POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master degree course in Electronic Engineering

Master Degree Thesis

Design and synthesis of a software-based transceiver PHY controller

Supervisor
prof. Maurizio Martina

Candidate
Annarita Ranco

Academic year 2017-2018

Abstract

Companies developing integrated circuits are expected to enhance their products' performance at every new release, while reducing size and power consumption. The demand for more elaborate and diverse functionality, together with a reduced time-to-market, irremediably raises costs and increases the probability of bugs. Even high-performance ASICs are not immune: the complexity of the design flow implies significant non-recurring engineering and production costs. Similar challenges affect the FPGA design flow, where the allocation of programmable logic requires considerable engineering effort. Moreover, due to the limited visibility of internal operations, isolating and back-tracing malfunctions are open challenges. Ericsson AB is exploring novel approaches to deal with this complex ecosystem.

This thesis investigates the feasibility and the benefits of a flexible design approach, by developing and characterizing a Proof-of-Concept (PoC) transceiver handler for high-speed link applications. The flexibility lies in the software-based controller, exploited to handle the reset and dynamic reconfiguration of a transceiver physical layer (PHY). The objective of the software implementation is to simplify error detection and *on-the-fly* modification compared to a traditional HW-based controller. The firmware, running on a Nios II soft-core processor, drives the control signals while monitoring the transceiver's status. Unexpected synchronization losses are handled by a dedicated Interrupt Service Routine.

The correct HW/SW interaction has been tested through simulation, whereas the software profiling proves that the timing requirements are met (only $167\mu s$ are spent on the reset sequence). Finally, the PoC has been benchmarked against an analogous system with a traditional HW-based controller, to evaluate the drawbacks of the introduction of a soft-core processor (in terms of logic utilization and power consumption).

Despite the promising engineering effort reduction, further research is required to scale up the system and move from the PoC stage towards product release.

Keywords: Proof-of-Concept; FPGA; Nios II; embedded processing; PHY

Acknowledgements

A sincere thank is reserved to my Ericsson supervisors: Anders Nordström and Tume Wihamre. Their recommendations, combined with the various challenges I have faced, contributed to a professional growth which goes beyond this thesis' achievements. I am also grateful to Mårten Kidd and Sunil Kallur Ramegowda from the Intel support team for their endless suggestions, and to Michael Rosendahl for sharing his expertise.

I wish to thank both my KTH examiner and supervisor, Prof. Ahmed Hemani and Dimitrios Stathis for their precious observations. A great thank is then reserved to my supervisor from Politecnico di Torino, Prof. Maurizio Martina, who has been closely monitoring the project's development on a regular basis.

Within Ericsson's multicultural and diverse environment, several people inspired me by sharing their experience, their culture and their vision of life. In particular, Matteo Bernini and Enrique Cobo played an irreplaceable role: their constant optimism and motivation guided me through this journey.

Above all, I owe a debt of gratitude to my mother and my best friend Francesca for their unconditional love and trust. During the crucial moments, when failure seemed close, they encouraged me to endure by reminding that giving up was not an option.

Contents

List of Tables

List of Figures

List of Acronyms

AHB Advanced High-performance Bus

ALU Aritmetic Logic Unit

AMBA Advanced Microcontroller Bus Architecture

API Application Programming Interface

APSoC All-Programmable SoC

ASIC Application-Specific Integrated Circuit

ATX Auxiliary Transmit

Av-MM Avalon Memory-Mapped

AXI Advanced eXtensible Interface

BER Bit Error Rate

BIST Bulit-In Self Test

CDC Clock Domain Crossing

CDR Clock Data Recovery

CFSM Communicating FSM

CPRI Common Public Radio Interface

CTLE Continuous Time Linear Equalization

DAC Digital to Analog Converter

DFE Decision Feedback Equalization

DSP Digital Signal Processor

EDS Embedded Design Suite

EIC	External Interrupt Controller
EPE	Early Power Estimation
ESD	Electrical Static Discharge
FFE	Feed-Forward Equalization
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
fPLL	fractional PLL
FSM	Final State Machine
GPP	General Purpose Port
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HFSM	Hierarchical FSM
HGS	Hierarchical Graph Scheme
HPP	High Performance Port
HPS	Hard Processor System
IC	Integrated Circuit
IDE	Integration Development Environment
IP	Intellectual Property
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISI	Inter-Symbolic Interference
ISR	Interrupt Service Routine
ITRS	International Technology Roadmap for Semiconductor
I2C	Inter-Integrated Circuit
JTAG	Joint Test Action Group

LFSR	Linear Feedback Shift Register
LOF	Loss Of Frame
LON	Loss Of Network Timing
LOS	Loss Of Signal
MAC	Media Access Layer
MDI	Medium-Dependent Interface
MMU	Memory Management Unit
MPU	Memory Protection Unit
NRE	Non-Recurring Engineering
OCM	On-Chip Memory
OCT	On-Chip Termination
OS	Operating System
OSI	Open System Interconnection
PCS	Physical Coding Sublayer
PD	Phase Detector
PFSM	Parallel FSM
PHFSM	Parallel Hierarchical FSM
PHY	Physical Layer
PIO	Parallel Input/Output
PL	Programmable Logic
PLL	Phase-Locked Loop
PMA	Physical Medium Attachment
PMD	Physical Medium Dependent
PoC	Proof-of-Concept
PRBS	Pseudo-Random Binary Sequence
PS	Processing System

Qsys Quartus System Integration Tool

RAM Random Access Memory

RBI Ring Bus Interface

RE Radio Equipment

REC Radio Equipment Control

RFID Radio-Frequency Identification

ROM Read-Only Memory

RTL Register Transfer Level

RX Receiver

SBT Software Build Tool

SerDes Serializer/Desirializer

SNR Signal to Noise Ratio

SoC System-on-Chip

TLB Translation Lookaside Buffer

TTM Time-To-Market

TX Transmitter

UART Universal Asynchronous Receiver and Transmitter

USB Universal Serial Bus

VCO Voltage Controlled Oscillator

VGA Variable Gain Amplifier

VHDL Very High Speed Integrated Circuit HDL

Chapter 1

Introduction

Nowadays, the demand for electronic devices is increasing (even if at a reduced rate) [1] and consequently new challenges arise [2]. Indeed, the trend is to realize more efficient products characterized by a reduced area, whose complex functionality addresses various fields (automotive, medical, communications, Internet of Things, and so on). Moreover, with the advent of the so-called "Post-Moore" area, the cost and the size are no longer the unique parameters to keep into account: low power consumption, variety of functions, introduction of new materials and technological process are instead the key factors [3] [4].

Companies developing Integrated Circuit (IC)s, in order to deal with this complex ecosystem, have to adopt new approaches: a possible way is to increase degree of flexibility of the products. One company which is investigating flexible solutions to address these challenges is Ericsson AB. This thesis project was proposed as a contribution to the evaluation of the feasibility and the benefits of such approach.

1.1 Problem Statement

Traditionally, ASIC were designed in order to fulfill these requirements, in addition to high-performance and low power consumption. The ICs design is tailored according to different costumer needs. However, because of the increasing demand for enhanced performance it is necessarily to deal with higher complexity which irremediably leads to higher costs. Apart from the extremely high Non-Recurring Engineering (NRE) costs related to the ASIC design, the complexity of the design itself leads to highly probable bugs. This complexity lies in the sequence of intermediate steps of the design flow. These steps can be grouped into two macro-phases: front-end (logical design) and back-end (physical design) [5]. ASICs lack of flexibility by definition, since the design is *hardened* in the silicon. Additionally, with the traditional ASIC implementation, the visibility of what is happening within the circuit during regular operations is limited and it is challenging to trace eventual errors. One company that has experienced this problem is Ericsson AB. An unexpected behavior was found in one of their ASIC, internally designed to implement the proprietary xIO-s protocol (for detailed information refer to *Case of study: Ericsson's ASIC* section). Despite the conspicuous amount of resources invested in the debug phase (in terms of time and human resources), isolating and solving the problem was extremely problematic. The bug was located within the traditional Final State Machine (FSM) developed through Register Transfer Level (RTL) design to handle an high-speed link. The bug was triggered by a rare combination of corner cases, and the final fix was to directly bypass the FSM. Consequently, all the required operations were performed by external Digital Signal Processor (DSP)s (by running SW) at the cost of reduced performance. The former *in extremis* solution would not have been necessarily in case of more versatile system. This study case, convinced Ericsson AB to investigate novel approaches able to overcome these drawbacks.

However, their concern is not limited to the ASIC domain. A similar analysis is also valid for Field-Programmable Gate Array (FPGA) whose purpose is no longer bounded to prototype development. Besides FPGAs flexibility and reusability, the performance gap with ASICs is also significantly reduced. However, the complexity is an issue even with programmable logic: a deep optimization of the logic resources is required in order to meet the desired performance. This implies to perform a subdivision of the Programmable Logic (PL) within the FPGA at design-time. Further modifications are not trivial and resource consuming. Nevertheless, another problem that Ericsson AB aims to solve is error detection and tracing to the root cause. Indeed, tracing errors is extremely demanding and, at the state-of-the-art, it requires a second or third order analysis of the log information provided by their system. Consequently, the verification effort and cost is increasing as well [6].

Although the Ericsson AB firsthand experience was presented, it is highly probable that the whole IC industry may experiment analogous situations. In other words, the real problem is to find a flexible solution which, despite the unavoidable performance degradation, is still able to meet the requirements.

An emerging solution is the usage of reconfigurable architectures which exploit FPGA [7]. The main idea is to dynamically configure the programmable logic of the FPGA in order to significantly increase the HW flexibility. On the other hand, the reconfiguration introduces an initial overhead which affects the performance. Moreover, an internal subdivision of the logic is required (re-configurable and static modules) hence the complexity increases as well. Until further researches will improve this promising technique, the problem will still be alive.

1.2 Objectives

As previously mentioned, this problem may involve many IC companies operating in various fields, thus, a general and portable solution is desirable. However, as a master thesis, the scope is narrowed to target high-speed link applications. Starting from the Ericsson AB case of study, the FSM responsible for the high-speed link handling should be implemented through SW running on a soft-core processor synthesizeable on PL.

Indeed, the purpose of the thesis is to propose a more flexible system architecture without significantly compromising the performance. Hopefully, this conceptual approach may be valid for other IC-design areas. More specifically, the flexibility is given by a SW implementation of the system controller. The embedded controller should handle the reset and partial reconfiguration of the physical layer of a transceiver. Hence, the generality lies in the design and development of the firmware: by changing it, in theory, any FSM can be substituted.

Because of the experimental nature of this thesis, the main goal is to realize and characterize a PoC of the previously mentioned system. The development targets the Intel Arria 10 System-on-Chip (SoC)[1], which is selected in order to simplify and speed-up the design process itself [8]. The PoC's characterization should enlighten pros and cons of the proposed HW/SW partition in order to determine possible area of applications and to suggest future work. This main goal can be split into intermediate tasks:

- Development flow standardization to perform system design comprehensive of HW/SW partition:
  - Hardware Description Language (HDL) description of the HW portion of the system (by also including the embedded micro-controller) which models the system's data path.
  - Development of the firmware to be downloaded into the micro-controller. The double aim is to check the status signals from the PHY and to drive the command signals to the PHY in order to control it. Basically, the firmware should model a FSM able to reset the PHY within few ms. Additionally, the reconfiguration feature of the PHY should be enabled by performing a sequence of atomic *read-modify-write* operation to write the reconfiguration registers.
- Functional simulation of the developed system to test the correct behavior both in operative mode and in corner cases such as loss of synchronization.
- System compilation and synthesis for the targeted device which is then *programmed* via micro-blaze USB connection.
- System characterization in terms of logic resource utilization, power consumption and software profiling.
- System benchmarking against a system with a traditional HW-based reset controller.

A PoC, able to satisfy such requirements, may be a starting point for future researches: a HW supervisor with trace-back capability and able to monitor ASICs and/or FPGAs interfaces realtime would be desirable.

[1]A SoC consists of the integration of one or more core (processor and/or DSP), a memory system, peripherals and external interfaces on the same chip.

1.3 Outline

The report contents are organized into six chapters. *Chapter 2* gives a brief overview of the IC state-of-the-art, presents a concrete case of application and described the fundamental concepts the thesis is based on, by also including an analysis of related work.

Afterwards, *Chapter 3* describes the methodologies and tools used for the PoC design, development and final characterization. A detailed description of the target SoC and soft-core processor is also provided.

The system implementation is fully covered by *Chapter 4* where separate subsections are reserved for the HW and SW architecture description, with a particular attention on the design choices.

After that, *Chapter 5* collects the results of the functional simulation, PoC benchmark and characterization.

To conclude the report, *Chapter 6* evaluates the results in order to determine the accomplishments of this thesis. Finally, based on this analysis, suggestions for improvements and future works are pointed out.

Chapter 2

Background and Literature Study

This chapter presents the fundamental knowledge this thesis is based on, starting from an overview of the IC trend. This overview is useful to contextualize this thesis, whose investigation starts from a case of study proposed by Ericsson AB. Afterwards, an analysis of the related works is carried out, by focusing on flexible and reconfigurable system controllers.

2.1 State-of-the-art of Integrated Circuits

Starting from the 1980s, thanks to the miniaturization process, the area scaling allowed more and more transistors to be integrated on a single chip according to Moore's prediction. The direct consequence was a significant enhancement of ICs functionality. The increased number of transistors per unit area led to dense ICs, customized for a specific purpose, and able to satisfy extremely high performance. In parallel, the digital design flow and system modeling considerably improved.

A description of the IC industry's evolution can be found in the introduction of the 2015 executive report of the International Technology Roadmap for Semiconductor (ITRS) [9]. This evolution comprehends different phases and the peak of the IC industry growth was reached during the 90s with an average pace of $17\%/year$. During this phase, ASICs production played a key role in the industrial revenues. Indeed, application specific IC were able to meet the required performance at a limited size due to their density of integration. Nevertheless, ASIC production has always been profitable only for a large scale of application because of the NRE costs related to both design and verification process [10]. Production costs are also recognizable [11]. Moreover, higher NRE combined with shorter Time-To-Market (TTM) increased the productivity gap [12]. This gap represents the divergence between the technological development (determined by the progress of technology) achievable on paper, and the actual one which is limited by the available resources (time and money). For what concern the design phase, such gap is still under control, whereas it is no longer negligible for the verification process [13]. Hence, additional effort should be primarily spent on the optimization of the verification process itself to reduce both time and human resources.

The traditional design flow, introduced by Gajski and Kuhn [5], is divided into two phases called front-end and back-end respectively. The front-end design starts with the RTL definition, followed by the functional verification, the correspondent synthesis, till the gate level verification. Therefore, it can be considered as the logical design. On the other hand, the physical design (back-end) is then implemented through placement and routing, post layout verification and final mask generation. Because of the complexity and the related costs, the trend has always been to use FPGA for the preliminary investigation phase and for prototyping [14]. The benefit is that "FPGA based implementations provide the flexibility of re-programming and quick delivery of the product to the market." [15]. However, this traditional division between FPGA and ASIC is not entirely valid at the moment. Indeed, the performance and the time spent on design and verification of large size FPGAs are comparable to ASICs one. The intrinsecally flexible FPGAs are becoming attractive also for small-end applications [16].

Moving towards SoC production is another emergent approach to deal with high complexity, short TTM and the overall cost explosion. An SoC-based design is characterized by the instantiation and connections of commercialized Intellectual Property (IP)s [9]. These IPs may be directly used as black boxes, or they may also be customized, through HDL code, with respect to the design's requirements [17]. The major advantage, of this technique, is the high-level approach of the system's design: the key is *which* functional blocks need to be implemented and not *how* (implementation details). Conversely, the main drawbacks are as follows:

- The system design approach requires a global optimization which may differ from locally optimized solutions. Moreover, the prediction of the individual choices' impact on the overhall system is not straigthforward [18]. Thus, the HW/SW partition plays a key role [12], especially for the control unit.
- The high level usage of the IPs as black boxes may shield the monitoring of the system activity [19] and, consequently, it may affect the debugging and the testing processes [20].

Within this complex ecosystem, the analysis of an optimal HW/SW partition plays a key role. The idea of a software-based controller was triggered by a malfunctioning experienced by Ericsson in one of their ASICs, as detailed in the following section.

2.2 Case of study: Ericsson's ASIC

Different types of radio system (LTE, WCDMA, GSM) can be represented as a continuous stream of IQ data, whose information can be extracted from the carrier. The ASIC under interest is composed of many serial links to implement Ericsson's proprietary xIO-s protocol. The latter is a frame based protocol used to provide multiple services such as the transportation of network timing, IQ data and packet data. Such protocol is based on basic frames, hyper frames and radio frames, and it exploits the 8B/10B encoding [22]. Basically, the xIO-s protocol is similar to the Common Public Radio Interface (CPRI) [1] where the lines carry both IQB data and IQ control data.

In order to provide the described protocol, each serial line is connected to the ASIC's core block through a SerDes. A SerDes configuration handler is also needed. A simplified block diagram with one serial link is illustrated in Figure 2.1.

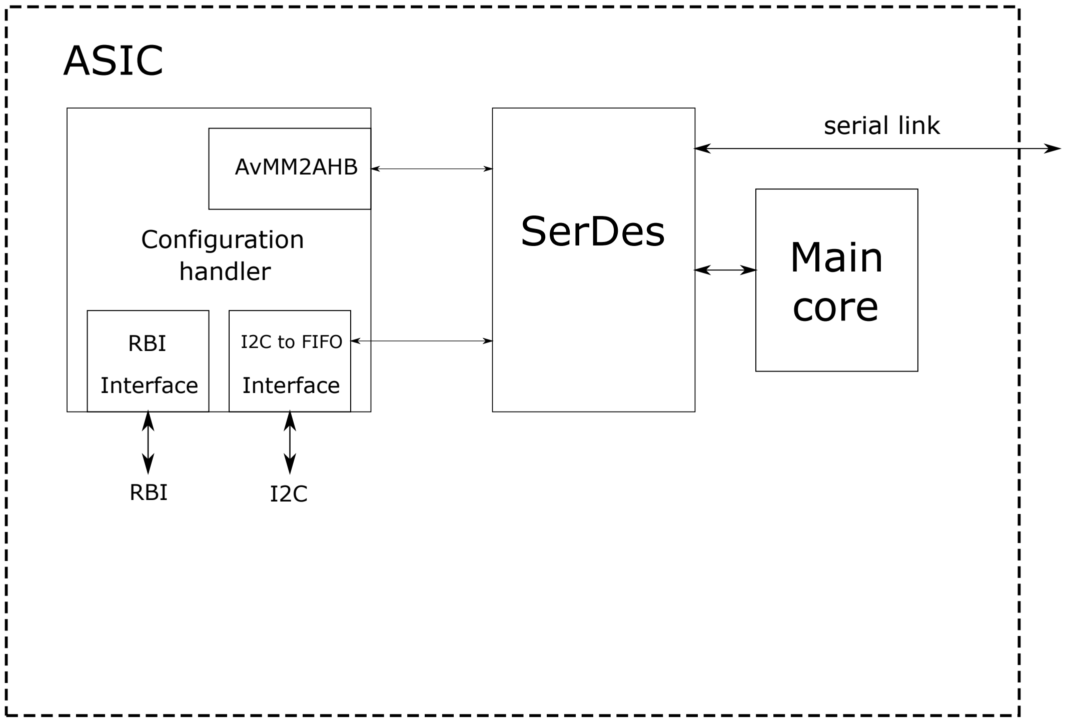


Figure 2.1: Simplified block diagram showing only one of the SerDes and the corresponding configuration handler within the ASIC.

The core blocks and the SerDes are configured independently. For what concern the core blocks, the configuration is performed through ring bus mails, whereas for the SerDes there is also the possibility to access the internal registers via Inter-Integrated Circuit (I2C). Thus, the configuration handler should support internal register interface (read and write) both via Ring Bus Interface (RBI) and I2C. Moreover, in order to communicate to the SerDes' registers, an Av-MM to Advanced High-performance Bus (AHB) bridge is required. With this system, the SerDes power-up, reset and reconfiguration are performed. Finally, for a proper handling, it is also necessary to take care of the Clock

[1]CPRI provides a standard framework to allow the communication between the Radio Equipment (RE) and the Radio Equipment Control (REC). The definition of the CPRI specifications [21] has been achieved through the cooperation of Ericsson, Huawei, NEC, Alcatel and Nokia.

Domain Crossing (CDC), since the SerDes' clock is asynchronous with respect to the system's clock.

During the regular activity the frequency of sending/receiving data depends on the serial link configuration which establishes the division factor (2X, 1X, or 0.5X) for the reference frequency (491.52 MHz). A reset is needed whenever one of the following conditions appear:

- The link does not perform the clock data recovery successfully.
- The main core is not able to find the frame synchronization on the link (for example due to cross talk or corrupted data).
- The synchronization is somehow lost.

In order to monitor the correct behaviour, dedicated status signals (supported by hardware) are kept into account:

- Loss Of Signal (LOS): an alarm is activated if the Bit Error Rate (BER) overcome the accepted threshold value.
- Loss Of Frame (LOF): an alarm is activated if the frame synchronization is lost.
- Loss Of Network Timing (LON): an alarm is activated if the timing synchronization is lost.

The LOS and LOF need to be inactive in order to properly receive packets, elsewhere a re-synchronization is required. Instead, in case of LON alarm there is no need for an HW re-synchronization and only a *bad-timing* indication is generated.

Therefore, the introduction of a controller able to monitor LOS, LOF and LON and to eventually perform a re-synchronization is required. This task is performed by a traditional FSM, whose simplified transition diagram is shown in Figure 2.2. At first, a power-up reset sequence is performed and the CDR is enabled. Frame synchronization is achieved through the following states (according to the protocol) and, in case of loss, the system is restarted.

An accurate debug revealed that the FSM needed to be reviewed with a consequent system re-designed since it was affected by unexpected behaviours. The bug localization and analysis required a great effort in terms of both time and human resources. It was problematic to reproduce such corner case because, even with an induced failure of the SW interface, it was affecting only 6% of the tests. Hence, a huge amount of trials was necessary to properly debug. Finally, the adopted solution was to bypass the FSM and to directly handle the SerDes through the SW interface. This example leads to an interesting consideration: if the system had been flexible enough to allow localized modifications within the FSM, many resources would have been saved, and the complete re-design would have been avoided.

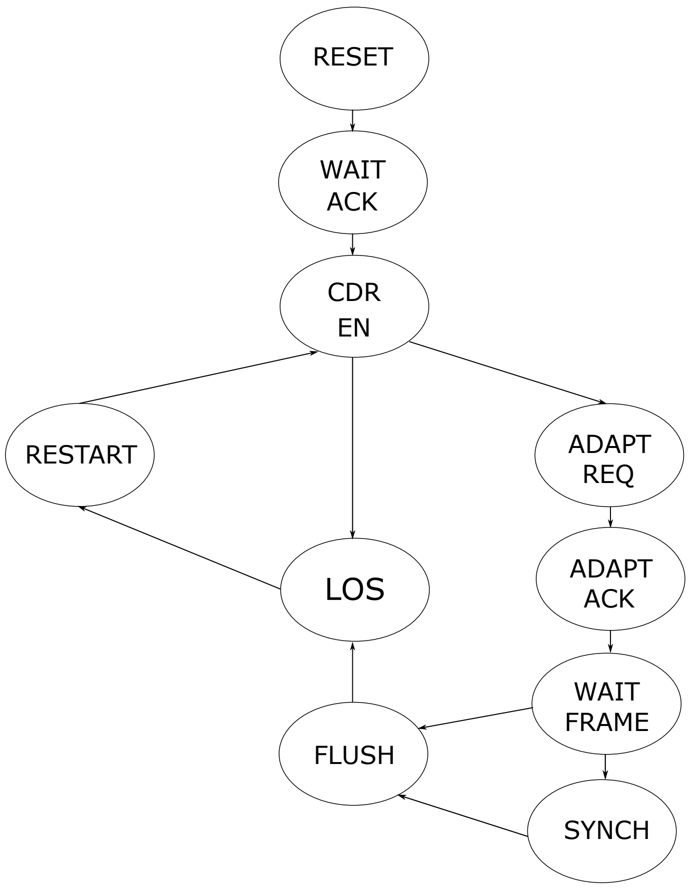


Figure 2.2: SerDes reset FSM simplified state transition diagram.

