ELSYS

INSTITUT FÜR LEISTUNGSELEKTRONISCHE SYSTEME

Development of an FPGA based Value Acquisition System for a Real-Time Control Platform

Thilo Wendt

TECHNISCHE HOCHSCHULE NÜRNBERG
INSTITUT ELSYS

www.th-nuernberg.de/elsys

Abstract

This report addresses the design and verification of an FPGA based solution for actual value acquisition in the domain of power electronics. In this connection, the FPGA acts as a master of a serial peripheral interface which is connected to the output of a 16 bit ADC. Besides the acquisition of the raw value, a hardware solution for post processing is included as well. The report presents a universal design methodology for FPGA projects as well as an analysis of the environment and the target platform, the solution is designed for. Furthermore, the verification with a static test bench has been performed in the project. The report contains an analysis of the opportunities and drawbacks of static test bench approaches in comparison to object oriented concepts. Finally, recommendations for further hardware developments within the project are given based on the outcomes of the development of the actual value acquisition system. Target audience are electrical engineers with a background in the design of digital hardware with hardware description languages. A descent understanding of VHDL is recommended.

Nomenclature

Abbreviations

a.k.a.	Also Known As
ADC	Analog Digital Converter
AXI	Advanced eXtensible Interface
DUT	Design Under Test
e.g.	exampli gratia
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FUNC	Functional Requirement
HDL	Hardware Description Language
i.e.	id est
IO	Input Output
IP	Intellectual Property
LSB	Least Significant Bit
MSB	Most Significant Bit
NOF	Non Functional Requirement
PCB	Printed Circuit Board
RPU	Real-Time Processing Unit
SAHT	Sample And Hold Time
SoC	System-on-a-Chip
SPI	Serial Peripheral Interface
SysML	System Modeling Language
VHDL	Very High Speed Integrated Circuit HDL

Contents

1 Introduction

1.1 Motivation

A crucial step in every control application is the acquisition of actual values from the target system. In the domain of power electronics, the application usually requires control cycles of a few microseconds because of the short time constants of the target system. Therefore, the component being responsible for the acquisition of actual values shall induce minimum latency. The goal of the current project is the development of an IP core for an FPGA, that is capable to operate a number of analog digital converters (ADC) simultaneously while maintaining a low resource footprint. The target platform is a Xilinx Zynq UltraScale+ SoC FPGA which is embedded in a system targeting the rapid prototyping of new control algorithms for power electronic devices. Equipped with a highly performant computation unit, the system, which is available under the label "UltraZohm", is capable to meet hard real time requirements even for highly dynamic target devices with a large number of switching states at high switching frequencies [1]. The UltraZohm hardware features a fast and precise ADC which shall be operated by the newly created IP core presented in this report.
The implementation on an FPGA is motivated by the capability to parallelize process by using a hardware implementation. In contrast to a processor, FPGA based implementation do not depend on the instruction set of the processor it is rather a custom hardware solution for the problem to be solved, which offers great parallelization opportunities. Furthermore, the SoC FPGA in the UltraZohm offers comprehensive digital signal processing hardware that is used for further post processing of the acquired information.
Besides the development of the hardware component, the design workflow for direct hardware development shall be examined for eligibility for the UltraZohm project. As depicted in [1] the usual workflow in the UltraZohm project is the code generation of hardware descriptions by using High Level Synthesis by Xilinx or the Matlab HDL coder by MathWorks. While these workflows are eligible for rapid prototyping, the solution may not be as optimized as a manually tailored component. Since the acquisition of actual values with a fixed external hardware is no subject to frequent major changes a more optimized solution is desirable. However, it must be examined, if the workflow is suitable for further developments in the UltraZohm project.

1.2 Structure of the Report

The report is subdivided in the following parts: After a short introduction to the problem and a justification of the presented development in the current chapter, theoretic fundamentals concerning system and logic design are given in chapter 2. The chapter contains a methodology description of the hardware design workflow applied in the project as well as a context and requirements analysis for the IP core. Finally, an abstract architecture is given in the chapter. The implementation as well as the verification of the components of the architecture is described in chapter 3. Considerations for future developments and a discussion about the methods applied in the project are given in the final chapter 4.

1.3 Project Goals

The goal of the current project is the development of a base component for actual value acquisition on a Xilinx Zynq UltraScale+ SoC FPGA. It is a contribution to the UltraZohm project and therefore, the existing hardware of the UltraZohm control system is being used. The solution shall offer maximum flexibility concerning software control options and further developments. Therefore, a strict hierarchic component design shall be applied in order to offer the possibility of design reuse and to ensure maintainability. With this approach, the complexity is kept at a minimum level. Furthermore, the solution shall maintain a low resource footprint while achieving the highest physically possible sampling rates of the ADC.

2 Theory

The following chapter contains an introduction to the methodologies and the system design on which the implementation described in chapter 3 is based. Section 2.1 contains a brief introduction to the logic design with finite state machines (FSM). Section 2.2 and 2.3 cover the system and requirements analysis, on which the architecture of the IP core depicted in section 2.4 is based.

2.1 Methodology

The following section gives a brief introduction to logic design with FSMs. Firstly, a decision for a suitable FSM design is given in section 2.1.1. Section 2.1.2 and 2.1.3 cover the implementation of the FSM in a hardware description language (HDL).

2.1.1 Finite State Machines for Logic Design

It is common practice to describe digital circuits using FSM [2, p. 45]. This comes with the advantages of a standardized design process and deterministic synthesis results. Moreover, the synthesis tools provide support for FSM structures [3, p. 171].
In general, an FSM that is implemented in hardware is composed of a state memory S a transition function δ and an output function λ [3, p. 171]. A distinction between different FSM types is made concerning the relation between the output signal Y^t, the current state Z^t the input signal X^t and the output function λ. Generally, FSMs can be categorized in Mealy, Moore and Medvedev types. [4] Within the scope of this project, the Mealy FSM with adjustments to the structure presented in [4] forms the basic FSM type for all implementation in the IP core. The final structure is illustrated by Fig. 2.1.

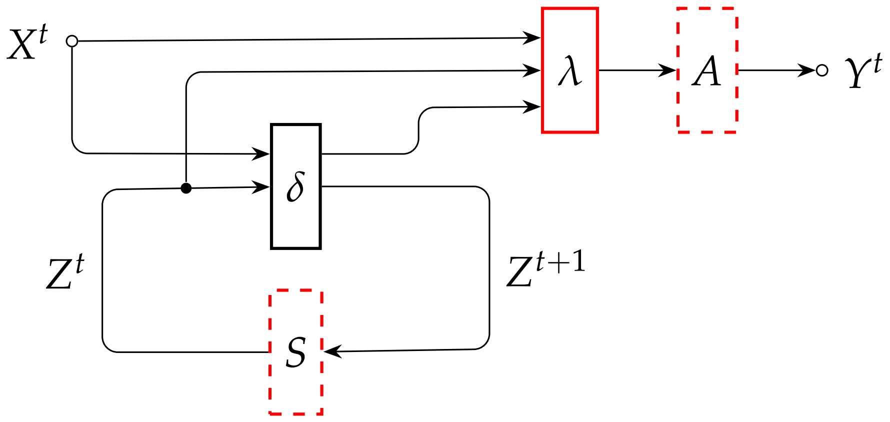


Figure 2.1: Basic FSM structure for the implementation in the IP core. Dashed components are sequential logic whereas solid components are combinational logic. All components marked in red are implemented in a single sequential process, whereas the transition logic δ is realized in a separate combinational process. The structure has been derived from [4] and [5].

The decision for the structure presented in Fig. 2.1 is explained within the following paragraphs. For externally available signals it is desirable to produce a hazard free output with a low jitter. Within the scope of the current project, the SPI of the ADC that shall be operated by the FPGA demands for a high signal quality [6]. In order to guarantee an output signal without hazards, an output register is required. Without the output register, the function λ is directly exposed to the physical interface which potentially leads to hazards due to the combinational nature of λ [5]. This requirement can either be satisfied by the Medvedev FSM where the state memory register directly forms the output signal or by a Mealy FSM extended with an output register A.
In a Medvedev implementation, the output vector is directly formed directly by the state memory S. This leads to a high verification effort in case of a large output vector and only few states being used [4]. In comparison to the Medvedev FSM the Moore and the Mealy FSM offer more flexibility since the output signal is not directly coupled to the state memory but it is formed by an arbitrary output function. The Mealy structure offers the possibility to produce different output signals while staying in the same state whereas the output of the Moore structure exclusively depends on the current state. Generally, a Mealy FSM can be converted to a Moore FSM but for every different output signal that is produced in the same state at the Mealy structure a new state in the Moore structure is created [2]. This leads to an increased number of states which is confusing for the implementation in VHDL. Therefore, the Mealy FSM with an output register has been considered as the basic structure for all FSM implementations in the IP core. In contrast to the structures presented in [4], the information about the current state Z^t is fed to the output function λ as well. This is necessary to distinguish between transitions from different states. [5] introduces another structure, where δ and λ are combined into a single process. Indeed, this offers the possibility to consider Z^t for the formation of the output signal as well. These approaches have been merged together to the structure presented in Fig. 2.1.

2.1.2 State Reduction of FSMs

In general, an FSM with a minimal number of states is desirable for the implementation in hardware in order to save resources. However, the first draft of an FSM usually does not lead to a minimal number of states. Therefore, a state reduction needs to be performed before implementing the FSM in hardware. The general concept of a state reduction is derived from [2, p. 39-44]. The key concept of the state reduction is based on the following principles [2, p. 39]: Two states are equivalent if

1. ...they produce the same output signal for the same input signal.
2. ...they transition to an equivalent state for the same input signal.

However, it is fundamental to these principles that two states which shall be tested for equivalency expect the same input signal. Consequently, it needs to be verified if there exist states that expect the same input signal in a first step. If this is not the case, no equivalent states exist in the FSM.

The method shall be illustrated on the example of the SPI master which is implemented in the ADC IP core. The state reduction is performed in the following five steps:

1. Definition of a potentially not minimal state machine.
2. Definition of a formal input alphabet X.
3. Definition of a formal output alphabet Y.
4. Examination of the states for equivalence.
5. Design of a minimal FSM.

Fig. 2.2 shows the FSM of the above mentioned SPI master. The definition of this FSM is not further explained in this section but a detailed description can be found in section 3.1.1.

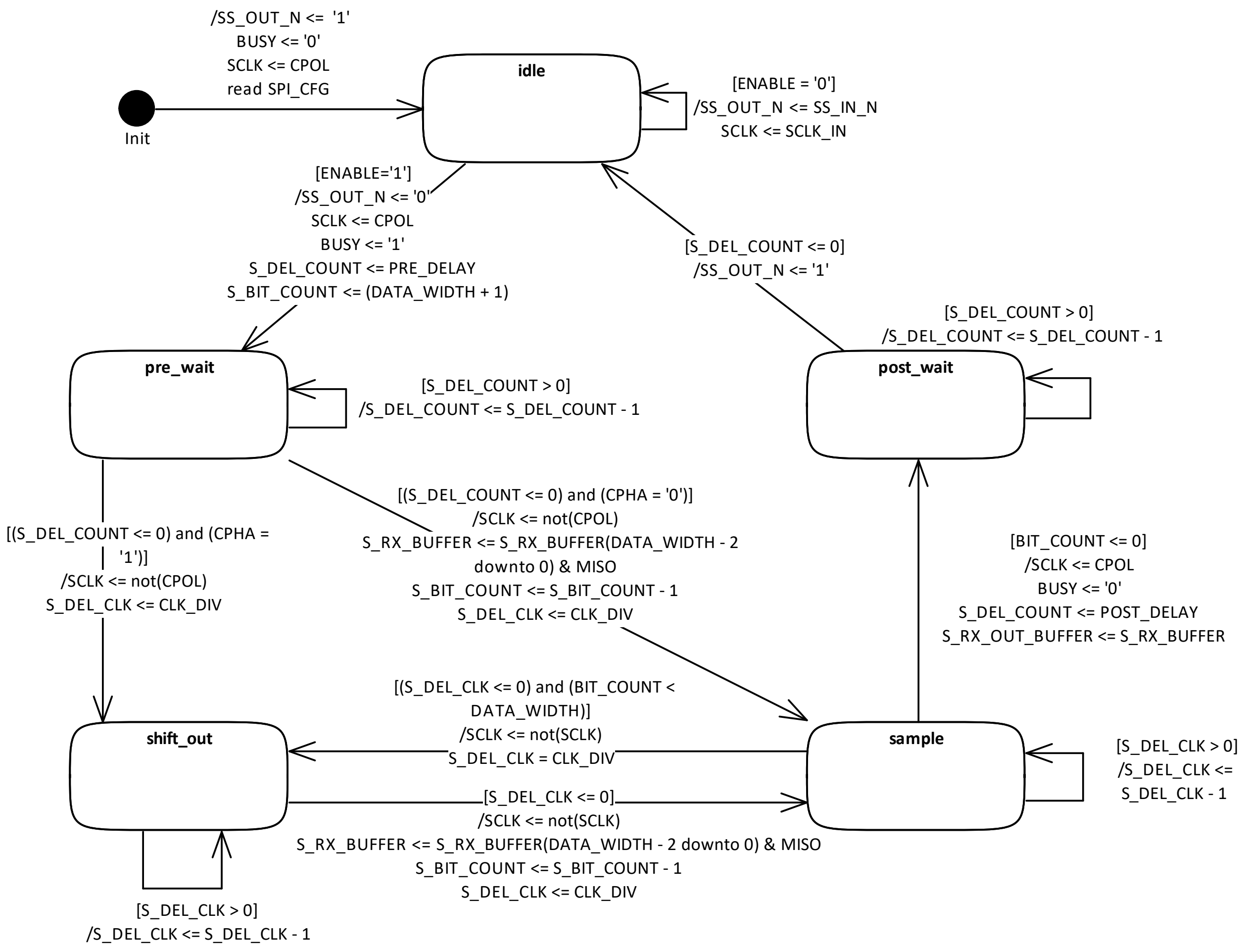


Figure 2.2: FSM of the SPI master. The conditions for the transition are written in square brackets while the output signal is described after the slash on the transition arrow.

Table 2.1: Input alphabet of the FSM of the SPI master. Boolean values like `ENABLE` or `CPHA` can be inserted in the alphabet directly while integer values must satisfy a certain condition in order to produce a Boolean value that can be modeled by a bit in the input vector.

Vector index	**Condition**
0	ENABLE
1	CPHA
2	S_DEL_COUNT > 0
3	S_DEL_COUNT ≤ 0
4	S_DEL_CLK > 0
5	S_DEL_CLK $\leq$
6	S_BIT_COUNT > 0
7	S_BIT_COUNT ≤ 0

However, this representation of the FSM is not suitable for a direct state reduction. The conditions marked on the transitions in the FSM form the input grammar which is based on the input alphabet. As a first step, a formal input alphabet is defined and input words are identified. The input alphabet is defined by a vector where every unique condition defines a bit. The encoding is shown in Table 2.1.
Based on the input alphabet defined in Table 2.1 an input grammar can be identified for the FSM from Fig. 2.2. The grammar is composed of input words that are formed by the input alphabet. Allowed values for the bits in the input vector are `true (1)` `false (0)` and `don't care (-)`.

Table 2.2: Input grammar of the FSM of the SPI master. The bits in the input vector are further explained in Table 2.1.

	Bit in input vector							
Transition	0	1	2	3	4	5	6	7
IDLE $\rightarrow$ IDLE	0	-	-	-	-	-	-	-
IDLE $\rightarrow$ PRE_WAIT	1	-	-	-	-	-	-	-
PRE_WAIT $\rightarrow$ PRE_WAIT	-	-	1	0	-	-	-	-
PRE_WAIT $\rightarrow$ SHIFT_OUT	-	1	0	1	-	-	-	-
PRE_WAIT $\rightarrow$ SAMPLE	-	0	0	1	-	-	-	-
SHIFT_OUT $\rightarrow$ SHIFT_OUT	-	-	-	-	1	0	-	-
SHIFT_OUT $\rightarrow$ SAMPLE	-	-	-	-	0	1	-	-
SAMPLE $\rightarrow$ SAMPLE	-	-	-	-	1	0	1	0
SAMPLE $\rightarrow$ SHIFT_OUT	-	-	-	-	0	1	1	0
SAMPLE $\rightarrow$ POST_WAIT	-	-	-	-	-	-	0	1
POST_WAIT $\rightarrow$ POST_WAIT	-	-	1	0	-	-	-	-
POST_WAIT $\rightarrow$ IDLE	-	-	0	1	-	-	-	-

After the definition of the input alphabet and the input grammar, the states can be examined for equal input words. For example the self transitions of PRE_WAIT and POST_WAIT expect the same input word but when transitioning to another state, the input words differ. In conclusion, no states with equal input grammars can be found and therefore the investigation for transitions to equal states and the examination of the output signal is not applicable. The considered FSM with the given input signals is therefore minimal.
Similar considerations have been performed for the other FSMs, which are explained in section 2.4. However, non of the considered FSMs could be minimized and therefore the procedure for the other FSMs is not shown in this report.

2.1.3 FSM Implementation in HDL

When a minimal FSM is found, it can be implemented in a hardware description e.g. in VHDL. Using three processes is the most convenient method to describe an FSM in VHDL. Referring to Fig. 2.1, S, A and λ are implemented in a single sequential process, whereas δ is realized by a combinational process. The states of the FSM are defined as an enumeration as shown in listing 2.1. The synthesis tool provides the attributes `fsm_encoding` to define the encoding of the states. For example the encoding can be set to one hot if this is required by the application. However, in this case no defined encoding is required and therefore the attribute is set to `auto` which allows the tool to determine the most eligible encoding. The attribute `fsm_safe_state` can be set to synthesize extra logic to avoid a deadlock in an undefined state [3, p. 52]. Fig. 2.1 shows that the signals `curstate`, which holds the current state corresponding to Z^t and the signal `nxtstate`, which holds the output of the transition function δ corresponding to the next state Z^{t+1}, are required for the operation of the FSM.

```
type state_type is (IDLE,PRE_WAIT,SHIFT_OUT,SAMPLE,POST_WAIT);
signal curstate, nxtstate : state_type := IDLE;
attribute fsm_encoding : string;
attribute fsm_encoding of curstate, nxtstate : signal is "auto";
attribute fsm_safe_state : string;
attribute fsm_safe_state of curstate, nxtstate : signal is "power_on_state";
```

Listing 2.1: State declaration of an FSM in VHDL.

