ConfSlaveCore

Reference Manual

Product Info	
Product Manager	Sven Meier
Author(s)	Sven Meier
Reviewer(s)	-
Version	1.0
Date	15.11.2018

Copyright Notice

Disclaimer

The information available to you in this document/code may contain errors and is subject to periods of interruption. While NetTimeLogic GmbH does its best to maintain the information it offers in the document/code, it cannot be held responsible for any errors, defects, lost profits, or other consequential damages arising from the use of this document/code.

NETTIMELOGIC GMBH PROVIDES THE INFORMATION, SERVICES AND PRODUCTS AVAILABLE IN THIS DOCUMENT/CODE "AS IS," WITH NO WARRANTIES WHATSOEVER. ALL EXPRESS WARRANTIES AND ALL IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF PROPRIETARY RIGHTS ARE HEREBY DISCLAIMED TO THE FULLEST EXTENT PERMITTED BY LAW. IN NO EVENT SHALL NETTIMELOGIC GMBH BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL, SPECIAL AND EXEMPLARY DAMAGES, OR ANY DAMAGES WHATSOEVER, ARISING FROM THE USE OR PERFORMANCE OF THIS DOCUMENT/CODE OR FROM ANY INFORMATION, SERVICES OR PRODUCTS PROVIDED THROUGH THIS DOCUMENT/CODE, EVEN IF NETTIMELOGIC GMBH HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

IF YOU ARE DISSATISFIED WITH THIS DOCUMENT/CODE, OR ANY PORTION THEREOF, YOUR EXCLUSIVE REMEDY SHALL BE TO CEASE USING THE DOCUMENT/CODE.

Overview

NetTimeLogic's Conf Slave is a helper core to access AXI registers from a Host via UART or Ethernet. It uses a proprietary protocol to convert the commands from the Host into AXI access as AXI4Lite Master. In addition, it contains a ROM which stores all address ranges, types and instance number of all Slaves connected (has to be filled manually).
It is meant to be used with NetTimeLogic's Universal Configuration Manager but can also be used by other software.

Key Features:

- Configuration of the cores via UART or Ethernet
- Status monitoring of the cores via UART or Ethernet
- Register access to all AXI addresses in the system (also 3rd party)
- Provide list of available cores and base addresses
- Proprietary protocol for the UART/Ethernet connection, can also be done from a terminal (UART only)

Revision History

This table shows the revision history of this document.

Version	Date	Revision
0.1	12.11.2018	First draft
1.0	15.11.2018	First release

Table 1: Revision History

Content

Definitions

Definitions	
Universal Configuration Manager	GUI solution which allows access to NetTimeLogic cores which uses the Conf Slave core

Table 2: Definitions

Abbreviations

Abbreviations	
AXI	AMBA4 Specification (Stream and Memory Mapped)
TB	Testbench
LUT	Look Up Table
FF	Flip Flop
RAM	Random Access Memory
ROM	Read Only Memory
FPGA	Field Programmable Gate Array
VHDL	Hardware description Language for FPGA's

Table 3: Abbreviations

1 Introduction

1.1 Context Overview

NetTimeLogic's Conf Slave is the FPGA part of NetTimeLogic's Universal Configuration Manager solution which is meant as a solution for configuring and supervising all NetTimeLogic's IP cores. It allows to configure the configuration registers of the individual cores and allows to supervise the status of the cores. The connection between the host and the target is done either via UART (often USB UART) or 100Mbit Ethernet and has its own protocol running on it. The solution consists of two parts, an FPGA part which is the Conf Slave described here and a GUI part. The FPGA part (Conf Slave allows the access to the registers, provides information about the cores in the system and makes a protocol and interface conversion between UART/Ethernet and AXI. The GUI part is the frontend for the user, it abstracts the communication interface and the individual registers and does the data representation. Multiple instances of the same core in a system are handled and can be configured individually.

