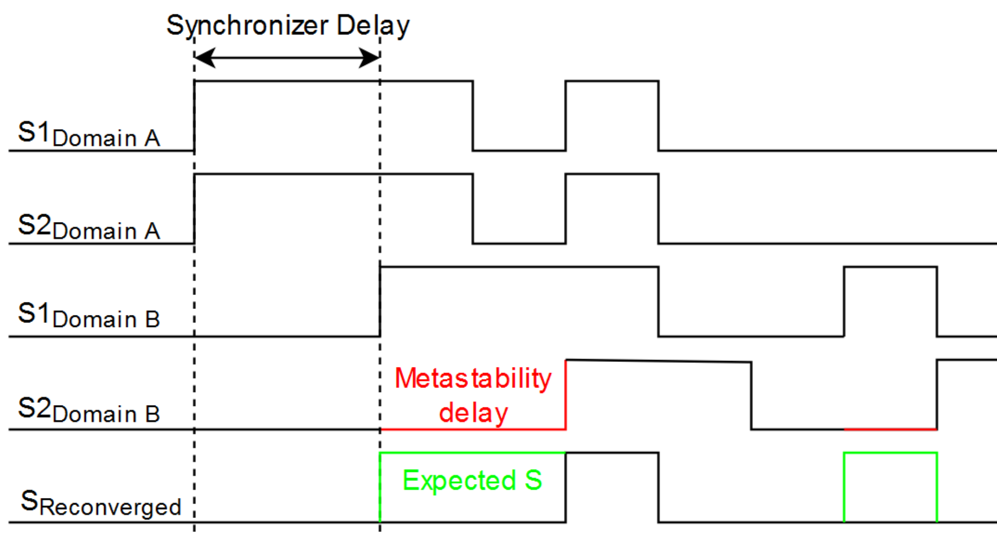

Figure 4.3: Reconvergence Signals

For example, considering a two bit signal S in clock domain A. This signal is sent to clock domain B, where it is synchronized by two separate paths, P1 and P2. One path for each bit. After crossing the clock domain boundary, these two paths reconverge at a logic element in clock domain B. In the Figure 4.2 depicted example, the signal is synchronized with two 2FF synchronizers, for each bit, one. The signal reconverges in an and logic block. If there are differences in the delays due to metastability, of one of the paths (P1 or P2), the signals may not align properly at the reconvergence point. This can lead to inconsistent data being latched into the downstream logic, causing functional errors. This signal delay and the resulting problems are visualized in Figure 4.3. The second signal bit (S2) suffers a delay caused by metastability after taking the second synchronizer path P2. This delay results in an unexpected and erroneous signal ($S_{Reconverged}$) after reconvergence. Even though S2 still correctly represents the signal changes in the B clock domain, just a cycle delayed.

This demonstrates how a small delay can cause multi bit signals to lose their integrity, and how small deviations from assumed signal changes can have a large impact on the resulting logic.

4.2.1 FIFO Violation

The first reconvergence violation is depicted in Figure 4.4, in which signals originating from the txclk domain reconverge after some binary logic blocks and multiplexers reconverge and are saved in the register r. Specifically, the signals txflip0 and txflip1 which are used to indicate to the system clock domain that a token was pulled to the transmission clock domain. This indication frees a token slot, in the system clock domain, that functions as a buffer memory. The tokens are a single 10 bit SpaceWire message, as described in Section 2.3.

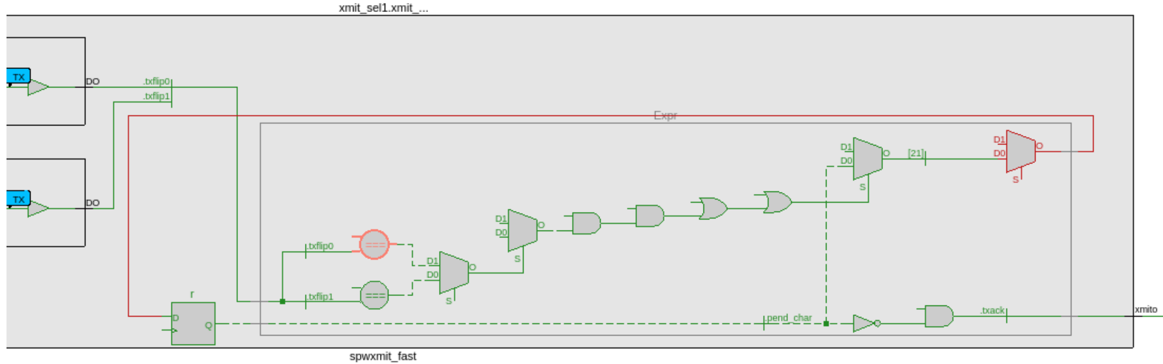

Figure 4.4: First Violation's Logic Circuit

As previously mentioned, reconvergence errors appear when multiple signals are dependent on one another in time, i.e. have a meaning together like a two bit signal. Therefore, the first step of analyzing the violation is to decern if the two signals have a temporal dependency. For that, the following points need to be checked.

- is there essentially a two/multi bit signal?
- does the clock domain boundary crossing have a synchronizer?
- does it have an effect if one or both are delayed for a few cycles?
- is this effect caused by the delay accounted for?