After the definition of the states, the combinational process for the transition function δ can be derived from the graphical representation from Fig. 2.2. The sensitivity list is composed of the signals from the input alphabet as defined in Table 2.1 and the current state. This corresponds to the signal flow from Fig. 2.1. The conditions for the state transitions can be directly copied from the graphical representation of the FSM. Listing 2.2 shows the implementation of the transition function. The description of S is shown in listing 2.3 and the implementation of λ and A are given in listing 2.4. Referring to Fig. 2.2, the listing contains the output signals given induced by the state transitions. The implementation matches the signal flow presented by Fig. 2.1.

```vhdl
transition: process(curstate, ENABLE, S_CPHA, S_DEL_COUNT, S_DEL_CLK, S_BIT_COUNT)
begin
  case curstate is
    when IDLE =>
      if (ENABLE = '1') then nxtstate <= PRE_WAIT;
      else                   nxtstate <= IDLE;
      end if;
    when PRE_WAIT =>
      if     ((S_DEL_COUNT <= 0) and (S_CPHA = '1')) then nxtstate <= SHIFT_OUT;
      elsif ((S_DEL_COUNT <= 0) and (S_CPHA = '0')) then nxtstate <= SAMPLE;
      else                                               nxtstate <= PRE_WAIT;
      end if;
    when SHIFT_OUT =>
      if  (S_DEL_CLK <= 0) then nxtstate <= SAMPLE;
      else                      nxtstate <= SHIFT_OUT;
      end if;
    when SAMPLE =>
      if     (S_BIT_COUNT <= 0)                         then nxtstate <= POST_WAIT;
      elsif ((S_DEL_CLK <= 0) and (S_BIT_COUNT > 0)) then nxtstate <= SHIFT_OUT;
      else                                                nxtstate <= SAMPLE;
      end if;
    when POST_WAIT =>
      if (S_DEL_COUNT <= 0) then nxtstate <= IDLE;
      else                       nxtstate <= POST_WAIT;
      end if;
    when others => nxtstate <= IDLE;
                   report "Undecoded State" severity note;
  end case;
end process transition;
```

Listing 2.2: VHDL description of the transition function δ.

```vhdl
 91  output_state_mem: process(CLK)
 92  begin
 93    if rising_edge(CLK) then
 94      if (reset_n = '0') then
 95        curstate            <= IDLE;
 96        S_DEL_COUNT         <= 0;
 97        S_DEL_CLK           <= 0;
 98        S_BIT_COUNT         <= 0;
 99        S_RX_OUT_BUFFER     <= (others => '0');
100        S_RX_BUFFER         <= (others => '0');
101        S_PRE_DELAY         <= (others => '0');
102        S_POST_DELAY        <= (others => '0');
103        S_CLK_DIV           <= (others => '0');
104        S_SCLK              <= CPOL;
105      else
106        curstate <= nxtstate;
```

Listing 2.3: VHDL description of the state memory S. The implementation features a synchronous reset which is implemented in line 94 to 104. The actual state memory is implemented in line 106.

```vhdl
107         case nxtstate is
108           -- Transition to IDLE
109           when IDLE =>
110             BUSY <= '0';
111             -- latch in SPI config
112             S_PRE_DELAY <= PRE_DELAY;
113             S_POST_DELAY <= POST_DELAY;
114             S_CLK_DIV <= CLK_DIV;
115             S_CPOL <= CPOL;
116             S_CPHA <= CPHA;
117             -- pull SS high at least for one clock cycle
118             case curstate is
119               when POST_WAIT =>
120                 SS_OUT_N <= '1';
121               when others =>
122                 if (MANUAL = '1') then
123                   SS_OUT_N <= SS_IN_N;
124                   S_SCLK <= SCLK_IN;
125                 else
126                   SS_OUT_N <= '1';
127                   S_SCLK <= S_CPOL;
128                 end if;
129             end case;
```

Listing 2.4: VHDL description of the output function λ. The given code is part of the process from listing 2.3. Therefore, it describes a part of the combinational circuit λ, which forms the input for the output register A. Line 107 shows is the output of the transition function δ a.k.a. the next state. Line 118 describes the input of the current state.

2.2 Context Analysis

The IP core is a contribution to the UltraZohm project. Therefore, a basic understanding of the UltraZohm is necessary. In brief, the UltraZohm is a general purpose platform for the rapid prototyping of control algorithms for power electronic systems. The system is build around a Xilinx Zynq UltraScale+ SoC FPGA providing several interfaces for the acquisition of actual values and the output of control signals. Fig. 2.3 shows the hardware of the UltraZohm. The outer interface of interest in this report is the Analog Adapter Board, since the ADCs for the value acquisition are located here [1]. In the following sections, an introduction to the hardware and software ecosystem of the UltraZohm is given.

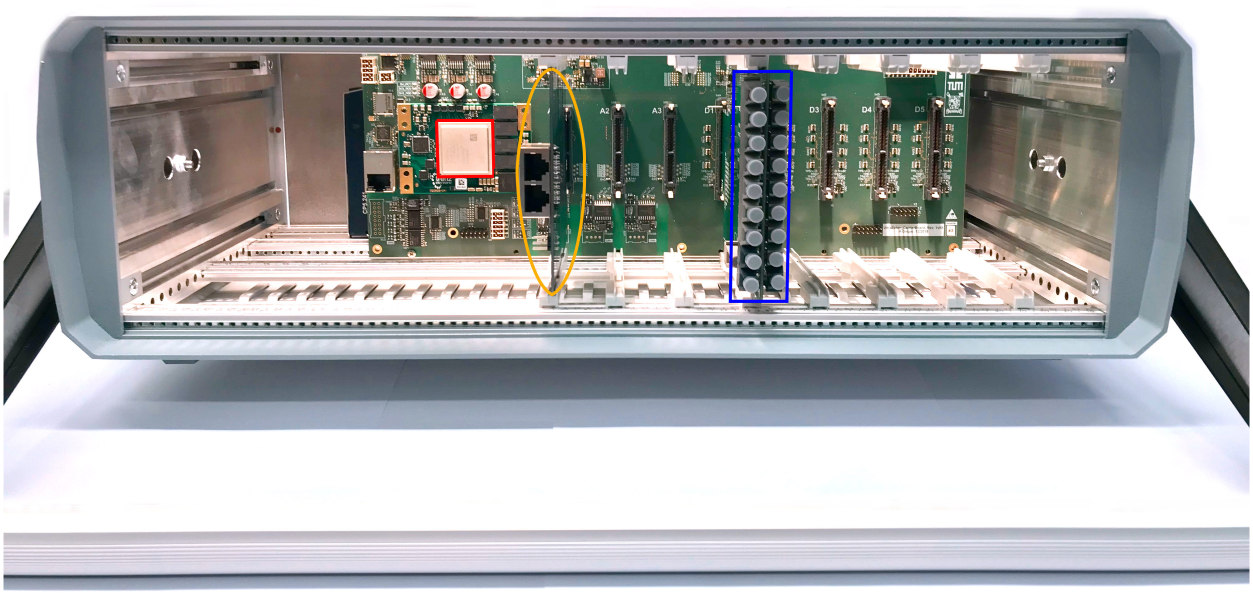


Figure 2.3: Hardware of the UltraZohm. From left to right: SoC (red), analog adapter board (yellow) and digital adapter board with optical transmitters (blue).

2.2.1 Hardware System Context

In order to abstract the system ”UltraZohm” a SysML representation is used within this section. Fig. 2.5 displays the UltraZohm system in it’s usual context. Since this project exclusively deals with the efficient acquisition of actual values from externally connected voltage source inverters, the interface of interest is the ”AnalogSignals” interface. The physical medium is a twisted pair cable which is connected to the UltraZohm via RJ45 connectors as shown in Fig. 2.3. Fig. 2.4 displays the connection from the outer interface of the UltraZohm to the Analog Adapter Board. The Analog Adapter Board is responsible for converting the analog value from the voltage source inverter to a digital value. It carries the LTC2311 ADC and appropriate signal adjustments. Within the scope of this report the main interface of interest is the connection from the LTC2311 via the Carrier Board to the SoC FPGA. This interface can be subdivided into the following components:

- The LTC2311 as an SPI slave
- The signal traces on the carrier board
- The input and output buffers in the FPGA
- SPI master implemented in the FPGA

Especially the SPI interface of the ADC and the routing on the printed circuit boards (PCB) feature significant delay times that must be taken in account when operating the system. A detailed analysis of the hardware situation is performed in section 3.1.1. The ADC IP core is the functional interface against the hardware of the ADC. It is responsible to control the data flow into the FPGA which is generated by the ADC.

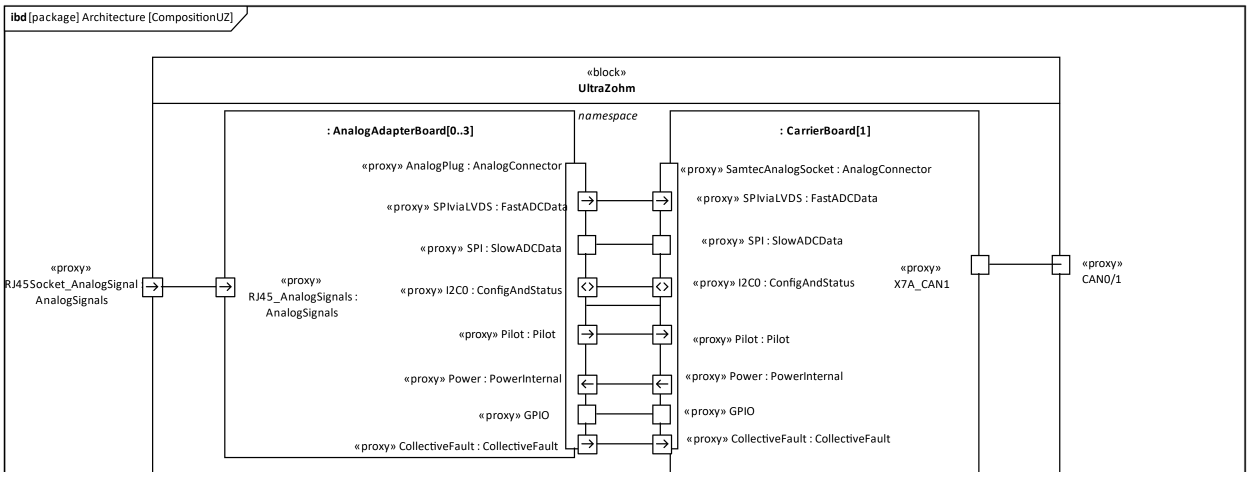

ibd [package] Architecture [CompositionUZ]
«block»
UltraZohm
: AnalogAdapterBoard[0..3]
namespace
: CarrierBoard[1]
«proxy» AnalogPlug : AnalogConnector
«proxy» SPIviaLVDS : FastADCData
«proxy» SPI : SlowADCData
«proxy» I2C0 : ConfigAndStatus
«proxy» Pilot : Pilot
«proxy» Power : PowerInternal
«proxy» GPIO
«proxy» CollectiveFault : CollectiveFault
«proxy» RJ45Socket_AnalogSignal AnalogSignals
«proxy» RJ45_AnalogSignals : AnalogSignals
«proxy» SamtecAnalogSocket : AnalogConnector
«proxy» SPIviaLVDS : FastADCData
«proxy» SPI : SlowADCData
«proxy» I2C0 : ConfigAndStatus
«proxy» Pilot : Pilot
«proxy» Power : PowerInternal
«proxy» GPIO
«proxy» CollectiveFault : CollectiveFault
«proxy» X7A_CAN1
«proxy» CAN0/1

Figure 2.4: Detailed SysML representation of the Analog Adapter Board and the Carrier Board. Within this project only the interface "SPIviaLVDS" that carries the fast ADC data is used.

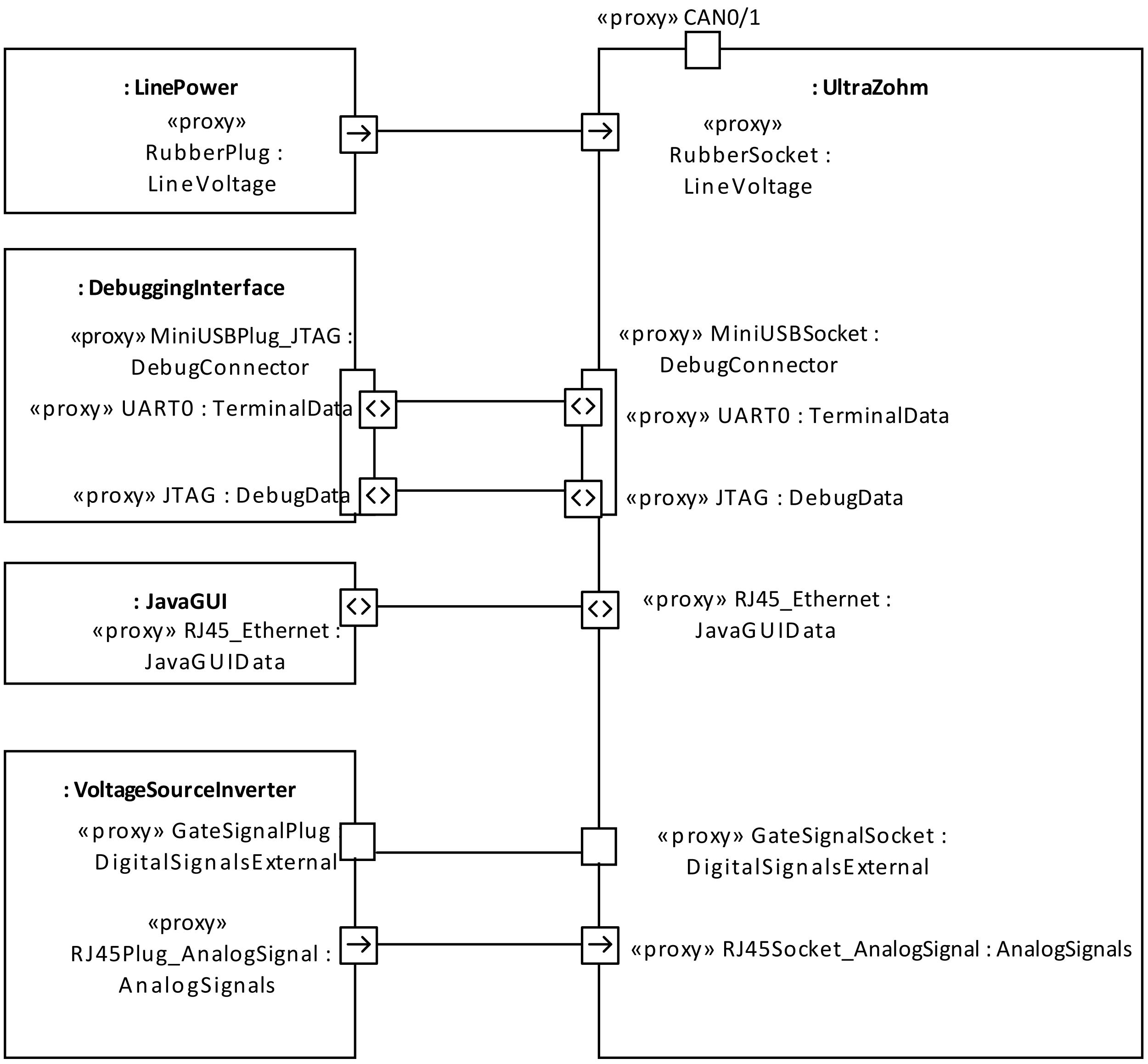

«proxy» CAN0/1
: LinePower
«proxy» RubberPlug : LineVoltage
: UltraZohm
«proxy» RubberSocket : LineVoltage
: DebuggingInterface
«proxy» MiniUSBPlug_JTAG : DebugConnector
«proxy» UART0 : TerminalData
«proxy» JTAG : DebugData
«proxy» MiniUSBSocket : DebugConnector
«proxy» UART0 : TerminalData
«proxy» JTAG : DebugData
: JavaGUI
«proxy» RJ45_Ethernet : JavaGUIData
«proxy» RJ45_Ethernet : JavaGUIData
: VoltageSourceInverter
«proxy» GateSignalPlug DigitalSignalsExternal
«proxy» RJ45Plug_AnalogSignal : AnalogSignals
«proxy» GateSignalSocket : DigitalSignalsExternal
«proxy» RJ45Socket_AnalogSignal : AnalogSignals

Figure 2.5: SysML representation of the UltraZohm in the context of it's surrounding systems. The interface of interest is the connection "AnalogSignals" from the Voltage Source Inverter to the UltraZohm.

2.2.2 Software System Context

While section 2.2.1 contains an analysis of the connections on the outside of the UltraZohm the following section describes the context of the IP cores inside the SoC FPGA. Fig. 2.6 displays the next level of detail starting at the representation of Fig. 2.4. The value generated by the ADC is transferred via SPI on LVDS to the FPGA where it is sampled by the ADC IP core. The IP core forwards this value to it's interface "RAW_VALUE" but also performs a conversion to an SI scaled unit e.g. Volt or Ampere. This value is available at the output "SI_VALUE". The IP core also features other interfaces e.g. a hardware trigger or status signals but these signals are omitted in the representation in Fig. 2.6.
Besides the FPGA the SoC also contains a dual core ARM Cortex R5 processor which is referred to as the Real-Time Processing Unit (RPU). In the current setup, hardware instantiated in the FPGA works as a hardware accelerator for the software running on the RPU. In order to transfer the data obtained from the ADC quickly to the software the tightly coupled memory (TCM) is used. This is basically a RAM with low access times for the RPU. The data from the ADC IP core is transferred to the AXI2TCM IP core which then writes the data to a predefined address in the TCM.