The key idea to overcome the problem is to realize a PoC able to target this application, but where the FSM is implemented through SW, so that, it is easily changeable. The PoC will be realized on FPGA where a transceiver (physical layer) reset and partial re-configuration will be handled at a SW level to model the SerDes management. In fact, the transceiver's PMA main component is a SerDes where the Serializer is part of the transmitter, whereas the Deserializer is part of the receiver.

2.3 Transceiver PHY

2.3.1 Introduction

A mean of communication characterized by a single entity port can be defined as a link. A link can be composed of a variable number of transceiver channels. A transceiver comprehends different components, in particular, the PHY is the circuitry which implements the physical layer functions of the Open System Interconnection (OSI) Reference model.

The former standardization was created in the 1980s to allow the communication between two generic systems obeying such standard. According to the OSI model, the networking components of every device (both HW and SW) can be represented as a seven layer structure as shown in Figure 2.3.

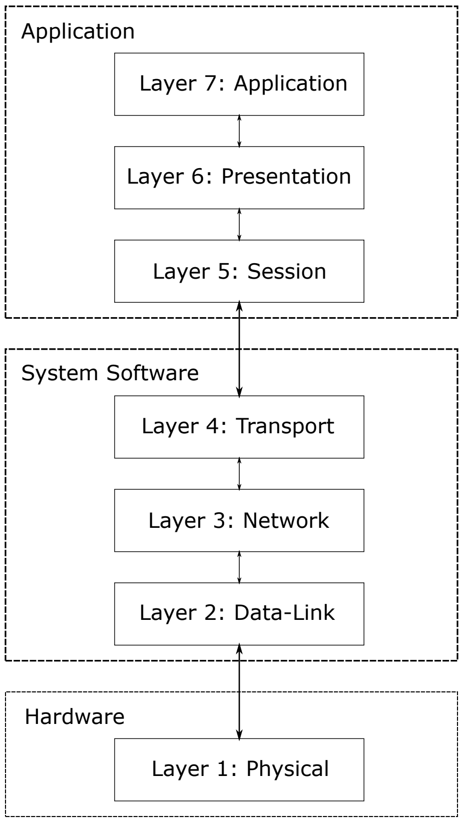


Figure 2.3: OSI seven layers model block diagram (inspired by [24]).

Specifically, the hardware components are represented by the physical layer (layer 1). “The Physical Layer provides the mechanical, electrical, functional, and procedural standards to access the physical medium.”, as directly stated by Day and Zimmermann [23]. The information to be transmitted is elaborated from the application layer down to the physical layer, and then, it is sent to the receiving network device over a transmission medium. Conversely, on the receiver side, the information collected at layer 1 is then *unpacked* to reach upper layers. Thus, the physical layer is directly connected to the communication medium in order to handle the voltage signal (low and high levels represent logical 1 and 0), both in transmission and reception [24]. An example of data transmission between two devices is shown in Figure 2.4, where the interaction between data link and physical layer is clearly illustrated.

By focusing on the physical layer architecture, it is composed of two main building blocks: TX and RX, which are both "mixed-signal circuits in nature" [25]. Indeed, the TX

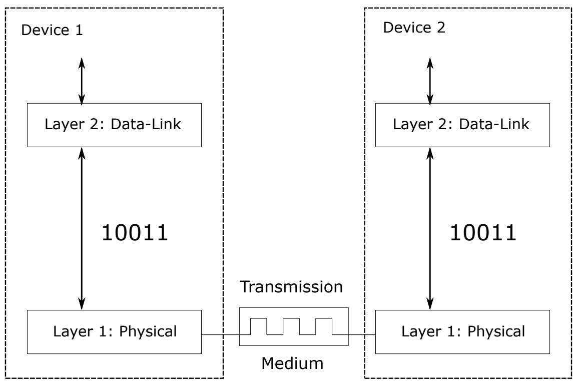


Figure 2.4: Data flow diagram between two network devices. The focus is on the lower levels of the OSI model: the data link layer (binary digits) and the physical layer (voltage level). The diagram is inspired by [24].

receives a stream of parallel data and converts it into an high-speed serial signal, which then reaches the RX side. The RX, in order to perform the reverse operation, needs to recover not only the data, but also the clock information. The conversion is internally performed by a serializer (in transmission), so that, from the input digital stream it is possible to obtain the high-speed analog signal.

An ethernet model can be taken as reference to clarify the link's architecture. The main components are the PMA and the Physical Coding Sublayer (PCS). Moreover, as it is shown in the block diagram[2] of Figure 2.5, the Physical Medium Dependent (PMD) is used as interface between the PMA and the physical ethernet cable through the Medium-Dependent Interface (MDI).

2.3.2 PMA

As previously introduced, the PMA is the analog front end of the transceiver, where the TX is responsible for the data serialization, whereas the RX takes care of data deserialization and clock recovery.

A termination circuit is a common requirement for both sides (TX and RX). This termination component is, generally, used to reduce the reflection phenomena that can affect the channel. It can also be used to identify whether the receiver is ready to accept the data stream. For example, this evaluation system is used in the PCIe case. Different types of termination are determined by the voltage level it is tied to. Typical examples are the power supply or ground; however, it is also possible to use a different DC-voltage, whose level is in between these two. In case of DC-coupled links, the termination type

[2]The presented system model is valid for devices with at least 100 Mbps data rate, such as Fast Ethernet and Gigabit Ethernet systems [26].

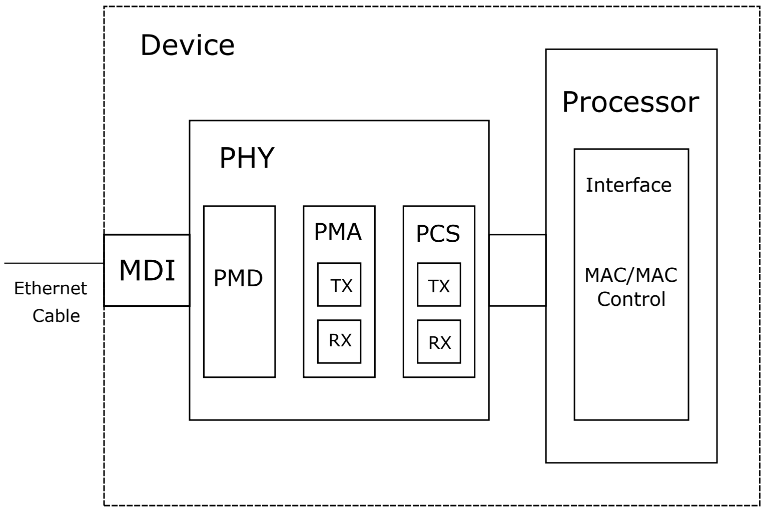


Figure 2.5: Simplified ethernet block diagram (inspired by [26]).

should be the same at both sides, whereas in case of AC-coupling RX and TX are allowed to have different termination types. In theory, both passive resistors and active transistors are valid components for the termination circuit, but because of the high-speed, the transistor's dependence of V_{DS} may cause unacceptable non-linearities. Thus, in most of the cases a resistor in series with a transistor is chosen as termination circuit. Finally, in order to preserve the resistance value even in case of variations (regarding the power supply, the temperature or the silicon process itself), a calibration circuit is also added.

Transmitter

A transmitter serializer and a transmitter buffer are the main components of the PMA's transmitter. They create the high-speed serial data and guarantee the correct voltage swing between the differential output pins (TXP and TXN). The serializer is responsible for the parallel-to-serial conversion, whereas the transmitter buffer performs both Feed-Forward Equalization (FFE) and line driving. The size of the data stream may vary according to the protocol: PCIe specifications define, for example, 8-bit or 10-bit stream of data. Additionally, Electrical Static Discharge (ESD) components may be used. These ESD are nothing but protecting diodes to shield the devices connected to the pad. However, in certain high-speed applications, the parasitic capacitance introduced by the ESD is no longer negligible, and it directly affects the link performance. A typical block diagram is presented in Figure 2.6.

For what concern the serializer architecture, many choices are available in relation to the number of taps. Moreover, the serializer requires a higher-frequence and a lower-frequence clock to perform the parallel-to-serial conversion and, typically, they are both given as input reference from an external clock generation block. A reference architecture

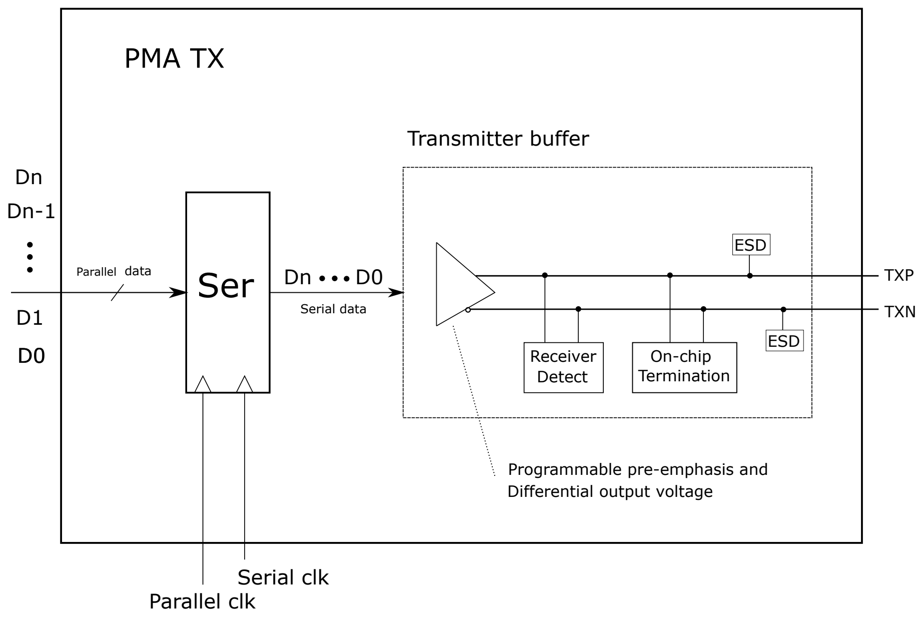


Figure 2.6: Typical PMA transmitter block diagram (inspired by [25]).

is described in [27], where a 8-bit wide input data stream is serialized by the architecture shown in Figure 1 of the same paper. The serialization is split into a chain of steps: firstly an 8:2 multiplexer separates the data stream into odd and even, which represents the input data of a 4-tap Finite Impulse Response (FIR) filter. This filter is made of four shift registers and 2:1 multiplexers, and it is also responsible for the FFE, with the aim to minimize the Inter-Symbolic Interference (ISI). In practice, the equalization consists of the output swing adjustment to preserve the integrity of the signal. In general, 2-tap FFE is sufficient for channels with high signal integrity, conversely for high-loss channels 3-tap or 4-tap FFE is required. With the data stream already split, the equalization can be processed at half rate. After this step, the equalized data flow through the 2:1 multiplexers to finally generate the serial output at full-rate. The output signal from the 2:1 multiplexers has to drive the input load of the line driver, and it may be large. Thus, it is important to include a programmable pre-driver able to adapt with respect to the high-speed specifications.

Receiver

The PMA receiver, in order to perform clock and data recovery, and equalization, acts as a mirror of the transceiver side, whose main components are, precisely, a receiver buffer and a receiver deserializer. Figure 2.7 shows the typical PMA receiver block diagram. The former system comprehends both digital and analog circuits. In particular, the Variable Gain Amplifier (VGA), On-Chip Termination (OCT), power management and CDR belong to the digital portion, whereas the Continuous Time Linear Equalization

(CTLE) and Decision Feedback Equalization (DFE) belong to the analog part.

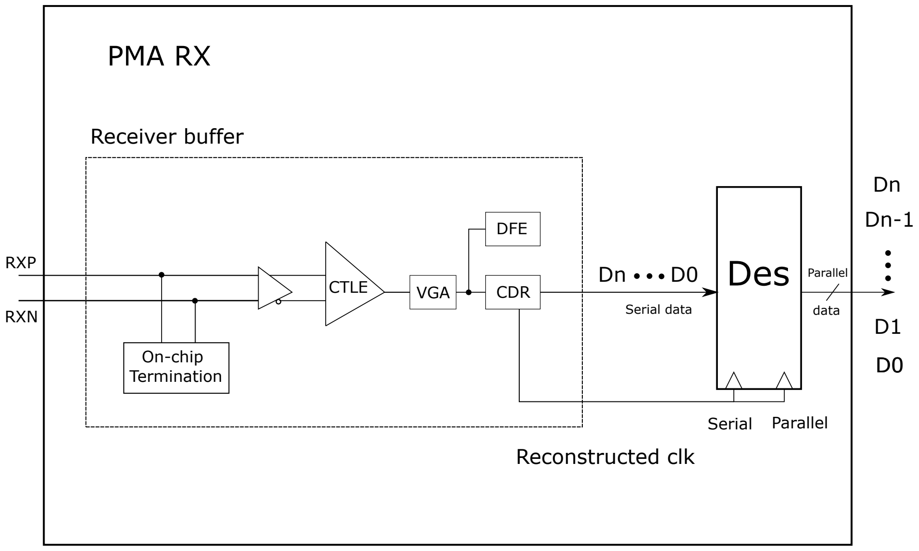


Figure 2.7: Typical PMA receiver block diagram (inspired by [25]).

With a continuous increasing in the data rate the ISI phenomena is no longer negligible, thus, many researches were carried out regarding the system's equalization. Within the PMA, the equalization is performed by the CTLE and DFE units.

For what concern the CTLE, the Cherry-Hooper [28] is one of the most used topology, due to its high bandwidth. Anyway, many other architectures were investigated, especially for adaptive equalizer [28], [29], [30], [31]. In practice, the CTLE recovers from signal attenuation due to the channel by supplying a frequency, whose peak is close to the Nyquist frequency. In other words, the high-frequency component of the input signal is amplified by the equalizer circuit, so that the low-pass behaviour of the physical medium is compensated. If the CTLE supports DC gain circuitry, the amplification is constantly applied through the whole spectrum, conversely in case of AC gain circuitry, the amplification mainly affects the high-frequency spectrum. The unbalanced content of the high-frequency components of the CTLE output waveform is well-analysed in Figure 4.23 of [25]. Finally, this compensation results in a wider aperture of the receiver eye diagram.

The DFE is also widely studied in literature, and a detailed description is provided by Bottacchi [32]. To give a conceptual introduction, the DFE is used to minimize, and possibly cancel, the post-cursor ISI: the previously received bits are re-used in order to create a weighted value to be added, or subtracted. With this strategy, the DFE is able to selectively amplify the high-frequency components, while keeping the noise component untouched. As a consequence, the Signal to Noise Ratio (SNR) is significantly improved. According to the DFE architecture, a fixed number of taps is present: an example is the 4-tap architecture [33].

According to this architecture, the DFE's main components are: adder, data sampler and Digital to Analog Converter (DAC)s. The sample of the incoming data (indicated as x_n), re-scaled by the VGA (g factor), is added (or subtracted) to the DACs output, so that the result (indicated as y_n) can be processed by the data sampler. The decision (indicated as d_n) represented as a +1 or -1 is then looped back to the DACs. Finally, the DACs receive the DFE coefficients (indicated as c_i), so that, through the DACs output the residual effect of the current bit can be compensated. Such behaviour may be modeled by the following transfer function:

$$y_n = g \cdot x_n + \sum_{i=1}^{4} c_i \cdot d_{n-1}$$

As previously mentioned, the CDR covers the most critical tasks of the receiver: data and clock recovery. Hence, it contributes to an increasing in the eye-diagram aperture, with a consequent improvement in the incoming signal tracking. Specifically, in presence of jitter, the CDR expands the timing margins by optimally settling the sampling edge of the clock. Among all the available CDR's architectures, the essential component, in charge of recovery, is the Phase-Locked Loop (PLL). The block diagram of a typical CDR's architecture is shown in Figure 2.8 where the key role is played by a phase comparator, usually referred as Phase Detector (PD), a low-pass filter and a controlled oscillator, which is, generally, a VCO.

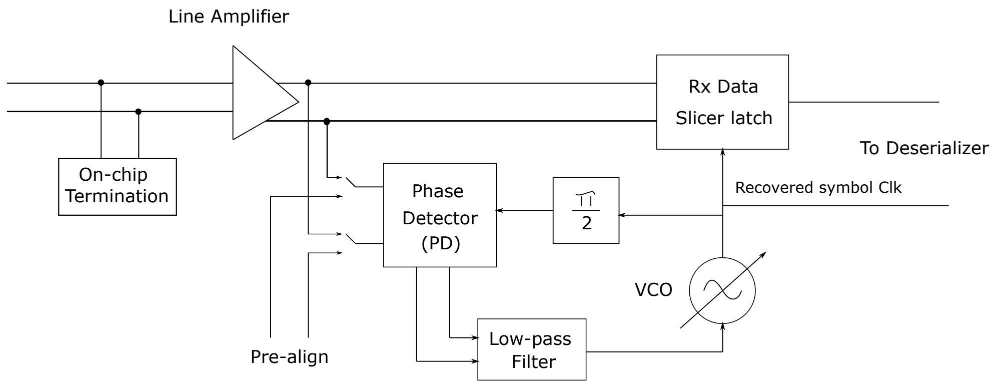


Figure 2.8: Typical CDR block diagram inspired by [34]. A current controlled oscillator may be used as an alternative to VCO.

The relative phase error is evaluated, by the phase detector, as the difference between the input reference frequency and the VCO. Such phase error (which can be identified with θ) is then filtered out by the low-pass filter, and it is ultimately looped back as VCO's input. Indeed, the function of the VCO is to convert the received value into outputs in fase and in quadrature with the input reference. "the quadrature edge is targeted to align with the center of the data eye for sampling the symbol values" [34]. The basic parameters for the PLL characterization are the lock-in and capture ranges. The lock range corresponds to the $\Delta\theta = 0$ condition, that is to say the interval in which the input reference can be optimally tracked. On the other hand, the capture range identifies the transition between lock and unlocked states.

The CDR's performance principally depends on the PD's architecture. Indeed, there are two different categories of PD:

- Linear phase detector: the relationship between the input and the phase error is modeled by a linear transfer function.
- Binary phase detector: only the phase error's sign is extracted as useful information. Hence, this behaviour is modeled by a step function.

Binary phase detector are widely used in CDR architectures and, within such category, the architectures proposed by Alexander [35] and Mueller-Muller [36] are remarkable. The Alexander topology establishes whether the clock edge is *early* or *late* with respect to the data edge, based on three samples (S_i) taken by consecutive clock edges (falling and rising). Thus, by aligning clock and data edge, the remaining edge can be settled in the middle of the data, in order to increase the timing margin. The corrective actions to be taken are described by Razavi [37], and they can be modeled by the following equations:

$$\text{Early:} \quad S_1 \oplus S_2 = 0 \;\&\; S_2 \oplus S_3 = 1 \quad => \quad \text{Clk freq reduction}$$

$$\text{Late:} \quad S_1 \oplus S_2 = 1 \;\&\; S_2 \oplus S_3 = 0 \quad => \quad \text{Clk freq increment}$$

$$\text{Others:} \quad S_1 \oplus S_2 = S_2 \oplus S_3 \quad => \quad \text{No Clk adjustment}$$

A different approach, which benefits systems with a reduced power consumption, is exploit in the Mueller-Muller PD. Indeed, in this case, the output error can be evaluated at half-rate since the information related to both current and previous data is exploited. The former behaviour can be modeled by the following equation:

$$Dt_n = D_n \cdot D_{n-1}(\theta_n - \theta_{n-1})$$

by identifying D as the sampled data (current n, and previous n-1), and θ as the phase error. Basically, if there is no change in the phase error in both cases, there is no need for updates; conversely, whenever the sign of the error phase is modified, a phase error information is evaluated based on the previous and current data.

2.3.3 PCS

The PCS acts as an interface between the Media Access Layer (MAC) and the PMA, hence its main function is to encode/decode the parallel data stream from/to the MAC sub-layer. Additional data manipulation, such as scrambling and descrambling, may be performed by the PCS too. Alike the PMA, the PCS can be split into transceiver and receiver side, which can be considered *mirrored.* Moreover, the system architecture varies according to the implemented protocol such as: fast Ethernet, Gigabit Ethernet or PCIe. For what concern Ethernet protocol, many studies can be found in literature with respect to different data rate, for example [38] targets 40-Gbps Ethernet, whereas [39] accurately targets both 40-Gbps and 100-Gbps Ethernet. Instead, [40] describes a typical PCS architecture for PCIe. The latter is shown in the block diagram of Figure 2.9.

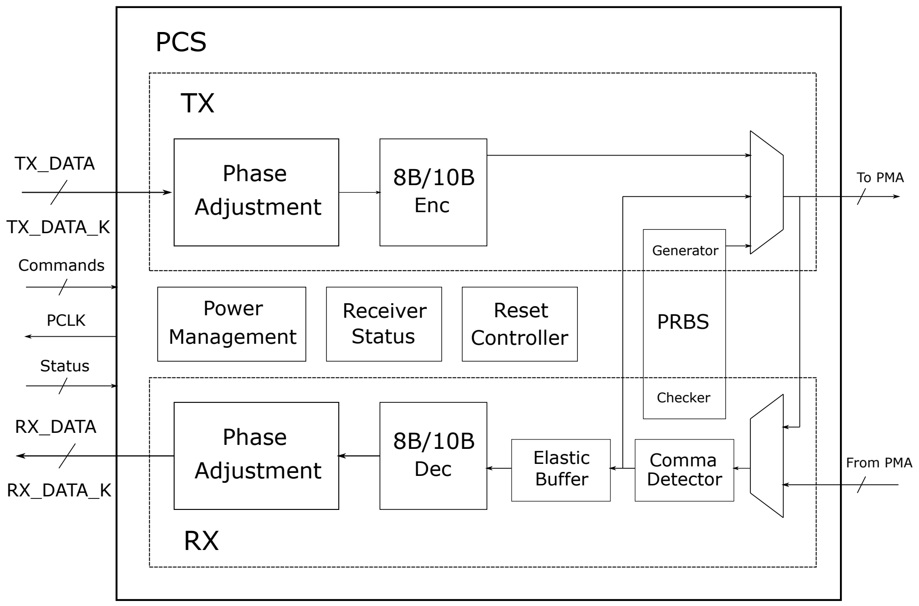


Figure 2.9: Typical PCS block diagram comprehensive of transmitter and receiver (inspired by [40]).

The PHY/MAC interface, according to the PCIe protocol, comprehends not only the transmitted and received data information, but also status and command signals, and finally, a synchronization signal indicated as `PCLK`.

- The `PCLK` frequency is modified so that the signaling rate can be adjusted while preserving the data path parallelism.
- The command signals handle the de-emphasis, voltage swing level, and voltage value across the PHY.
- The status signals report the condition of the power management (to track transitions), receiver detection, data rate (to track changes), and eventual errors in the received data.