The two txflip signals work independent from each other, indicating the status of the corresponding token space, i.e. transmitted or waiting to be transmitted. Therefore, the answer for the first point is a single bit signal. Secondly, checking the transit of the clock domain boundary. Where 2FF synchronizers were employed. These synchronizers are partly visible in the logic circuit diagram in Figure 4.4, on the left side. The synchronizers are here a good choice because both signals work independently. Which was the answer for the first point. The effects a delay induced due to metastability would have on the function needs to be determined. In this case, one of two outcomes could appear. One, both signals are delayed that results in a blocked buffer and a delayed delivery of tokens. Two, one of the signals is delayed. That would change the order of delivery, since the non delayed token is able to be transmitted. This switch

in token delivery is for applications that only require a single byte neglectable or if the packages do not need to have a defined order to them. For packages that require a defined token order, the corresponding ordering information would need to be inside the transmitted token. This imposes a restriction in the usage of the IP-core. In the case of delayed token delivery, the IP-core sends a "NULL" message, as described in Section 3.1, and is therefore not critical. Furthermore, this is only relevant for very large frequency differences, since the time to send a token must be less than the delay added to the time it takes for the token to cross the clock domain boundary and for the status signals to change.

The NULL message is a form of control code in which is used to keep the data link active, in the event that no data or control messages are sent and ensure a connection i.e. enabling disconnect detection. [16] The composition can be seen in the Figure 4.5.

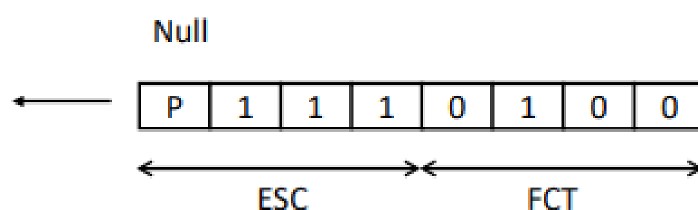

Figure 4.5: SpaceWire Null Message[16]

To conclude, by evaluating the three points, the CDC violation can be ignored if the token order is not a strict requirement or the frequency differences of both clocks are small.

4.2.2 Reset Violation

This violation was detected due to the reset signal rst, resetting the flip-flop in the tx-clock domain of three 2FF synchronizers, as depicted in Figure 4.6. Since it is part of the reconvergence violations, which would affect the txen, sysflip1 and sysflip0 signal, if metastability was to occur on the rst signal. Furthermore, the reset signal there-

fore crosses the clock domain boundary without a synchronizer and would therefore constitute as an CDC violation.

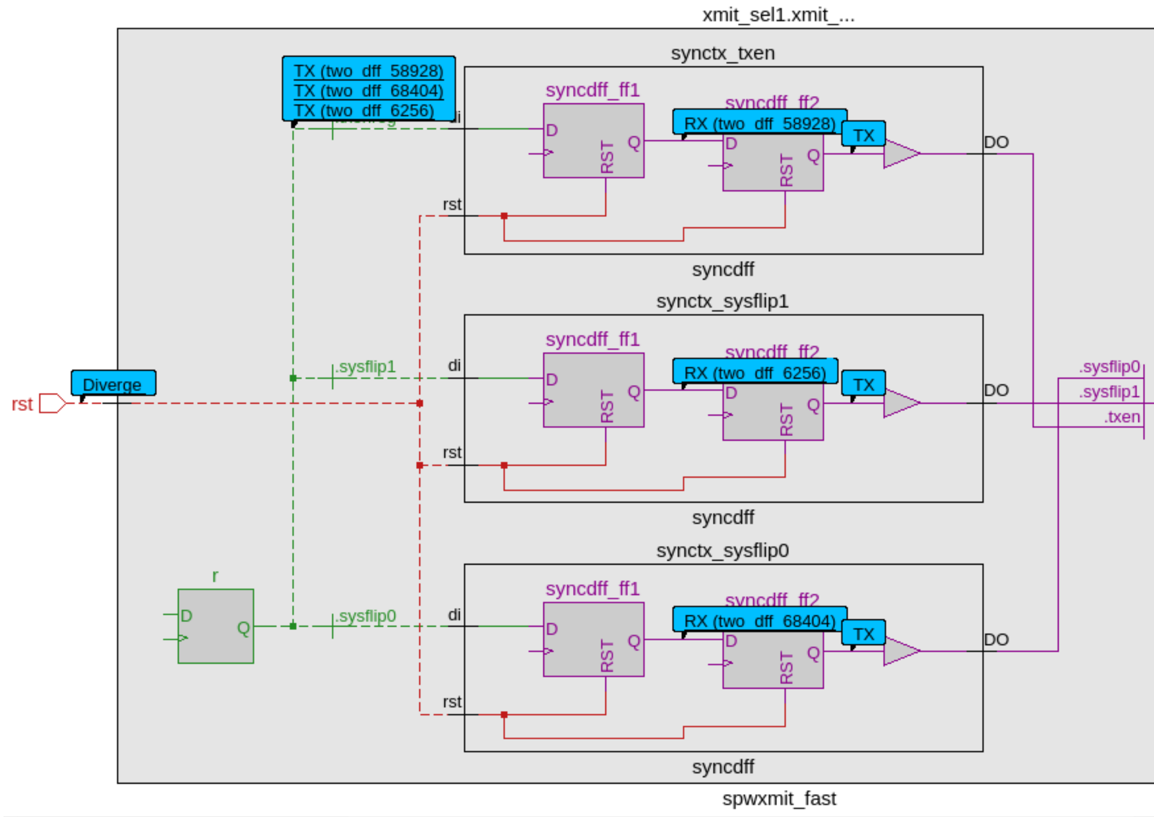

Figure 4.6: Second Violation/ reset violation

A reset signal constitutes a special case when it comes to CDC issues. Assuming a reset synchronizer would be implemented, this synchronizer would often have a reset port on their own, presenting us with the same problem. Furthermore, this approach would need the reset to be pulled to the corresponding value for long enough to propagate through the synchronizer. In this case it would be enough to simply pull the reset for long enough to settle even in the event of metastability occurring. Notably, the reset does not use a synchronizer due to those reasons.