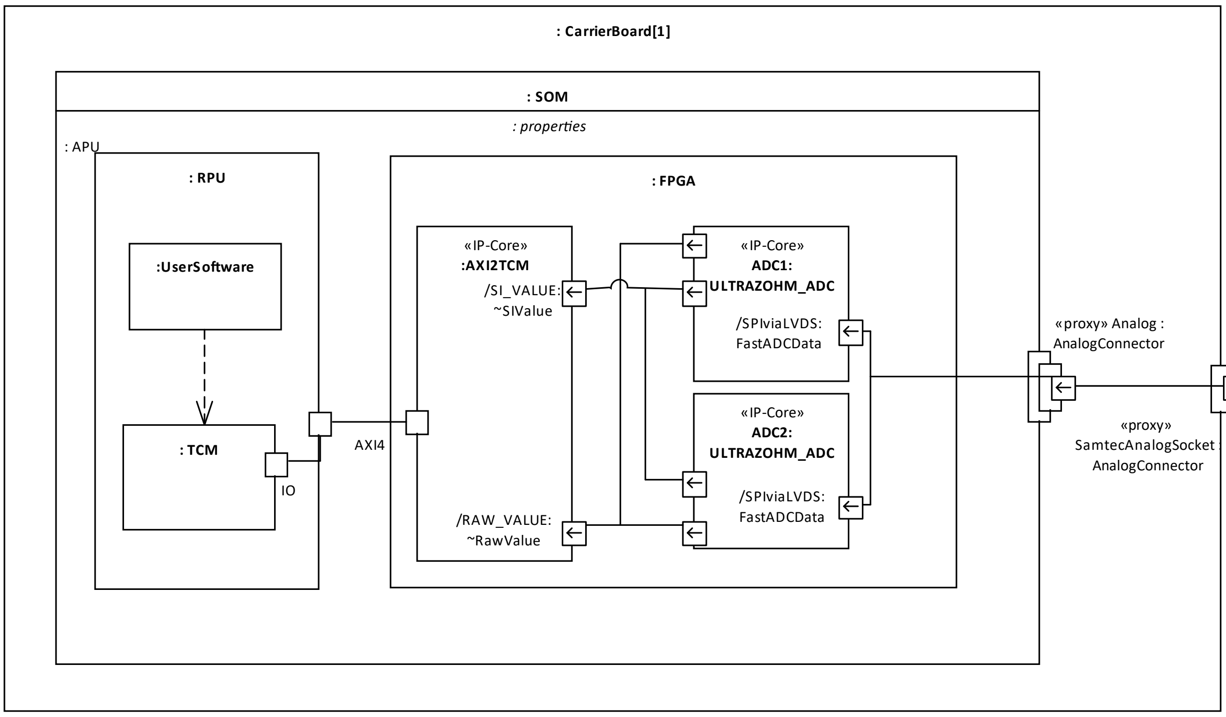


Figure 2.6: SysML representation of the UltraZohm in the context of it's surrounding systems. The interface of interest is the connection „AnalogSignals" from the Voltage Source Inverter to the UltraZohm.

2.3 Requirements

The requirements analysis is split up in a functional and a non-functional analysis. Every requirement is identified by a unique ID which carries the prefix `FUNC` for functional requirement and `NOF` for non-functional requirements. The functional analysis is carried out in SysML use case diagrams. Other components associated with the requirement are displayed in the use case diagrams as well. The association is illustrated with a general dependency in SysML.

2.3.1 Functional Requirements

Fig. 2.7 shows the main functional requirements of the IP core. In brief, the IP core shall control the physical interface of the ADC and process the raw value obtained from the ADC. These actions shall be controllable and configurable by the software application which is running on a processor in the SoC. `FUNC1` and `FUNC3` are further subdivided. The corresponding use case diagrams are displayed in Fig. 2.8 for `FUNC1` and Fig. 2.9 for `FUNC3`.

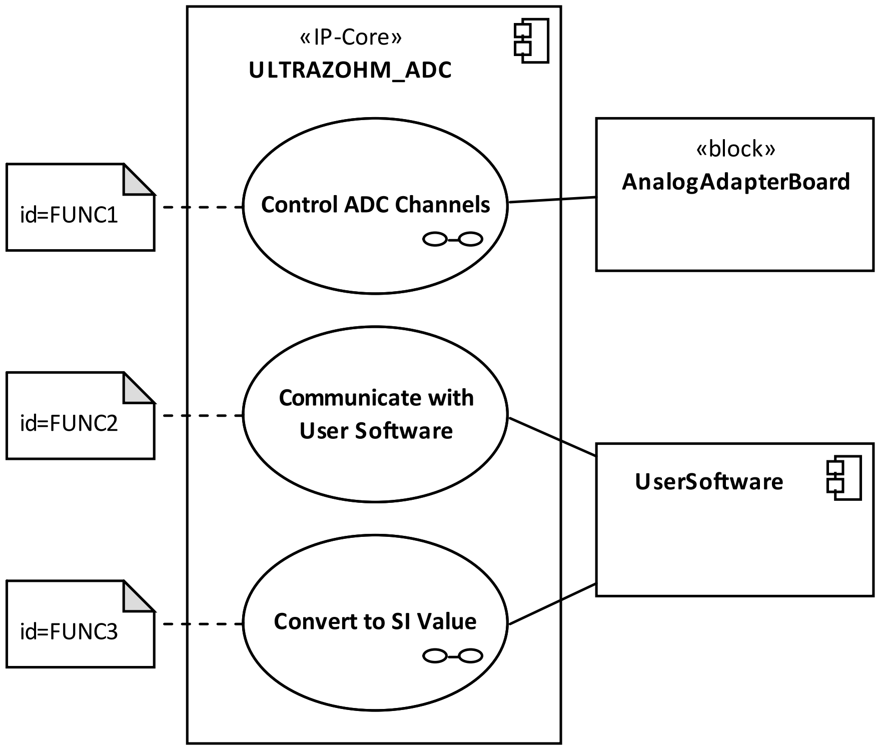


Figure 2.7: Top level use case diagram of the ADC IP core

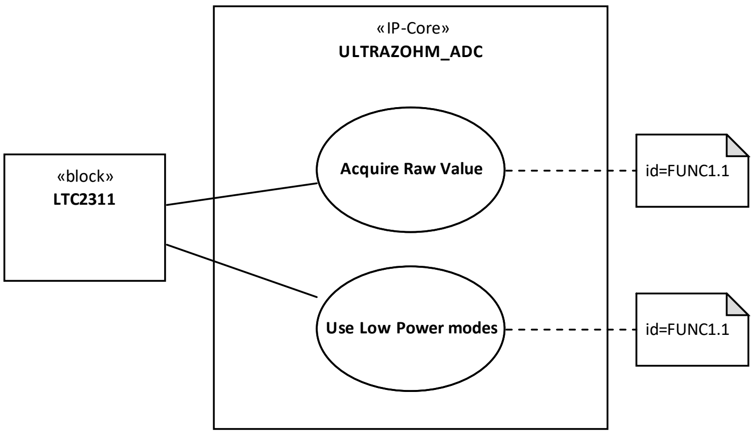


Figure 2.8: Sub use cases of the requirement "ControlAdcChannels" (a.k.a. `FUNC1`)

The functional requirement "ControlAdcChannels" (a.k.a. `FUNC1`) is further analyzed in Fig. 2.8. Indeed, the main requirement is the acquisition of values from the ADC but is still desirable to be able to use the low power modes of the chip. This feature will probably never be used in laboratory environments but it is handy to have it implemented when applying the IP core to a production application.

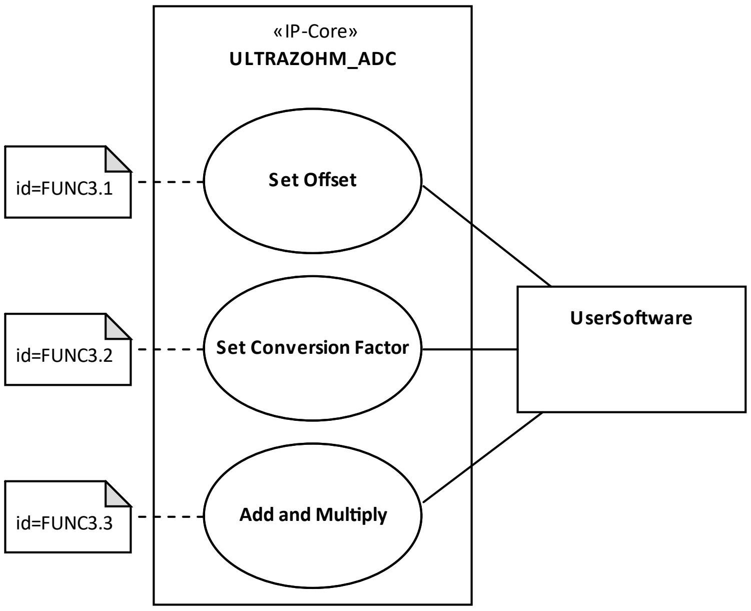


Figure 2.9: Sub use cases of the requirement "ConvertToSiValue" (a.k.a. `FUNC3`)

Furthermore, the hardware capabilities of the FPGA shall be used to process the raw value after acquisition. It shall be possible to add an offset to the raw value and to multiply this sum with a conversion factor. With a proper selection of the offset and the conversion factor the raw value can be converted to an SI-scaled value e.g. Volt or Ampere. The offset and the conversion factor shall be adjustable by the software application. These requirements are illustrated in Fig. 2.9

2.3.2 Non-Functional Requirements

Besides the functional requirements that describe what the ADC IP core is supposed to do, a number of non-functional requirements that describe the way the functions are implemented must be specified. An important distinction is made between features that shall be adjustable during runtime and those that are only changeable before the synthesis of the hardware description. This synthesis is comparable to the compilation of high-level software code e.g. a program written in C to the hardware specific machine language of the target platform.

NOF1 The bit width of the output value of the ADC shall be adjustable before synthesis.

NOF2 The ADC shall be operated over SPI where the ADC IP core acts as an SPI master. This is required by the ADC since it features an SPI.

NOF3 It shall be possible to select a slice of the result vector of the conversion. The width of the slice is determined before synthesis by selecting the MSB and the LSB from the vector.

NOF4 The IP core shall feature an Advanced eXtensible Interface (AXI) 4 Lite as a communication interface to the software application.

NOF4.1 The following parameters shall be configurable by software through AXI 4 Lite:

- CPOL and CPHA of the SPI
- Timing parameters of the SPI
- Selection which ADCs shall be triggered when receiving a trigger signal
- Value that is added to the raw value of the ADC (a.k.a offset)
- Value with which the sum of the offset and the raw value is multiplied (a.k.a conversion factor)

NOF4.2 The following control events can be triggered by software via AXI 4 Lite:

- Adjustment of the trigger mode
- Trigger by software
- Reset by software

NOF5 The offset and the conversion factor shall be unique to every single ADC that is connected to the SoC.

NOF6 The IP core shall provide a continuous trigger mode, where all ADCs are triggered as frequent as possible and a triggered mode, where a conversion is either triggered by software or by a hardware port.

NOF7 The raw value and the converted value shall be output as a separate vectors at the hardware interface of the ADC. The results from all connected ADCs are concatenated to a single output vector.

NOF8 Signed integer in two's complement is the format for the following values:

- The raw value of the ADC (given by the LTC2311)
- The offset and the conversion factor
- The output of the conversion unit

No floating point conversion of the converted value shall be performed in the IP core.

NOF9 The IP core shall be organized in synchronous groups. In one synchronous group a number of ADCs is controlled with a single SPI clock signal and a single SPI slave select signal. Fig. 2.10 illustrates the requirement.

NOF9.1 The number of groups and the number of ADCs per group is adjustable before synthesis.

NOF9.2 Each group can be triggered independently

NOF10 The IP core shall include an optional LVDS buffer. The inclusion of the LVDS buffer is adjustable before synthesis.

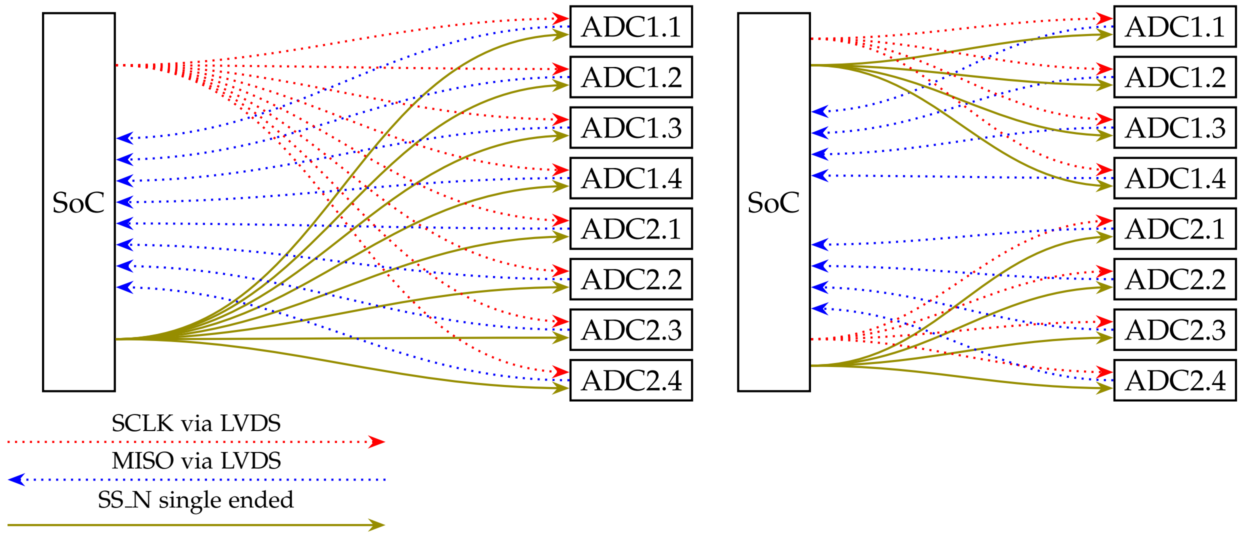


Figure 2.10: Connection between the SoC and the ADCs. The representation on the left side shows the default case, where all ADCs are controlled synchronously by a single SS_N and SCLK signal. The figure on the right side shows the configuration, where the ADCs are subdivided in two groups and each group is controlled independently.

2.4 Architecture

Based on the requirements analysis from section 2.3 an architecture for the implementation of the IP core has been created. An abstract representation of this architecture is given in the SysML component diagram in Fig. 2.11.

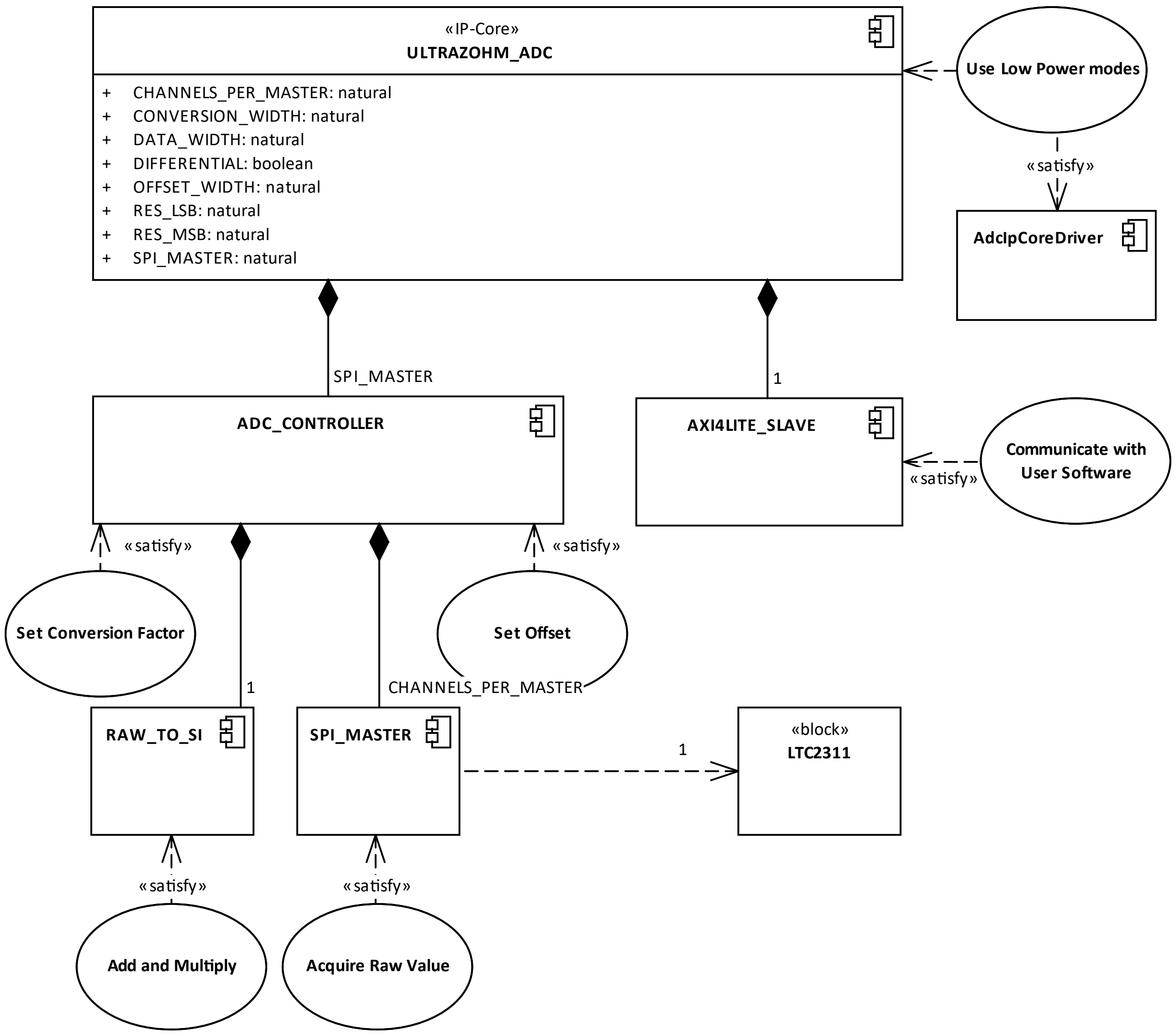


Figure 2.11: Component diagram of the ADC IP core. The functional requirements from section 2.3.1 are shown in this representation in order to illustrate the relations between the components and the requirements. This way, every component is functionally justified and the responsibilities are clearly disposed.