Within the PCS, the main building blocks provide the following functionality:

- Phase adjustment: it is nothing but a FIFO which compensates for phase difference that may arise between the internal transmitter low-speed clock, and the external clock (MAC clock).

- Reset Controller: it is an internal FSM able to perform the reset of both the analog and digital portions of the PHY. After the power-up, as soon as the calibration is completed with the lock acquisition, a TX analog reset is asserted. It follows the assertion of a TX digital reset and, whenever the CDR locks to the receiving data, a RX digital reset is performed. At this point, the power-up reset sequence is completed, and finally a RX reset signal is asserted another time to empty the previously received data. Lastly, the system jumps into operative mode state.

- 8B/10B Encoder/Decoder: both the encoder (at the transmitter side) and the decoder (at the receiver side) exploit the 8B/10B transmission code, implemented by IBM in the 1980s [22]. Nowadays, this transmission code is still widely used in order to translate 8-bit into 10-bit (encoder), thus achieving DC balance, and automatically performed error check through disparity detection. Different architectures are available in literature and, in particular, in the reference system [40], an M-codec and a P-codec implementations are exhaustively described and compared, in terms of area and dynamic power consumption.

- Comma Detector: it aligns the received de-serialized data at the PMA interface.

- Elastic Buffer: it provides a frequency compensation similar to the one performed by the phase adjustment. In particular, the difference between the local and the recovered clock needs to be kept under a certain threshold, for example ±600 ppm for the implementation proposed in [40].

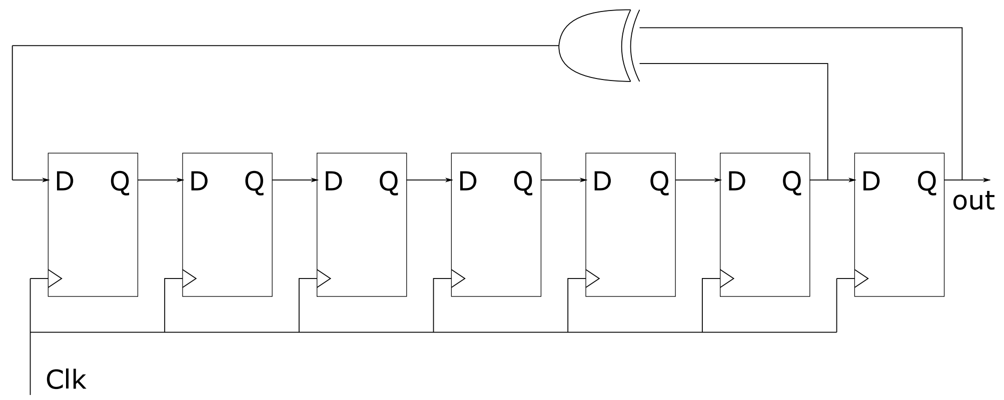


Figure 2.10: Typical PRBS7 implementation with XORed LFSR's architecture (inspired by [40]).

- Pseudo-Random Binary Sequence (PRBS) generator and checker: they are used to verify and characterize high-speed links. Indeed, they are typically activated in case of loopback configuration, in order to test the correct data transfer between the transmitter (PRBS generator) and the receiver (PRBS checker). Additionally, the actual data rate reached during operative mode is measured. In this way, the test can be directly performed without developing the upper levels of the protocol stack. Moreover, the random sequence acts as a wide set of stimuli for the system, so that it is easier to locate corner case bugs. The term *pseudo-random* specifies that, although the produced sequence models a random stream of data (independent from any other element), it is, in fact, deterministic. The pattern is generated by a Linear Feedback Shift Register (LFSR). LFSRs have a broad range of applications and, above all, they are widely used in Cryptography and Bulit-In Self Test (BIST). Hence, a lot of researches involving LFSR's architectures and optimization were carried out. [41] presents a detailed analysis of the state-of-the-art of LFSR, where the key metric is the power consumption. The traditional LFSR's XOR configuration (PRBS7) is presented in the block diagram of Figure 2.10.

Basically, feedback paths are originated from some of the registers' output which are referred as *taps*. Such taps are then exclusively XORed together before closing the loop, in order to generate the pseudo-random sequence. The taps' choice determines the polynomial equation the random sequence is originated from. Frequently, the PRBS are identified with the degree of the polynomial they are originated from, for example:

$$\text{PRBS7:} \quad G(x) = 1 + x^6 + x^7$$

$$\text{PRBS9:} \quad G(x) = 1 + x^5 + x^9$$

For what concern the PRBS7 example, the output sequence length is $m = 2^N - 1$, where N is the number of bits of the shift-register ($N = 7$, $m = 127$). This implies that, after every 127 binary bits, the sequence will start over by repeating itself. Hence, it is possible to forecast the next pattern from the previous one. The PRBS checker is based on this mechanism: starting from the received pattern, it determines the next sequence as the reference to compare with the actual sequence that will be received later on.

2.4 Related work

The topic of HW/SW system's co-partition was widely investigated to increase performance, flexibility and re-usability of the whole system. Besides, this thesis benefits from the analysis of relevant work. Specifically, few researches, proposing innovative implementation of the system's controller, are considered.

2.4.1 Partially Reconfigurable Microprogrammed Controllers

The interesting concept, introduced in [42], is the implementation of a partially reconfigured controller in FPGA. The main advantage of this realization is the considerably reduction in prototyping time. Indeed, apart from the controller implementation, an innovative prototyping flow is introduced. The traditional logic controller implemented in FPGA is here sub-divided: an addressing module (AM) is responsible for the microinstructions address, and a control memory (CM) stores the address information. This microprogrammed controller decomposition is exemplified by the block diagram of Figure 2.11, where an input stimulus X is applied to the AM which produces an excitation functions (T) that serves as input data for the counter (CT). The CT's goal is to to provide the address (A) to the CM, so that the desired microinstruction (Y) is extracted from the memory itself.

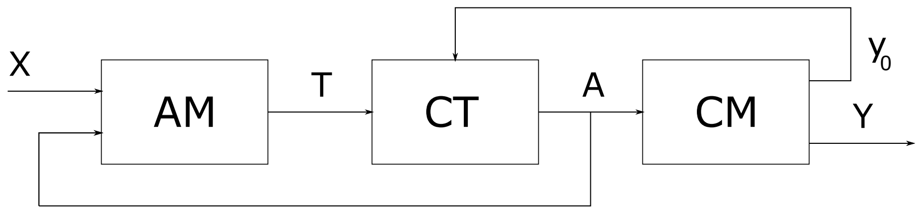


Figure 2.11: Typical structure of a microprogrammed controller (inspired by [42]).

Therefore, with this configuration, the designer is free to choose how to implement the memory block in order to save FPGA logic resources. Nevertheless, the proposed solution, considerably benefits from the application of *partial reconfiguration* to the CM. The concept of partial reconfiguration has been studied since the 1980s [43], [44] and nowadays, thanks to novel researches and the toolchain improvements, it has become mature[3]. Since only the content memory requires a replacement, the difference-based partial reconfiguration proposed by Xilinx [45], is recommended. Only the difference between the actual design and the new one is stored within the partial bit-stream, whose size is considerably reduced. The initial overhead (in terms of configuration time and size) of the main bit-stream creation in the implementation step still remains. On the other hand, the modification of the memory content, by relaying on the previously prepared differential bit-stream, is very fast. Despite the limited range of applications (design with minimal modifications), it is still relevant because it represents a fully-hardware implementation of a more flexible controller.

2.4.2 Parallel Logic Controllers

An even more interesting approach is the combination of partial reconfiguration concepts and HW/SW architectures [46]. This research is focused on customizable logic controllers, where various functionalities are defined in SW running on a Processing System (PS) and then physically realized in HW through PL implementation. The controller's model is based on Parallel Hierarchical FSM (PHFSM) mapped into reconfigurable logic, whose

[3]For the interest reader, a good overview of the reconfigurable SoC FPGA trends is presented in [7]

communication is controlled by SW through high-performance interface composed of General Purpose Port (GPP) and High Performance Port (HPP) [4].

A parallel controller is defined as "one of the processing elements that gets input from the controlled system and generates output that ensure the desired functionality" [46]. Based on that, the dual-core APSoC implementation emphasizes those aspects:

- Usage of modular, hierarchical and parallel HW resources (PL), based on HFSM, Communicating FSM (CFSM), and PHFSM, whose behaviour can be modified by the SW running on the PS. In particular, hierarchical FSM, alike for the procedure handling in SW, are based on modules calling the one to each other according to the Hierarchical Graph Scheme (HGS), so that the control algorithm is modeled step-by-step. A more detailed description of the state transition, with the modules' call, is presented in [47], and [48], whereas, an example of HFSM usage for controlling purpose is reported in [49]. On the other hand, parallel FSMs allow more than one module to be executed in parallel, with the same mechanism of hierarchical call and return, which requires a stack memory (implemented in the PL). Figure 2.12 displays a simplified model of execution for both HFSM (on the left), and PFSM (on the right).

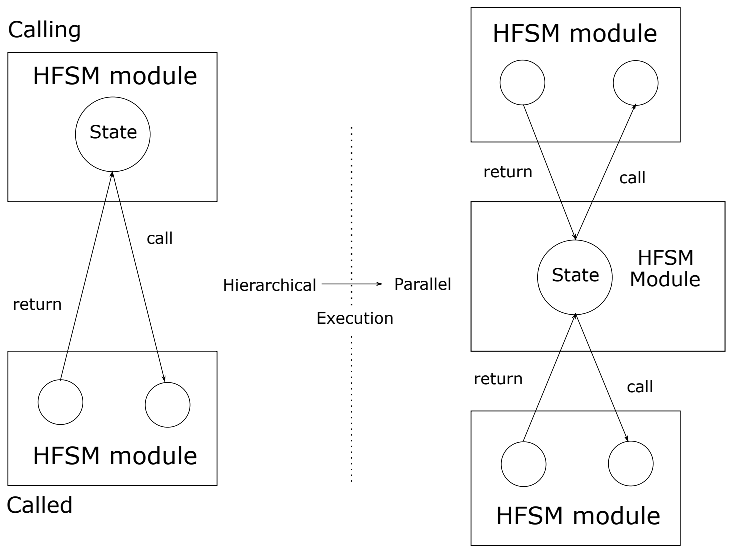


Figure 2.12: Examples of execution of HFSM on the rigth and PFSM on the left (inspired by [46]).

The described FSM modules are handled by the PS SW which acts as high-level controller able to request a run-time reconfiguration. The SW checks the operation

[4]The implementation targets All-Programmable SoC (APSoC) and in particular the Zynq-7000 so that the technical terminology introduce by Xilinx in [50] is used.

of the lower-level modules by periodic signal monitoring, and by interrupt. This way of proceeding is similar to the one adopted in this thesis, where both interrupt and polling mechanism are used in order to monitor the controlled sub-system. Conversely, in this case the control unit is partitioned into two layer: PS with the SW and PL with the HW by leading to higher system complexity. Finally, after the check, the PS may conclude whether the controlled devices require an improved management algorithm and, a consequent modules' update through reconfiguration.

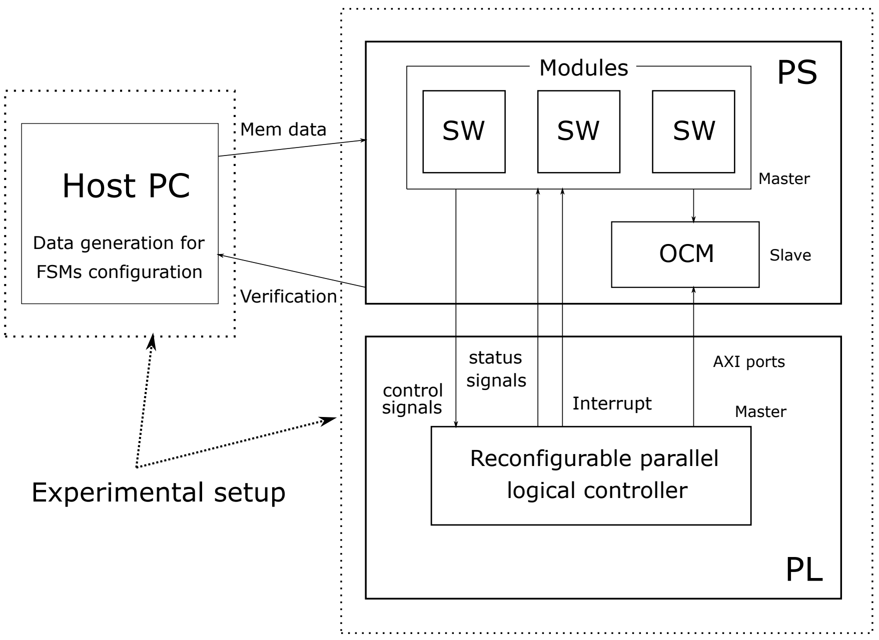


Figure 2.13: Hardware/Software architecture. The communication with the host PC serves as experimental setup (inspired by [46]).

- SW handled reconfiguration of the control circuit FSM according to the models and methods introduced by Sklyarov [48]. The reconfiguration models may be either static (bit-stream uploaded into the PL) or dynamic (at run-time when the bit-stream is loaded).
- Co-partitioned SW/HW architecture as shown in the simplified block diagram of Figure 2.13. The usage of a customized interface (GPP, HPP, signal transfer through registers), allows a separate and independent development of HW and SW, in order to speed up the system design. Moreover, the HPP interface is also used for the FSMs reconfiguration and the correspondent information are extracted from the On-Chip Memory (OCM) filled in from the host PC. Thus, the PS acts as a master by copying the data from the PC to the OCM (the slave). Afterwords, once the data are locally stored, the relocation into the PL occurs in a burst mode with the sequence of *init*, *read*, *load*, *done* [46].

Chapter 3

Design Methodologies and Preliminar Analysis

This chapter gives a brief introduction of the methodologies and the tools used to design, implement and characterize the PoC. In particular, the Intel Arria 10 board is described, by focusing on the architecture, available IPs, and Nios II soft-core processor, whose design flow is also introduced.

3.1 System-On-Chip for prototyping: Arria 10

Among all the possible prototyping technques, underlined within the state-of-the-art analysis, the choice of a SoC implementation cuts down the design time and complexity. For what concern the design flow, it is considerably simplified by the toolchain provided together with the target SoC board. For this reason the Intel Arria 10 SoC (TMSC 20-nm process technology) is used as target development board [8].

3.1.1 Introduction

A minimal block diagram of the Arria 10 architecture is shown in Figure 3.1. This board is addressed as SoC FPGA by Intel because of its *hybrid* nature. Indeed, it is composed of two main parts with the correspondent development toolchain:

- Hard Processor System (HPS) (20-nm, second generation). It consists of a dual-core ARM Cortex-A9 MPCore processor [51], integrated with a set of peripherals and a multiport memory controller. This integration with the FPGA, and the hard IPs, on the same SoC, allows the HPS to control the logic configuration while running software applications. The SW development for the HPS is simplified and optimized by the Embedded Design Suite (EDS) which comprehends: hardware libraries for the HW low-level access, Eclipse Integration Development Environment (IDE) based on ARM DS-5 for the software development, and additional configuration tools.

- FPGA fabric PL. The Logic Core is free to operate independently from the HPS

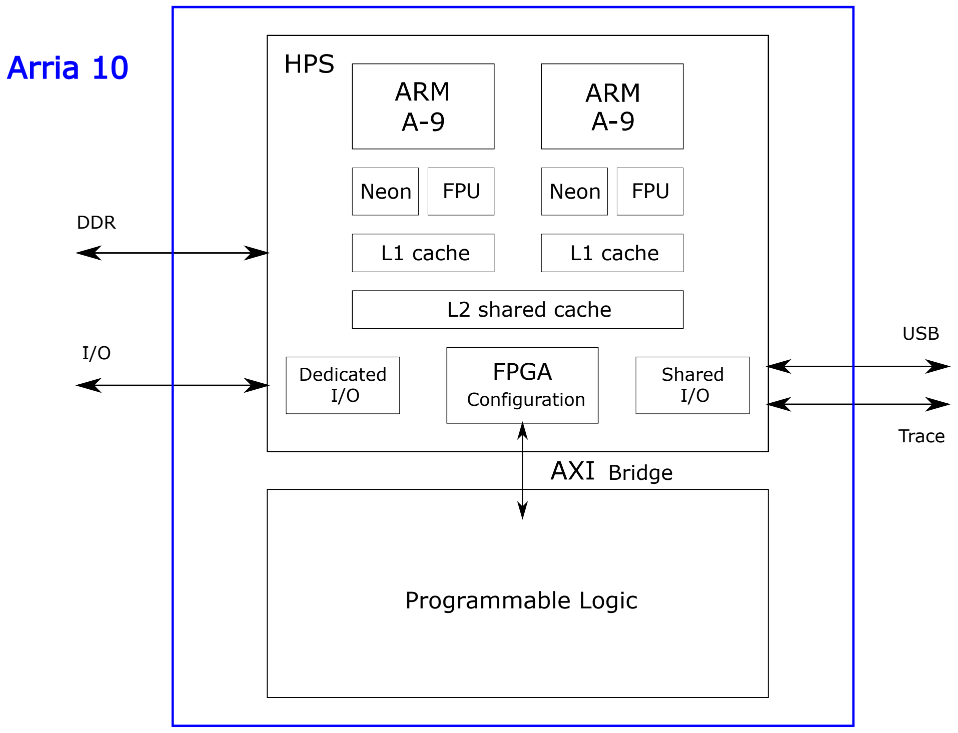


Figure 3.1: Intel Arria 10 block diagram.

but, in case of required communication, the high-performance system interconnect is exploited. This interconnection system is based on the ARM AMBA AXI[1] bus bridges. The logic fabric allows to tailor the design by exploiting off-the-shelf Intel's IPs or by implementing custom ones. In particular, the Quartus tool is used for system design (through HDL), compilation, synthesis, timing analysis and power estimation. More specifically, Quartus System Integration Tool (Qsys) is used at design time to instantiate and connect IPs through Av-MM interfaces.

This duality combines the PL's flexibility with cost minimization (size reduction), and power consumption cut down, due to separate frequency management (between HPS and FPGA). Additionally, the system reliability is improved by separating the HPS boot process from the logic configuration. Finally, it is possible to choose between bare-metal or Operating System (OS) supervised applications (Wind River VxWorks, Micrium μC/OS-II, μC/OS-III, open-source).

[1]The Arm Ltd. Advanced Microcontroller Bus Architecture (AMBA) has been widely used as a standard for IPs connection within SoCs [52]. Advanced eXtensible Interface (AXI) represents a AMBA interface able to connect order of 100 master-slave within a SoC.

3.1.2 Nios II soft-core processor

Besides the HPS, Arria 10 also supports the instantiation and configuration of a soft-core[2] processor, within the fabric logic, which is named Nios II. This processor may be preferred in case of limited complexity, where there is no need for the HPS' resources.

Introduction

The key point, that distinguish Nios II from any other off-the-shelf microcontroller, is the possibility to customize it (similarly to a generic IP) to meet various requirements. This shaping is not limited to the internal hardware components, but it may also be extended to the Instruction Set Architecture (ISA). By adapting the set of instructions to the SW/HW partitioning, the optimal trade-off between performance, size and power consumption can be reached.

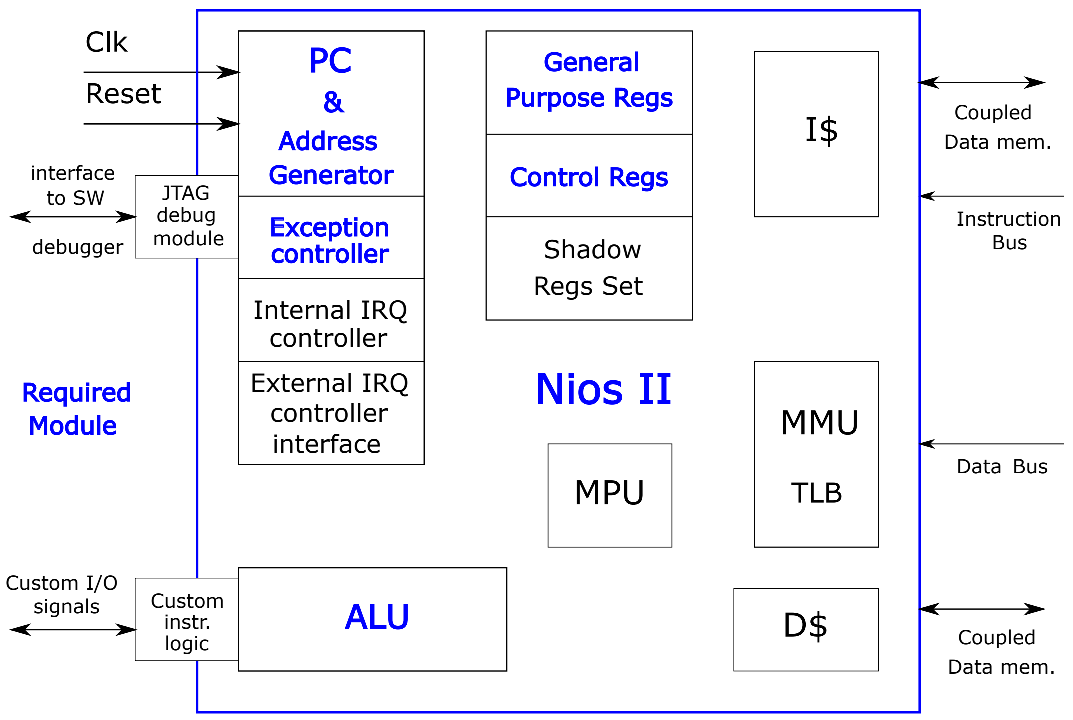


Figure 3.2: Nios II block diagram.

As it can be seen from the block diagram of Figure 3.2, the necessary hardware (colored in blue) is minimal, whereas various additional blocks may contribute to the customization. Alternatively, this functional units may be emulated via software; for

[2]*Soft* indicates the processor flexibility. Indeed it can be configured on a system-by-system basis to target any Altera FPGA family.

example, if complex mathematical computations are not required it may be convenient to perform division operations in SW.

Nios II range of applications is broad, ranging from high-speed image acquisition [54] to multi-functional signal generator [55], embedded web server [56], or Radio-Frequency Identification (RFID) reader [57]. Nevertheless, it represents an optimal solution for prototyping: a preliminary design can be easily synthesized and analyzed in a reduced amount of time while leaving room for further step-by-step refinements.

Technical details

Nios II is a 32-bit RISC[3] processor core, which acts as a general-purpose microcontroller by including peripherals and internal memories. The technical details are described in [53], whereas the main components are as follows:

- 32-bit datapath, ISA, and address space. It is classified as an Harvard architecture since separated instruction and data buses are supported.
- 32-bit internal registers: general purpose ones plus optional set of shadow registers (up to a total of 63)[4].
- 32 level-sensitive Interrupt Request (IRQ) inputs plus External Interrupt Controller (EIC), which is an external interface able to reduce interrupt latency. The *precise exception* management is carried out by a basic, nonvectored controller. Precise means that, whenever the exception occurs, before handling it, all the previous instructions need to be committed and no later instruction execution has the right to start.
- Aritmetic Logic Unit (ALU) able to perform arithmetic, relational, logical, shift and rotate operations on the general purpose registers. In case of unimplemented instructions (not defined in hardware), an exception is raised, so that, such functionalities can be performed in software instead. Additionally, the ALU is directly connected to the programmable logic by letting room for customized instructions.
- On-chip peripherals and interfaces, used to manage external memories. The organization is configurable and may vary from system to system. Both instruction and data cache may be internally added, whereas tightly-coupled instruction and data memory port may connect Nios II to on-chip memory, outside the core, in case of performance-critical applications, where the key requirement is low-latency. In other words, they may perform as cache memories but without dealing with cache load, invalidate, and flush overhead. Additionally, Av-MM master ports may be used as interconnection standard. The address map of both memories and peripherals is not fixed and has to be defined at design time.