The main problem with resets is not the activation, but the release, where the logic starts to operate again. It is important that the functions start at the same time. Otherwise, CDC problems can cause delays in the initial signal changes of individual signals. These signals usually form a larger logic circuit that experiences issues similar

to reconvergence errors. In the case of the SpaceWire IP-core, this problem is reduced by using a state machine as the primary control instance. In order to enter the next operating state, a number of conditions must be met, allowing any delayed signals and logic to catch up and resynchronize. Due to those reasons, the reset violation warrants no changes.

4.2.3 Pointer Crossing bitcnt & headptr

The SpaceWire IP-core uses a FIFO synchronizer for the received data. This FIFO consists of multiple memory blocks, that are referenced with an address pointer, which points to one of tree memory registers, each of them being one block, for data transfer. This pointer allows for continuous operation by switching between the write and read dependency, for the FIFO memory, accordingly. This dependency on both clock domains means that this pointer needs to be synchronized in a fitting manner. The synchronization was implemented in the IP-core as a 2FF synchronizer for each bit of the pointer, as depicted in Figure 4.7, which can result in erroneous behavior, due to delay caused by metastability as described in Section 2.2.2.2.

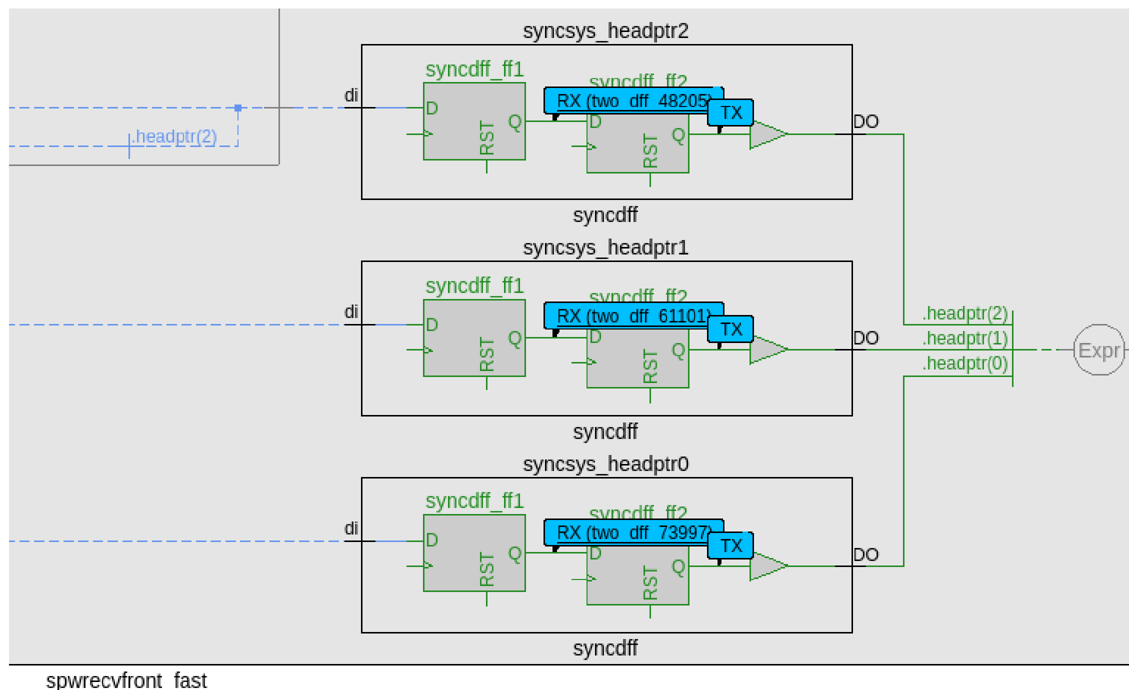

Figure 4.7: Pointer Error Diagram

Furthermore, the implemented pointer increments normally and not via graycode, what is a major risk. With graycode the 2FF Synchronizer would be a good solution as mentioned in Section 2.2.2.3. Therefore to fix this issue, a graycode counter should be substituted for the regular one.