All components from Fig. 2.11 represent a single VHDL file that contains the hardware description of the module. Subcomponents are instantiated in the parent component. The number of subcomponents that are instantiated in a parent component is given by the parameter of the composition relationship. The top level component `ULTRAZOHM_ADC` also provides the interface

that is visible to the user of the IP core. This interface is composed of the hardware ports as well as the design parameters with which the IP core can be customized to the application. Furthermore, the top level component contains the instantiation of the differential input and output buffers which is not shown in the representation of Fig. 2.11. In combination with design parameter `DIFFERENTIAL` the top level components fulfills the requirement `NOF10`
The design parameters `SPI_MASTER` and `CHANNELS_PER_MASTER` determine the number of groups and the number of synchronously controlled ADCs per group. In Fig. 2.11 a group is represented by an instance of the component `ADC_CONTROLLER` that contains a number of SPI masters which is adjusted with the parameter `CHANNELS_PER_MASTER`. In conclusion, `NOF9` and `NOF9.1` are fulfilled by the modular architecture of the IP core. Besides `FUNC1` the component `SPI_MASTER` also satisfies the requirements `NOF2` and `NOF1`. The fulfillment of `NOF1` is not visible in the representation above but it is explained in section 3.1.
`NOF4` is satisfied by the component `AXI4LITE_SLAVE` that provides the communication interface to the software application. As shown in section 3.1, `NOF4.1` and `NOF4.2` are implemented by the modules `AXI4LITE_SLAVE` and `ULTRAZOHM_ADC`.
The components `ULTRAZOHM_ADC ADC_CONTROLLER` and `SPI_MASTER` are implemented as finite state machines. The state machines are implemented as a variant of Mealy machine with an output register as described in section 2.1. In the following sections, all subcomponents of the ADC IP core are described in detail.

3 Results

The following chapter contains the description of the components presented in section 2.4. For this purpose, the FSMs, which define the functional behavior of the components are presented but the implementation in VHDL is omitted, since it follows the methodology presented in section 2.1. The source code is available at the UltraZohm project repository which can be accessed without restrictions. Furthermore, the verification flow and results are described in section 3.2.

3.1 Implementation

Within the following sections, the components shown in Fig. 2.11 are explained in detail. While the design of the FSMs that describe the functionality of the modules is explained and justified, the implementation in HDL is omitted in this chapter. It follows the principle introduced in section 2.1.3. The development follows a bottom up approach. The design of the SPI master is explained first. Subsequently, the integration in the components `ADC_CONTROLLER` and `ULTRAZOHM_ADC` is explained. Finally, the design of the user interface formed by the AXI and the hardware interface is presented.

3.1.1 SPI Master

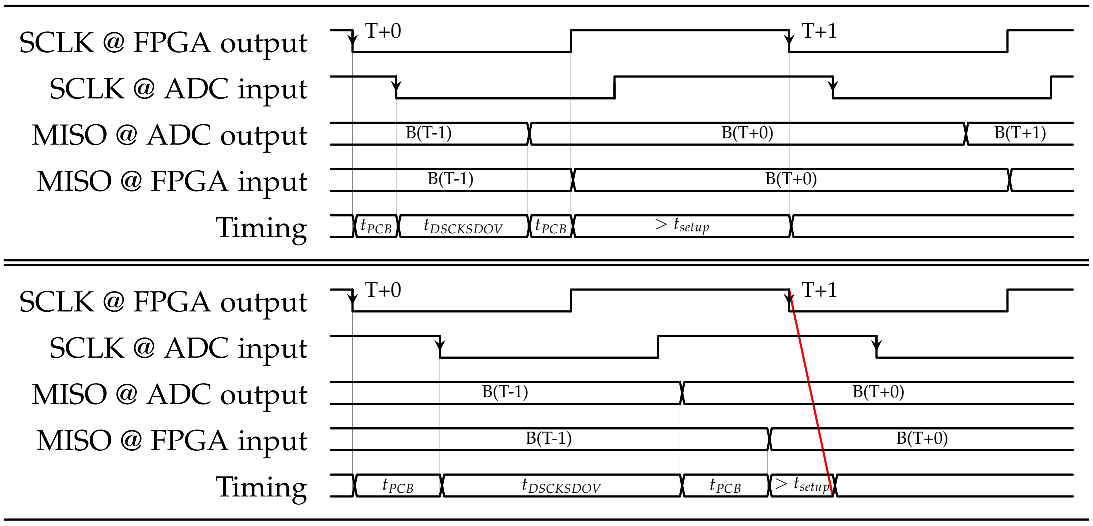
In the following section, an analysis of the hardware situation is given. This includes hardware interface of the LTC2311, the PCB layout between the SoC and the LTC2311 as well as the IO interface of the SoC.

Analysis of the hardware interface

Besides a standard CMOS interface, the LTC2311 offers the possibility to transfer data via low voltage differential signals (LVDS) [6, p. 23], which has been used in the implementation of the UltraZohm hardware [7]. LVDS offers a more robust communication for high-speed signals than a single ended CMOS connection [8, p. 868]. Moreover, the LVDS standard can be used effortless because the SoC FPGA offers integrated support [9, p. 13].
As a first step, the serial interface of the ADC must be analyzed. The interface is designed similarly to a standard SPI, however it features significant differences which must taken into account when designing the counterpart in the FPGA. Within the scope of this project, the signal names of the interface have been adapted to the standard SPI naming scheme. Table 3.1 shows the mapping of the signal names to the data sheet of the LTC2311.

Table 3.1: Signal definitions for the scope of this project.

Data Sheet	**Renamed**	**Description**
$\overline{CNV}$	SS_N	**S**lave **S**elect low active
SCK	$SCLK$	**S**erial **Cl**oc**k**
SDO	$MISO$	**M**aster **I**n **S**lave **O**ut



Timing diagram 3.1: Timing of the signals between the SoC and the ADC with $f_{SCLK} \approx 50\,\text{MHz}$ in the upper diagram and $f_{SCLK} \approx 100\,\text{MHz}$ in the lower diagram. The periods mentioned in the `Timing` row are explained in Table 3.2. The trigger instant for the ADC to output a new bit is marked with an arrow. The red line in the lower diagram shows the timing violation that may occur for the furthest ADC.

Timing diagram 3.1 illustrates the timing of the SPI between the FPGA and the LTC2311. The *SS_N* signal initiates a new conversion of an analog value and enables the serial interface of the LTC2311. *SCLK* is the clock signal for the conversion unit of the ADC but it also controls the read out process of the buffer which contains the result of the *previous* sample. On each falling edge a new bit of the *previously* converted sample is output as a serial bit stream on *MISO*. This behavior is further discussed in section 3.1.3.
In the following, an analysis of the behavior of the serial interface is carried out, which forms the basis for the FSM presented in Fig. 3.1. Timing diagram 3.1 shows the situation between the FPGA and the ADC. The diagram illustrates that the FPGA must sample a new bit on the falling edge of *SCLK*. Since a falling edge also triggers the ADC to output a new bit after a delay of $t_{DSCKSDOV}$, the bit sampled by the ADC corresponds to the *previous* falling edge. Considering the first falling edge of *SCLK* as instant *T+0* the bit *B(T-1)* is sampled on this falling edge.
Especially important is the considerations of the delay times invoked by the transmission line between the FPGA and the ADC which is referred to as t_{PCB} in the diagram. t_{PCB} mainly determines the maximum frequency for *SCLK*, which can be derived from timing diagram 3.1. The timing condition for a proper transfer is expressed in (3.1).

$$f_{SCLK} < \frac{1}{2 \cdot t_{PCB} + t_{DSCKSDOV} + t_{setup}} \tag{3.1}$$

While $t_{DSCKSDOV}$ and t_{setup} are given by the hardware, t_{PCB} depends on the physical distance of the ADC from the SoC. Considering t_{PCB} from Table 3.2 the highest frequency is given in (3.2) for the furthest ADC.

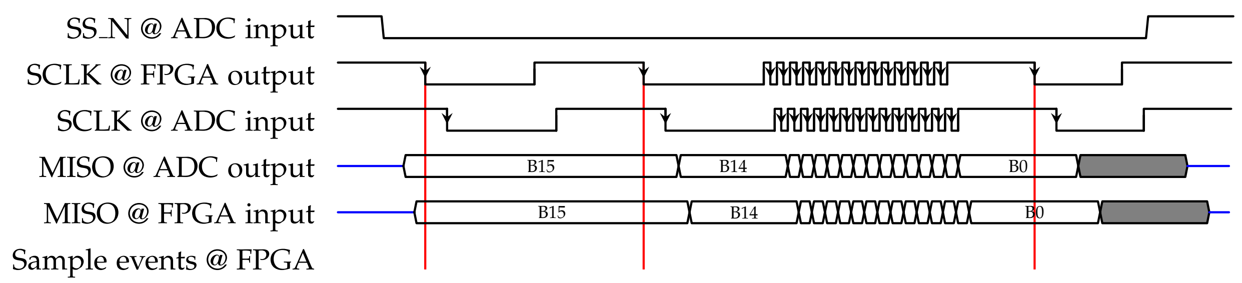
$$f_{SCLK} < \frac{1}{2 \cdot 0.9\,\text{ns} + 7.4\,\text{ns} + 1\,\text{ns}} \cong 98\,\text{MHz} \tag{3.2}$$

After the data sheet the ADC can be driven with up to 105 MHz but due to the hardware setup this is not possible for all ADCs in the given configuration. Indeed, the period considered for $t_{DSCKSDOV}$ is the worst case value and the data sheet also specifies a lower typical delay time of 4 ns. With the typical delay time, a proper transfer is possible but when driving the ADC with up to 100 MHz the above mentioned circumstances must be considered, since the timing depends on the individual chip.

Table 3.2: Relevant delay times of the SPI to the ADC adapter board [6].

Abbreviations	**Value**	**Description**
t_{PCB1}	0.9 ns	Propagation delay to the first analog adapter card slot
$t_{DSCKSDOV}$	7.4 ns	Data valid delay from falling edge of SCLK

The aforementioned characteristics of the hardware interface still match the behavior of a standard SPI. However, the LTC2311 features differences on the beginning of the transfer. Timing diagram 3.2 shows a complete transfer of a sample from the ADC to the FPGA as it is implemented in the IP core. The diagram illustrates the issue, that the MSB is sampled twice, while the LSB is truncated with a standard SPI. The problem can not be solved by adjusting the `CPHA` parameter, which determines the edge of *SCLK* on which the first sample is taken by the SPI master [10], since the sampling must take place on the first edge as illustrated by timing diagram 3.1. Consequently, this issue has been resolved in the current implementation by sampling 17 bits per transfer and truncating the MSB.



Timing diagram 3.2: Timing of a complete transfer from the ADC to the FPGA. The bits on *MOSI* are sampled by the IP core on the instants marked with red vertical lines. For a 16 bit sample 17 sample events take place where the MSB is sampled twice, which is corrected in the final result.

Implementation of the SPI master

Based on the hardware analysis, the SPI master module of the IP core can be designed. In the following sentences an explanation of the implementation, which is composed of the entity declaration from listing 3.1 the signal declaration from listing 3.2 and the FSM from Fig. 3.1, is given. The entity definition is the interface for the upper component in the hierarchy, which is the ADC Controller. The implementation in HDL is explained in section 2.1.3.
The reset state of the FSM is `IDLE`. In this state, a manual control of the signals SS_N and $SCLK$ is possible which is required to enter the sleep and nap modes of the LTC2311. The trigger event $ENABLE =' 1'$ immediately starts the transfer by pulling SS_N to 0. The LTC2311 requires a certain period between the falling edge of SS_N and the first falling edge of $SCLK$, which is referred to as the `PRE_DELAY` that can be configured with the corresponding parameter via AXI. It is implemented with the `PRE_WAIT` state, in which the FSM stays for `PRE_DELAY` + 1 system clock cycles. The actual transfer happens in the sates `SHIFT_OUT` and `SAMPLE`. A duplex communication is not implemented but a new bit for a communication to an SPI slave would be shifted on the serial bus on the transition from `SHIFT_OUT` to `SAMPLE`. The $SCLK$ edge, on which the bit on the serial bus is sampled, can be adjusted with the parameter `CPHA`. If this parameter is 0 the sampling takes place on the first edge which is required for the LTC2311. The sampling of the bit on the bus takes place on the transition from `SAMPLE` to `SHIFT_OUT`. The $SCLK$ frequency can be adjusted with the `CLK_DIV` parameter which is adjustable by software as well. The clock scaling is implemented by the counter S_DEL_CLK running in the states `SHIFT_OUT` and `SAMPLE` which is reset on the transition between these states. The actual clock frequency of $SCLK$ is given by (3.3). Therefore the maximum $SCLK$ frequency is $f_{SystemClock}/2$.

$$f_{SCLK} = \frac{f_{SystemClock}}{2 \cdot (CLK\_DIV + 1)} \tag{3.3}$$

After the transmission completes, the state `POST_WAIT` is entered which satisfies the requirement for a delay between the last $SCLK$ edge and the rising edge of SS_N [6]. Similar to the `PRE_WAIT` state, the time is adjusted by the corresponding parameter and is given with `POST_DELAY` + 1 system clock cycles. The discard of the double-sampled MSB is implemented by setting the S_BIT_COUNT, which controls the number of bits to transfer, to `DATA_WIDTH` + 1 while maintaining the width of the input register S_RX_BUFFER. This way the first bit is discarded with the sampling of the LSB. This issue is illustrated in timing diagram 3.2. The total latency from the trigger to a valid result at the output vector is given by (3.4). The equation outputs the latency in system clock cycles.

$$n_{lat,SPI} = 4 + PRE\_DELAY + 32 \cdot (CLK\_DIV + 1) + POST\_DELAY \tag{3.4}$$

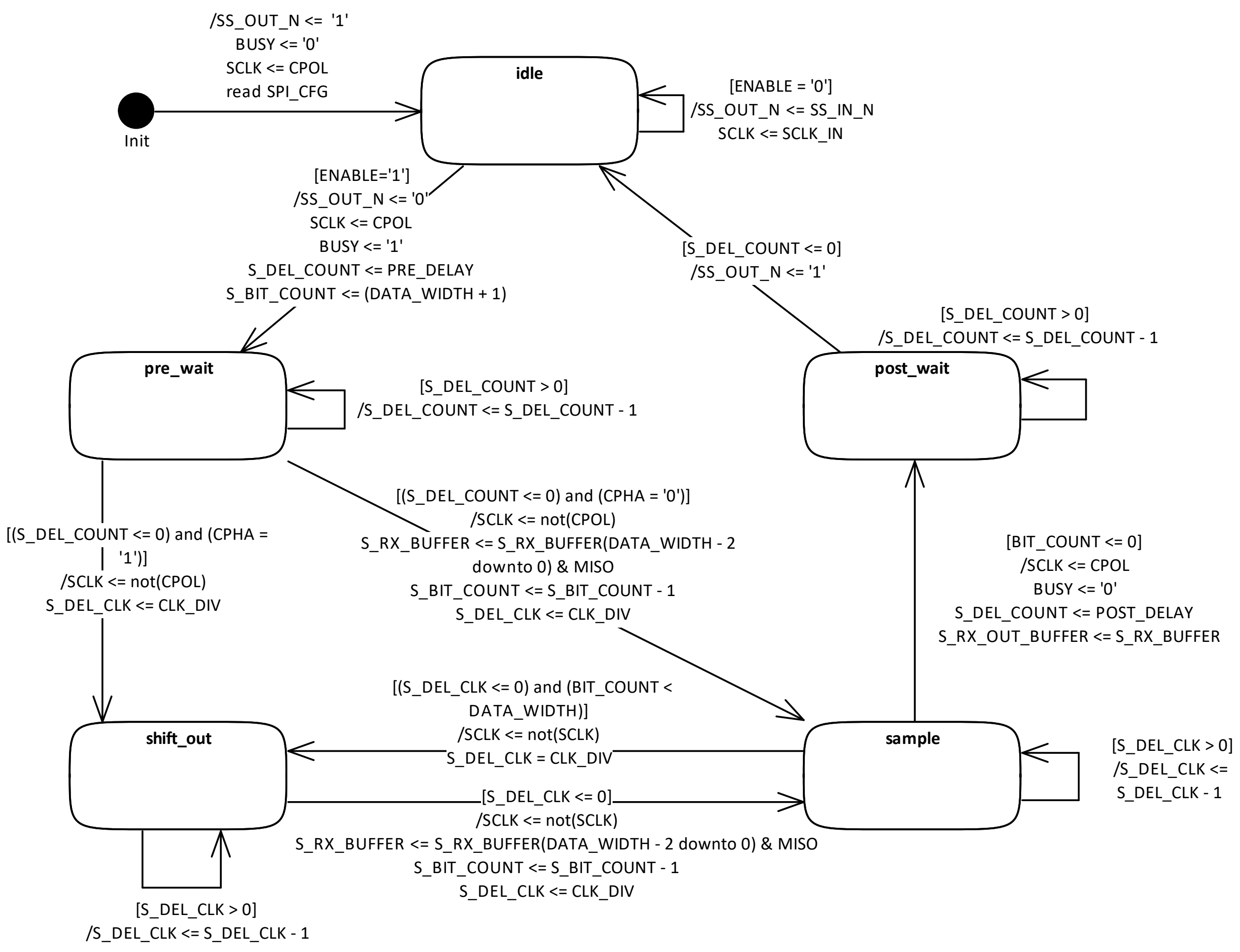


Figure 3.1: FSM of the SPI master. The conditions for the transition are written in square brackets while the output signal is described after the backslash on the transition arrow.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library work;
use work.ADC_LTC2311_PKG.all;

entity SPI_MASTER is
  generic(
    DATA_WIDTH       : natural := 16; -- Number of bits per SPI frame
    CHANNELS         : natural := 1   -- Number of slaves that are controlled with
     ↪   the same SS_N and SCLK

    );
  port (
    CLK          : in std_logic;
    RESET_N      : in std_logic;
    -- SPI ports
    RX_DATA      : out std_logic_vector((CHANNELS * DATA_WIDTH) - 1 downto 0);
    CPHA         : in std_logic;
    CPOL         : in std_logic;
    SCLK         : out std_logic;
    SCLK_IN      : in std_logic;
    MISO         : in std_logic_vector(CHANNELS - 1 downto 0);
    SS_OUT_N     : out std_logic;
    SS_IN_N      : in std_logic;
    MANUAL       : in std_logic;
    -- Control Ports
    BUSY         : out std_logic;
    ENABLE       : in std_logic;
    PRE_DELAY    : in std_logic_vector(C_DELAY_WIDTH - 1 downto 0);
    POST_DELAY   : in std_logic_vector(C_DELAY_WIDTH - 1 downto 0);
    CLK_DIV      : in std_logic_vector(C_CLK_DIV_WIDTH - 1 downto 0)
    );
end SPI_MASTER;
```

Listing 3.1: Entity declaration of the SPI master. Ports do not carry a prefix.