[3]A description of a RISC architecture and a comparison with CISC may be found in [58] and [59].

[4]Shadow registers are accessed by the OS' kernel to faster up context switching.

- Optional dedicated single-precision floating point instructions (double precision operations may be performed through software emulation).
- JTAG connector which is used to start and stop the processor for debugging purpose.
- Optional Memory Management Unit (MMU) with TLB or Memory Protection Unit (MPU). The MMU takes care of the virtual (4 GB of address space) to physical address mapping (4 GB of memory) by supporting system paging (the size of both page and frame is 4 KB). The translation process is accelerated by an hardware Translation Lookaside Buffer (TLB). In case of MMU instantiation it is enabled by default and the caches (data and instruction) are virtyally-indexed, physically-tagged. The MPU guarantee an higher degree of memory protection by dividing the memory area in different regions (up to 32 for instruction and 32 for data). Moreover, read and write permissions are required to access specific regions. It is important to consider that MMU and MPU are mutually exclusive.

3.2 PoC's Design Flow

The PoC's development is carried out through a sequence of steps which, starting from the design lead to the synthesis and board programming. The process deeply relies on Intel's guideline and toolchain, since they allow to independently carry on the hardware and firmware development. Consequently, the overall design time is considerably reduced. In a nutshell, the flow (shown in Figure 3.3) comprehends the following steps:

- Creation of a Quartus project with detailed information about the target board.
- System design with the integration of the building blocks through Qsys. The Quartus Prime system integration tool is responsible for the hardware definition (IPs features selection), and consequent generation. The following set of files are generated during the process:
  - An HDL (both VHDL[5] and Verilog are supported) hardware description of the system, comprehensive of both Nios II and additional IPs.
  - A file with .sopcinfo extension with an hardware description of the system, which is readable by the Software Build Tool (SBT). In this way, the software can be transparently written to precisely target such hardware platform.
  - An optional HDL testbench for functional simulation of the generated system. This testbench generation process sets up simulation libraries and .tcl scripts for simulation tools of different vendors (Cadence, Mentor Graphics, Synopsys and Aldec).

[5] Very High Speed Integrated Circuit HDL (VHDL).

- Firmware development, on top of the HW architecture (described by the .sopcinfo file), through the Nios II SBT for Eclipse, or an alternative software development environment based on GNU C/C++. In both cases, it is possible to start the SW development while the underlying hardware still need to be completed. Out of compilation, the .elf file is used by the Quartus Programmer to physically upload the code into the board. Consequently, a real-time debugging of the firmware running on board can be performed. Generally, to allow such functionality, a JTAG debug unit is inserted within the Nios II subsystem to perform read/write operation through the SBT's console.
- Quartus project compilation and synthesis of the project. This step produces a set of reports with detailed information about the logic utilization (in ALMs), timing analysis and placement of the target device. Finally, the synthesizable .sof file is uploaded onto the board by the Quartus programmer via micro-blaze Universal Serial Bus (USB) connection.

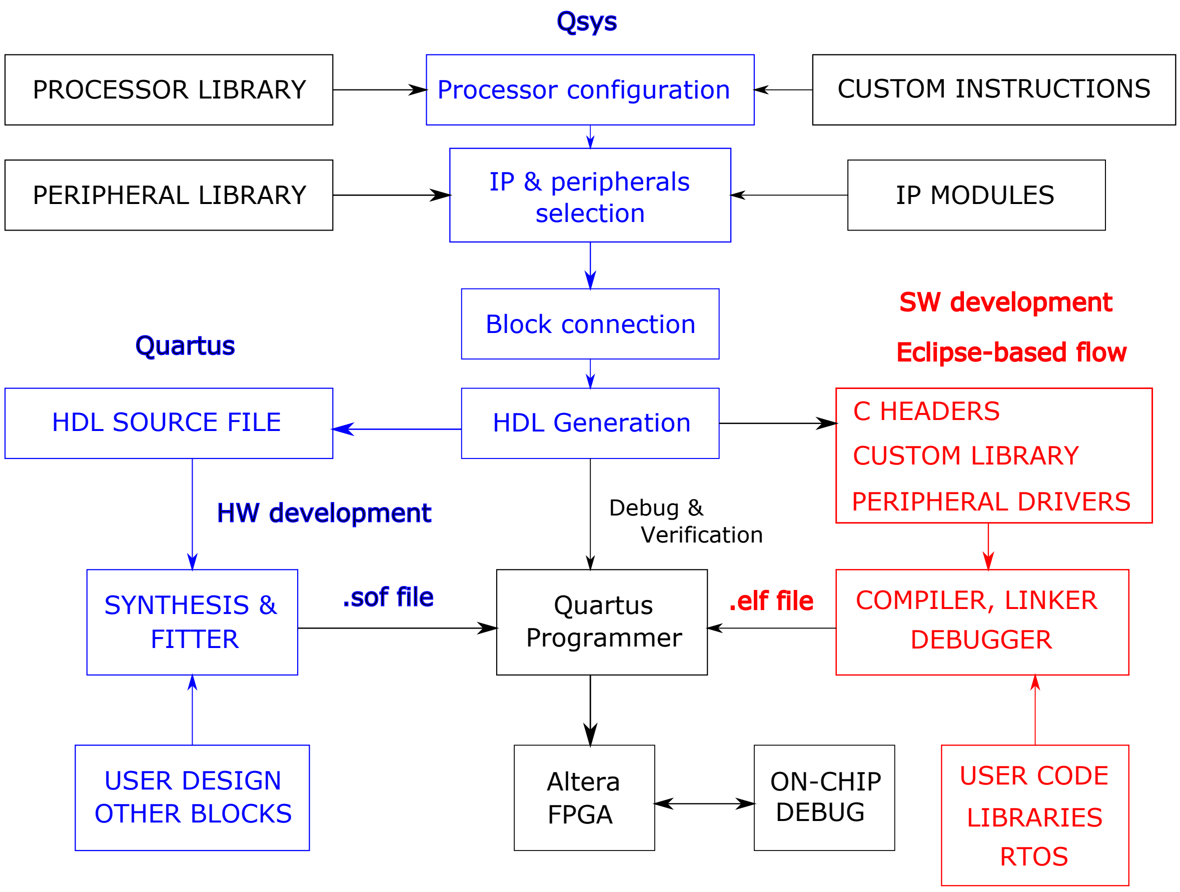


Figure 3.3: PoC design flow to target the Arria 10 board with the Nios II soft-core processor.

3.3 PoC's characterization

Logic utilization (in ALMs), power consumption (static and dynamic), and software performance are evaluated in order to characterize the PoC. As previously mentioned, the logic resource information is directly provided within the compilation reports. Instead, the power consumption estimation is based on two different techniques: Early Power Estimation (EPE) and PowerPlay Power Analyzer The correspondence accuracy,[6] after the power models, is:

- PowerPlay Power Analyzer: ±20% from silicon (by using an input file with the toggle rate evaluated during simulation).
- EPE spreadsheet: ±20% from the Power Analyzer and ±30% from silicon (in 90% of the tested designs).

Additionally, the estimation accuracy is affected by many parameters:

- Toggle rate. As already mentioned the toggle rate may have a huge impact on the final estimation. A possible way to increase the accuracy is to separate the system sub-blocks and separately perform the resource usage and toggle activity evaluation. If the same sub-blocks have been already in use for other designs it is suggested to rely on that information.
- Airflow. The effect of the fan across the FPGA needs to be included within the thermal model. Consequently, thermal simulations are required in order to evaluate the accuracy of such model.
- Temperature. Since the temperature has a huge influence on the final evaluation, it is fundamental to consider the temperature of the air around the device (which is generally higher than the ambient outside the system).

For what concern the power estimation, it is dived into static[7] and dynamic contribution (Table 3.1), whereas the possible signal behaviours are defined as follows [60]:

- Toggle rate: it is the average amount of the transition of the signal values (0 to 1 or vice-versa) within the time unit. Thus, the unit of measurement is properly $transitions/s$. Quartus models full rail-to-rail switching.
- Static probability: it is the window of time, within the device operation, where the signal has logic value '1'. When dealing with small technologies node (under 90 nm) the state-dependent leakage may significantly contribute to the static power.

[6]This is true for "the majority of the designs" as stated by Intel in the *Power Analysis* section of the Quartus user guide [60].

[7]The I/O DC bias power and transceiver DC bias power are exceptionally accounted for in the I/O and transceiver sections instead of being computed as PL's leakage.

Power Estimation	**Description**	**Signal Activity**
Static	Leakage thermal power dissipated by the PL	Static probability
Dynamic	Portion of the total power due to signal activity	Toggle rate

Table 3.1: Total power consumption contribution with the related signal activity.

For what concern the software performance, it minimally affects the dynamic power consumption: the firmware only determines the signal activity of the control and status signals to be driven from/to the Nios II.

3.3.1 Early Power Estimator (EPE)

The Excel-based EPE performs a thermal analysis based on the input parameters: device family and serial number, logic utilization and reference temperature (it can be either the junction temperature or the range of ambient temperature). The main features (detailed in the EPE user guide [61]) are:

- Early stage power estimation: it may be performed at a brainstorming phase since the rough power evaluation is based on of the hypothetical logic usage. This helps the designers, at the very first step, to evaluate whether their approach is going to meet the power consumption's constraints. Moreover, the power supply, voltage regulators, heat sink, and cooling system can be designed based on this first estimation.
- Early stage thermal analysis.
- Intermediate stage power estimation: as long as the design evolves it is recommendable to update the EPE estimation.
- Late stage power estimation: after compilation, it is possible to export a .csv file with accurate information, so that it can be used as input file for the EPE spreadsheet in order to increase the estimation's accuracy.

3.3.2 Quartus PowerPlay Power Analyzer

The PowerPlay Analyzer performs an accurate estimation based on the fitter information, operative conditions and signal activities (from the .vcd file extracted from simulation), as illustrated in Figure 3.4.

The usage of the Power analyzer is fundamental in the late stage of the design to check whether the power budge is met. However, a high-accurate estimation requires:

- Accurate power models of device circuitry.
- Accurate knowledge of the device operating conditions.

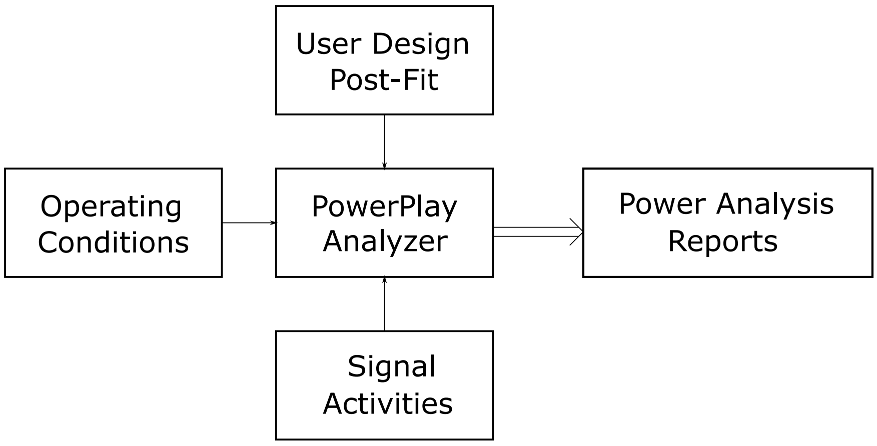


Figure 3.4: PowerPlay Power Analyzer high-level flow (inspired by [60]).

- Accurate toggle rate data on each signal. This can be achieved by using a modular design (every sub-modules activity is evaluated) or a complete design simulation approach (the signal activities for every hierarchical level are collected within a unique output file). In both cases, data collected under various simulation conditions (used to exhaustively stimulate the design) may be evaluated in parallel to contribute to a more accurate the power estimation.
- Glitch filtering. The Power Analyzer interprets a glitch as "two signal transitions so closely spaced in time that the pulse, or glitch, occurs faster than the logic and routing circuitry can respond" [60]. Therefore, glitch filtering avoids the computation of non-physical signal activities and the consequent overestimation of the dynamic power consumption.

Besides the output's report summary, many detailed parameters are specified such as the operating condition used, the thermal power dissipated by block, the thermal power dissipation by hierarchy, the signal activities and the current drawn from voltage supplies. A summary of the PowerPlay features in comparison with the EPE ones is presented in Table 3.2.

3.3.3 Software profiling

The profiling serves to measure the performance of Nios II subsystem, to identify bottlenecks and to propose firmware improvements. This analysis is unavoidable for complex systems, especially in case of multithreading applications which deeply rely on context switching. Many profiling techniques are available: *GNU profiler*, *Intel performance counter* and *high-resolution counters.*

The major requirement for the thesis is to complete the transceiver's reset sequence within few ms, and, possibly to recover from synchronization loss even faster. In other

Characteristic	**EPE**	**PowerPlay Analyzer**
Design phase	Any time (the accuracy varies)	Post-fit
Accuracy	30% from silicon	20% from silicon
Input	• Resource usage estimates • Clock requirements • Operation conditions • Fixed toggle rate	• Post-fir design • Clock requirements • Default signal activity • Operation conditions • Simulation results • Signal activity per node (or entity)
Output	• Total thermal power dissipation • Static contribution • Dynamic contribution • Off-chip power dissipation • Current drawn from voltage supplies	• Total thermal power • Static contribution • Dynamic contribution • Thermal I/O power • Thermal power by design hierarchy • Thermal power by block type • Thermal power dissipation by clock domain • Off-chip power dissipation • Device supply currents

Table 3.2: Comparison between the EPE and the PowerPlay Power Analyzer (inspired by [60]).

words, a precise time of execution is the key metric. Consequently, the ideal profiling technique would be non-intrusive and with high-resolution. The high-resolution counter was selected due to its limited resource utilization, and its light implementation of the performance measurement functions. The high resolution is embedded in the 64-bit counter with 1 μs tick counting. A detailed description, together with the performance evaluation, is reserved to the *Analysis and Results* Chapter.

Chapter 4

System design and development

This chapter meticulously describes the PoC's implementation. The description begins with the analysis of the hardware components and their interconnections. The bare-metal application is then built on top of such system due to the HAL contribution. Afterwards, the software architecture is presented, with a particular attention on exception handling. A reference for hardware and software development of a Nios II-based system is [62].

4.1 Hardware design

The PoC's HW architecture designed and synthesized on Arria 10 is shown in Figure 4.1.

Conceptually, the system can be divided into two units:

- System handler: composed of the Nios II soft-core processor and its peripherals (OCM, JTAG UART[1], Timer and PIO[2]). The Nios II (economy version) was configured with the minimum functionality: OCM for both the reset vector and the exception vector memory, JTAG debug module, without chache.
- Transceiver: composed of the Intel Transceiver Native PHY and the external PLL for the reference clock.

The system clocking requires separate sources for the presented units:

- Input reference clock of 50 MHz for the system handler (Nios processor and peripherals). Specifically, the Arria 10 *CLK_50M_FPGA* reference signal was used.
- Input reference clock of 125 MHz for the transceiver. The former clock is used as reference for both the external PLL and the transceiver CDR. Specifically, the Arria 10 *CLK_ENET_FPGA_P* reference signal was used.

[1]UART contains serial lines for data communication. It is frequently used together with RS-232 interface for embedded system's application.

[2]PIO refers to the Intel IP notation but is alike the GPP introduced in the *Background and Literature Study* chapter.

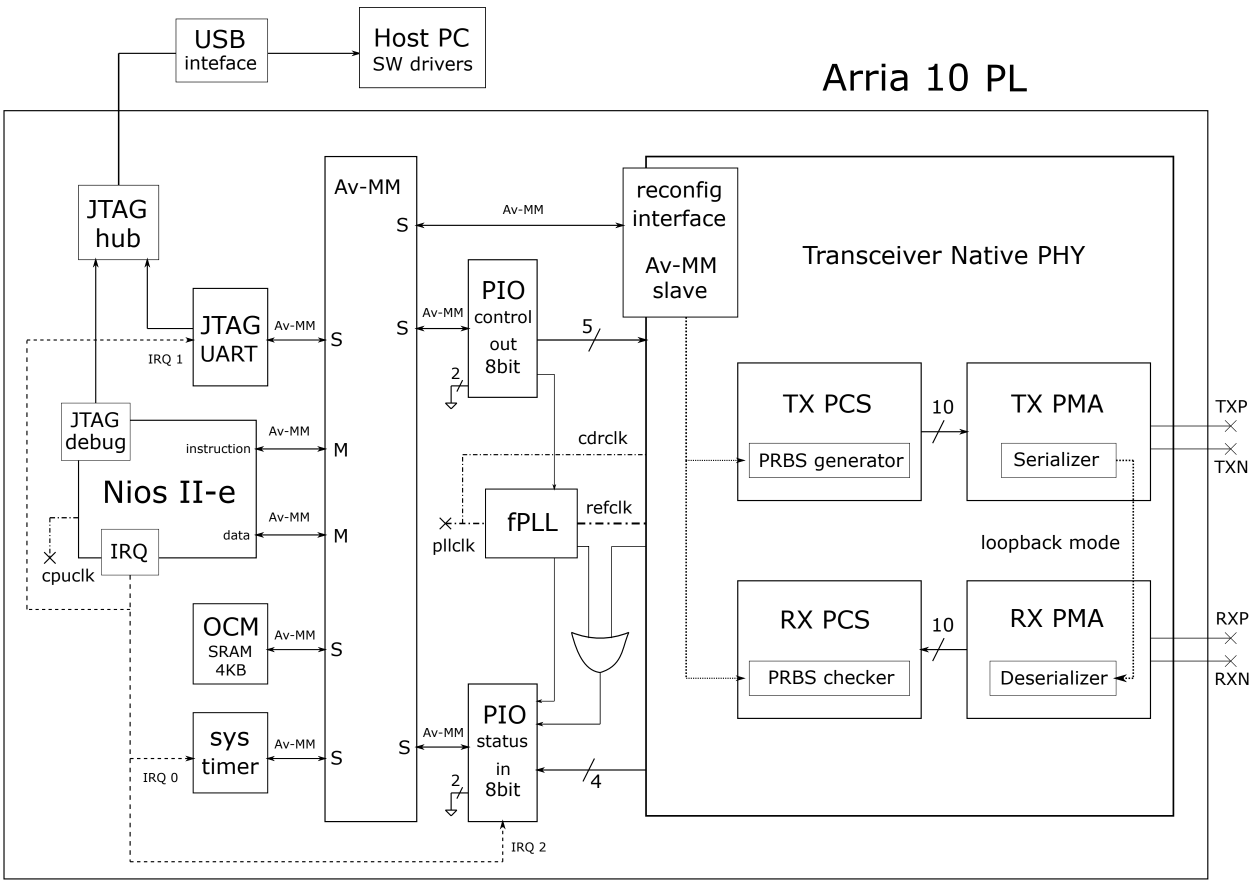


Figure 4.1: Conceptual block diagram of the HW architecture.

Finally, the communication between such units is handled by the Av-MM interface. The following sections give an overview of the mentioned block units and their interconnection.

4.1.1 Nios II's peripherals

The Nios II soft-core processor can be interconnected to I/O peripherals within the same system, hereinafter called Nios subsystem. The designer can directly use the set of peripheral within Intel's libraries or customized ones. In the former case, a wrapper is needed to adapt the IP's HDL description to the Av-MM standard.

For what concern the Intel peripherals, many configuration parameters are available, for example enable/disable interrupts or size selection. Additional device drivers[3] are

[3]This term generally indicates a collection of SW routines coded to access the I/O cores. A more detailed description of the so-called HAL is provided within the *Software Design* section.

provided to facilitate the communication with the higher level of abstraction (software C code).

Nios II processor utilizes a memory-mapped scheme, so that any generic I/O core is *identified* as a collection of registers. Basically, the access mechanism consists of write/read operations to/from the set of registers mapping the addressed peripheral.

The following sections give detailed analysis of the Nios II subsystem's peripherals.

On-chip memory (OCM), SRAM

The available OCM provides memory space for both instruction and data. Moreover, it can be deeply customized through the following configuration parameters:

- Memory type: Random Access Memory (RAM) or Read-Only Memory (ROM).
- Size: expressed by the data width (from 8-bit up to 1024-bit) and the total memory size (in bytes).
- Read latency: set to 1 or 2 according to the Av-MM read protocol.
- Additional protection on read request and ECC parameters.
- Memory initialization: it can be automatically performed by Qsys when assigning a base address or manually specified by inserting the folder path of the initialization file. This can be used for the HW/SW system simulation but also to include the memory initialization within the .sof file (synthesizable code).

The memory requires a reset and a clock signal as input parameters, and it acts as a Av-MM slave. Therefore, a master needs to be connected to this interface: in the specific case Nios II serves as master. The memory size is computed as the minimum value able to allocate the firmware (comprehensive of libraries and drivers).

For the proposed system an SRAM of 4 KB with 1 cycle of read latency and manual initialization was chosen. No additional protection was used.

JTAG UART

The Intel JTAG UART exploits a JTAG controller for serial communication instead of using a RS-232 interface. This IP is treated as any other serial port for character exchange between a host PC and the PL. A Conceptual block diagram is illustrated in Figure 4.2.

The FIFO internal buffer width and the interrupt threshold may be customized at design-time. These buffers are used to enhance the read/write performance, and they can be directly synthesized into registers (instead of memory blocks) for a deep optimization.

The register map can be divided into data and control registers. The data field, precisely contains the byte to be exchanged. Conversely, the control register monitors the interrupt requests and the available space within the buffers.

The implemented JTAG UART has a buffer depth of 32 bytes and an IRQ threshold set to 8. Finally, the IRQ priority was set to 1, where the range varies from 0 (highest priority) up to 2 (lowest priority).

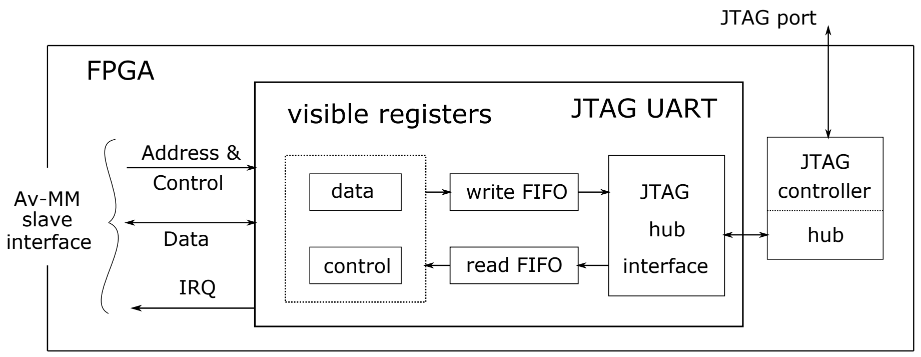


Figure 4.2: Conceptual block diagram of the JTAG UART (inspired by [62]).

Interval Timer Core configured as system clock

From the HW perspective, the internal timer core is based on a count-down timer which, starting from a specified value, keeps on counting until it reaches 0. This defined the so-called *timeout period* which has to be a multiple of the *timer frequency.* After the timeout period, a correspondent bit is set and an IRQ is raised. A simplified block diagram is illustrated in Figure 4.3, whereas a summary of the configuration parameters is reported in Table 4.1.