Code 4.1: Graycode Implementation

```
case gray_cntr_head is
    when "00" | "10" =>
            vrx.headptr(0) := (not rrx.headptr(0));
  when "01" =>
    vrx.headptr(1) := not rrx.headptr(1);
  when "11" =>
    vrx.headptr(2) := not rrx.headptr(2);
    gray_cntr_head <= (others => '0');
  when others =>
    gray_cntr_head <= (others => '0');
end case;
gray_cntr_head <= gray_cntr_head + 1;
```

A possible implementation of a graycounter is depicted in the VHDL-Code 4.1. It works by using a case statement and an additional counter signal, gray_cntr_head. By negating the individual bits of the pointer in the appropriate order.

vrx.headptr
000 → 001 → 011 → 010 → 110 → 111 → 101 → 100
gray_cntr_head
00 → 01 → 10 → 11 → 00 → 01 → 10 → 11

Table 4.1: Gray code Counter Incrementation

The archived gray code incrementation is depicted in Table 4.1, while the order in which the bits need to be negated is archived with regularly incrementing the counter signal gray_cntr_head in conjunction with the case statement. The same pattern was put at the system clock counter to match the adjusted counter at the rxclock domain.

The same violation was also detected for the bitcnt signal and got the same adjustment. The bitcnt signal is used to detect activity on the SpaceWire bus. It increments

whenever the rxclock domain receives new bits. This is done because the system clock domain monitors the synchronized bitcnt signal to determine whether it has increased since the previous system clock cycle, and therefore needs to prepare to receive the data.

4.3 Multi-bit signal across clock domain boundary

This violation describes essentially the same issue as the pointer crossing bicnt and headptr. Where the multibit pointer signals are transmitted with only an 2FF synchronizer. After implementing the graycode counter, as described in Section 4.2.3 and running the analyses again, the violation does not show again and is therefore resolved. It is important to note that this type of false positive, where a non existent violation is detected, is a common occurrence in violation detection tools. These tools are designed with a conservative approach, often identifying a higher number of potential violations. This conservative approach is intentional, aiming to ensure that all potential errors, especially those that might be missed (false negatives), are flagged for review. While this can lead to the identification of non-issues (false positives), it is a necessary trade-off to prevent critical undetected errors that could compromise the integrity of the design if left unaddressed.

4.4 FIFO pointer mismatch

The FIFO pointer mismatch violation occurs within the FIFO synchronizer. In the potential violation depicted in Figure 4.8, both the txclock and the sysclock signal control the s_mem, and the outgoing signal based on both is detected to be the violation, since it is a synchronizer it is not uncommon to have two clock domains converging.

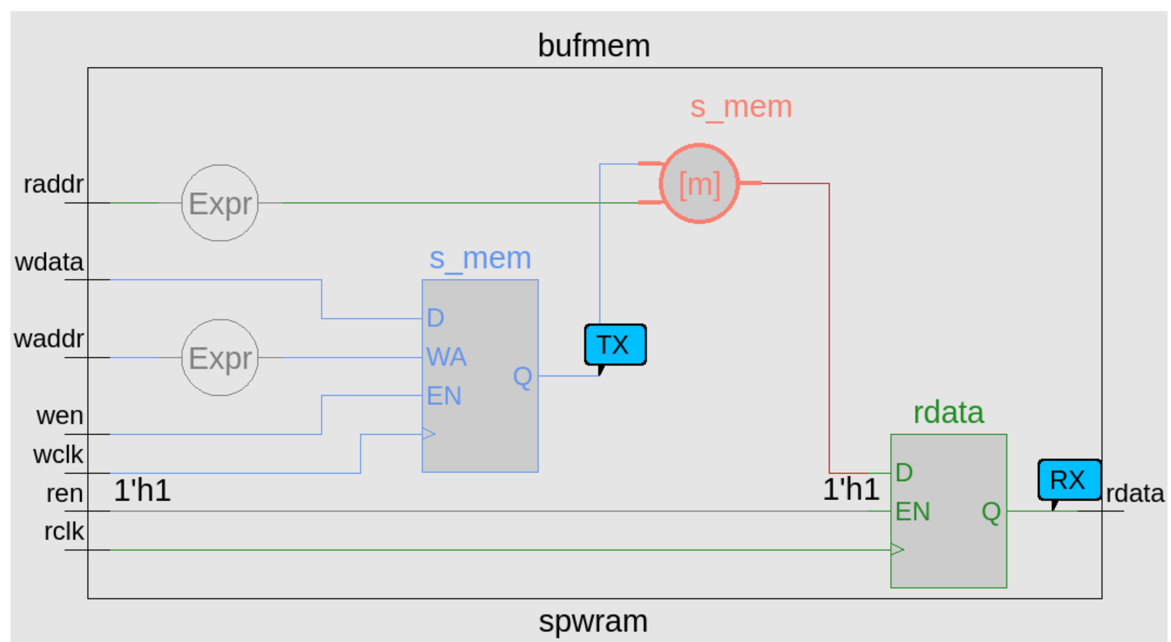

Figure 4.8: FIFO violation

The violation is likely attributable to the utilization of a memory array of the mem_type, as previously referenced in Section 3.1.2, which is identified by the CDC tool as a single coherent signal array, despite its actual composition being separate blocks. These blocks can be read and written to independently with the corresponding clock, although only one of each of the read and write operations can occur simultaneously. Given that multiple blocks are employed, it is necessary for the address pointer to be different in order to access the FIFO. Only when this condition is met can the write operation be initiated. Consequently, the mismatch represents a desired state for access operations. This allows us to ignore the violation, as the same portion of the synchronizer memory cannot be accessed by both clock domains.

4.5 Verification

Since CDC violations appear irregular, testing with hardware is very difficult and may not even yield results even if a CDC violation is present, especially for complex elements such as this. Due to this reason, this is being omitted and only the tool handling is tested.