```
signal S_PRE_DELAY     : std_logic_vector(C_DELAY_WIDTH - 1 downto 0);
signal S_POST_DELAY    : std_logic_vector(C_DELAY_WIDTH - 1 downto 0);
signal S_CLK_DIV       : std_logic_vector(C_CLK_DIV_WIDTH - 1 downto 0);
signal S_DEL_COUNT     : integer range -1 to 255; -- 8 bit integer
signal S_DEL_CLK       : integer range -1 to 65535; -- 16 bit integer
signal S_BIT_COUNT     : integer range -1 to DATA_WIDTH + 1;
-- The bits from the SPI slave are clocked into the S_RX_BUFFER
signal S_RX_BUFFER : std_logic_vector((CHANNELS * DATA_WIDTH) - 1 downto 0);
-- S_RX_BUFFER <= S_RX_OUT_BUFFER after transmission ended
-- RX_DATA <= S_RX_OUT_BUFFER -> data avail. until next transmisson complete
signal S_RX_OUT_BUFFER : std_logic_vector((CHANNELS * DATA_WIDTH) - 1 downto
 ↪  0);
signal S_SCLK          : std_logic;
signal S_CPOL          : std_logic;
signal S_CPHA          : std_logic;

-- State definition for the FSM
type state_type is (IDLE,PRE_WAIT,SHIFT_OUT,SAMPLE,POST_WAIT);
signal curstate, nxtstate : state_type := IDLE;
```

Listing 3.2: Signal declaration of the SPI master. Signals carry the prefix S_ in order to distinguish them from ports.

3.1.2 Conversion Unit

The addition of an offset to the raw value gathered from the ADC and the multiplication with a conversion factor is performed by a single DSP48E2 slice. Fig. 3.2 shows the relevant parts of a DSP48 slice for the application. The component incorporates a input registers for all operands. The result of the pre-adder is the sum of the input values A and D and it is multiplied with the value B by the multiplier. The DSP48 slice implements a pipeline structure. The application presented in this report can make use of a pipeline structure since multiple values are acquired from the ADC concurrently and they must be further processed quickly while maintaining a low resource footprint. After an initial latency of three clock cycles for the first value, a valid result is output every clock cycle. This behavior is illustrated by the implementation in VHDL in listing 3.4 and the corresponding timing diagram 3.3. The implementation is based on the code example given in [3, p. 98]. Timing diagram 3.3 and listing 3.4 illustrate that the timing of the conversion is deterministic and it can be calculated before synthesis. This has been utilized in the component `ADC_CONTROLLER`.

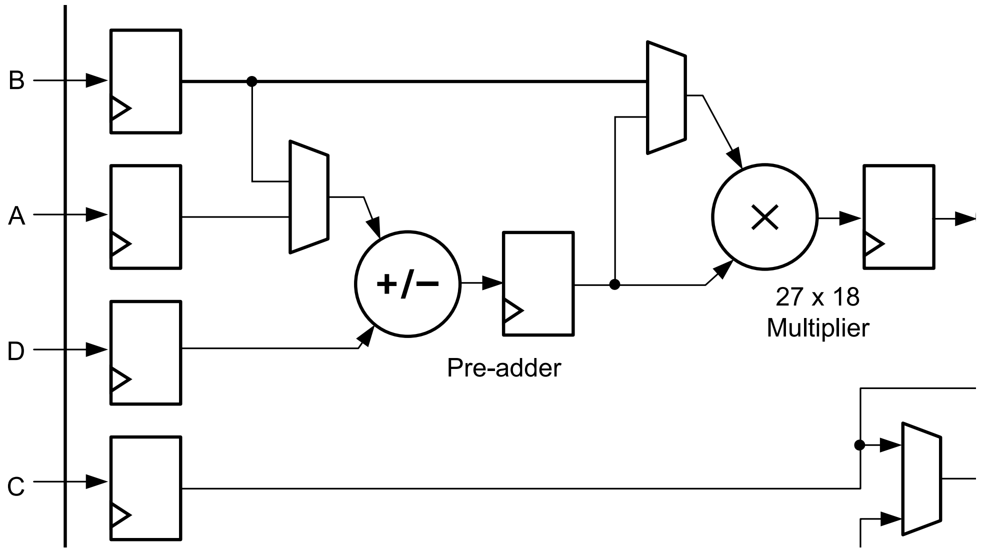


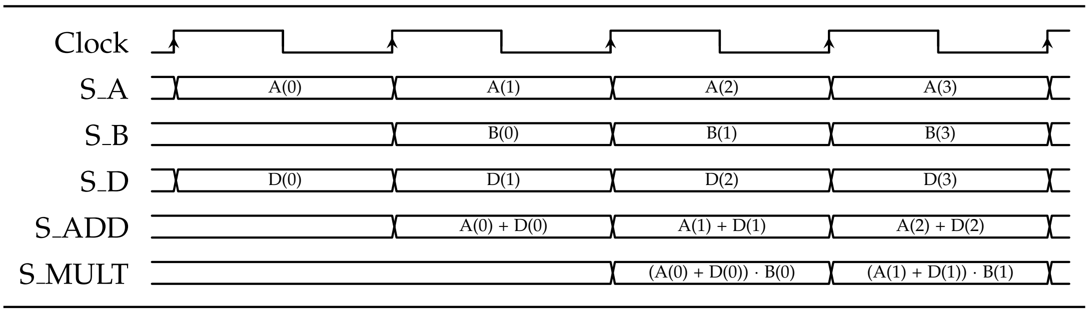
Figure 3.2: Overview of a DSP48 slice from [11, p. 7]. The figure only shows the relevant parts of a DSP48 slice for the application. Input C is not used in the application.

```vhdl
signal S_A          : signed(AWIDTH - 1 downto 0);
signal S_B          : signed(BWIDTH - 1 downto 0);
signal S_D          : signed(DWIDTH - 1 downto 0);
signal S_ADD        : signed(AWIDTH downto 0);
signal S_MULT       : signed(AWIDTH + BWIDTH downto 0);
-- make sure a DSP slice is used instead of slice logic
attribute use_dsp of Behavioral : architecture is "yes";
```

Listing 3.3: Signal declaration of the conversion unit.

```
S_A <= signed(AIN);
S_B <= signed(BIN);
S_D <= signed(DIN);
if SUBADD = '1' then
  S_ADD <= resize(S_A, AWIDTH + 1) - resize(S_D, AWIDTH + 1);
else
  S_ADD <= resize(S_A, AWIDTH + 1) + resize(S_D, AWIDTH + 1);
end if;
S_MULT <= S_ADD * S_B;
```

Listing 3.4: Implementation of a DSP48 slice in VHDL. *S_A*, *S_B* and *S_D* hold the content of the input registers. *S_ADD* holds the result off the addition and *S_MULT* contains the final result of the conversion. The corresponding timing can be found in timing diagram 3.3.



Timing diagram 3.3: Illustration of the signals in a DSP48 slice. Due to the pipeline structure, multiple sets of values can be processed concurrently. After the initial latency of three clock cycles, a value is output on every rising edge until all values are processed.

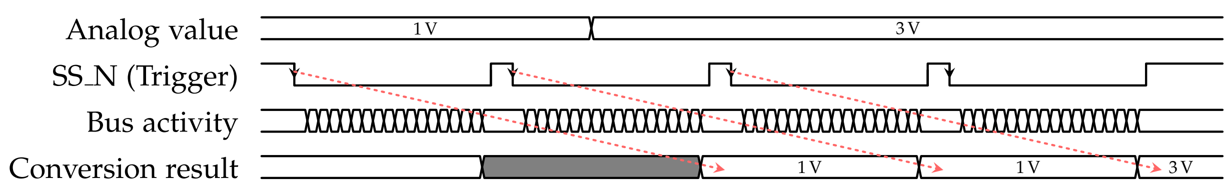
3.1.3 ADC Controller

The component ADC_CONTROLLER integrates the SPI master and the conversion unit. The raw value delivered by the SPI master is piped through the conversion unit both results are presented at the interface of the component. In order to satisfy NOF3, the ADC controller provides the generics RES_MSB and RES_LSB. With these generics, a slice from the output vector of the multiplier can be selected. The generics are available at the interface of the top level component, which is described in appendix A.3. Furthermore, the output format of the ADC controller satisfies NOF7, since the results are concatenated to a vector for the raw and another vector for the post-processed values. Since the SPI master only serves as a media access control, the ADC controller needs to consider the functional characteristics of the LTC2311. It features the property that the chip outputs the result of the *previously* converted sample. This issue is illustrated by timing diagram 3.4. Usually, the user expects the IP core to return the sampled analog value in the moment of the trigger event. In fact, with the behavior of the LTC2311, the user obtains the value from the previous trigger event which can be at anytime in the past. Therefore, a dummy sample must be taken before reading out the value of interest. This has been considered in the implementation of the ADC controller. Fig. 3.3 shows the functional description of the ADC controller which has been designed as an FSM.
In the following, the FSM of the ADC controller is explained. The IDLE state (reset state) is similar to IDLE in the SPI Master. A manual control of the signals *SCLK* and *SS_N* is possible, if the appropriate port on the components interface is set to '1'. The signals are directly connected to the SPI master. A conversion is initiated by setting the *ENABLE* port to '1', during which no manual control of the SPI master is possible. Therefore, the signal *S_MANUAL* is pulled to '0' during the active operation of the ADC controller. As explained above, the first sample returned by the ADC needs to be discarded. This behavior is induced by setting the signal *S_DUMMY_SAMPLE* to '1' for the first sample which lets the FSM directly return into the OCCUPIED state after the first sample. After this first dummy sample, the FSM transitions between the states OCCUPIED, SPI_TRANSFER and CONVERTING until the desired number of samples, which is determined by the signal *S_SAMPLES*, has been taken from the ADC. In case of the SPI master, the transition from SPI_TRANSFER to CONVERTING is triggered by the falling edge of the *BUSY* signal of the SPI master which indicates the end of the conversion. For the conversion unit the number of clock cycles required for the addition and multiplication is determined during synthesis as explained in section 3.1.2. Therefore, the ADC controller can stay in the CONVERTING state for a predefined number of clock cycles. This number is simply determined by adding the initial latency of three clock cycles of the conversion unit to the number of ADCs (i.e. values to be processed) that are connected to the IP core. The total latency of transfer and conversion is given in system clock cycles by (3.5) where $n_{lat,SPI}$ is given by (3.4).

$$n_{lat,result} = n_{lat,SPI} + CHANNLES\_PER\_MASTER + 6 \tag{3.5}$$

After the processing of all values, the FSM returns into the IDLE state, if the *ENABLE* signal is '0' after the last sample. Otherwise, a new series of samples is taken immediately without taking a dummy sample. This behavior allows the operation in a continuous mode. The implementation of the FSM follows the principle explained in section 2.1.3.
Besides the functional implementation of the FSM, the ADC controller also holds the offset and the multiplication factor for the conversion of the raw value as well as the number of samples taken

per trigger. The VHDL process to update these values is independent from the implementation of the FSM. Therefore, the parameters can be updated at any time also during an ongoing series of samples. For the offset and the multiplication factor, the ADC controller offers a distinct memory location for each ADC channel which satisfies NOF5.
The components `SPI_MASTER` and `RAW_TO_SI` are integrated in the ADC controller as a standard VHDL component instances. One single SPI master and conversion unit are instantiated per ADC controller unit.



Timing diagram 3.4: Conversion behavior of the LTC2311. The red arrows show the mapping between the trigger events and the corresponding results delivered by the LTC2311. The diagram shows the power up behavior. After the first transfer, an unknown value is returned by the ADC because the output register of the ADC contains a random power up value.

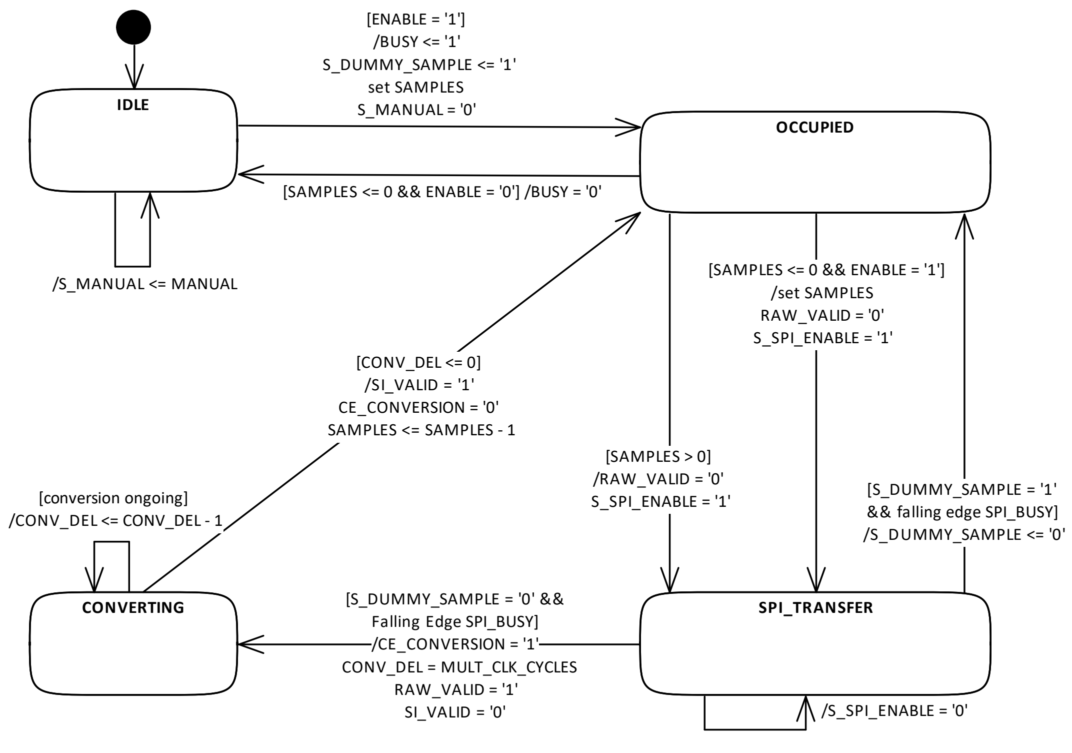


Figure 3.3: FSM of the ADC controller

3.1.4 Advanced eXtensible Interface 4 Lite

The AXI4 Lite provides the connection between the software driver that may run on a processor of the SoC and the hardware IP core. The IP core acts as an AXI slave. Within the scope of this report, the implementation details of an AXI slave are not further discussed since the Xilinx provides an AXI module as a starting point for custom IP core development. For further details about the AXI see [12]. For the current project, the AXI implementation that is available when following the "Create and package IP" wizard has been adapted for the application. See [13, p. 40-42] for details about the wizard. The tool outputs a top level module with the ports for the AXI as well as the implementation of the AXI, which is instantiated in the top level module. The architecture is illustrated by Fig. 2.11. Indeed, the tool offers various customization options for the newly created AXI peripheral but the AXI implementation is still generic and needs to be adapted for the application. In the following paragraphs, only the adaptions applied to the template provided by Xilinx are presented.
The IP core requires ten 32 bit AXI registers. In order to utilize the whole address space, the number of slave registers needs to match a power of two because the encoding of the address is binary. Therefore, an AXI peripheral with 16 registers has been created. Unused registers may be used for further enhancements of the IP core. Listing 3.5 displays the registers that are used for the application. In contrast to the implementation given by Xilinx, meaningful names have been given to the signals. All registers used by the application are connected to the corresponding port on the entity of the AXI component. This way, the content of the registers is available in the top level component.

```
signal ADC_CR                :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_SPI_CR            :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_SPI_CFGR          :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_MASTER_CHANNEL    :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_CHANNEL           :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_MASTER_FINISH     :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_MASTER_SI_FINISH  :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_MASTER_BUSY       :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_CONV_VALUE        :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal ADC_AVAILABLE         :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg10             :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg11             :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg12             :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg13             :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg14             :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg15             :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
```

Listing 3.5: Signal declaration of the AXI registers. `C_S_AXI_DATA_WIDTH` is a generic which determines the width of a single AXI register. It has a value of 32. The registers 10 to 15 are not used by the application. The registers `ADC_MASTER_FINISH`, `ADC_MASTER_SI_FINISH` and `ADC_MASTER_BUSY` are read only.

Further adjustments have been done in the process that performs the write action from the software to the AXI register. All read only register from listing 3.5 have been removed from this process, because the software shall not be able to write to these memory locations. Moreover, an acknowledgment mechanism has been added to the process. Specific actions e.g. the software trigger are acknowledged by resetting the corresponding bit by hardware. For this purpose, ports carrying the write request for the specific AXI register have been added to the entity of the AXI component. The acknowledgment logic is displayed in listing 3.6.

```
if (P_ADC_CR_IN(C_TRIGGER) = '0') and ADC_CR(C_TRIGGER) = '1' then
  ADC_CR(C_TRIGGER) <= '0';
end if;

if (P_ADC_CR_IN(C_CONV_VALUE_VALID) = '0') and ADC_CR(C_CONV_VALUE_VALID)
 ↪   = '1' then
  ADC_CR(C_CONV_VALUE_VALID) <= '0';
end if;
```

Listing 3.6: Acknowledgment mechanism in the AXI component. Ports are prefixed with `P_` and constants with `C_`. The constants determining the position of the bit in the register are defined in a package globally for the IP core.

Lastly, a process has been added to the AXI implementation that fills in the information in the read only registers. Ports on the entity are available that take the information given by the top level component. Listing 3.7 shows the implementation of this feature.