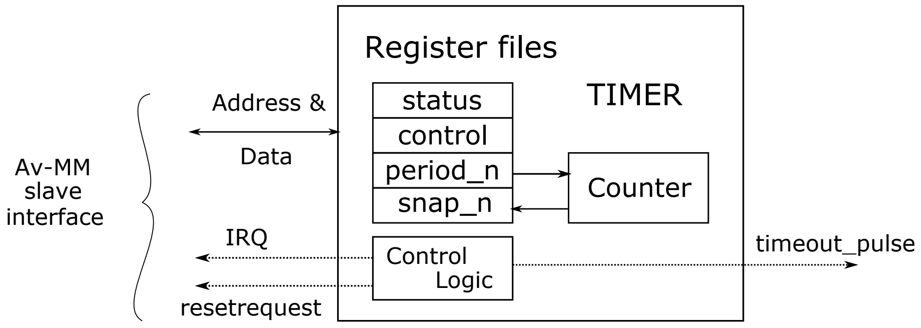


Figure 4.3: Conceptual block diagram of the Interval Timer Core.

The register map allows the user to directly access 32-bit registers which are internally based on 16-bits registers. Among the whole set of registers (summarized in Table 4.2), the key role is played by the status and control registers.

Additionally, a customized functionality can be selected through SW configuration:

- System clock timer: it is generally used in application which requires a scheduler. Indeed, it is configured in free-running mode by default, so that the counter is stopped in case of IRQ. After reaching 0, the counter reloads the value specified within the timeout register. As a final result, the time is measured in clock ticks (similarly to a periodic heartbeat). The tick resolution depends on the clock

Parameter	**Value**	**Description**
Timeout period	time units or # of clock	Set the initial value of the timeout period register
Counter size	32 or 64 bits # of clock	Set the timer's width
Preset HW configurations	• Simple periodic IRQ • Watchdog • Full-featured	IRQ are periodically generated Reset in case of malfunction Variable period and start/stop handled by an embedded processor
Register options	• Writable period • Readable snapshot • Start/Stop	Enable to access the count-down period Enable to read the current count value Enable to access the start/stop bits of the control registers
Output signals	• Timeout pulse • Reset on timeout	Enable to output the *timeout_pulse* signal (1 high pulse after timeout) Enable to output the *resetrequest* signal (reset the system after timeout)

Table 4.1: Internal timer core configuration parameters.

Offset	**Register**	**Activity**
0	Status	Check the timeout (to bit) and the running state (run bit)
1	Control	• ito -> interrupt enabled • cont -> running mode • start -> start counter (bit to 1) • stop -> stop counter (bit to 1)
2	Periodl	Timeout period bits [15 ÷ 0]
3	Periodh	Timeout period bits [31 ÷ 16]
4	Snapl	Counter snapshot bits [15 ÷ 0]
5	Snaph	Conter snapshot bits [31 ÷ 16]

Table 4.2: Internal timer core register mapping (inspired by [62]).

frequency and it can be set by software.

- Timestamp driver: this configuration guarantee high-resolution time measurements. Basically, by sampling the monolitically increasing counter, in the crucial point of the code, an accurate time information is extracted.

In conclusion, the designed system exploits an internal core timer configured as system clock driver with:

- Counter size of 32-bits.
- Preset configuration: full-featured.
- Register option: readable snapshot enabled.

General Purpose I/O

A PIO core basically serves as interface between the Av-MM interconnection and the actual general-purpose I/O port. The latter does not necessarily reside on board, but it may also be an off-chip external device connected to the FPGA. A functional block diagram is shown in Figure 4.4, whereas the configuration parameters are summarized by Table 4.3.

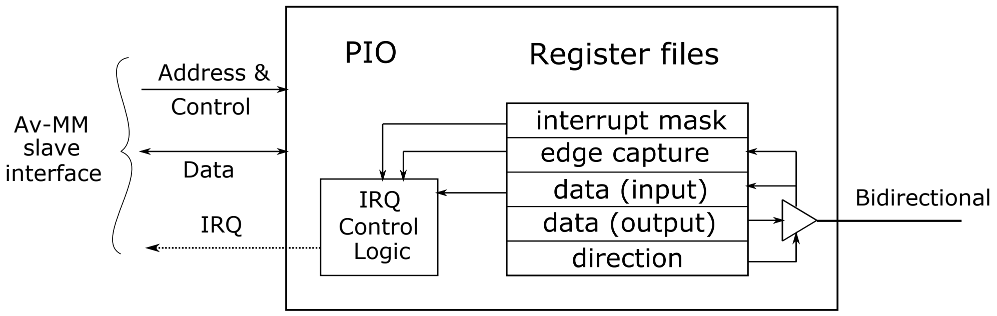


Figure 4.4: Conceptual block diagram of a PIO (inspired by [62]).

To read/write different registers the correspondent offset parameter needs to be specified, as reported in Table 4.4. Note that, despite both input and output registers share the same offset they work independently.

The designed system makes use of two PIOs: one to read the status signals from the transceiver (and PLL), and the other to drive the control signals in order to reset the transceiver. They are configured as follows:

- PIO status signals:
  - Width: 8-bit.
  - Direction: input.
  - Edge Capture: FALLING.
  - Interrupt: LEVEL, with IRQ priority set to 2.

Parameter	**Value**	**Description**
Width	$1 \div 32$	Specify the number of bits
Direction	• Bidirectional • Input • Output • InOut (both)	Select one tristate bidirectional bus Only enable input capture Only drive the output Capture input and drive output simultaneously onto separate unidirectional buses
Edge Capture	• OFF • FALLING • RISING • BOTH	Disable input capture Capture input on falling edge Capture input on rising edge Capture input on both edges
Interrupt	• OFF • LEVEL • EDGE	Disable interrupt IRQ triggered by input level IRQ triggered by input capture

Table 4.3: PIO configuration parameters.

Offset	**Register**	**Activity**
0	Data	Read input data or write output data
1	Direction	Set the PIO direction
2	Interrupt mask	Enable the IRQ by setting the correspondent bits to 1
3	Edge capture	Check the detected edge (bit to 1) and clear (set the bit to 1)
4	Out-set	Set the output bits (by writing 1)
5	Out-clear	Clear the output bits (by writing 1)

Table 4.4: PIO register mapping (inspired by [62]).

- PIO control signals:
  - Width: 8-bit.
  - Direction: output.
  - Edge Capture: OFF.
  - Interrupt: OFF.

4.1.2 Avalon-MM Interface

Avalon interfaces achieve two main goals:

- Increase the communication performance within a complex system. A traditional centralized resource (such as a bus) becomes a bottleneck as soon as the data transfer within the system increases. Conversely, Avalon interfaces allows the internal communication through interconnect fabric according to the standardized timing properties. The interconnection automatically comes together with arbitration functionality, in order to avoid contentions (typical of centralized resources).
- Simplify the design of complex systems. Generally, an increase in complexity is followed by an increased number of interconnected sub-blocks communicating the one to each other. Indeed, most of the time is spent in the design of customized interfaces to suit various requirements of each sub-block. Conversely, with the Avalon interface a master-slave interconnection between different sub-blocks is automatically optimized.

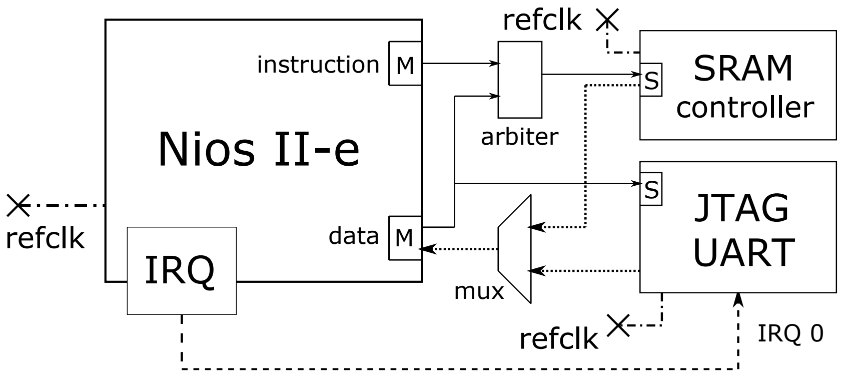


Figure 4.5: Conceptual block diagram of an Av-MM interconnect with a master (Nios II) and two slaves (JTAG UART and SRAM controller). Both data and instruction are connected to the SRAM controller, thus an arbiter is provided. The mux allows the master to select one slave at a time. The picture is inspired by [62].

The focus is on the Memory-Mapped (Av-MM) one. It handles address-based read-/write communication between a master, which initiates a transaction, and one or more slave(s) that respond(s) to this request. Typically, microprocessors, memories, UARTs and timers are connected through Av-MM. A conceptual block diagram with a Nios II (master), a JTAG UART and a SRAM controller (slaves) is illustrated in Figure 4.5.

Avalon-MM slave interface signals

The standardization defines a set of interface signals; the major ones are as follows:

- Read: single bit signal which is asserted to denote an on-going read transfer.
- Write: single bit signal which is asserted to denote an on-going write transfer.

- Address: specify the memory location to be accessed within the slave address space. The address width can be selected within the range (1 ÷ 32) bits.
- Readdata: value provided *by* the slave as result of a read operation. The width varies between (8 ÷ 1024) bits (power of 2 values).
- Writedata: value provided *to* the slave during a write operation.

Avalon-MM slave interface properties and timing

The Av-MM slave interface delineates plenty of properties related to the read/write operation timing. Based on that, all the transactions are synchronous with the reference clock. These are the fundamental properties:

- Read/Write WaitTime: introduce a variable number of wait state(s) to prolong the read/write signal. This feature is very useful in case of slow slave sub-blocks.
- Setup/Hold Time: after asserting/deasserting the address and data, it is the time to wait before the consequent read/write signal assertion/deassertion.
- ReadLatency: time required to have the data available after the read signal's assertion.
- Waitrequest: alternative to fixed wait-states (as previously described). The slave exploits the *waitrequest* signal to stall the transfer (read/write operation) until it is ready. Keeping *waitrequest* asserted for N clock cycles is alike setting read/write WaitTime to N.

Different scenarios of read/write transactions are illustrated by the timing diagram of Figure 4.6 (no wait-state) and 4.7 (readWaitTime = 1, writeWaitTime = 2). Note that, zero wait-states implies to immediately generate the response in the same clock cycle of the request and this may decrease the frequency.

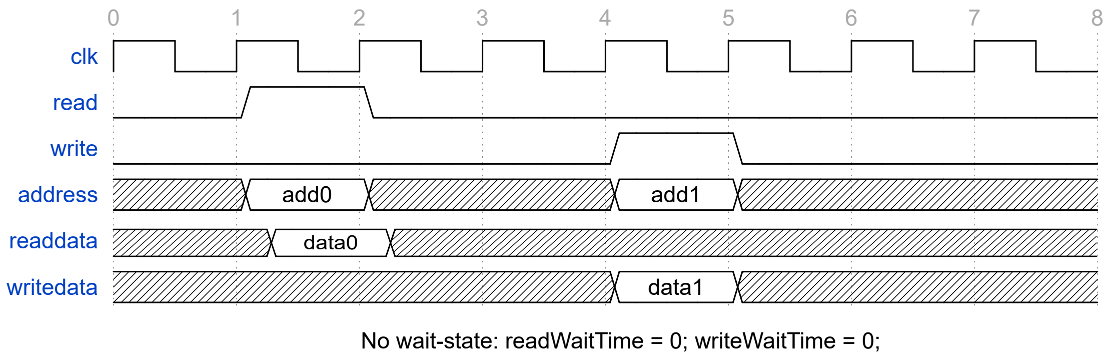


Figure 4.6: Timing diagram of Av-MM read followed by a write transactions with no wait-states.

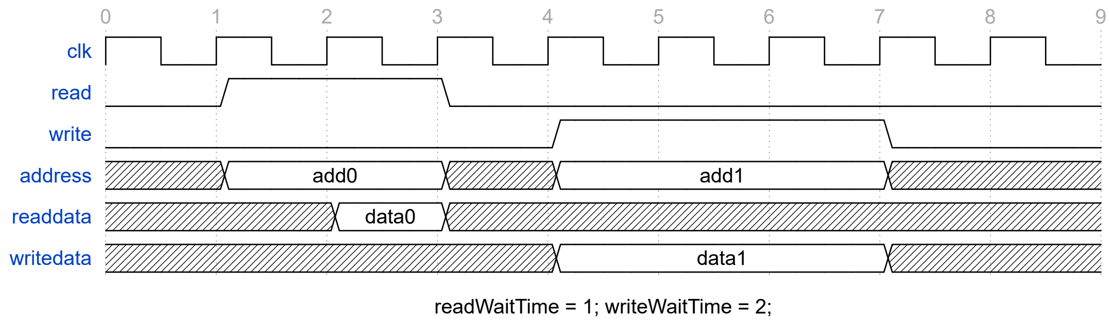


Figure 4.7: Timing diagram of Av-MM read followed by a write transactions with fixed wait-states.

4.1.3 Transmitter PLL: fractional PLL

The clocking generation and distribution is vital for the correct behaviour of high-speed links. In this regard, the transmitter and receiver have distinct contribution:

- At the RX side the channel's PLL is used as CDR to provide the recovered clock and data signals.
- At the TX side an external PLL is used to enhance design's flexibility and transparency. Precisely, two different clock signals need to be generated:
  - High-speed serial clock for the PMA's serializer.
  - Low-speed parallel clock for both PMA and PCS.

  Note that, in case of non-bounded[4] channel configuration (as in the designed system), only the serial clock generated by the external PLL is driven to the transceiver. Conversely, the parallel clock is internally generated, for example a possible solution is a direct connection:

  tx_clkout –> *tx_coreclkin*

For every group of three channels (they can be grouped into three or six), an external PLL can be selected. Generally the Auxiliary Transmit (ATX) PLL is chosen due to its high jitter performance. Instead, fPLL is recommended for relatively low data rate: $(1 \div 12.5)$ Gbps. Since the designed transceiver data rate is within such interval (1.25 Gbps) a fPLL was utilized. An fPLL is similar to the CDR's architecture previously described, as it is clear in the block diagram of Figure 4.8.

The fPLL is a quite complex IP which leaves room for a deep customization. For the proposed design, the default configuration[5] was chosen:

- fPLL mode: transceiver.

[4]A bounded channel is preferred when there are high constraints on the clock skew. With the former configuration the skew between multiple transceiver channels is reduced.

[5]Refer to [63] for the complete list of parameters and the correspondent explanation.

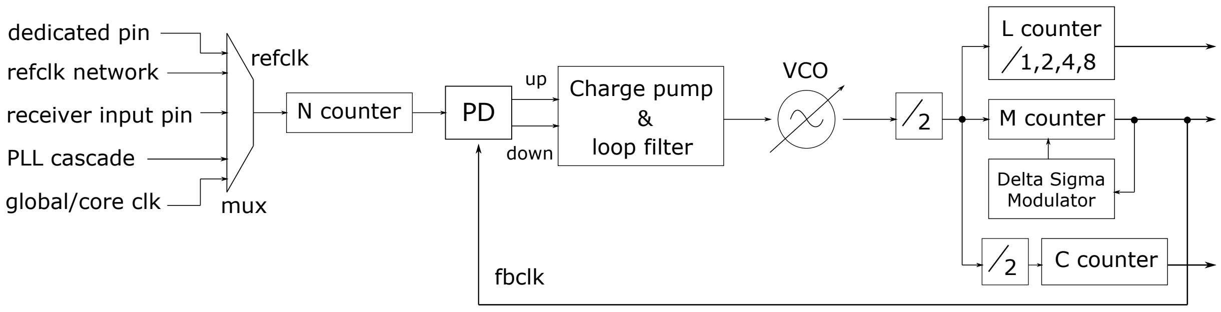


Figure 4.8: Conceptual block diagram of a fPLL, inspired by [63].

- Protocol mode: basic.
- Desired reference clock frequency: 125.0 MHz.
- Number of PLL reference clock: 1.
- Bandwidth: high.
- Operation mode: direct.
- Output frequency: 625.0 MHz.
- Dynamic reconfiguration: disabled.

For what concern the input reference clock many options are available (refer to Figure 4.8). Typically, a dedicated FPGA reference clock pin is used. However, the main requirement is to have a stable and free-running clock signal at power-up to complete a proper calibration.

4.1.4 Transceiver Native PHY

As introduced in the *Background and Literature Study*, the transceiver PHY is mainly composed of PMA and PCS. The Transceiver Native PHY supports both PMA and PCS under the assumption that it is then connected to a MAC IP (the Low Latency Ethernet 10G MAC for example) or to data generator and analyzer directly implemented in the PL. Moreover, it supports many protocols such as: PCIe, various Gigabit Ethernet, CPRI and many others. Finally, if no protocol is specified, the key role is played by the selected data rate and by the transceiver architecture.

The transceiver also required the interconnection with an external PLL (the previously described fPLL) and an external reset controller so that, at the end, it can be viewed as a self-sustained system. The reset controller IP provided by Intel is the natural choice since it is already optimized for such application. Conversely in our case, the Nios subsystem provides the functionality of the reset controller. Intel proposes a dedicated design flow [63]:

- Native PHY IP Core configuration.

- Generation of the transceiver system (Native PHY, reset controller, external PLL) and of the transceiver reconfiguration interface (optional).
- Connection to a MAC IP or data generator and analyzer (implemented in the PL).
- Compilation of the design and verify the functionality through simulation.

Alike the fPLL, many transceiver's parameters are configurable. The layout of the selected configuration (default) is as follows:

- Transceiver configuration rules: basic/custom (Standard PCS).
- Transceiver mode: TX/RX Duplex.
- Data rate: 1 channel at 1250 Mbps.
- TX bonding mode: not bonded.
- Internal loopback: enable *rx_seriallpbken* port.
- Standard PMA/PCS interface width: 10 bits.
- Dynamic reconfiguration: enabled.

PMA architecture

The PMA functionality detailed in the *Background and Literature Study* chapter is here reflected by the Intel PMA architecture. Hence, this subsection is meant to underline the differences.

At first, it is noticeable that the Intel architecture allows to perform a serial loopback (and a reverse loopback) between the TX's serializer and RX's CDR (refer to the dot-line connection in Figure 4.1).

Another peculiarity is the Av-MM interface, which is used to modify the configuration post-instantiation. This feature specifically targets the VGA, CTLE and DFE which cannot be automatically configured. Interestingly, the DFE supports 11 taps which guarantee ISI removal from the sequence of bits (from the current bit till the next 11). The supported modes are listed below:

- Disabled Mode: tap values are set to 0.
- Manual Mode: tap values are manually set (and eventually modified through Av-MM registers).
- Adaption Enabled Mode: tap values are optimized by the Adaptive Parametric Tuning Engine.

Table 4.5 gives a final overview of receiver equalization modes:

The PMA comprehends a channel PLL configured as CDR, whose block diagram is illustrated in Figure 4.9. The key component is the LTR/LTD Controller which handles the lock modes:

Receiver Equalization	Modes
CTLE adaption mode	Manual, Triggered (PCIe Gen3 only)
DFE adaption mode	Adaption enabled, Manual, Disabled
# of fixed taps	3, 7, 11

Table 4.5: Summary of receiver equalization modes (based on Table 251 of [63]).

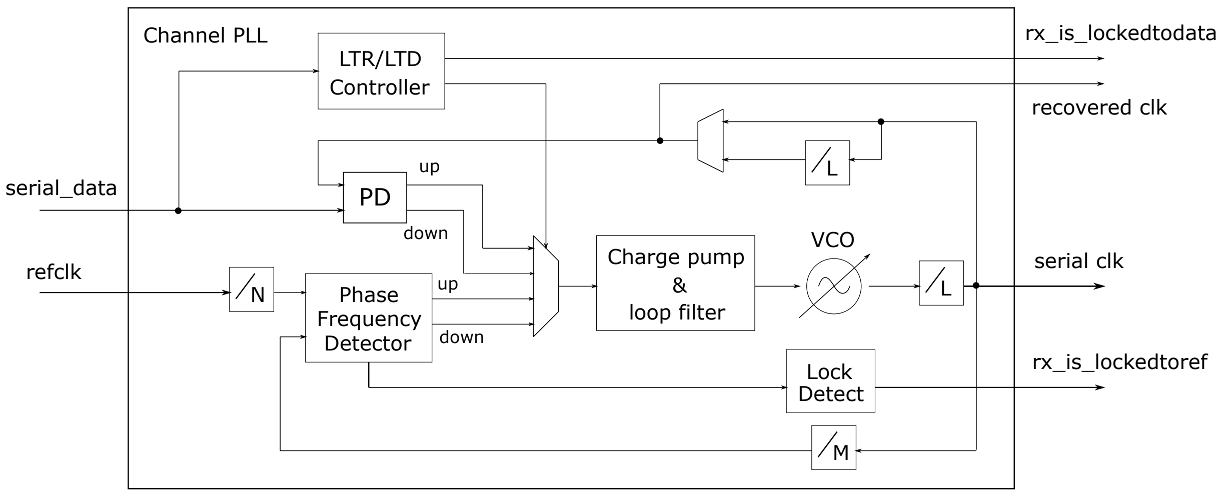


Figure 4.9: Conceptual block diagram of a the Native PHY PMA CDR, inspired by [63].

- Lock-to-Reference Mode (LTR): the CDR tracks the input reference clock (indicated as refclk in Figure 4.9. The PD is inactive since the charge pump is controlled by the phase frequency detector instead. The latter tunes the VCO so that, when the lock condition is achieved, the *rx_is_lockedtoref* signal is asserted.
- Lock-to-Data Mode (LTD): the CDR tracks the input serial data instead of the reference clock. In this mode the PD is used. The PD controls the charge pump (and the VCO as a cascade) with respect to the phase difference. The time required to reach the lock condition depends of the amount of incoming data and the required PPM of phase difference. The *rx_is_lockedtodata* is asserted to indicate the lock acquisition.

By default the lock mode handling is automatically performed by passing from LTR to LTD during regular operation. However, if a quicker CDR lock time is required a manual lock mode handling can be performed. This manual control is done through two additional input ports: *rx_set_locktoref* and *rx_set_lockedtodata*.

To conclude, a list of the chosen configuration parameters is reported:

- PMA supply voltage for the transceiver: 0.9 V.

- PMA configuration rules: basic.
- CDR reference clock frequency: 125 MHz.
- PPM detector threshold: ±1000 PPM.
- PMA CTLE mode: manual.
- PMA DFE mode: disabled.
- PMA output ports: enable *rx_is_lockedtodata*.
- PMA output ports: enable *rx_is_lockedtoref*.
- PMA output ports: enable PRBS verifier control and status ports.

PCS architecture

The transceiver Native PHY PCS architecture is very similar to the one described in the *Background and Literature Study*. Thus, hereinafter only the difference between the two architectures are analyzed. Firstly, here the reset controller is external.

Secondly, the notation is slightly different. The Phase Adjustment block at the TX side (refer to Figure 2.9) is here denoted as TX FIFO. Similarly, the Elastic Buffer task here is performed by the RX FIFO. The RX Phase Adjustment is replaced by the combination of a rate Match FIFO and a Word Aligner blocks.