In order to verify that the tool was correctly setup and violations can be safely detected. A violation inducing construct was embedded into the IP-core. This is done to be separate to the IP-cores function and therefore does not influence its workings.

The construct consists of some generic logic in the system and txclock domain. With signals that correspond to those domains, as can be seen in the VHDL-Code 4.2. The logic incorporates combinatoric and sequential elements with the xor and if statements respectfully.

Code 4.2: Induced violation

```
1      --induced violation
2      signal induced_violation : std_ulogic_vector(2 downto 0);
3      signal induced_violation_tx : std_ulogic;
4
5  begin
6
7  process (clk) is
8  begin
9      if rising_edge(clk) then
10 ...
11         induced_violation <=  std_ulogic_vector(unsigned(
       induced_violation) + 1);
12         induced_violation_out <= induced_violation_tx;
13     end if;
14 end process;
15
16 process (txclk) is
17 begin
18     if rising_edge(txclk) then
19       if induced_violation_tx = induced_violation(2) then
20       induced_violation_tx <= (not induced_violation_tx) xor
       induced_violation(0);
```

```
        end if;
      end if;
end process;
```

It is important to have a connection between an major port i.e. out of the ip core and the violating inducing part. Because the tool only registers violations that have an effect on the output, because it traces the signal paths starting from the inputs and ending on the IP-core outputs. Therefore, the verification construct does need a connection to the output port, that can be seen in the VHDL-Code 4.3.

Code 4.3: Port declaration

```
port(
...
induced_violation_out : out std_ulogic
);
```

After running the CDC analyses, two violations were detected by the tool, proving the setup was done right. One violation for each signal flow direction, as can be seen in Figure 4.9.

Violation (7)							
Single-bit sign...							
	induced_violation_tx	induced_violation[0]	no_sync_41940	clk	txclk	spwstream	spwstream
	induced_violation_out	induced_violation_tx	no_sync_45160	txclk	clk	spwstream	spwstream

Figure 4.9: Induced Violation

The detected violations are the xor combinatoric in line 20 of VHDL-code 4.2 and the directs signal assignment in line 12.

5 Discussion

The aim for this paper was to examine a SpaceWire IP-core for CDC violations. Furthermore, found problems should be corrected to facilitate the IP-cores usability and reliability. The performed CDC analyses has found a total of six CDC potential violations.

One out of those six violations could be safely ignored. One created some usage restrictions. One additional could be ignored, if the reset signal pulls low long enough and the last three described the same issue. The issue concerned the address pointer of the FIFO synchronizer employed by the SpaceWire IP-core, that is used to transfer the received and transmitted data from the receiver clock domain to the system clock domain where the data is interpreted and used. This issue could be resolved by replacing the address pointer incrementing scheme from a linear counting scheme (1,2,3,4,...) to a graycode scheme, as described in Section 2.2.2.2.

The changes done to the IP-core are based on optimal synchronizer schemes, that are described in Section 2.2.2. Improving the signal integrity. Therefore, the results indicate a better performance of the IP-core in regards to CDC relating issues, enabling better use for future terrestrial research projects in the department.

The analyses were done via a static analyses, even though other methods are available like simulation, which results are potentially able to reduce false positives, although the overhead also increases drastically. With the gray code correction, the reliability of the IP-core could be increased. In addition, a limitation could be identified that the IP-core is only suitable for single bytes, packets where the order of the tokens is irrelevant, or it must be encoded in the transmitted byte. This restriction is only applicable when the clock frequencies are far apart. If the IP-core appears to not work

properly, other verification methods can be used, or the problems could be caused by non CDC issues.

The initial installation and execution of the tool encountered delays due to unforeseen issues with the virtual machine. To address these challenges, a Linux machine was subsequently utilized. The primary issue was the prolonged loading, compiling, and input response times, which significantly impacted the usability and efficiency.

6 Conclusion

6.1 Summary

In this thesis, we explored the principles of digital electronics with a focus on FPGA-based systems. From this understanding, we explored the principal of metastability, with a focus of clock domain crossing and the arising issues from this, as well as methods to mitigating them, e.g. synchronizers and the avoidance of CDC if possible. A brief examination of VHDL, the language used by the IP-core, was also conducted, to be able to understand its intricate workings.

In addition, we examined the operation of SpaceWire networks and how the IP-core manages its clock domain boundary crossings. To ensure robust CDC handling, we used the Questa CDC tool for a static analysis and identified six potential CDC violations. After examining the tool and its setup. Upon further investigation, we determined that only four violations were relevant, as the tool tends to over-report violations to minimize false negatives, as automatic code analyzers often do.

One violation was reported, the same problem for two different signals. An additional violation was reported for the same problem. These violations are is part of the FIFO synchronizer used by the SpaceWire IP-core to pass messages across the clock domain boundary, to the transmission and from the receiver domain. The solution was to replace the linear pointer incrementation with a gray code incrementation scheme, which should reduce missing and scrambled SpaceWire tokens. The last violation could be partially ignored, but it creates a usage restriction of the IP-core. Where the order of two tokens could be changed, changing the bit stream. This risk can either be accepted for applications where this is neglected, or the sequence must be encoded in the token.

In conclusion, a CDC analysis could be successfully performed and several issues be found and corrected if possible. The IP-core has even after the corrections some usage restriction. Due to those I would advise against use in critical parts of a application and limiting the difference of the clock frequencies, that control the different clock domains.