```
process (S_AXI_ACLK) is
begin
  if (rising_edge(S_AXI_ACLK)) then
    if S_AXI_ARESETN = '0' then
      ADC_MASTER_FINISH    <= (others => '0');
      ADC_MASTER_SI_FINISH <= (others => '0');
      ADC_MASTER_BUSY      <= (others => '0');
    else
      ADC_MASTER_FINISH    <= P_ADC_MASTER_FINISH;
      ADC_MASTER_SI_FINISH <= P_ADC_MASTER_SI_FINISH;
      ADC_MASTER_BUSY      <= P_ADC_MASTER_BUSY;
    end if;
  end if;
end process;
```

Listing 3.7: Operation of the read only AXI registers. Ports are prefixed with `P_` in order to distinguish them from signals.

3.1.5 Top Level Module

The top level module of the IP core integrates the ADC controller and the AXI4 Lite. Besides that, the module implements the continuous and the triggered operation mode as well as a manual mode in order to control sleep and nap modes of the LTC2311. Moreover, it controls the adjustment of the offset and the multiplication factor as well as the number of samples taken per trigger event. Similar to the ADC controller, the trigger modes and the manual mode are implemented in an FSM, while the adjustment of the operation parameters is performed independently in a separate process.
Fig. 3.4 shows the FSM implemented in the top level module. The transitions are mainly controlled by the content of the AXI registers. Appendix A.1 contains a detailed description of the AXI registers. The FSM considers only the control register (*S_ADC_CR*) and the SPI control register (*S_ADC_SPI_CR*). Furthermore, an ongoing conversion is taken into consideration by reading the signal *NOT_BUSY*. Fig. 3.5 displays the activity diagram that is executed on every transition to the triggered state including self transitions. It shows that the IP core features a hardware trigger for real time requirements and a software trigger via AXI4 Lite for uncritical purposes. The trigger event is executed by pulling the *ENABLE* port of the ADC controller to '1'. In continuous mode, the port is pulled to '1' as long as the FSM is in the state `CONTINUOUS`. Finally, a manual mode can be enabled in order to use nap and sleep modes of the LTC2311. This mode can only be enabled if no conversion is ongoing and if the IP core is in triggered mode. In manual mode, the corresponding bits for *SS_N* and *SCLK* from the `ADC_SPI_CR` register are simply forwarded to the ADC controller component. The implementation of the different trigger modes fulfills NOF6. Furthermore, NOF9.2 is satisfied by the implementation of the hardware trigger as vector, where every bit triggers an ADC controller which represents a group of ADCs. This functionality is also available for the software trigger mode, where the channels are selected by the `ADC_MASTER_CHANNEL` register.

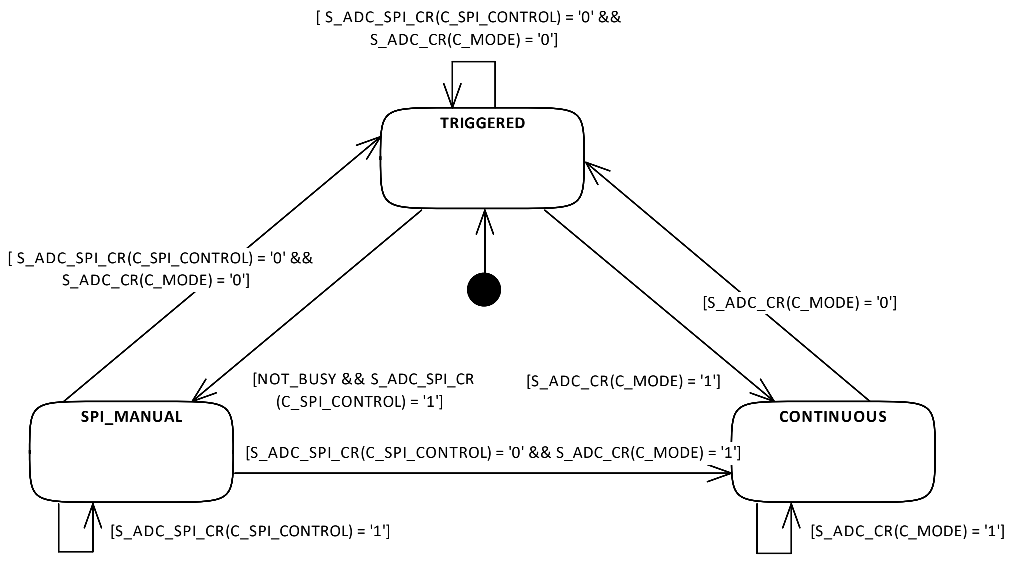

[S_ADC_SPI_CR(C_SPI_CONTROL) = '0' &&
S_ADC_CR(C_MODE) = '0']
TRIGGERED
[S_ADC_SPI_CR(C_SPI_CONTROL) = '0' &&
S_ADC_CR(C_MODE) = '0']
[S_ADC_CR(C_MODE) = '0']
[NOT_BUSY && S_ADC_SPI_CR
(C_SPI_CONTROL) = '1']
[S_ADC_CR(C_MODE) = '1']
SPI_MANUAL
CONTINUOUS
[S_ADC_SPI_CR(C_SPI_CONTROL) = '0' && S_ADC_CR(C_MODE) = '1']
[S_ADC_SPI_CR(C_SPI_CONTROL) = '1']
[S_ADC_CR(C_MODE) = '1']

Figure 3.4: FSM of the top level module.

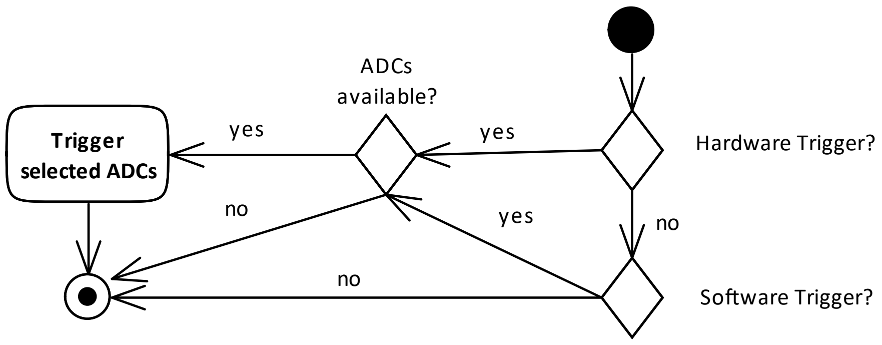


Figure 3.5: Activity diagram, that is executed on every transition to the `TRIGGERD` state including self transitions. ADCs can be unavailable if they are put into sleep or nap mode, which is indicated by resetting the corresponding bit in the `ADC_AVAILABLE` register.

The initial total latency from the application of a trigger signal to the output of the first valid result on the interface of the IP core is given by (3.6). For every additional sample, another latency of $n_{lat,result}$ clock cycles is added to the initial delay.

$$n_{lat,total} = n_{lat,result} + n_{lat,SPI} + 3 \tag{3.6}$$

Besides the implementation of the operating modes, the top level module also controls the adjustment of the offset, the multiplication factor and the number of samples taken per trigger event. Listing 3.8 shows a part of the port mapping of the ADC controller component in the top level module. The ADC controller features ports for setting the operation parameters and a port that carries the value from the AXI register described in appendix A.1.9. The process presented in listing 3.9 sets the control ports on the ADC controller based on the user request. The selection, which channels shall be updated is performed via the `ADC_MASTER_CHANNEL` register as described in appendix A.1.4. The encoding from Table A.2 as well es the update mechanism can be found in the lines 307 to 327 of listing 3.9.

```
-- Control Ports
SET_CONVERSION  => S_SET_CONVERSION(i),
SET_OFFSET      => S_SET_OFFSET(i),
SET_SAMPLES     => S_SET_SAMPLES(i),
SI_VALID        => S_ADC_MASTER_SI_FINISH(i),
RAW_VALID       => S_ADC_MASTER_FINISH(i),
BUSY            => S_ADC_MASTER_BUSY(i),
-- Value Ports
VALUE   => S_ADC_CONV_VALUE,  -- input for conversion or offset value
```

Listing 3.8: Extraction from the port map of the instantiation of the ADC controller in the top level module. The ports *SET_OFFSET SET_CONVERSION SET_SAMPLES* control the adjustment of the corresponding operation parameter while *VALUE* carries the value from the AXI register.

```vhdl
proc_set_conversion: process(S_CLK)
begin
  if rising_edge(S_CLK) then
    if (S_RESET_N = '0') then
      S_SET_CONVERSION <= (others => '0');
      S_SET_OFFSET <= (others => '0');
      S_ADC_CR_IN(C_CONV_VALUE_VALID) <= '1';
    elsif (S_ADC_CR(C_CONV_VALUE_VALID) = '1') then
      -- reset the update request
      S_ADC_CR_IN(C_CONV_VALUE_VALID) <= '0';
      -- update the values
      S_SET_OFFSET     <= (others => '0');
      S_SET_CONVERSION <= (others => '0');
      S_SET_SAMPLES    <= (others => '0');
      -- overwrite default if adjustment has been requested
      case S_ADC_CR(C_CONFIG_VALUE_MSB downto C_CONFIG_VALUE_LSB) is
        when "000" =>
          S_SET_OFFSET     <= S_ADC_MASTER_CHANNEL(SPI_MASTER - 1 downto 0);
        when "001" =>
          S_SET_CONVERSION <= S_ADC_MASTER_CHANNEL(SPI_MASTER - 1 downto 0);
        when "010" =>
          S_SET_SAMPLES    <= S_ADC_MASTER_CHANNEL(SPI_MASTER - 1 downto 0);
        when others =>
          S_SET_OFFSET     <= (others => '0');
          S_SET_CONVERSION <= (others => '0');
          S_SET_SAMPLES    <= (others => '0');
      end case;
    else
      S_ADC_CR_IN(C_CONV_VALUE_VALID)  <= '1';
      S_SET_CONVERSION                 <= (others => '0');
      S_SET_OFFSET                     <= (others => '0');
      S_SET_SAMPLES                    <= (others => '0');
    end if;
  end if;
end process proc_set_conversion;
```

Listing 3.9: Process to adjust the operation parameters of the IP core.

3.2 Verification

The following sections describe the verification workflow. Section 3.2.1 describes the approach of a static test bench which has been used in the project, while the setup and the results of the verification on hardware are presented in section 3.2.2.

3.2.1 Verification by Simulation

Since the synthesis and implementation of a VHDL description for the target platform requires significant computational time and resources it is important to perform at least a functional verification of the design by simulation. There exist several test bench approaches, where the concept of a static test bench written in VHDL is the simplest [4]. This approach has been used within the scope of this report. The test bench consists of VHDL description with an empty entity and an instantiation of the design under test (DUT). Furthermore, a stimulus is created with a sequential process. The output of the digital circuit can be inspected in a waveform window. This is an interactive and manual verification method, since the executive engineer has to examine the output of the circuit manually, which is suitable for simple designs. However, for a throughout verification, a checker function should be implemented which performs the verification automatically based on a high level description of the desired behavior. Within the scope of this report, such a checker function has not been implemented because the behavior of the design is difficult to describe in software and the level of complexity is still suitable for an interactive verification.
Even though the approach of a static test bench is simple and straightforward to use, the simulation of an AXI bus effortful since a functional model for an AXI master is required. The AXI master operates the AXI slave that is implemented in the design. The solution offered by Xilinx is called AXI verification IP (AXI VIP) which is framework for building object oriented test benches in SystemVerilog [14]. Since this solution requires SystemVerilog programming skills, which are not available at the executive engineer, it has not been used within the scope of this project. Therefore, only components without an AXI have been verified by simulation.
In the following paragraphs the verification of the component `ADC_CONTROLLER` is described. This component incorporates all other components without an AXI. Therefore, it illustrates the verification process for the whole design.
After the setup of the test bench with an instance of the DUT, a stimulus needs to be created. Since the IP core interfaces the LTC2311, which is a physical component outside the SoC which can not be included in the simulation, a functional model for it's interface has been created. The behavior of the serial interface is described in listing 3.10. The generation of the stimulus itself is straightforward. The ports of the instantiated ADC controller are connected to test signals which are driven by a VHDL process. After adjusting the operating parameters a trigger is initiated. These actions are presented in listing 3.11.

Fig. 3.6 shows the simulation result of the test bench. Another advantage of the verification by simulation is the availability of all signals in the design. In a synthesized design, it must be decided in advance which signals shall be visible after the synthesis and implementation. Moreover, when using state machines, the names of the states are visible in plaintext in the simulation whereas the probe in the synthesized design only shows the encoding as a binary number. With the presented approach, the functional verification of subcomponents can be done effortless and time efficient.

```
spi_slave : process (S_SCLK, S_SS_OUT_N, S_MISO)
            begin
    if falling_edge(S_SS_OUT_N) then
        S_MISO <= (others => S_TX_DATA(S_TX_BIT_COUNT)) after (DCNVSDOV + 2 *
         ↪  PCB_DEL);
    end if;

    -- generate new output value on falling edge
    if falling_edge(S_SCLK) then
        S_MISO <= (others => 'X') after (HSDO + 2 * PCB_DEL);
        if (S_TX_BIT_COUNT > 0) then
            S_TX_BIT_COUNT <= S_TX_BIT_COUNT - 1;
        end if;
    end if;
    if S_MISO(0) = 'X' then
        S_MISO <= (others => S_TX_DATA(S_TX_BIT_COUNT)) after (DSCKSDOV - HSDO);
    end if;

    -- reset bit counter for next transmission
    if rising_edge(S_SS_OUT_N) then
        S_TX_BIT_COUNT <= TEST_DATA_WIDTH;
    end if;
end process spi_slave;
```

Listing 3.10: Functional model of the LTC2311. The model behaves exactly like the real component. It obeys to the *SCLK* and *SS_N* signal which is generated by the IP core. Delay times have been modeled as well with `after` statements. See [6] for the specification of the considered delay times. The signal *S_TX_BIT_COUNT* is the index of the vector that is transferred to the SPI master.

```
223     -- set offset and conversion for channel 1
224     S_CHANNEL_SELECT <= (1 => '1', others => '0');
225     S_OFF_CONV <= std_logic_vector(to_signed(OFFSET_1, S_OFF_CONV'length));
226     S_SET_OFFSET <= '1';
227     wait for CLOCK_PERIOD;
228     S_SET_OFFSET <= '0';
229
230     S_OFF_CONV <= std_logic_vector(to_signed(CONVERSION_1, S_OFF_CONV'length));
231     S_SET_CONVERSION <= '1';
232     wait for CLOCK_PERIOD;
233     S_SET_CONVERSION <= '0';
234
235     -- set number of samples
236     S_OFF_CONV <= std_logic_vector(to_signed(SAMPLES, S_OFF_CONV'length));
237     S_SET_SAMPLES <= '1';
238     wait for CLOCK_PERIOD;
239     S_SET_SAMPLES <= '0';
240
241     wait for CLOCK_PERIOD;
242     -- start transfer
243     S_ENABLE <= '1';
244     wait for CLOCK_PERIOD;
245     S_ENABLE <= '0';
246     wait for 4 * (CLOCK_PERIOD * (TEST_CLK_DIV + 2) * TEST_DATA_WIDTH
247             + CLOCK_PERIOD * (TEST_DELAY + 1));
```

Listing 3.11: Setup and trigger of the ADC controller in the test bench environment. In lines 224 to 241, the operation parameters are set, where *OFFSET_1*, *CONVERSION_1* and *SAMPLES* are constants that are set in the signal definition of the test bench. The trigger event happens in the lines 243 to 245 and the transfer takes place in lines 246 and 247.

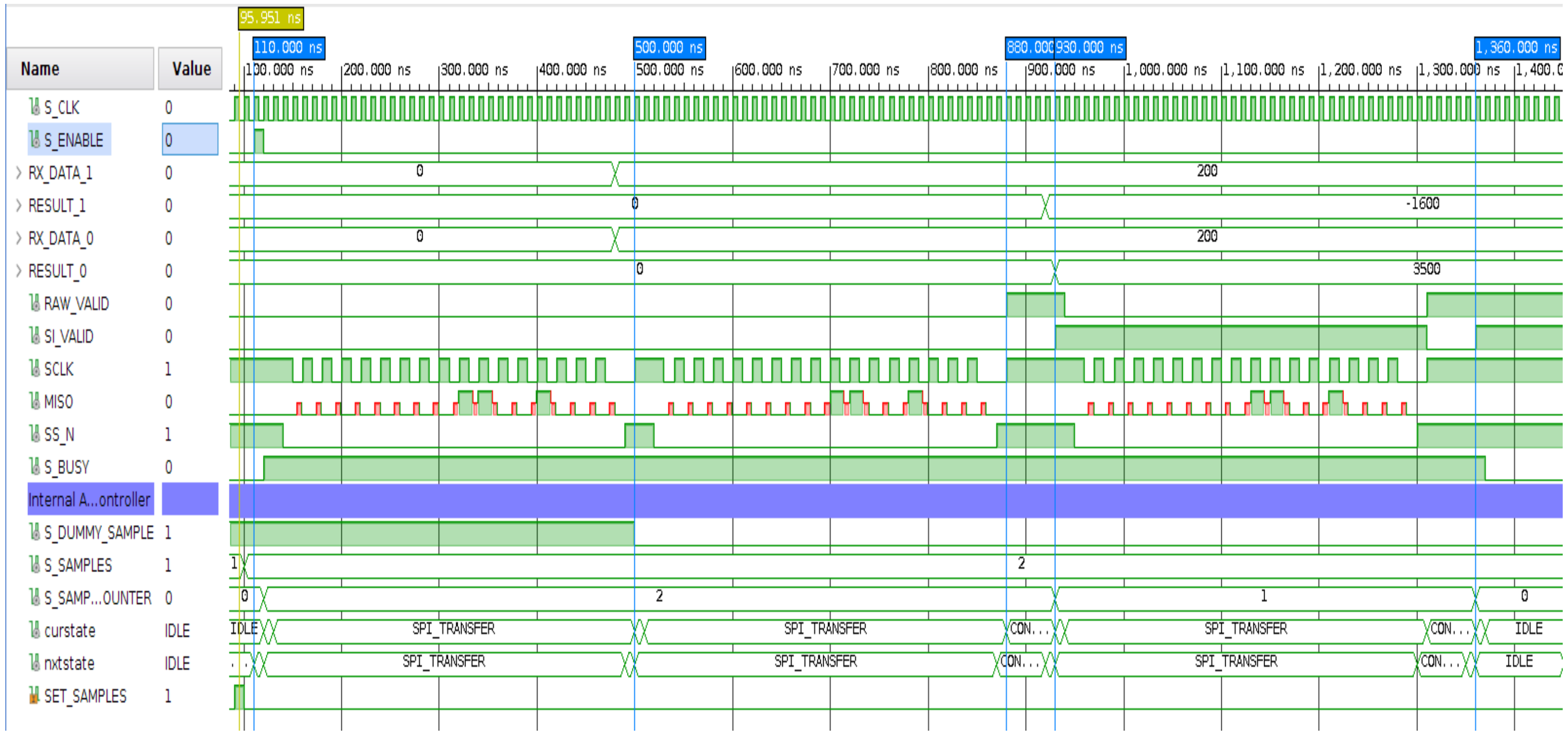


Figure 3.6: Simulation result of the test bench.

Marker	Event
110 ns	Start of transfer of the dummy sample
500 ns	End of transfer of the dummy sample
880 ns	First raw value is valid
930 ns	First converted value valid
1360 ns	Series of two samples finished

3.2.2 Verification on Hardware

After the verification by simulation, the VHDL design can be synthesized and implemented for the target platform. In the synthesis step, a netlist is being generated from the VHDL description. In a traditional electrical engineering this step is comparable to the creation of a schematic. The implementation step is comparable to a place and route procedure in PCB design. Indeed, both of these steps are fully automated by the synthesis tool but they can be manipulated by using synthesis attributes and various design constraints [3] [15].
The verification process on hardware can be split up in two sections. As a first step, the signals inside the IP core must be exposed to a probe in order to be able to observe them during operation. Then, a functional verification of the IP core can be performed. These steps are described in detail within the following paragraphs.

Setup of the IP core for Debugging

The first step in hardware debugging is the setup of a block design in Vivado. When using software running on a CPU a Zynq UltraScale+ MPSoC block is placed into the block design, which represents the SoC device including processor cores. Alternatively, the JTAG-to-AXI master core can be used to run transactions with TCL commands [16, p. 161]. Within the scope of this report, the IP core has been driven from a control application running on the ARM Cortex R5 processor. The AXI of the IP core is then connected to the appropriate interface on the aforementioned components.
In order to connect the hardware interface of the IP core to the appropriate pins on the package of the SoC, ports are created in the block design. In case of the ADC IP core the mapping of the physical pin on the SoC package to the pin on the package of the LTC2311 must be determined. The connection of the ports in the block design to the appropriate physical pin is controlled by the physical constraint `PACKAGE_PIN`. Besides the determination of the correct pin, the `IOSTANDARD` constraint must be set as well to instruct the synthesis tool to connect the correct IO buffer to the pin (e.g. an LVDS buffer for the MISO and SCLK lines). In case of differential inputs (e.g. the *MISO* lines) the internal termination resistor can be used by setting the constraint `DIFF_TERM_ADV`.
For the observation of the internal signals the internal logic analyzer (ILA) is used. The ILA is an IP core offered by Xilinx to which signals from a synthesized design can be connected. In order to connect the signals to the ILA core, the "Netlist Insertion Debug Probing Flow" is used [16, p. 128]. The synthesis tool offers the `MARK_DEBUG` synthesis attribute that must be assigned to signals being connected to the ILA core. Additionally the `KEEP` attribute can be assigned as well. This instructs the synthesis tool to keep the signal in the design as it has been declared instead of absorbing it which facilitates the debugging in the probe window and may help to find errors in the schematic generated by the synthesis tool. Listing 3.12 shows the assignment of the attributes required for debugging to a signal. Ideally, this assignment is performed on all signals that shall be connected to the ILA core before synthesis. Alternatively, the signals can be marked for debugging after synthesis and before implementation as well, but this induces significant manual workload.