Note that, in case of external data generator and checker, the RX is not able to distinguish the starting point of the received sequence of bits. Hence, with the 8B/10B transmission code, the special sequence (k28.5) is sent for synchronization. In the designed system things get more complicated since no 8B/10B encoder/decoder is used. Therefore, a synchronization FSM is needed. With this aim the optional *rx_bitslip* port is enabled and used: an edge causes the slip of a single bit. The FSM repetitively slips a bit of the sequence (by driving a *rx_bitslip* edge) until synchronization is reached. This problem is directly avoided while testing with the internal PRBS generator and checker.

The final difference regards the PRBS generator and checking handling. The Native PHY internally has an Av-MM-based dynamic reconfiguration interface to handle it. This interface can be optionally enabled and it requires an Av-MM master to communicate with. This clearly shows the advantage of the Nios usage: the reset controller IP would have required the instantiation of an additional master block. A detailed description of how to enable and customize the PRBS generator and checker is provided later on within the *Software design* section. A summary of the PCS configuration parameters follows:

- PCS FIFO mode: low latency for both FIFO TX and RX.
- PCS byte serializer mode: disabled both in TX and RX.
- PCS 8B/10B Encoder/Decoder: disabled.
- PCS rate match FIFO: disabled.
- PCS Word aligner mode: bitslip with external port for word synchronization in RX.

4.2 Hardware/Software interaction via HAL

A set of SW routines called *drivers* is usually developed to access the underlying HW through a higher-level code, typically in C programming language. Moreover, a set of predesigned libraries is recommended to automatically define the system configuration parameters, such as the base address of each module, and the signal types, like `alt_u8` for unsigned char. A great advantage of the Intel design flow is, precisely, that a basic set of drivers and libraries is already integrated within the HAL framework. The HAL serves as a coherent and transparent interface that allows to program in a flexible way without developing different drivers for every specific target device. This considerably cuts off the firmware development time.

4.2.1 HAL overview and software hierarchy

The HAL can be classified as an intermediate paradigm in between *desktop-like embedded system* and *barebone embedded system.* A comparison is interesting from both the software hierarchy and initialization process perspective. To start with the software hierarchy, a comparison is illustrated by the simplified hierarchies of Figure 4.10.

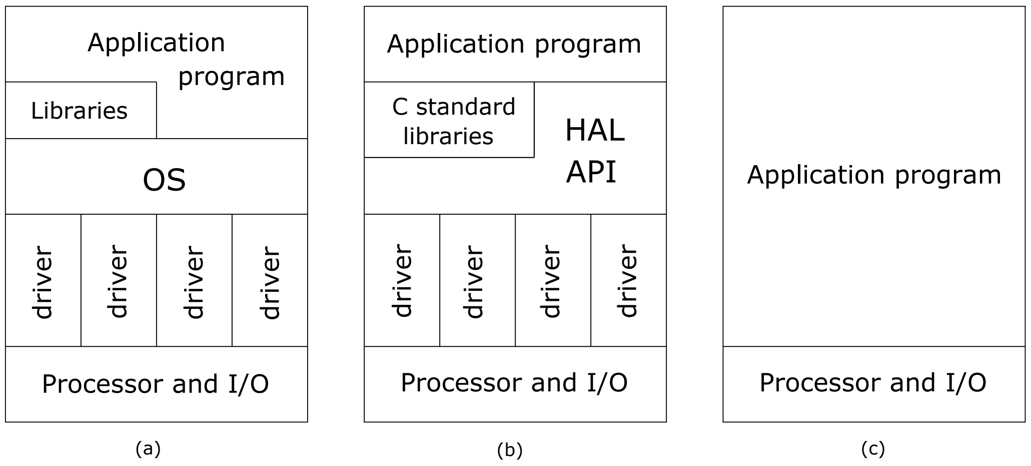


Figure 4.10: Software hierarchy comparison: (a) desktop-like embedded system, (b) HAL-based system, (c) barebone embedded system. The picture is inspired by [62].

Hence, the HAL represents a compromise between the complexity of a completely mediated HW access via OS (desktop) and a fully direct application-hardware interaction (barebone). It provides similar desktop-fashion functionality without requiring an OS. As shown in Figure 4.10, (b) the software hierarchy is made of:

- Device drivers: simple routines to access the I/O resources.
- Application Programming Interface (API): set of utility functions such as timing management, interrupt handling or Unix compatible functions.

- C standard library: ANSI C library functions.

Depending on the I/O device and the design requirements, it is possible to use API and C standard libraries or, to perform direct read/write access to bypass the HAL. Indeed, the HAL usage is not mandatory and it is not recommended in case of devices with HW-specific features (*non-generic devices*). In other words, non-generic devices (such as PIO cores) do not include a mapping to the HAL. Noteworthy examples of device classes compliant with the HAL's model are: character-mode (JTAG UART), timer and flash memory devices. To summarize, an HAL-compliant device driver comprehends:

- A collection of macros to read/write I/O registers (mandatory).
- Status variables to monitor the device's current state.
- A collection of initialization routines.
- Interrupt Service Routine (ISR) and functions to manage hardware interrupt processing.
- BSP configuration components integration.

4.2.2 HAL initialization process

For the initialization process which comprehends HW setup (such as cache flushing, stack configuration, drivers initialization and interrupts enabling) several paradigms are available. In desktop-like systems the OS takes care of the run-time environment preparation and of the program execution scheduling and coordination. As a consequence, the application program is free to start using the libraries and I/O services. Conversely, in a barebone system a start-up code is required to initialize the processor and I/O devices, to set-up the interrupt service and, finally, to coordinate all the operations. Once again, the HAL paradigm lies in between. This means that the environment cannot be dynamically prepared (as with the OS), and that pre-defined initialization routines are firstly executed to perform the set-up tasks. Basically, if all the devices are HAL-compliant, the application program can assume that all the resources and libraries are available without manually performing any initialization task. A summarized list of the initialization steps follows (refer to [62] for the full list):

- Flushing the caches, configuring the stack pointer and the global pointer registers.
- Copying the data to the selected memory module (such as the OCM).
- Programming the interrupt controller and enabling interrupts.
- Initializing the drivers via `alt_sys_init()` function call.
- Preparing the C standard I/O channels.
- Calling `main()` and, ultimately, calling `exit()`.

4.2.3 BSP file structure

Within the HAL environment, a run-time SW, specifically targeted for the Nios II may be provided by the Board Support Package, identified as BSP. At firmware development time, BSP libraries are integrated with user application files to create a bootable image. The BSP libraries comprehend drivers, HAL API code and headers and the compiled object files. Precisely, the drivers configuration is performed through BSP; for example the character mode device is specified and the interval timer core is customized as system timer or as timestamp timer. It is also possible to select a reduced set of C standard libraries in order to considerably reduce the memory size.

4.2.4 I/O access methods and design choices

Based on the mentioned overview, different methods to access I/O devices can be distinguished:

1. via standard library functions.
2. via HAL-compliant functions.
3. via customized device drivers.
4. via direct low-level read/write operations (not recommendable).

The main advantages of methods 1 and 2 is the development of robust, transparent and flexible firmware with no need for manual initialization. Method 3 targets non-compliant HAL devices, since it generally guarantees good performance (optimized code with respect to the device specification) at a price of the initial set-up development. This solution can be optimal in case of small systems with a reduced amount of devices with a limited complexity so that the development time is still acceptable.

Consequently, the design implementation was chosen as follows:

- JTAG UART access via method 2. The natural choice would be to associate the JTAG UART to the `stdout` and `stdin` stream flows by exploiting the high-level C standard `stdio` libraries. This would allow the usage of the `printf()` and `scanf()` functions. As a drawback a considerable amount of memory is required. The reduced set of drivers provided by the HAL, includes the `sys/alt_stdio.h` libraries which features `alt_printf` and `alt_putstr` functions. These functions support limited write on console functionality but with a much smaller memory size with respect to the standard C solution. Moreover, the JTAG UART is automatically initialized by the `ALTERA_AVALON_JTAG_UART_INIT(JTAG_BASE, jtag_name)` belonging to the `alt_sys_init` function.

- Timer configuration and access via method 3. In most of the cases involving a timer core, it is advisable to use the HAL framework. However, for this specific design, customized drivers were developed to reach a resolution of $5\mu s$, required to perform a correct reset sequence. Such resolution is not guaranteed by the usage of the HAL `usleep(unsigned int t)` function. Specifically, the `set_timer_prd(alt_u32 timer_base, alt_u32 prd)` function was implemented to handle the initial configuration.

- PIO communication via method 3. In this case, the design choice coincides with the recommended one, since PIOs are categorized as non-compliant HAL devices. To perform read/write operations `pio_read(base)` and `pio_write(base, data)` driver routines were developed. Finally, a `init_pio()` function was developed to perform the set-up tasks.

- Transceiver dynamic reconfiguration via method 3. In this case, since the reconfiguration was used only for enabling the PRBS generator and checker a customized routine `rd_mod_wr(base, address, rd_mask, wr_mask)` was developed within the `gpio_macros.h` file. The latter routine is used to perfom an *atomic* read-modify-write operation, as required by [63].

4.3 Software design

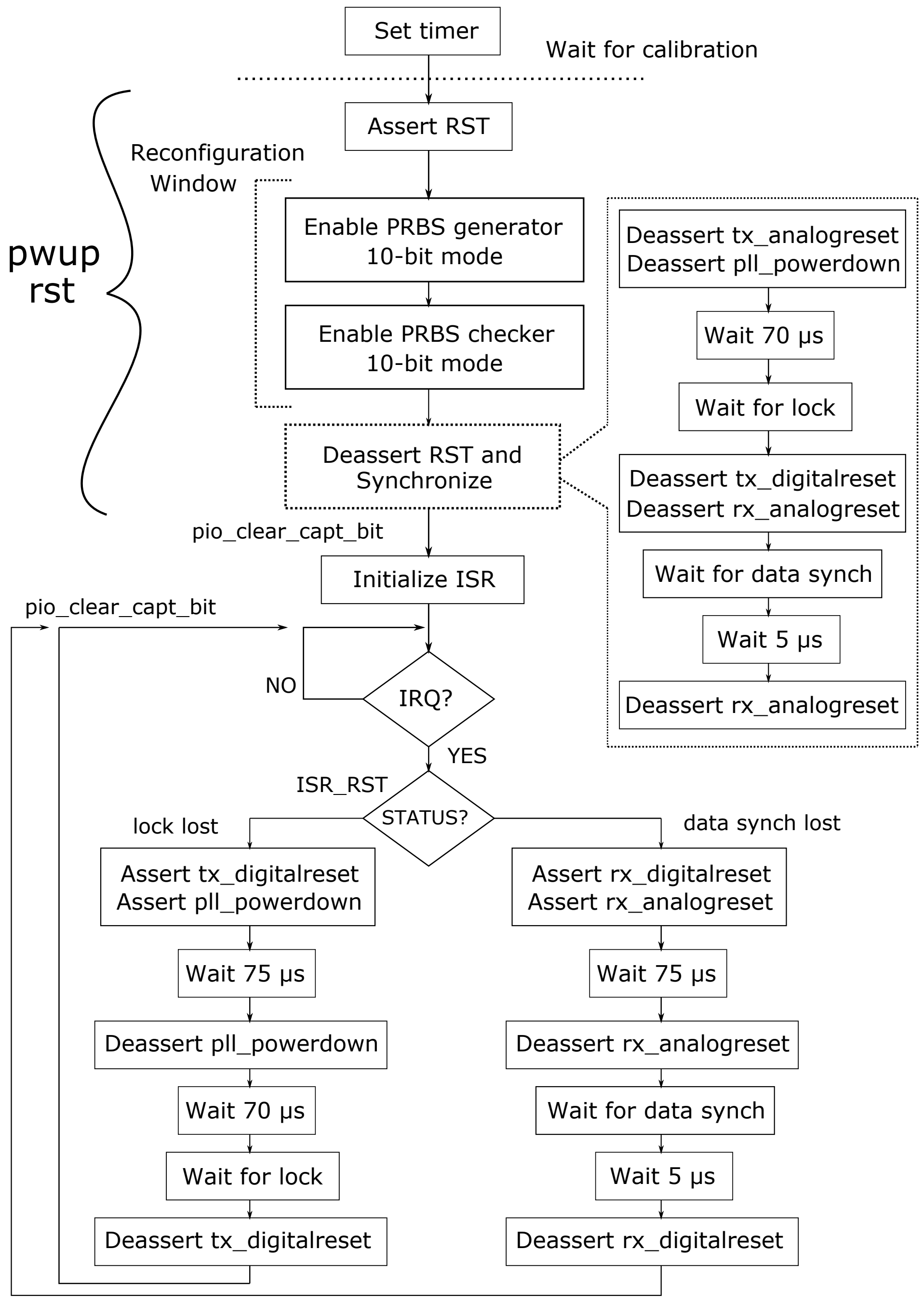


Figure 4.11: Software execution flow of the main() code.

The C-based execution flow, for the transceiver handling, is exemplified in Figure 4.11. The objectives of the code are as follows:

- System reset at power-up.
- Dynamic reconfiguration of the transceiver (limited to the PRBS generator and checker).
- Status register readback with synchronization loss detection.
- Synchronization loss recovery via ISR.

The firmware is based on low-level drivers developed for the PIO and the timer in addition to a main source file implementing high-level functions. A graphic scheme is illustrated in Figure 4.12, whereas the main function's description is summarized in Table 4.6.

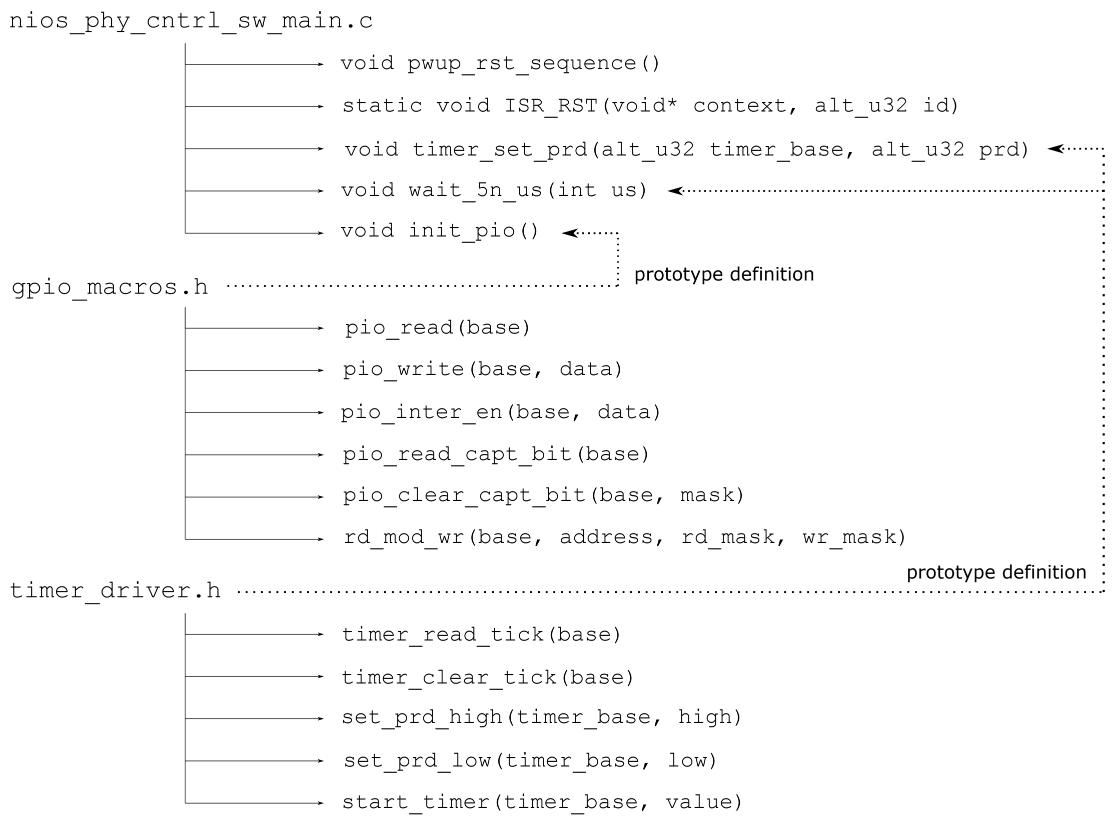


Figure 4.12: Software organization of the functions which are called in the main source file and the developed drivers' routines.

Function Prototype	**Description**
`void pwup_rst_sequence()`	Read/write operations from/to PIOs for the power-up reset sequence and to the PHY's dynamic interface for reconfiguration
`static void ISR_RST( void* context, alt_u32 id)`	Recover from synchronization loss by performing a TX or RX reset
`void timer_set_prd( alt_u32 timer_base, alt_u32 prd)`	Set the timeout period to prd and select the timer's mode (free-running)
`void wait_5n_us(int us)`	Increment ntick counter until the inserted value in μs (resolution: $5\mu s$)
`void init_pio()`	Enable input capture and register the ISR_RST (`alt_irq_register()`)

Table 4.6: Brief description of the functions in the main source file.

4.3.1 Reset and dynamic reconfiguration

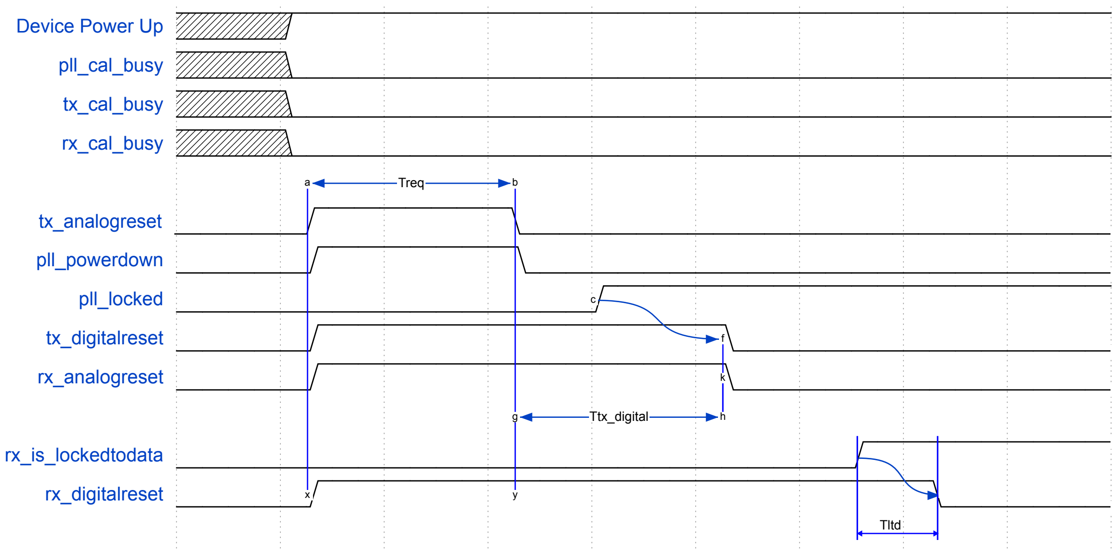
This section explains how to perform the transceiver reset and dynamic reconfiguration by using the Nios II subsystem as embedded controller. Firstly, the timing requirements are analyzed in *Power-up reset sequence* according to the Intel documentation [63]. Secondly, the peripherals' handling is detailed, with a particular interest in the PIO (*PIO handling*) and the timer core (*Timer handling*). Similarly, it is explained how to interact with the PHY's reconfiguration interface. Note that the dynamic reconfiguration is confined to the PRBS generator and checker (*Transceiver dynamic reconfiguration interface handling*).

Power-up reset sequence

The Transceiver Native PHY comprehends a set of input signals whose assertion, in a rigid timing sequence, correctly performs the reset. The timing diagram of the full transceiver reset sequence, as stated in [63], is illustrated in Figure 4.13. Noteworthy input signals are:

- `tx_analogreset`: in charge of the PMA reset (TX side).
- `tx_digitalreset`: in charge of the PCS reset (TX side).
- `rx_analogreset`: in charge of the PMA reset (RX side).
- `rx_digitalreset`: in charge of the PCS reset (RX side).

- `pll_powerdown`: input signal of the external PLL. It is in charge of the fPLL reset.



Minimun intervals: Treq = 70 μs; Ttx_digital = 70 μs; Tltd = 4 μs

Figure 4.13: Timing diagram of the transceiver's reset sequence, according to [63]. Note that the listed intervals state the minimum time to be waited for correct behaviour.

The output status signals used to check the synchronization (LTD/LTR) are:

- `*_cal_busy`[6] output signals of both the fPLL and the transceiver which are de-asserted at calibration completion. It is strictly required to wait until the calibration is completed to start driving the reset sequence.
- `pll_locked`: output signal of the external PLL. It is asserted as soon as the lock is acquired by the fPLL.
- `rx_is_lockedtodata`: asserted as soon as the CDR switches from LTR to LTD mode. As a consequence, data synchronization is guaranteed.

PIO handling

The reset sequence is performed by the Nios II which drives the control signals and reads back the correspondent status signals. The PIOs act as intermediate layer between the controller and the transceiver. Hence, asserting a control signal implies to write the value '1' to the mapped bit within the PIO control registers. Similarly, monitoring a status signal requires the reading of the mapped bit within the PIO status registers. When a set

[6]The * symbol is used as short notation for: `pll_cal_busy`, `tx_cal_busy` and `rx_cal_busy`.

of signals needs to be driven/monitored, masks are applied to complete the correspondent write/read operation.

Let's imagine to sample the signal's level as high or low at a certain instant of time. Now this set of values can be mapped into a binary sequence corresponding to the content of the PIO registers. The mapping assumes that a high-level is associated to a logical '1' whereas a low-level is associated to a logical '0'. This means that, to reproduce the reset sequence (Figure 4.13), it is sufficient to write the correspondent binary sequence (to the PIO control) at every time instants the reset sequence was divided into. To better clarify, a small portion of the reset sequence (shown in Figure 4.14) is *driven* by the simplified C code in Listing 4.1.

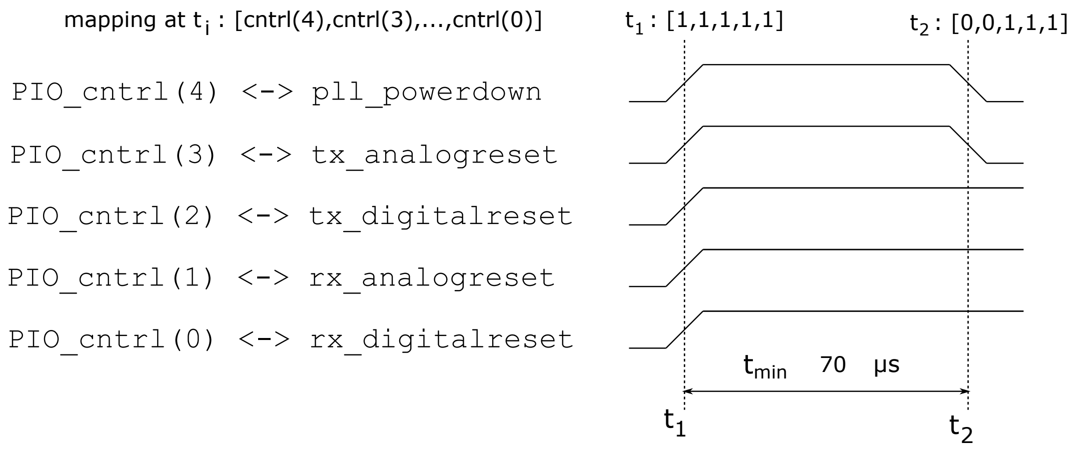

mapping at t_i : [cntrl(4),cntrl(3),...,cntrl(0)]
t_1 : [1,1,1,1,1]
t_2 : [0,0,1,1,1]
PIO_cntrl(4) <-> pll_powerdown
PIO_cntrl(3) <-> tx_analogreset
PIO_cntrl(2) <-> tx_digitalreset
PIO_cntrl(1) <-> rx_analogreset
PIO_cntrl(0) <-> rx_digitalreset
t_{min} 70 μs
t_1
t_2

Figure 4.14: Fraction of the reset sequence with the mapping to the PIO registers.