6.2 Outlook

The adjusted IP-core can be used in a variety of development projects in the DLR, with some restrictions. That could be initially be identified. In any case, the knowledge about the IP-core and its limitations could be increased.

It is also possible to apply other CDC analysis methods to the IP-core. For which this bachelor thesis builds a good basis. These methods are mentioned in the section 3.2 and include simulation and extending the use of formal analysis with user defined properties.

Bibliography

[1] V. Taraate, *Digital Logic Design Using Verilog Coding and RTL Synthesis*, 2nd ed. Springer Nature Singapore Pte Ltd., 2022, ISBN: 978-981-16-3198-6.

[2] P. J. Ashenden, *The Designer's Guide to VHDL*, 2nd ed. Morgan Kaumann Publishers, 2002.

[3] J. J. P. Meseguer, "Design and optimization of a space camera with application to the phi solar magnetograph," Ph.D. dissertation, Technischen Universität Carolo-Wilhelmina, 2013, ISBN: 978-3-942171-72-4. [Online]. Available: `http://dnb.d-nb.de/`.

[4] M. W. Winfried Gehrke, *Digitaltechnik Grundlagen, VHDL, FPGAs, Mikrocontroller*, 8th ed. Springer-Verlag GmbH, 2022, ISBN: 978-3-662-63953-5.

[5] R. B. Frank Kessel, *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs*. Oldenbourg Wissenschaftsverlag, 2006, ISBN: 978-3-486-57556-9.

[6] K. Borchers, "Decentralized and pulse-based clock synchronization in spacewire networks for time-triggered data transfers," Ph.D. dissertation, Universität Würzburg, 2020. [Online]. Available: `https://elib.dlr.de/140212/`.

[7] H. Foster. "The 2022 wilson research group functional verification study." (Oct. 16, 2022), [Online]. Available: `https://blogs.sw.siemens.com/verificationhorizons/2022/10/16/part-1-the-2020-wilson-research-group-functional-verification-study-2/`. acessed: 06.07.2024.

[8] M. Bartik, "Clock domain crossing — an advanced course for future digital design engineers," in *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, 2018, pp. 1–5. DOI: `10.1109/MECO.2018.8406004`.

[9] S. Hatture and S. Dhage, "Multi-clock domain synchronizers," in *2015 International Conference on Computation of Power, Energy, Information and Communication (ICCPEIC)*, 2015, pp. 0403–0408. DOI: 10.1109/ICCPEIC.2015.7259493.

[10] N. Karimi, Z. Kong, K. Chakrabarty, P. Gupta, and S. Patil, "Testing of clock-domain crossing faults in multi-core system-on-chip," in *2011 Asian Test Symposium*, 2011, pp. 7–14. DOI: 10.1109/ATS.2011.68.

[11] A. B. Chong, "Clock domain crossing verification challenges," in *2021 2nd International Conference on Electronics, Communications and Information Technology (CECIT)*, 2021, pp. 383–387. DOI: 10.1109/CECIT53797.2021.00075.

[12] C. Lin, W. He, Y. Sun, *et al.*, "A metastability risk prediction and mitigation technique for clock-domain crossing with single-stage synchronizer in near-threshold-voltage multivoltage/ frequency-domain network-on-chip," *IEEE Journal of Solid-State Circuits*, vol. 59, no. 2, pp. 616–625, 2024. DOI: 10.1109/JSSC.2023.3283961.

[13] Y. Tao, "An introduction to assertion-based verification," in *2009 IEEE 8th International Conference on ASIC*, 2009, pp. 1318–1323. DOI: 10.1109/ASICON.2009.5351246.

[14] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny, *The Power of Assertions in SystemVerilog*. Springer, 2010, ISBN: 978-1-4419-6599-8.

[15] S. Chalana, S. Mitra, S. Bharath, R. Bhimireddy, S. K. Manickam, and S. Kumar, "Enhancing coverage of clock domain crossing assertion verification leveraging formal," in *2023 IEEE Women in Technology Conference (WINTECHCON)*, 2023, pp. 1–6. DOI: 10.1109/WINTECHCON58518.2023.10277198.

[16] "Ecss-e-st-50-12c rev.1," European Cooperation for Space Standarisation, Tech. Rep., 2019. [Online]. Available: https://ecss.nl/standard/ecss-e-st-50-12c-rev-1-spacewire-links-nodes-routers-and-networks-15-may-2019/.

[17] J. Akhundov, "Implementation of the global physical time for the domain model of the virtual path of the dlr hand-arm system," Chemnitz University of Technology, Tech. Rep., Jun. 2013. [Online]. Available: https://elib.dlr.de/87115/.

[18] J. van Rantwijk, “Spacewire light version 20110709,” Chemnitz University of Technology, Tech. Rep., 2013. [Online]. Available: `https://opencores.org/projects/spacewire_light`.

[19] S. EDA, *Siemens eda questa® cdc user guide*, 2nd, Available at `https://example.com/manual.pdf`, Siemens Digital Industries Software, Example City, CA, 2022.