```
attribute KEEP : string;
attribute MARK_DEBUG : string;
attribute KEEP of MY_SIGNAL : signal is "true";
attribute MARK_DEBUG of MY_SIGNAL : signal is "true";
```

Listing 3.12: Assignment of the attributes required for debugging to a signal.

The next step is the synthesis of a netlist from the VHDL description without performing the implementation. After the synthesis, the ILA core is injected in the netlist. This is achieved by opening the synthesized design and following the "Set Up Debug" wizard. All signals, to which the `MARK_DEBUG` attribute has been assigned, appear in the dialog and can be connected to the ILA core. When all aforementioned steps have been performed successfully, the implementation and the generation of the bitstream, which is loaded on the device, can be started.
While this procedure is suitable for debugging purposes, it has some drawbacks concerning production usage. The assignment of the `KEEP` and `MARK_DEBUG` attributes may prevent the synthesis tool to find the most efficient solution because it is forced to keep signals that me be obsolete in a more efficient implementation. Therefore, ILA cores should be removed from the design in production.

Functional verification on Hardware

After the IP core has been setup for debugging, measurements on test signals can be performed. Fig. 3.7 shows the setup with which the measurements on the IP core have been performed. In order to ensure reproducibility, the equipment as well as a technical verification of the setup is presented in appendix B.1.

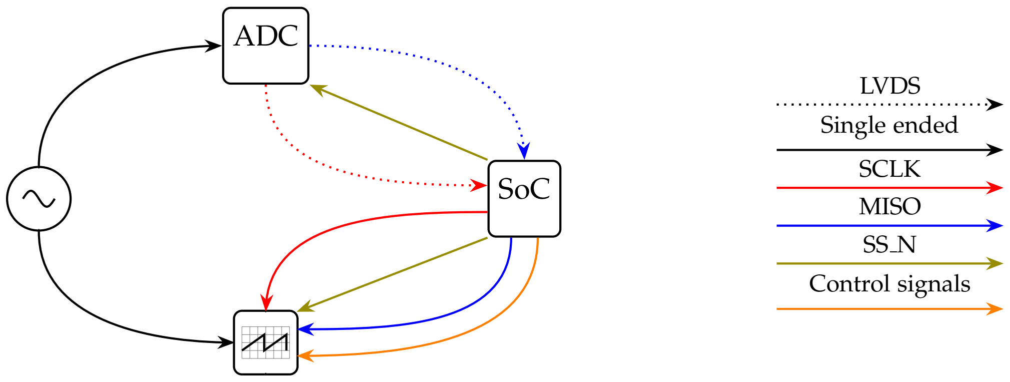
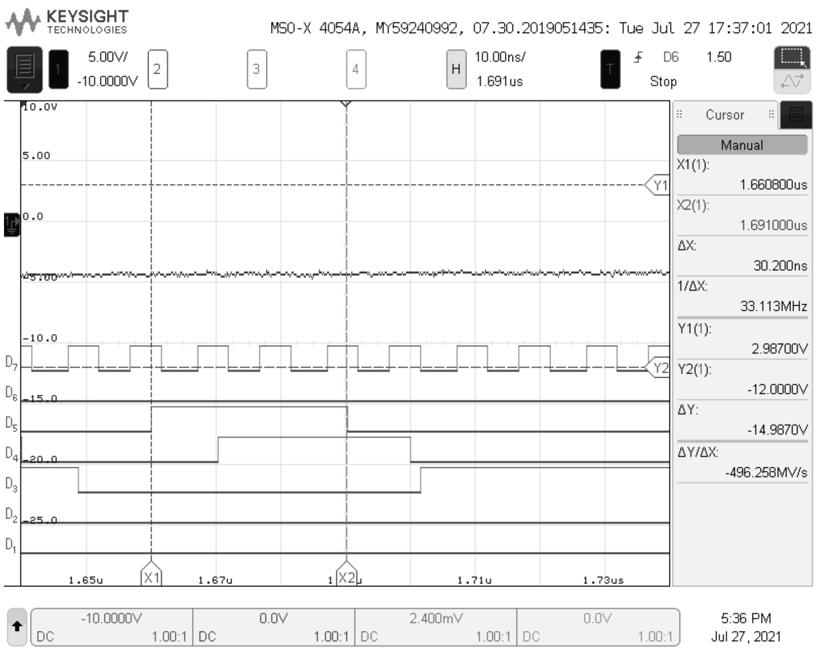


Figure 3.7: Block diagram of the measurement setup for the verification on hardware. The signals received from the ADC are piped through the FPGA and can be observed at the oscilloscope as well.

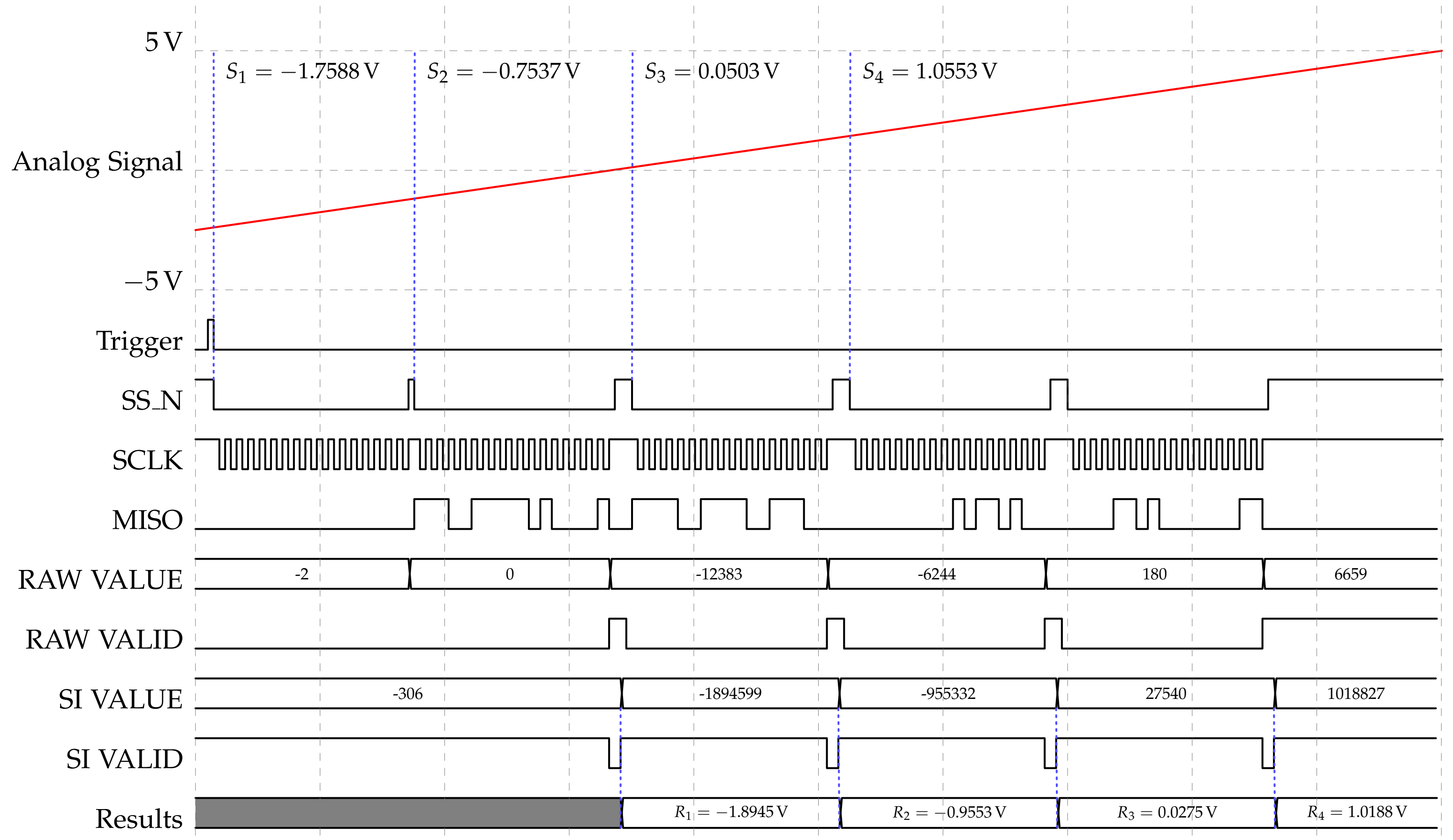
Within the scope of this project, the quality of the measurement result is not investigated. Information about that can be found in [6] and [7]. In fact, the timing of the measurements is the scope of the following investigations.

As a test signal, a 20 kHz triangle has been fed to the ADC. Timing diagram 3.5 reveals the results of the measurement of the test signal. In order to asses the quality of the result, the output of the ADC has been compared to the output of the oscilloscope. While the timing of the sampling events in relation to the output of the result correlates with the investigations from section 3.1.3, especially the second value given by the ADC features a significant difference to the result of the oscilloscope. This may be induced by inaccuracies of the oscilloscope, however the results R_3 and R_4 are closer to the samples S_3 and S_4. The difference between these samples is mainly the sample and hold time (SAHT) in which the sampling capacitor of the ADC is connected to the analog signal. This period is given by the length of the high state of the *SS_N* signal [6], which is shown in Fig. 3.8 for R_2. If this period is to short, the capacitor may not be charged accurately and the conversion result of the ADC does not match the actual value of the analog signal. The minimal SAHT also depends on the driving strength of the analog signal which can differ in varying applications. Furthermore, input filtering of the analog signal may require higher SAHTs as well.
Besides the inaccuracy of the second sample, the limitation of the maximal operating frequency of the IP core is depicted by the measurement. In order to drive the LTC2311 with the maximum *SCLK* frequency of about 100 MHz, a system clock frequency of 200 MHz must be applied to the IP core as explained in section 3.1.1. The high time of the *SS_N* signal for S_2 is currently fixed to three system clock cycles which results in a period of $3/200\,\text{MHz} = 15\,\text{ns}$. After the data sheet of the LTC2311, the minimal SAHT is 28.5 ns [6]. Therefore, the SAHT is out of specification for the second sample when driving the IP core with 200 MHz. As a consequence, the SAHT should be adjustable after a further revision of the IP core.



Cursor
Manual
X1(1): 1.660800us
X2(1): 1.691000us
ΔX: 30.200ns
1/ΔX: 33.113MHz
Y1(1): 2.98700V
Y2(1): -12.0000V
ΔY: -14.9870V
ΔY/ΔX: -496.258MV/s

Figure 3.8: Length of the SS_N high pulse S_2. The signal is on digital channel 5. The difference between the cursors X_1 and X_2 show the pulse width which is at 30 ns. This still meets the requirements of the LTC2311, however it may be to short for the application.



Timing diagram 3.5: Result of the measurement taken to investigate the timing accuracy of the IP core. The data from the diagram has been derived from a measurement with an oscilloscope combined with the output of the ILA. The analog slope, on which the measurement is performed, is represented by the red curve in the upper region of the diagram. The samples $S_{1..4}$ have been derived from the measurement of the oscilloscope whereas the results $R_{1..4}$ present the value output by the ADC. Timing scale: 2 µs per division.

4 Summary and Discussion

4.1 Method Discussion

In the following section the applied methods for design and verification are being discussed and possible enhancements are being depicted. The design methodology for the VHDL code by using FSMs has been proven to be a versatile and well structured approach. Besides the explicit support by the synthesis tool, the method produces deterministic results and is simple to reproduce. Since the design and implementation of an FSM in VHDL is a strictly standardized process, it would be possible to enhance the method with code generation. In this case, the engineer would design a graphical representation of the FSM and a program generates the VHDL description of the FSM. Concerning the architecture of the IP core, the definition of interfaces and the encapsulation of functionality as described in section 2.4 turned out to be a promising approach as well. VHDL directly supports the implementation of hierarchical designs with the instantiation and generation of subcomponents as performed in the ADC controller and the top level component. Nevertheless, an abstract design and the clear allocation of requirements to components as described in the sections 2.3 and 2.4 is a vital preparation step in advance to the definition of the interfaces and the implementation of the components.
Enhancements can be applied to the verification procedure. Within the scope of this project, only a functional verification after the implementation in VHDL has been performed in order to avoid time consuming implementation runs and to take advantage of the simulation features as described in section 3.2.1. For a throughout verification two more simulation runs are required:

1. Simulation of the netlist that is generated by the synthesis. It needs to be verified, that the synthesis tool does not change the behavior of the initial VHDL description.

2. Simulation of the implemented design. The synthesis tool outputs a netlist that includes propagation delays of combinational circuits as well. A static timing analysis can be applied on the implementation which detects possible violations of timing requirements. Additionally, functional errors that may be induced by the implementation run can be detected as well.

With this procedure, the behavior of the design on hardware is simulated as realistic as possible. It may even outperform the hardware debugging with an ILA core since it's injection induces major changes to the netlist and consequently to the implemented design. However, it is mandatory to create a fully automated test bench in order to perform the above mentioned simulation runs. An interactive verification as performed in section 3.2.1 is inefficient and error-prone [4]. A further enhancement is the application of an object oriented test bench. In contrast to a static solution, an object oriented approach separates the creation of the stimulus and the automated verification from the actual DUT. Therefore, major parts of the test bench can be reused, while a static solution needs to be tailored for every new design. The functional description of the DUT is usually given in a high level programming language like SystemVerilog or SystemC which offer a more comfortable modeling support than VHDL [4].

4.2 Summary

Referring to the project goals from section 1.3 the success of the project is estimated in the following sentences: The solution for actual value acquisition proposed in the current report offers the maximum achievable amount of flexibility concerning adaptions during runtime. Even more flexibility is offered by the various design parameters described in appendix A.3. However, another synthesis and implementation run is required after the adjustment of the design parameters. Nevertheless, the combination of a well defined AXI and the customization options offered by the design parameters leads to a flexible solution while maintaining a low resource footprint. Especially the pipeline implementation of the post processing described in section 3.1.2 leads to high throughput while the resource requirements concerning DSP slices are cut down by the factor of eight in comparison to non pipelined implementation and for the current external hardware. Furthermore, the FSM implementation of the majority of components enables a simple access for future developments by other engineers, because the process is straightforward and easy to reproduce. The strict hierarchic implementation also enables partial design reuse for future projects. Small shortcomings as described in section 3.2.2 can be fixed effortless in a minor revision. In conclusion, the technical outcome of the project is considered to be successful.
Considering the method discussion from section 4.1, the workflow presented in the report is not eligible for future developments in the UltraZohm project. The workflow as it has been performed in the current report has major drawbacks concerning the verification. This may be uncritical in laboratory environments when keeping the dangers concerning moving parts and electrical energy to a safe level. However, when porting the application to production in higher power classes, a consistent verification of all hardware components is unavoidable. Indeed, the setup of a throughout verification solution requires major human resources which can not be provided by the UltraZohm community because the project is targeting the development of control algorithms and not the verification of HDL components. Furthermore, the majority of UltraZohm users is not familiar with VHDL. Therefore, the presented workflow of a manual HDL coding and verification is considered to be ineligible for further developments in the UltraZohm project.

4.3 Outlook

The IP core presented in the present report forms a solid basis for a value acquisition system. Nevertheless, further enhancement and developments in the signal chain are required. This includes the possibility to adjust the sampling time of the ADC as depicted in section 3.2.2. With this feature, the user can react to different driving capabilities of the signal applied to the ADC. Furthermore, an operation at the maximum sampling frequency would be possible as well. The current solution has limitations in this concern as described in section 3.2.2.
With an adjustable number of samples taken on a single trigger event, the IP core is ready to feed other post processing in the signal chain. For example, the implementation of an oversampling unit that performs linear regression on a set of samples as presented in [17] may be a useful development for the project. In this case, another hardware interface that carries the number of the sample in the current set should be considered as a further enhancement of the ADC IP core itself.

Concerning the hardware verification, the usage of an object oriented test bench may be considered also for code generated components. In this case, the behavioral description of the generated code is available in an environment, the average user of the UltraZohm is familiar with. Therefore, no knowledge about hardware verification domain is required by the end user. It needs to be investigated, if it is possible to feed this description to a test bench, which automatically monitors the behavior of the implementation running on hardware against the high level description in software. This step is of major importance when bringing the application to production, since it needs to be verified that no behavioral changes have been induced during the various processing steps of the high level description. Since the generation of the stimulus, the verification against the high level description and the interface to the hardware component are decoupled in an object oriented test bench, this may be promising approach for hardware verification in the UltraZohm project [4].

List of Figures

List of Timing Diagrams

List of Tables

List of Listings

Bibliography

[1] S. Wendel, A. Geiger, E. Liegmann, *et al.*,
"UltraZohm — A Powerful Real-Time Computation Platform for MPC and Multi-Level Inverters,"
In *2019 IEEE International Symposium on Predictive Control of Electrical Drives and Power Electronics (PRECEDE)*,
May 2019,
Pp. 1–6.
DOI: `10.1109/PRECEDE.2019.8753306`.

[2] J. Bäsig,
"Automaten und ihre Anwendung,"
Lecture Notes,
Nuremberg, 2017.

[3] Xilinx,
"Vivado Design Suite User Guide: Synthesis,"
2020,
P. 295.
[Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf` (visited on 08/05/2021).

[4] J. Bäsig,
"Rechnergestützter Schaltungsentwurf,"
Lecture Notes,
Nuremberg, 2019.

[5] M. K. Patel. "Finite state machines — FPGA designs with VHDL documentation,"
FPGA designs with VHDL. (2017),
[Online]. Available: `https://vhdlguide.readthedocs.io/en/latest/vhdl/fsm.html` (visited on 08/08/2021).

[6] A. Devices,
"Datasheet LTC2311,"