Listing 4.1 Extract of the C code to perform the reset sequence.

```c
#include "system.h"       //System configuration, base registers
    definition
#include "gpio_macros.h"  //PIO handling
#include "timer_drv.h"    //Timer handling

#define ASS_RST          0x1F
#define DEAS_TXANALOGRST  0x07

int main()
{
  pio_write(CNTRL_SIG_BASE, ASS_RST);            //Assert reset
  wait_5n_us(15);                                //Wait for 75 us
  pio_write(CNTRL_SIG_BASE, DEAS_TXANALOGRST);   //Deassert
    tx_analogreset
}                                                //pll_powerdown
```

Timer handling

An interval timer core was used in the system clock configuration. Its usage is fundamental to correctly temporize the writing operations (to the PIO) by interposing a `wait` function in between. The key idea is to increment a counter (`ntick`) within a loop until the desired value (input parameter) is reached. This is achieved by checking the timer's status register: if the `tick` was reached then this field is cleaned and the `ntick` is incremented by one unit. In order to set the minimum interval of time (`tick`) to $5\mu s$, with the system running at $50MHz$, 250 clocks are required: $5\mu s = 20ns \cdot 250$. Therefore, at initialization phase the timeout period is set to $5\mu s$ by the following function call:

```
timer_set_prd(TIMER_SYS_CLK_BASE, 250)
```

where `TIMER_SYS_CLK_BASE` value is specified within the included `system.h` file. The wait function is shown by Listing 4.2 and it is called in line 11 of Listing 4.1.

Listing 4.2 wait_5n_us(int us) function

```
int ntick = 0;
while(ntick < us)      //wait for multiple of 5 us
{
  if(timer_read_tick(TIMER_SYS_CLK_BASE) == 1)
  {
    timer_clear_tick(TIMER_SYS_CLK_BASE);
    ntick++;
  }
}
```

Transceiver dynamic reconfiguration interface handling

Each channel was provided with a separate Av-MM-based reconfiguration interface to allow a concurrent interaction. This lets the channel free to dynamically adapt in case of changing requirements. In this thesis, the communication with the reconfiguration interface was limited to enable and configure the PCS's PRBS generator and checker. The access to the transceiver's programmable space via Av-MM requires read/write operation which are compliant with the Av-MM specifications. Furthermore, all writes transactions need to be *read-modify-write* or, in other words, *atomic*. This prevents accidental modifications of two or more features sharing the same reconfiguration address (with interleaved bits).

The PRBS9 configuration was chosen by considering the 10-bit width PCS/PMA interface and the data rate below $3Gbps$. To enable this configuration, a precise sequence of steps so-called *direct reconfiguration flow* [63] is followed:

1. Assert the digital reset.
2. Assert the analog reset. This step is required in case of reconfiguration of data rate, protocol mode, or PRBS generator/checker enabling/disabling.

3. Perform a sequence of read-modify-write operations to the specific address to force a defined bit configuration.
4. Deassert the digital reset.
5. Deassert the analog reset.

An additional 3b step may be added to perform re-calibration. This is necessary in case of data rate or protocol mode modification. This general flow allows the transceiver to be modified at any time during regular operation at the expense of an additional reset. Since the requirement is to enable and continuously use the PRBS, the best option is to directly exploit the *reconfiguration window* at start-up. The reconfiguration window is the interval of time, within a reset sequence, in which it is allowed to perform the reconfiguration. For example, in the extract 4.1, the reconfiguration code should be inserted between line 10 and line 12.

A read-write-operation selectively forces (based on the specified masks) some bits (within the input address' location) to the desired value, by keeping all the rest untouched. This is done by forcing the logical '1's while reading (via bitwise AND against a rd_mask) and the logical '0's while writing (via bitwise OR against a wr_mask) as implemented in the following routine:

```
#define rd_mod_wr(base, address, rd_mask, wr_mask) IOWR(base,
    address, ( IORD(base, address) & rd_mask) | wr_mask );
```

where IORD and IOWR are implemented in the HAL's `io.h` source file. Listing 4.3 presents an extract of code to enable the PRBS9 generator according to [63] bit sequences.[7]

Listing 4.3 Reconfiguration window -> Enable PRBS generator 10bit mode

```
rd_mod_wr(TRANS_BASE, 0x006, 0x30, 0x4C); // 1) address 0x006,
                                              bits "01--1100"
rd_mod_wr(TRANS_BASE, 0x007, 0x0F, 0x20); // 2) address 0x007,
                                              bits "0010----"
rd_mod_wr(TRANS_BASE, 0x008, 0x8C, 0x00); // 3) address 0x008,
                                              bits "-000----"
rd_mod_wr(TRANS_BASE, 0x110, 0xF8, 0x04); // 4) address 0x110,
                                              bits "-----100"
```

4.3.2 Synchronization loss recovery via ISR

During the transceiver regular operation, a synchronization loss may occur. This may dramatically affects the system's performance and reliability, thus the system should be able to recover as fast as possible to minimize the impact. Nevertheless, if the system

[7]The symbol - is used to indicate the bit(s) to preserve.

recovers fast *enough*, the impact is almost negligible. In order to have a reactive response, the loss recovery was handled through interrupt. Specifically, whenever the affected signals experience a falling edge, the correspondent ISR is executed to perform a *selective* reset. Selective because the loss is usually confined to the receiver side (falling edge on `rx_is_lockedtodata`) or to the transmitter side (falling edge on `pll_lock`). In the first case, only the RX is reset whereas when it comes to the TX loss, only the digital portion (and the fPLL) needs to be reset. An introduction of the Nios' exception handling system follows.

Nios II exception handling

Any event disrupting the regular flow of execution can be identified as *exception* and needs to be handled. Nios II's *exception handler* consists of a set of SW routines to serve exceptions, and to eventually let the ISR takes control. There is a single exception handler whose code resides at the so-called *exception address* location. Because of the non-vectored controller, the exception handler has to discriminate the type of exception associated to the given address. Besides the different types of exception, the hardware-based ones play a key role in the designed system. Indeed, an HW interrupt is triggered by an HW IRQ PIO-based (falling edge of the status signals). Hardware interrupts are generally managed by dedicated ISR. To summarize, Nios II reacts to interrupts in the following way:

- Keep the information of the actual system's configuration by saving the status register in `eastatus`.
- Set `ienable` to disable HW IRQs with lower or equal priority (it can be defined by SW).
- Save the information of the next instruction to be executed post-interrupt.
- Serve exception.

The registered ISRs are then collected together in a lookup table and they are identified by a unique IRQ ID (assigned in the `system.h` file according to the HW design). The HAL API assists in the design, registration and maintenance of the ISRs. To exploit these functions, every ISR has to match the following prototype (the one expected by the *IRQ register* function):

```
void isr(void* context, alt_u32 id)
```

where `id` is the unique IRQ identifier and `context` is a pointer to information useful for the specific ISR it is associated to. Such ISR is registered by calling the following HAL function:

```
int alt_irq_register(alt_u32 id,
                     void* context,
                     void (*isr)(void*, alt_u32));
```

where `isr` points to the function to be executed to serve the IRQ identified by `id`. A successful registration automatically enables the interrupt on return from the register function itself. Later on, `alt_irq_enable()` and `alt_irq_disable()` may be used to easily enable/disable interrupt based on the id information. Rarely, the enable/disable process is required for all the interrupt and also in that case HAL provides ad-hoc functions (`alt_irq_enable_all()` and `alt_irq_disable_all()`). If a system heavily relies on interrupt, it is worth monitoring *latency*, *response time* and *recovery time*. The latency is the interval of time between the interrupt generation and the execution of the first instruction at the exception address. The response time is the interval of time between the interrupt generation and the execution of the first ISR instruction. Finally, the recovery time is the required time to switch from the execution of the last instruction to the regular operation.

PIO edge capture handling via ISR

As already explained, at design time, the PIO can be configured with additional features: edge capture and IRQ. The first feature is translated into an additional *edge capture* register where, in case of condition met (rising, falling or generic edge), the correspondent bit is set to '1'. Conversely, by including the IRQ, an additional circuit to serve such interrupt request is added. From the HAL perspective, the additional information is automatically included within the `system.h` file, as in the following example targeting the PIO status (called `STATUS_SIG`):

- `#define STATUS_SIG_BIT_CLEANING_REGISTER 1`
- `#define STATUS_SIG_CAPTURE 1`
- `#define STATUS_SIG_EDGE_TYPE "FALLING"`
- `#define STATUS_SIG_IRQ 2`
- `#define STATUS_SIG_IRQ_TYPE "EDGE"`

The combination of edge capture and IRQ was widely used to handle the transceiver. The idea is to recover from a synchronization loss through an ISR whose IRQ is triggered by a falling edge captured by the PIO status. Note that only `pll_locked` (mapped to `PIO_status(0)`) and `rx_is_lockedtodata` (mapped to `PIO_status(2)`) should trigger an IRQ. Thus, a specific routine was developed within the `gpio_macros.h` to selectively enable the interrupt on the PIO signal specified by the input mask: `pio_inter_en(base, mask)`.

However, another issue is present: the PIO status has a unique IRQ as a whole. This implies that only one ISR can be registered with such id. As a consequence, the unique ISR has to handle both types of synchronization loss: at TX and RX side. This introduces an initial overhead since the ISR has to check the edge capture register to recognize the affected signal. Alternatively, an HW modification is required: additional PIOs need to be implemented to develop separated ISRs. However, this is no longer sustainable for complex systems (for example where a battery of links is under control) because of the *explosion* of the resource utilization.

For this thesis, the ISR's overhead is irrelevant, thus no additional HW were implemented. To summarize, the code was developed according to the following steps:

- The ISR (`ISR_RST`) was developed.
- During and after power-up reset the edge capture register was cleaned by the `pio_clear_capt_bit(base, mask)` routine. The input mask selectively targets `PIO_status(0)` and `PIO_status(2)`.
- A PIO status initialization was perform to enable interrupt and to register the ISR by calling `alt_irq_register`.

The `ISR_RST` simply checks for the edge capture register, by isolating the interested bits, and consequently performs a reduced reset sequence. Listing 4.4 illustrates the `ISR_RST`'s flow, with the detailed code for the RX reset sequence. The correspondent timing diagram is illustrated in Figure 4.15. Noteworthy is the dependence between the falling edge on the `rx_is_lockedtodata` and the rising edge of the associated IRQ. Similarly, the TX reset sequence is implemented by asserting/deasserting `tx_digitalreset` and `pll_powerdown` signals, according to the timing diagram of Figure 4.16.

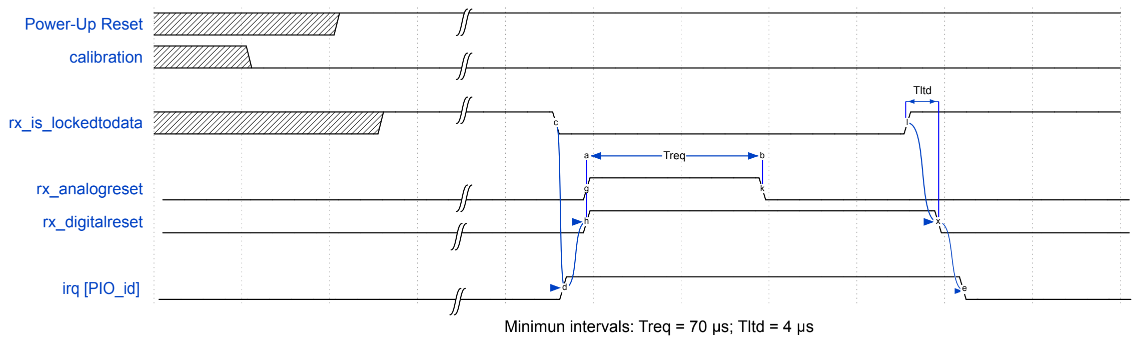


Figure 4.15: Timing diagram of the RX reset sequence.

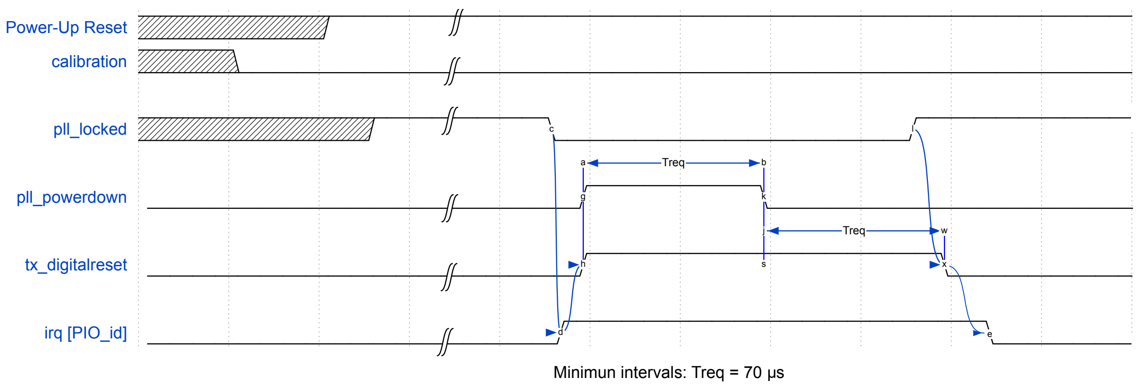


Figure 4.16: Timing diagram of the TX reset sequence.

Listing 4.4 Extract of static void ISR_RST(void* context, alt_u32 id)

```
alt_u8 status_capt;
status_capt = pio_read_capt_bit(STATUS_SIG_BASE);
status_capt &= 0x05;        //Only pll_locked and rx_is_lockedtodata

if(status_capt == lock_is_lost) // == 0x01 ?
{
  //Perform TX reset sequence
}
else
  if(status_capt == lockedtodata_is_lost) // == 0x04 ?
  { //Perform RX reset sequence
    pio_write(CNTRL_SIG_BASE, 0x03);  //Assert rx reset
    wait_5n_us(15);
    pio_write(CNTRL_SIG_BASE, 0x01);  //Deassert analog reset
    do{status_read = pio_read(STATUS_SIG_BASE);}  //Wait for LTD
    while((status_read & 0x04) != 0x04 );
    wait_5n_us(1);
    pio_write(CNTRL_SIG_BASE, 0x00); //Deassert digital reset
    pio_clear_capt_bit(STATUS_SIG_BASE, lockedtodata_is_lost);    //
  Clear capture register
  }
pio_inter_en(STATUS_SIG_BASE, 0x05);  //Enable IRQ
```

Chapter 5

Analysis and Results

The analysis starts from the functional verification through simulation. In this phase, the regular operation and deviation from the expected behavior are simulated. It follows the system synthesis and implementation on the target board with consequent performance evaluation in terms of logic resource utilization, power consumption and software profiling. Finally, the proposed system is benchmarked against an alternative solution with an HW-based system handler.

5.1 Functional simulation

A set of libraries was generated by Qsys for simulation purpose (simulator compliant model of the system). Those libraries, together with the customized top-level testbench, were compiled, elaborated and finally simulated. A memory initialization file (with the compiled firmware) mapped to the OCM allows to test the HW/SW interactions. In order for such test to be effective, the memory initialization file should be included within the simulation directory.

The simulation was structured in a systematic way based on various steps which stimulates the same architecture with different signals. In order to feature a *debug mode*, the implemented system (refer to Figure 4.1) was slightly modified: additional multiplexers and AND gate were introduced. The resulting block diagram is illustrated in Figure 5.1.

The debug mode is used to reset the transceiver independently from the Nios II, by manually driven external signals. Additional external signals are then forced in regular mode (with debug mode off) to emulate the transceiver's loss of synchronization. In this way the Nios II ability to react via ISR is checked.

The progressive set of tests is summarized as follows:

1. Test the transceiver behavior after a *manually* performed reset. Manually means

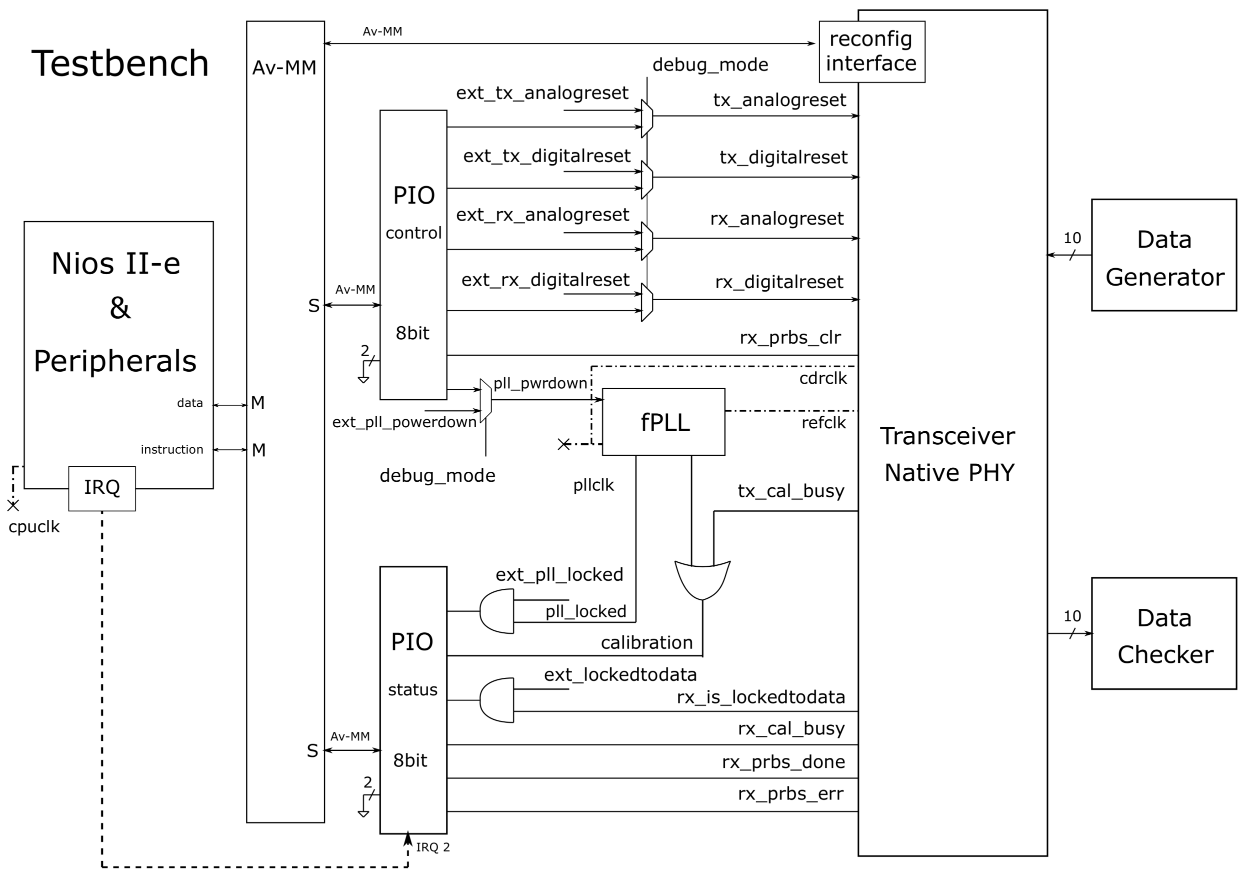


Figure 5.1: Simplified block diagram of the top-level testbench. The Nios II-e system handler comprehends the processor, JTAG UART, OCM and timer, whose interconnection through Av-MM are the same as in the designed system. The debug mode value is assigned as a parameter. When PRBS generator and checker are disabled, an external data generator and checker are used.

that the signal are driven within the top-level testbench by directly define the logical values over time. This first step aims to check the high-speed link functionality independently from the controller's interference.

2. Switch off debug mode to give the control to the Nios II. In this mode, the transceiver is sensitive to the PIO signals instead of the external ones. Therefore, the Nios correct warm up via `pwup_rst_sequence` function call is monitored.
3. Test the system ability to recover from an unexpected behavior by emulating a synchronization loss at the receiver side (falling edge on the externally driven `ext_lockedtodata`) or at the transmitter side (falling edge on the externally driven `ext_pll_locked`). This step plays a key role in the debugging of the `ISR_RST`.

4. Enable the PCS's built-in PRBS generator and checker and perform step 2) and 3) again with this configuration.

During the simulation of the transceiver's regular mode of operation there would be no reason to experience a synchronization loss. Hence, such unexpected behavior was induced by manually acting on the external signals connected to the PIO status. In order to preserve the regular operations, the following connections were performed:

```
PIO_status(0) <= pll_locked AND ext_pll_locked
PIO_status(2) <= rx_is_lockedtodata AND ext_lockedtodata
```

To simulate the regular operating mode `ext_pll_locked` and `ext_lockedtodata` are fixed to '1' so that the PIO status listens to the fPLL and the transceiver. Conversely, to induce a synchronization loss a falling edge is manually driven to the external signals to trigger the PIO status IRQ and, to consequently execute the `ISR_RST`. In order to simplify the debug process, it is convenient to include the Nios II IRQ associated to the PIO status, to the set of waveforms checked during simulation.

The verification process is successful if the same simulation waveform are obtained in case 1) 3) and 4) and if such waveforms are compliant to the specifications [63]. The same should happen in case of an induced loss of synchronization in both 3) and 4).

5.2 Synthesis and performance evaluation

The verified system was compiled, synthesized, programmed onto the Arria 10 SoC board and, finally, characterized in terms of logic utilization, power consumption and software profiling (refer to *PoC's characterization*). The same procedure was carried out for the benchmark system. The key difference is that the system handler functionality is implemented in HW by the Intel *Transceiver PHY Reset Controller* IP. The reset controller interface is targeted to be connected to the transceiver Native PHY.

5.2.1 PoC's benchmark against a traditional HW-based controller

The aim of this benchmarking is to *evaluate* the performance drawback of the SW implementation of the system's controller (hereinafter called *NiosII controller*). The Intel Transceiver PHY Reset Controller IP is used as traditional HW-based controller. This IP is specifically designed to reset transceiver native PHY by monitoring the PLL lock activity. The complete system, hereinafter called *HW controller*, is realized through the direct connection of the reset controller to the transceiver. In order to perform this comparison, external data generator and checker have been used to provide/check data to/from the transceiver in both cases.

The first metric to be compared is the resource utilization. The post-compilation data are collected in Table 5.1. As expected, the introduction of a soft-core processor leads to an increase of logic utilization, registers and memory usage. In particular, the data shows an increment in logic utilization of approximately 18 times. The impact depends on the system's requirements and on the total amount of available programmable logic.

Resource	**HW controller**	**NiosII controller**
Logic utilization (ALM)	51 / 251,680 (< 1%)	955 / 251,680 (< 1%)
Total registers	99	1684
Total pins	12 / 812 (1%)	10 / 812 (1%)
Total block memory bits	0 / 43,642,880 (0%)	43,008 / 43,642,880 (< 1%)
Total HSSI RX channels	1 / 48 (2%)	1 / 48 (2%)
Total HSSI TX channels	1 / 48 (2%)	1 / 48 (2%)
Total PLLs	2 / 96 (2%)	2 / 96 (2%)

Table 5.1: Comparison of the logic utilization between the Intel-based system (HW controller) and the proposed SW-based solution (NiosII controller). The data are collected from the Quartus reports.