Appendix

A.1 Full VHDL example code

Code A.1: VHDL example Full-adder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity example is --FUll_adder
  port(
    cary    : in  std_ulogic;
    X       : in  std_ulogic_vector (1 downto 0);
    Y       : out std_ulogic_vector (1 downto 0);
    clk_i   : in  std_ulogic;
    rst_i   : in  std_ulogic
  );
end entity example;

architecture rtl1 of example is
  signal safe : std_ulogic_vector (1 downto 0);
begin
  p1: process(clk_i)
  begin
    if(rising_edge(clk_i)) then
      if(rst_i = '1') then
        Y   <= "00";
        safe<= "00";
      else
        safe(0)<= cary xor (X(0) xor X(1));
        safe(1)<= (X(1) and X(0)) or ((X(0) xor X(1)) and cary);
        Y<=safe;
```

```
28        end if;
29      end if;
30    end process p1;
31
32  end architecture rtl1;
```

Code A.2: VHDL example Full-adder test-bench

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity tb is
6   end tb;
7
8   architecture rtl of tb is
9
10    component example is
11      port(
12        cary    : in  std_ulogic;
13        X       : in  std_ulogic_vector (1 downto 0);
14        Y       : out std_ulogic_vector (1 downto 0);
15        clk_i     : in  std_ulogic;
16        rst_i     : in  std_ulogic
17      );
18    end component example;
19
20    signal sig_cary           : std_ulogic:= '0';
21    signal sig_X              : std_ulogic_vector (1 downto 0):= "00";
22    signal sig_Y              : std_ulogic_vector (1 downto 0):= "00";
23    signal sig_clk_i      : std_ulogic := '0';
24    signal sig_rst_i      : std_ulogic := '1';
25    signal cnt                 : integer := 0;
26
27    for i_example: example use entity work.example(rtl1);
28
29  begin
30    i_example: example
31      port map (
32        cary      => sig_cary,
33        X         => sig_X,
```

```
34        Y          => sig_Y,
35        clk_i      => sig_clk_i,
36        rst_i      => sig_rst_i
37      );
38
39    p_rst_gen: process
40    begin
41      wait for 50 ns;
42      sig_rst_i <= '0';
43    end process p_rst_gen;
44
45    p_clk_gen: process
46    begin
47      wait for 10 ns;
48      sig_clk_i <= '1';
49      wait for 10 ns;
50      sig_clk_i <= '0';
51    end process p_clk_gen;
52
53    p_stim: process(sig_clk_i)
54    begin
55      if(rising_edge(sig_clk_i)) then
56        if(sig_rst_i = '1') then
57        else
58          cnt <= cnt + 1;
59          if cnt = 7 then
60          sig_X <= "01"; end if;
61          if cnt = 14 then
62          sig_cary <= '1'; end if;
63          if cnt = 21 then
64          sig_X <= "11"; end if;
65          if cnt = 28 then
66          sig_X <= "00"; end if;
67        end if;
68      end if;
69    end process p_stim;
70  end architecture rtl;
```

A.2 Relevant IP-core codes

Code A.3: syndff

```
library ieee;
use ieee.std_logic_1164.all;

entity syncdff is
    port (
        clk:        in  std_logic;          -- clock (destination domain
    )
        rst:        in  std_logic;          -- asynchronous reset,
    active-high
        di:         in  std_logic;          -- input data
        do:         out std_logic           -- output data
    );
    -- Turn off register replication in XST.
    attribute REGISTER_DUPLICATION: string;
    attribute REGISTER_DUPLICATION of syncdff: entity is "NO";
end entity syncdff;

architecture syncdff_arch of syncdff is
    -- flip-flops
    signal syncdff_ff1: std_ulogic := '0';
    signal syncdff_ff2: std_ulogic := '0';
    -- Turn of shift-register extraction in XST.
    attribute SHIFT_EXTRACT: string;
    attribute SHIFT_EXTRACT of syncdff_ff1: signal is "NO";
    attribute SHIFT_EXTRACT of syncdff_ff2: signal is "NO";
    -- -- Tell XST to place both flip-flops in the same slice.
    -- attribute RLOC: string;
    -- attribute RLOC of syncdff_ff1: signal is "X0Y0";
    -- attribute RLOC of syncdff_ff2: signal is "X0Y0";

    -- -- Tell XST to keep the flip-flop net names to be used in timing
    constraints.
    -- attribute KEEP: string;
    -- attribute KEEP of syncdff_ff1: signal is "SOFT";
    -- attribute KEEP of syncdff_ff2: signal is "SOFT";
begin
    -- second flip-flop drives the output signal
```

```
35     do <= syncdff_ff2;
36     process (clk, rst) is
37     begin
38         if rst = '1' then
39             -- asynchronous reset
40             syncdff_ff1 <= '0';
41             syncdff_ff2 <= '0';
42         elsif rising_edge(clk) then
43             -- data synchronization
44             syncdff_ff1 <= di;
45             syncdff_ff2 <= syncdff_ff1;
46         end if;
47     end process;
48 end architecture syncdff_arch;
```