Jul. 2016.
[Online]. Available: `https://www.analog.com/media/en/technical-documentation/data-sheets/231116fa.pdf` (visited on 04/19/2020).

[7] S. Lukas and E. Liegmann. "Analog Adapter Board V3 — Documentation,"
Documentation of the UltraZohm. (2020),
[Online]. Available: `https://docs.ultrazohm.com/hardware/adapter_cards/analog/LTC2311_16_v3.html` (visited on 08/09/2021).

[8] P. Horowitz and W. Hill,
The Art of Electronics,
Third edition.
New York, NY: Cambridge University Press, 2015,
1192 pp.,
ISBN: 978-0-521-80926-9.

[9] Xilinx,
"UltraScale Architecture SelectIO Resources User Guide,"
2019,
P. 360.
[Online]. Available: `https://www.xilinx.com/support/documentation/user_guides/ug571-ultrascale-selectio.pdf` (visited on 08/05/2021).

[10] Motorola,
"SPI block guide,"
Feb. 2004.
[Online]. Available: `https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/S12SPIV4.pdf` (visited on 04/21/2021).

[11] Xilinx,
"UltraScale Architecture DSP Slice User Guide,"
2020,
P. 76.
[Online]. Available: `https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf` (visited on 08/26/2021).

[12] Xilinx,
"AXI Reference Guide,"
2017,
P. 175.
[Online]. Available: `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf` (visited on 08/28/2021).

[13] Xilinx,
"Vivado Design Suite User Guide: Creating and Packaging Custom IP,"
Xilinx,
2020,
P. 113.
[Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1118-vivado-creating-packaging-custom-ip.pdf` (visited on 08/04/2021).

[14] Xilinx,
"AXI Verification IP Product Guide,"
2019,
P. 97.
[Online]. Available: `https://www.xilinx.com/support/documentation/ip_documentation/axi_vip/v1_1/pg267-axi-vip.pdf` (visited on 08/30/2021).

[15] Xilinx,
"Vivado Design Suite User Guide: Implementation,"
2021,
P. 208.
[Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug904-vivado-implementation.pdf` (visited on 08/31/2021).

[16] Xilinx,
"Vivado Design Suite User Guide: Programming and Debugging,"
2021,
P. 432.
[Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals_j/xilinx2021_1/ug908-vivado-programming-debugging.pdf (visited on 08/31/2021).

[17] P. Landsmann,
"Sensorless Control of Synchronous Machines by Linear Approximation of Oversampled Current,"
Dissertation, Technische Universität München, München, 2014.
[Online]. Available: https://nbn-resolving.org/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20141119-1220061-0-8.

A Description of the IP core

A.1 AXI Registers

A.1.1 ADC Control Register

Address offset: 0x0
Software control register of the IP core.

Table A.1: Description of the AXI register ADC_CR.

Bit(s)	Name	Default	Access	True ('1')	False ('0')
0	MODE	0	RW	Continuous mode: The core is triggered as frequent as possible	Triggered mode: The core must be triggered by software or by hardware where hardware trigger is prioritized
1	TRIGGER	0	RW	Start conversion of the channels selected in synchronously. If a selected channel is busy when the bit is set a new conversion is only started after the ongoing conversion has terminated.	The bit is reset by hardware after the conversion has been started synchronously.
2	SW_RESET	0	RW	Trigger a reset of the IP core by software. All registers and ports are reset to their default values.	Reset finished. Reading from this bit always returns 0.
3	CONV_VALUE_VALID	0	RW	Indicates that the value in is valid. After setting the bit the value will be updated in the selected channels as soon as the selected channels are not busy anymore.	Reset by hardware after the value in is read.
[4..6]	CR_CONFIG_VALUE_[0..2]	000	RW	See Table A.2	See Table A.2

Table A.2: Description of the AXI register ADC_CR.

6	5	4	Access	Description	Encoding
0	0	0	RW	The value in the ADC_VALUE register is the offset	Signed two's complement
0	0	1	RW	The value in the ADC_VALUE register is the conversion factor	Signed two's complement
0	1	0	RW	The value in the ADC_VALUE register is the number of samples per trigger	Unsigned integer
0	1	1	RW	Reserved	
1	0	0	RW	Reserved	
1	0	1	RW	Reserved	
1	1	0	RW	Reserved	
1	1	1	RW	Reserved	

A.1.2 SPI Control Register

Address offset: 0x4
The SPI interfaces can be controlled manually with this register in order to use sleep and nap modes of the ADC. The signal *SS_N* and *SCLK* only can be controlled manually if the selected master channels are not busy. Check ADC_MASTER_BUSY as a status indicator.
Furthermore, the clock polarity and the sample phase are set with this register. This setting applies globally to all SPI masters instantiated.

Table A.3: Description of the AXI register `ADC_SPI_CR`.

Bit(s)	Name	Default	Access	True ('1')	False ('0')
0	SPI_SS_N	0	RW	Set the *SS_N* signal to high	Set the *SS_N* signal to low
1	SPI_SS_N_STATUS	0	RW	The SS_N signal is high	The SS_N signal is low
2	SPI_SCLK	0	RW	Set the SCLK signal to high	Set the SCLK signal to low
3	SPI_SCLK_STATUS	0	RW	The SCLK signal is high	The SCLK signal is low
4	SPI_CONTROL	0	RW	Enable manual control of the SPI. SPI_SS_N_STATUS and SPI_SCLK_STATUS are only valid when SPI_CONTROL is true and the selected channels are not busy.	Disable manual control of the SPI
5	SPI_CONTROL_STATUS	0	RW	Manual control of the SPI interface is possible.	Manual control of the SPI interface is not possible.
6	SPI_CPOL	1	RW	IDLE state of the SCLK signal is logic high	IDLE signal of the SCLK signal is logic low
7	SPI_CPHA	0	RW	Sample on the second edge of SCLK	Sample on the first edge of SCLK

A.1.3 SPI Configuration Register

Address offset: 0x8
Setting for:

- DCNVSCKL (a.k.a PRE_WAIT)
- DSCKLCNVH (a.k.a POST_WAIT)
- Number of system clock cycles per half SCLK cycle - 1 (a.k.a CLK_DIV)

See [6, p. 22] for illustration. The values given indicate the number of system clock cycles for the time described.

Table A.4: Description of the AXI register `ADC_SPI_CFGR`.

Bit(s)	Name	Default	Access	Description	Encoding
0 - 15	CLK_DIV	0	RW	Number of system clock cycles per half SCLK period - 1	Unsigned integer (binary)
16 - 23	PRE_WAIT	0	RW	Number of system clock cycles between the falling edge of SS_N and first SCLK edge - 1. a.k.a DCNVSCKL	Unsigned integer (binary)
24 - 31	POST_WAIT	0	RW	Number of system clock cycles between the last rising edge of SCLK and the rising edge of SS_N - 1. a.k.a DSCKLCNVH	Unsigned integer (binary)

A.1.4 Master Channel Selection

Address offset: 0xC
Encoding: One-Hot
This register is used for two different functions:

1. Update of the configuration values such as offset, conversion factor and number of samples per trigger. In order to specify which individual ADC channels shall be updated, the SPI master as well as the ADC which is controlled by the selected SPI master channel must be selected. The individual channel selection is done in ADC_CHANNEL
2. Channel selection for software trigger: When setting the software trigger bit in the ADC_CR all channels selected in ADC_MASTER_CHANNEL are triggered by software. When using hardware trigger the content of this register is ignored.

Table A.5: Description of the AXI register `ADC_MASTER_CHANNEL`.

Bit(s)	Name	Default	Access	True ('1')	False ('0')
0 - 31	ADC_MASTER_0 - ADC_MASTER_31	0	RW	The master is selected for the specified operation	The master is not selected for the specified operation

A.1.5 ADC Channel Selection

Address offset: 0x10
Encoding: One-Hot
When updating the offset and conversion factor select the channel on the SPI masters selected in ADC_MASTER_CHANNEL that shall be updated.

Table A.6: Description of the AXI register ADC_CHANNEL.

Bit(s)	Name	Default	Access	True ('1')	False ('0')
0 - 31	ADC_CH_0 - ADC_CH_31	0	RW	The individual ADC channel is selected for the specified operation	The individual ADC channel is selected for the specified operation

A.1.6 Transmission Ended Register

Address offset: 0x14
Encoding: One-Hot
This register indicates that an SPI master unit finished with the transmission of the raw value from the SPI master i.e. the value on the hardware port RAW_VALUE is valid for the indicated channels.

Table A.7: Description of the AXI register ADC_MASTER_FINISH.

Bit(s)	Name	Default	Access	True ('1')	False ('0')
0 - 31	ADC_MASTER_0 - ADC_MASTER_31	0	R	The transmission on the specified master channel finished	There is a transmission ongoing on the master channel

A.1.7 Addition and Multiplication Ended Register

Address offset: 0x18
Encoding: One-Hot
This register indicates that an SPI master unit finished with the addition and the multiplication of the raw value i.e. the value on the hardware port SI_VALUE is valid for the indicated channels.

Table A.8: Description of the AXI register `ADC_MASTER_SI_FINISH`.

Bit(s)	Name	Default	Access	True ('1')	False ('0')
0 - 31	ADC_MASTER_0 - ADC_MASTER_31	0	R	The processing the specified master channel finished	There is processing ongoing on the master channel

A.1.8 Conversion Ongoing Register

Address offset: 0x1C
Encoding: One-Hot
The indicated master channels are currently busy i.e. a transmission or a multiplication is ongoing.

Table A.9: Description of the AXI register `ADC_MASTER_BUSY`.

Bit(s)	Name	Default	Access	True ('1')	False ('0')
0 - 31	ADC_MASTER_0 - ADC_MASTER_31	0	R	The specified master channel is busy	The specified master channel is not busy

A.1.9 Configuration Value Register

Address offset: 0x20
The register takes the following values:

- The offset (Encoding: Signed two's complement).
- The multiplication factor (Encoding: Signed two's complement).
- The number of samples taken per trigger event (Encoding: Unsigned integer).

The distinction between these values is done by the setting of ADC_CR register described in Table A.1 and A.2.

A.1.10 ADC Available Indicator

Address offset: 0x24
Encoding: One-Hot
The indicated master channels are currently not available because they are either in sleep mode or in nap mode. This register is set by software and used by the hardware in order to prohibit a trigger when an ADC is not available.

Table A.10: Description of the AXI register `ADC_MASTER_AVAILABLE`.

Bit(s)	Name	Default	Access	True ('1')	False ('0')
0 - 31	ADC_MASTER_0 - ADC_MASTER_31	0	RW	The specified master channel is available	The specified master channel is not available

A.2 IO Signals

Table A.11: IO Signals of the IP core

Port Name	Direction	Port Definition	Reset State	Description
RAW_VALUE	O	std_logic_vector(DATA_WIDTH * CHANNELS_PER_MASTER * SPI_MASTER - 1 downto 0)	′0′	Raw value outputed by the ADC
SI_VALUE	O	std_logic_vector((SPI_MASTER * CHANNELS_PER_MASTER * (RES_MSB - RES_LSB + 1)) - 1 downto 0)	′0′	Converted Value = (RAW_VALUE + OFFSET) * CONVERSION
RAW_VALID	O	std_logic_vector(SPI_MASTER - 1 downto 0)	′0′	The value on port RAW_VALUE is valid. High active.
SI_VALID	O	std_logic_vector(SPI_MASTER - 1 downto 0)	′0′	The value on port SI_VALUE is valid. High active.
TRIGGER_CNV	I	std_logic_vector(SPI_MASTER - 1 downto 0)	–	Hardware trigger input to trigger a conversion.
SCLK	O	std_logic_vector(SPI_MASTER - 1 downto 0)	′1′	SCLK signal for each individual SPI master. Only available if DIFFERENTIAL = true.
SCLK_DIFF	O	std_logic_vector(2 * SPI_MASTER - 1 downto 0)	logic 1	Differential SCLK signal for each individual SPI master. Only available if DIFFERENTIAL = true.
SS_N	O	std_logic_vector(SPI_MASTER - 1 downto 0)	′0′	SS_N signal for each individual SPI master.
MISO	I	std_logic_vector(CHANNELS_PER_MASTER * SPI_MASTER - 1 downto 0)	–	Data input for each individual ADC. Only available if DIFFERENTIAL = false.
MISO_DIFF	I	std_logic_vector(2 * CHANNELS_PER_MASTER * SPI_MASTER - 1 downto 0)	–	Differential data input for each individual ADC. Only available if DIFFERENTIAL = true.

A.3 Design Parameters

Table A.12: Design parameters of the IP core

Parameter Name	Allowable Values	Default	Type	Description
DATA_WIDTH	1 - 24	16	natural	Data output width of the connected SPI slave (i.e. ADC)
CHANNELS_PER_MASTER	1 - 32	4	natural	Number of SPI slaves that are controlled synchronously by one SPI master
SPI_MASTER	1 - 32	2	natural	Number of independent SPI masters
OFFSET_WIDTH	1 - DATA_WIDTH	16	natural	Bit width of the offset value which is added to the raw value
CONVERSION_WIDTH	1 - 18	18	natural	Bit width of the conversion value the sum of the offset and the raw value is multiplied with
RES_LSB	0 - DATA_WIDTH + CONVERSION_WIDTH - 1	6	natural	LSB of the result vector of the DSP48 block which is connected to the IP core SI_VALUE output
RES_LSB	0 - DATA_WIDTH + CONVERSION_WIDTH - 1	23	natural	MSB of the result vector of the DSP48 block which is connected to the IP core SI_VALUE output
DIFFERENTIAL	true false	true	boolean	If true differential buffers are instantiated for SCLK and MISO ports. Otherwise standard CMOS buffers are instantiated

B Environment

B.1 Measurement Setup

The measurements presented in section 3.2.2 are performed with the following equipment:

- Signal generator: Keysight InfiniiVision DSO-X 4034A Mixed Signal Oscilloscope
- Oscilloscope: Keysight InfiniiVision MSOX 4054A Mixed Signal Oscilloscope
- Single ended analog probe: Keysight N2894A 700 MHz. Fig. **??** shows the result of the probe trimming.
- Single ended digital probe: Keysight N2756A
- DC voltage source: Rohde und Schwarz HM7042-5

B.2 Software Tools

In order to ensure reproducibility, a listing of the tools and versions used to design and implement the system are presented in the following paragraphs:

VHDL Synthesis and Simulation For the synthesis and the simulation of VHDL descriptions Xilinx Vivado Version 2020.1 has been used.

SysML Concerning the system modeling domain, Enterprise Architect 14 (EA) by Sparx Systems has been used. Originally, the tool has been developed for software engineering with UML, but with the rise of SysML, Sparx System also integrated this functionality in Enterprise Architect. The tool has been chosen, because the developer already has experience with it and because it is open for automated post-processing like code generation and automated model verification.