Secondly, the power consumption was compared. The comparison is based on both the EPE (Table 5.2) and the PowerPlay Analyzer (Table 5.3) estimations[1]. An increment in the logic utilization, generally implies an increase in the static power consumption, since this power contribution evaluates the leakage of the system. Conversely, the static power consumption of the HW controller and the NiosII controller is almost the same. This result is proven in case of both EPE and PowerPlay analyzer. Probably, this is due to the tool resolution: the logic utilization increment is negligible, thus, the power estimation is not enough accurate to reveal the correspondent increment in static power consumption. In conclusion, due to the limited complexity of the proposed system, the impact on the power consumption is so small that it cannot be reveal through the available tool. As a consequence, it is not possible to generalize this result.

Based on the collected data, and, by limiting the analysis on the specific case of study, the conclusion is that it is worthy to pay for the additional flexibility provided by the SW introduction.

[1]For more details about the EPE and PowerPlay power estimation methods, refer to *Design Methodologies and Preliminar Analysis* Chapter.

[2]The routing thermal dynamic power contribution has been included within the dynamic contribution.

Power contribution (W)	**HW controller**	**NiosII controller**
Logic	$1.38 \cdot 10^{-4}$	$8.35 \cdot 10^{-5}$
Clock	$4.82 \cdot 10^{-4}$	$1.73 \cdot 10^{-4}$
PLL	0.116	0.116
I/O	$3.60 \cdot 10^{-4}$	$4.73 \cdot 10^{-5}$
Transceiver (XCVR)	0.247	0.247
Static power	1.485	1.485
Total thermal power	1.850	1.849

Table 5.2: EPE thermal power estimation: comparison between the Intel-based system (HW controller) and the proposed SW-based solution (NiosII controller).

PoC power contribution (W)	**HW controller**		**NiosII controller**	
	standby	dynamic	standby	dynamic
Clock[2]	0.00	$9.44 \cdot 10^{-3}$	0.00	$5.78 \cdot 10^{-3}$
PLL	$7.88 \cdot 10^{-3}$	0.103	$7.88 \cdot 10^{-3}$	0.103
Register cell[2]	0.00	$1.10 \cdot 10^{-4}$	0.00	$1.10 \cdot 10^{-4}$
I/O Analog	$8.00 \cdot 10^{-5}$	$2.70 \cdot 10^{-4}$	$5.00 \cdot 10^{-5}$	0.00
I/O Digital	$1.10 \cdot 10^{-4}$	$2.0 \cdot 10^{-5}$	$9.00 \cdot 10^{-5}$	0.00
Transceiver (XCVR)	0.188	0.158	0.188	0.153
Static power	1.534		1.533	
Total thermal power	1.890		1.880	

Table 5.3: PowerPlay Analyzer thermal power estimation: comparison between the Intel-based system (HW controller) and the proposed SW-based solution (NiosII controller); under an Ambient Temperature of 25 °C, and a Junction temperature of 28.4 °C. The data are collected from the *Thermal Power Dissipation by Block Type* report, in order to have a detailed comparison.

PoC's performance evaluation

In conclusion, additional measurements were carried out for the PoC with the internal PRBS generator and checker enabled. The resource utilization information is collected in

Table 5.4, whereas the thermal power consumption is summarized in Table 5.5. Nevertheless, it is interesting to notice that, the value obtained with the EPE are close to the more accurate ones provided by the PowerPlay Analyzer. This would suggest that it is beneficial to perform an EPE estimation (at least at the beginning of the design process).

Resource	**PoC**
Logic utilization (ALM)	1,084 / 251,680 (< 1%)
Total registers	1976
Total pins	9 / 812 (1%)
Total block memory bits	43,008 / 43,642,880 (< 1%)
Total HSSI RX channels	1 / 48 (2%)
Total HSSI TX channels	1 / 48 (2%)
Total PLLs	2 / 96 (2%)

Table 5.4: Resource utilization of the PoC. The data are collected from the Quartus reports.

PoC power contribution (W)	**EPE**	**PowerPlay**	
		standby	dynamic
Clock[3]	$1.73 \cdot 10^{-4}$	0.00	$5.47 \cdot 10^{-3}$
PLL	0.116	$5.47 \cdot 10^{-3}$	0.103
I/O	$3.77 \cdot 10^{-5}$	$1.30 \cdot 10^{-4}$	0.00
Transceiver (XCVR)	0.247	0.188	0.153
Static power	1.485	1.533	
Total thermal power	1.849	1.880	

Table 5.5: PoC's thermal power estimation through EPE and PowerPlay Analyzer. The PowerPlay report specify the thermal power consumption (static and dynamic) for each block under an Ambient Temperature of 25 °C, and a Junction temperature of 28.4 °C. In this table such information has been summarized to have a comparison with the EPE estimation.

[3]The routing thermal dynamic power contribution has been included within the dynamic contribution.

5.2.2 PoC's software profiling

The PoC's characterization is enriched by SW profiling. The performance evaluation was performed by an additional high-resolution timer (interval timer core configured as timestamp timer), through `alt_timestamp()` function call at the top and the bottom of the code under analysis. The timer management is simplified by the HAL's `alt_timestamp.h`, which also contains an initialization function and the `alt_timestamp()` itself. Unfortunately this driver requires additional memory: the OCM size was increased to 8 KB. Two different measurements were carried out: time interval of the power-up reset and time spent within the `ISR_RST`. This was done by associating a couple of variable (`start_time` and `stop_time`) to each measurement. Finally, the computed intervals (`end_time - start_time`) were printed to console (due to the JTAG UART configured as `stdout`):

- power-up reset sequence : $167\mu s$
- recover from data synchronization loss (RX reset) : $207\mu s$
- recover from lock synchronization loss (TX reset) : $268\mu s$

These data are inevitably affected by the timer's overhead: the `timestamp()` function call requires many clock cycles to be performed. However, by considering the order of magnitude of hundreds of μs, it is reasonable to consider such contribution as negligible.

For what concern the power-up reset, the measured value is considerably below the threshold of ms, and, it is also close to the optimum value[4] of approximately $165\mu s$.

Conversely, for what concern the `ISR_RST`, despite the results are still acceptable, there is room for further studies and improvements. Indeed, the approximate optimum would be: $155\mu s$ for the TX reset and $90\mu s$ for the RX reset, where the included switching time between LTR and LTD was hypothetically set to $10\mu s$.

To sums up, this analysis not only proves that the proposed firmware fulfills the timing requirements (with a minimum amount of resource utilization), but it also identifies the starting point for future improvements.

[4]This value was computed based on the delays manually introduce, via `wait_5n_us()` to reproduce the timing sequence. Note that the time required to switch from LTR to LTD is supposed to be $10\mu s$. This estimation may not be accurate since it is affected by different factors.

Chapter 6

Conclusion

The continuously increasing demand for minimally-sized, highly complex, and low-power electronic devices, implies a shorter TTM, while raising the probability of error-prone systems. An additional degree of flexibility has a double benefit: simplification of bugs localization, and consequent correction; on the other hand the possibility of tailoring the functionalities to meet slightly modified requirements.

The abstraction of the system's controller, through SW implementation, represents a possible contribution to flexibility. This approach, proposed by Ericsson AB to address high-speed links' applications, was investigated in the master thesis by developing and characterizing a PoC, whose peculiarity is a software-based controller. Indeed, the firmware, running on a Nios II soft-core processor, substitutes a traditional HW-based FSM to control the reset and dynamic reconfiguration of a transceiver PHY. In details, Nios II is in charge of driving the control signals, while monitoring the transceiver's status. Additionally, a dedicated ISR allows the system to recover from unexpected synchronization losses.

A top-level testbench was written to perform a functional simulation of the PoC. In this phase, the correct interaction between the firmware and the underlying HW was proven. Afterwards, the effectiveness of the transceiver's reset and the consequent operative mode were tested. Finally, the system's ability to recover from malfunctioning (caused by manually induced loss of synchronization) was checked. Additionally, through software profiling, the Nios II performance were evaluated, and the obtained result, clearly underlines that the timing requirements are met.

Finally, the PoC was benchmarked against a system with a traditional HW-based controller (Intel Transceiver PHY Reset Controller IP). The logic utilization data show that, the introduction of a soft-core processor increments the resource utilization of approximately 18 times. In this case, the complexity of the system is reduced respect to the capability of the Intel Arria 10. However, it is advisable to evaluate the impact of such increment for more complex systems. For what concern the power consumption, no direct conclusion can be drawn: the results would suggest that the adopted techniques are not accurate enough to differentiate between the two solutions. Thus, it is reasonable to assume that only a considerable increment in the static power consumption (correspondent to an increment in the logic utilization), would have been revealed.

The developed PoC successfully demonstrates the benefits of a SW-based controller implementation in terms of reduced engineering effort required to design, perform modification (there is no need to re-synthesize) and debug. On-the-fly modifications are also simplified: the transceiver can be dynamically reconfigured through software read-modify-write operations. On the other hand, the clear limitation is the reduced complexity of the proposed system. Consequently, this promising approach requires more researches in order to evaluate the impact of a scaling-up.

6.1 Future Work

The overcome the PoC's limitations, and move towards the release of newer FPGA/ASIC, additional explorations are suggested:

- The PoC may be improved by the introduction of an external flash memory, containing the hardware description information (.sof file), and the developed firmware (.elf file). In this way, there would be a direct boot from flash, without relying on a host PC. A connection with the host PC would be limited to a debug purpose.
- Besides monitoring the system conditions, the controller should desirably perform error traceback, so that it would act as HW *supervisor*. More specifically, the supervisor should be able to localize the errors, and to analyze error flag traces in order to identify the root cause. This approach would simplify the debug process, and consequently, cut down the correspondent time and cost.
- An examination of possible alternatives for the embedded controller would be beneficial. The starting point may be the comparison with the *fast* version of the Nios II, which should improve the performance such as the interrupt latency, at a price of additional logic utilization. Noteworthy would also be the exploitation of the SoC's functionalities through a HPS-based solution.
- To intensely experience the benefits of flexibility the system complexity has to increase. For example, it would be interesting to adapt the firmware in order to control various transceivers. This scaling-up needs to be investigated in terms of both HW and SW, so that an optimal partitioning may be reached. Moreover, with the additional complexity, further questioning arise: whether an OS-supervised implementation would be advantageous, for example. Nevertheless, the feasibility is limited by the logic utilization and power consumption constraints, and therefore, an accurate measurement is required for such key metrics.

Bibliography

[1] "Rising Demand for Electronic Devices to Drive the Global Semiconductor Deposition Market Through 2020, Says Technavio," 28-Jun-2016. [Online]. Available: `http://www.businesswire.com/news/home/20160628005321/en/Rising-Demand-Electronic-Devices-Drive-Global-Semiconductor`. [Accessed: 05-Oct-2017].

[2] T. Hooper, "Top Electronics Manufacturing Trends and Challenges in 2017," Pannam, 21-Jan-2017. [Online]. Available: `https://www.pannam.com/blog/top-trends-and-challenges-in-electronics-manufacturing/`. [Accessed: 05-Oct-2017].

[3] H. Wu, J. Kang, M. Chi, X. Zhang, T. Feng, and Poren, "The technology trend of IC manufacture during Post Moore's era", in *2017 China Semiconductor Technology International Conference (CSTIC)*, 2017, pp. 1 –7.

[4] J. S. Vetter, E. P. DeBenedictis, and T. M. Conte, "Architectures for the Post-Moore Era", *IEEE Micro*, vol. 37, no. 4, pp. 6 –8, 2017.

[5] D. D. Gajski and R. H. Kuhn, "Guest Editors' Introduction: New VLSI Tools", Computer, vol. 16, no. 12, pp. 11 –14, Dec. 1983.

[6] H. D. Foster, "Why the design productivity gap never happened", in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 581 –584.

[7] H. Abdelkrim, S. B. Othman, and S. B. Saoud, "Reconfigurable SoC FPGA based: Overview and trends", in *2017 International Conference on Advanced Systems and Electric Technologies (IC_ASET)*, 2017, pp. 378 –383.

[8] Intel (2017). "Intel Arria 10 Device Overview", *A10-OVERVIEW*, version 2017.09.20.

[9] "2015 ITRS 2.0 Executive Report", *International Technology Roadmap for Semiconductor (ITRS)*, 2015.

[10] S. Sabbavarapu, K. R. Basireddy, and A. Acharyya, "A New Dynamic Library Based IC Design Automation Methodology Using Functional Symmetry with NPN Class Representation Approach to Reduce NRE Costs and Time-to-Market", in *2014 Fifth International Symposium on Electronic System Design*, 2014, pp. 115 –119.

[11] A. Solomatnikov et al., "Chip Multi-Processor Generator", in *2007 44th ACM/IEEE Design Automation Conference*, 2007, pp. 262 –263.

[12] E. Larsson, Ed., "Design Flow", in *Introduction to Advanced System-on-Chip Test Design and Optimization*, Boston, MA: Springer US, 2005, pp. 5 –19.

[13] H. D. Foster, "Why the design productivity gap never happened," in 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2013,

pp. 581 –584.

[14] Padmanabhan, T. R., & Bala Tripura Sundari, B. (2005). "Introduction to VLSI Design". In *Through Verilog HDL* pp. 1 –9. Hoboken, NJ, USA: John Wiley & Sons.

[15] V. A. Chandrasetty, "FPGA Application Design", in *VLSI Design: A Practical Guide for FPGA and ASIC Implementations*, V. A. Chandrasetty, Ed. New York, NY: Springer New York, 2011, pp. 17 –46.

[16] S. Ramachandran, Ed., "Introduction to Digital VLSI Systems Design", in *Digital VLSI Systems Design: A Design Manual for Implementation of Projects on FPGAs and ASICs Using Verilog*, Dordrecht: Springer Netherlands, 2007, pp. 3 –31.

[17] K. S. Mohamed, "Introduction", in *IP Cores Design from Specifications to Production: Modeling, Verification, Optimization, and Protection*, K. S. Mohamed, Ed. Cham: Springer International Publishing, 2016, pp. 1 –11.

[18] E. Larsson, Ed., "Introduction", in *Introduction to Advanced System-on-Chip Test Design and Optimization*, Boston, MA: Springer US, 2005, pp. 1 –4.

[19] E. Larsson, Ed., "Design for Test", in *Introduction to Advanced System-on-Chip Test Design and Optimization*, Boston, MA: Springer US, 2005, pp. 21 –52.

[20] E. Larsson, Ed., "An Integrated Framework for the Design and Optimization of SOC Test Solutions", in *Introduction to Advanced System-on-Chip Test Design and Optimization*, Boston, MA: Springer US, 2005, pp. 187 –214.

[21] Ericsson AB; Huawei Technologies Co. Ltd; NEC Corporation; Alcatel Lucent; Nokia Networks. CPRI *Specification* V7.0, 2015.

[22] A. X. Widmer and P. A. Franaszek, "A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code", *IBM J. Res. Dev.*, vol. 27, no. 5, pp. 440 –451, Sep. 1983.

[23] J. D. Day and H. Zimmermann, "The OSI reference model", *Proc. IEEE*, vol. 71, no. 12, pp. 1334 –1340, Dec. 1983.

[24] T. Noergaard, "Chapter 2 - Know Your Standards", in *Embedded Systems Architecture (Second Edition)*, Newnes, 2013, pp. 21 –85.

[25] H. Zhang, S. Krooswyk, and J. Ou, "4.1.1 Embedded Clock Architecture", in *High Speed Digital Design - Design of High Speed Interconnects and Signaling*, Elsevier.

[26] T. Noergaard, "Chapter 6 - Board I/O", in *Embedded Systems Architecture (Second Edition)*, Newnes, 2013, pp. 261 –293.

[27] R. A. Philpott, J. S. Humble, R. A. Kertis, K. E. Fritz, B. K. Gilbert, and E. S. Daniel, "A 20Gb/s SerDes transmitter with adjustable source impedance and 4-tap feed-forward equalization in 65nm bulk CMOS", in *2008 IEEE Custom Integrated Circuits Conference*, 2008, pp. 623 –626.

[28] E. M. Cherry and D. E. Hooper, "The design of wide-band transistor feedback amplifiers", *Proc. Inst. Electr. Eng.*, vol. 110, no. 2, pp. 375 –389, Feb. 1963.

[29] J. Lee, "A 20-Gb/s adaptive equalizer in 0.13-μm CMOS technology", *IEEE J. Solid-State Circuits*, vol. 41, no. 9, pp. 2058 –2066, Sep. 2006.

[30] H. Higashi, S. Masaki, M. Kibune, S. Matsubara, T. Chiba, Y. Doi, H. Yamaguchi, H. Takauchi, H. Ishida, K. Gotoh, and H. Tamura, "A 5–6.4-Gb/s 12-channel transceiver with pre-emphasis and equalization", *IEEE J. Solid-State Circuits*, vol. 40, no. 4, pp. 978 –985, Apr. 2005.

[31] D. Dunwell and A. C. Carusone, "Gain and equalization adaptation to optimize

the vertical eye opening in a wireline receiver", in *IEEE Custom Integrated Circuits Conference 2010*, 2010, pp. 1 –4.

[32] S. Bottacchi, "Decision Feedback Equalization", in *Multi-Gigabit Transmission over Multimode Optical Fibre*, John Wiley & Sons, Ltd, 2006, pp. 509 –583.

[33] Chen L, Zhang X, Spagna F., "A scalable 3.6-to-5.2 mW 5-to-10 Gb/s 4-tap DFE in 32 nm CMOS", in *ISSCC dig tech papers*, Feb. 2009. p. 180 –81.

[34] W. Redman-White et al., "A Robust High Speed Serial PHY Architecture With Feed-Forward Correction Clock and Data Recovery", *IEEE J. Solid-State Circuits*, vol. 44, no. 7, pp. 1914 –1926, Jul. 2009.

[35] Alexander, J., "Clock recovery from random binary signals", *Electron. Lett.*, 1975, 11, (22), pp. 541 –542.

[36] K. Mueller and M. Muller, "Timing Recovery in Digital Synchronous Data Receivers", *IEEE Trans. Commun.*, vol. 24, no. 5, pp. 516 –531, May 1976.

[37] B. Razavi, "Design of integrated circuits for optical communications", *1st ed.*. Boston: McGraw-Hill, 2003.

[38] W. P. Ranjula, R. M. A. U. Senarath, D. P. D. Senaratna, G. D. S. P. Senaratne, and S. Thayaparan, "Implementation techniques for IEEE 802.3ba 40Gbps Ethernet Physical Coding Sublayer (PCS)", in *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2015, pp. 1 –5.

[39] G. Nicholl, M. Gustlin, and O. Trainin, "A Physical Coding Sublayer for 100GbE [Applications Practice]", *IEEE Commun. Mag.*, vol. 45, no. 12, pp. 4 –10, Dec. 2007.

[40] L. Qihao, W. Huihui, Z. Feng, Z. Jianzhong, L. Junsheng, and L. You, "An efficient physical coding sublayer for PCI express in 65nm CMOS", in *2012 International Symposium on Intelligent Signal Processing and Communications Systems*, 2012, pp. 625 –629.

[41] Y. G. P. Kumar, B. S. Kariyappa, and M. Z. Kurian, "Implementation of power efficient 8-bit reversible linear feedback shift register for BIST", in *2017 International Conference on Inventive Systems and Control (ICISC)*, 2017, pp. 1 –5.

[42] R. Wisniewski, M. Wisniewska, and M. Adamski, "Effective Partial Reconfiguration of Logic Controllers Implemented in FPGA Devices", in *Design of Reconfigurable Logic Controllers*, A. Karatkevich, A. Bukowiec, M. Doligalski, and J. Tkacz, Eds. Cham: Springer International Publishing, 2016, pp. 45 –55.

[43] S. I. Kartashev and S. P. Kartashev, "Dynamic Architectures: Problems and Solutions", *Computer*, vol. 11, no. 7, pp. 26 –40, Jul. 1978.

[44] S. P. Kartashev and S. I. Kartashev, "Software problems for dynamic architectures: Adaptive assignment of hardware resources", in *Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International*, 1978, pp. 775 –780.

[45] Xilinx, "Two Flows for Partial Reconfiguration: Module Based or Difference Based", in *XAPP290 (v1.2) Application Note*, September 9, 2004.

[46] V. Sklyarov, I. Skliarova, and J. Silva, "Synthesis and Implementation of Parallel Logic Controllers in All Programmable Systems-on-Chip", in *Design of Reconfigurable Logic Controllers*, A. Karatkevich, A. Bukowiec, M. Doligalski, and J. Tkacz, Eds.

Cham: Springer International Publishing, 2016, pp. 15 –29.

[47] V. Sklyarov, “Hierarchical finite-state machines and their use for digital control”, *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, vol. 7, no. 2, pp. 222 –228, Jun. 1999.

[48] V. Sklyarov, “Reconfigurable models of finite state machines and their implementation in FPGAs”, emphJ. Syst. Archit., vol. 47, no. 14, pp. 1043 –1064, Aug. 2002.

[49] R. Jayaraman, H. B. Basiron, and P. M. Sanga Pillai, “Programming Microcontroller via Hierarchical Finite State Machine”, in Intelligent Robotics Systems: Inspiring the NEXT, 2013, pp. 454 –463.

[50] Xilinx, “Zynq-7000 All Programmable SoC”, in *UG585 (v1.12.1)*, December 6, 2017.

[51] ARM Ltd. (2010). “Cortex-A9 MPCore Technical Reference Manual”, Revision r2p2, ARM DDI 0407F (ID050110), 30 April 2010.

[52] Madl, G., Pasricha, S., Bathen, L., Dutt, N., & Zhu, Q. (2006). “Formal performance evaluation of AMBA-based system-on-chip designs”. *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, 311 –320.

[53] Intel Altera (2016). “Nios II Gen2 Processor Reference Guide”, NII51001, version 2016.10.28.

[54] Y. Hailin, “Design of the high-speed image acquisition system based on NiosII”, in *IEEE 2011 10th International Conference on Electronic Measurement Instruments*, 2011, vol. 3, pp. 225 –227.

[55] R. Yue, Y. Wen-Ji, and W. Jinming, “An FPGA Based Multi-functional Signal Generator Using SOPC Design Methodology”, in *2016 3rd International Conference on Information Science and Control Engineering (ICISCE)*, 2016, pp. 1257 –1261.

[56] P. G. Salunke and A. M. Sayyed, “Design of embedded web server based on NIOS-II soft core processor”, in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016, pp. 488 –492.

[57] Y. Chen and F. H. Zhang, “The Base Band Design for UHF RFID Reader Based on Nios II”, in *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing*, 2008, pp. 1 –4.

[58] W. Stallings, “Reduced instruction set computer architecture”, *Proc. IEEE*, vol. 76, no. 1, pp. 38 –55, Jan. 1988.

[59] Bhandarkar, D. “RISC versus CISC: A tale of two chips”, *ACM SIGARCH Computer Architecture News*, 25(1), 1997, pp. 1 –12.

[60] Intel, “Intel Quartus Prime Standard Edition Handbook Volume 3 Verification”, *Updated for Intel Quartus Prime Design Suite: 17.1*, QPS5V3 2017.11.06, pp. 161 –182.

[61] Intel, “Early Power Estimator for Intel Arria 10 FPGAs User Guide", *UG-01164*, 2017.03.13 p. 57.

[62] P. P. Chu, “Embedded SoPC Design with Nios II Processor and VHDL Examples", *Hoboken, NJ, USA: John Wiley & Sons, Inc.*, 2012.

[63] Intel, “Intel Arria 10 Transceiver PHY User Guide", in *UG-01143*, 2017.04.20.