Code A.4: spwram

```
1  --
2  -- Synchronous two-port RAM with separate clocks for read and write
       ports.
3  -- The synthesizer for Xilinx Spartan-3 will infer Block RAM for this
       entity.
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.numeric_std.all;
8
9  entity spwram is
10     generic (
11         abits:      integer;
12         dbits:      integer );
13     port (
14         rclk:       in  std_logic;
15         wclk:       in  std_logic;
16         ren:        in  std_logic;
17         raddr:      in  std_logic_vector(abits-1 downto 0);
18         rdata:      out std_logic_vector(dbits-1 downto 0);
19         wen:        in  std_logic;
20         waddr:      in  std_logic_vector(abits-1 downto 0);
21         wdata:      in  std_logic_vector(dbits-1 downto 0) );
22 end entity spwram;
```

```
23
24 architecture spwram_arch of spwram is
25     type mem_type is array(0 to (2**abits - 1)) of
26                         std_logic_vector(dbits-1 downto 0);
27     signal s_mem:    mem_type;
28 begin
29     -- read process
30     process (rclk) is
31     begin
32         if rising_edge(rclk) then
33             if ren = '1' then
34                 rdata <= s_mem(to_integer(unsigned(raddr)));
35             end if;
36         end if;
37     end process;
38     -- write process
39     process (wclk) is
40     begin
41         if rising_edge(wclk) then
42             if wen = '1' then
43                 s_mem(to_integer(unsigned(waddr))) <= wdata;
44             end if;
45         end if;
46     end process;
47 end architecture;
```

A.3 makefile and filelist

A.5: makefile

```
1
2
3 # Static CDC
4 ##########################################
5 run_vl: clean compile_vl cdc debug
6 run_vh: clean compile_vh clock cdc debug
7
8 ###### Define Variables ################
9 VLIB = ${QHOME}/share/modeltech/linux_x86_64/vlib
```

```
10 VMAP = ${QHOME}/share/modeltech/linux_x86_64/vmap
11 VLOG = ${QHOME}/share/modeltech/linux_x86_64/vlog
12 VCOM = ${QHOME}/share/modeltech/linux_x86_64/vcom
13
14 DUT = spwstream
15 ###### Compile Design ################
16 compile_vl:
17   $(VLIB) work
18   $(VMAP) work work
19   $(VLOG) -f scripts/filelist_vl
20
21 compile_vh:
22   $(VLIB) work
23   $(VMAP) work work
24   $(VCOM) -f rtl/flist.vh
25
26 ###### Generate a Clock Report ###########
27 clock:
28   qverify -od Output_Results -c -do " \
29   do rtl/directives.tcl; \
30   cdc setup -d $(DUT) -cdc_report cdc.rpt; \
31   cdc run -d $(DUT); \
32   cdc generate report cdc_detail.rpt; \
33   exit"
34
35 ###### Run CDC Analysis ###################
36 cdc:
37   qverify -od Output_Results -c -do " \
38   do rtl/directives.tcl; \
39   cdc run -d $(DUT); \
40   cdc generate report cdc_detail.rpt; \
41   exit"
42
43 ###### Debug Results ######################
44 debug:
45   qverify Output_Results/cdc.db &
46
47 ###### Clean Data #########################
48 clean:
49   qverify_clean
```

```
50    \rm -rf work modelsim.ini *.wlf *.log replay* transcript *.db
51    \rm -rf Output_Results transcript.cmd.tcl mal_cmds.tcl
52    \rm -rf myProj_SVA myProj_SVA.zpf cdc_detail.rpt
53
54 ############################################
55 # Regressions
56 ############################################
57
58 REGRESS_FILE_LIST = \
59    Output_Results/cdc.rpt
60
61 regression: clean compile_vl cdc
62    @rm -f regress_file_list
63    @echo "# This file was generated by make" > regress_file_list
64    @/bin/ls -1 $(REGRESS_FILE_LIST) >> regress_file_list
65    @chmod -w regress_file_list
66
67 (15 downto 12) <= (others => erg(32));
68 end if;
```

A.6: filelist.vh

```
1  rtl/spwpkg.vhd
2  rtl/spwrecv.vhd
3  bench/vhdl/spwlink_tb.vhd
4  rtl/streamtest.vhd
5  rtl/spwlink.vhd
6  rtl/spwram.vhd
7  rtl/spwrecvfront_fast.vhd
8  rtl/spwrecvfront_generic.vhd
9  rtl/spwxmit.vhd
10 rtl/spwxmit_fast.vhd
11 rtl/syncdff.vhd
12 rtl/spwstream.vhd
```

A.7: directives.tcl

```
1 # Define clocks
2 netlist clock clk -period 50
3 netlist clock txclk -period 40
4 netlist clock rxclk -period 40
```

```
5
6 # Add CDC constraints
7 cdc reconvergence on
8 cdc preference reconvergence -depth 1 -divergence_depth 1
9
10 # not inferred siganls
11 netlist port domain spw_si -clock rxclk -posedge
12 netlist port domain spw_di -clock rxclk -posedge
13 netlist port domain rxread -clock clk
14 netlist port domain txdata -clock clk
15 netlist port domain txflag -clock clk
16 netlist port domain txwrite -clock clk
17 netlist port domain time_in -clock clk
18 netlist port domain ctrl_in -clock clk
19 netlist port domain tick_in -clock clk
20 netlist port domain txdivcnt -clock clk
21 netlist port domain linkdis -clock clk
22 netlist port domain linkstart -clock clk
23 netlist port domain autostart -clock clk
```

A.4 Overview of the used AI-based Tools

TODO tabelle sauber machen

Tool	**Description of usage**
Deepl	Translation of singular words, grammatical and syntax checks throughout the whole paper
ChatGPT	introductory text SpaceWire, theoretical basis, Single Source Reconvergence of synchronizers

Table A.1: Usage of AI-Tools