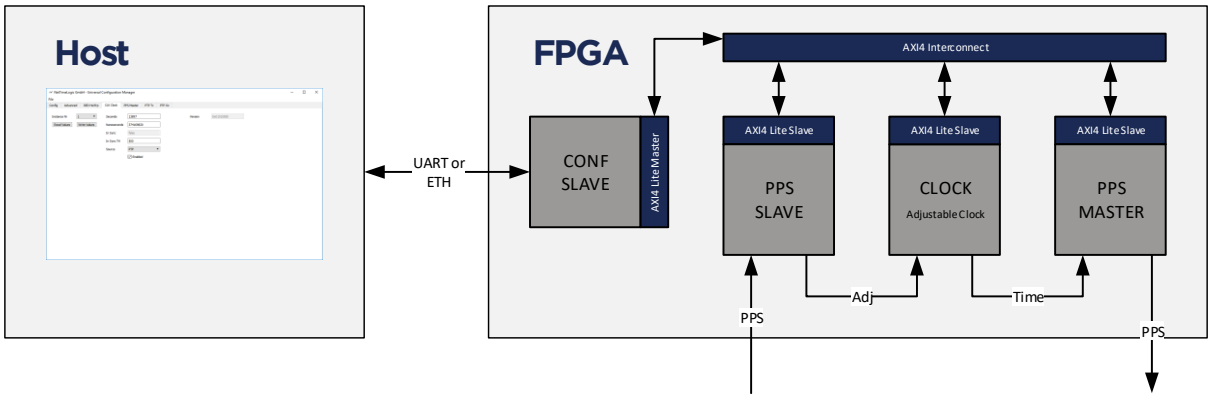

Figure 1: Context Block Diagram

1.2 Function

The Universal Configuration Manager allows to read and write registers via the Conf Slave core which converts between a proprietary UART/Ethernet protocol and AXI. It first tries to connect to the Conf Slave core and asks for a specific acknowledge (if in UART mode also baud rate). If it received the expected acknowledge it reads the configuration ROM in the Conf Slave core to get the information about the instantiated cores like base address and instance number.

Then the registers can be written and read. The Universal Configuration Manager knows the core types and will have a specific configuration and status GUI for each core type. It can however also just read and write specific addresses which are not auto detected.

1.3 FPGA Architecture

The core is split up into different functional blocks for reduction of the complexity, modularity and maximum reuse of blocks. The interfaces between the functional blocks are kept as small as possible for easier understanding of the core.

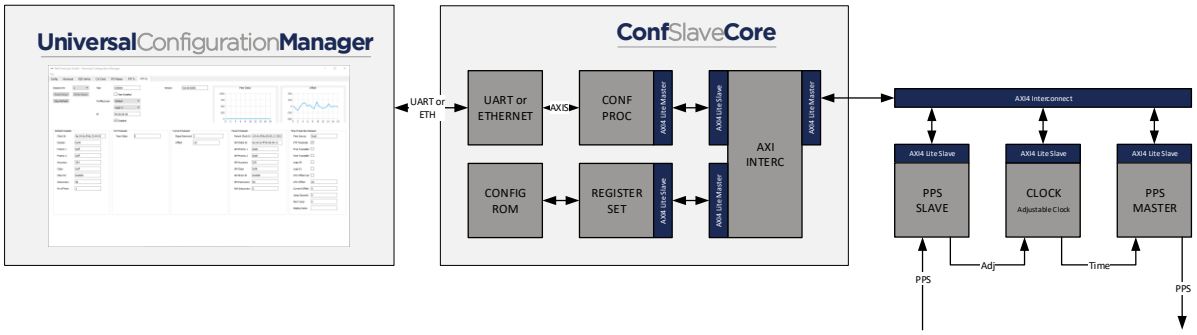

Figure 2: Architecture Block Diagram

UART or ETHERNET

This block converts the UART or Ethernet data stream into AXIS and vice versa.

Conf Processor

This block parses the protocol data received from the UART/ETH block, converts it into AXI access and generates responses towards the host via the UART/ETH block.

AXI interconnect

This block connects the internal Registerset with the AXI Master in the Conf Processor and connects to an external AXI interconnect for accessing all other registers.

Register Set

This block allows reading the configuration from the Config ROM.

Config ROM

This block stores all the information about the instantiated slaves in the ROM. The configuration has to be passed to the Conf Slave core via a structure via generics.

2 Interface and Protocol Basics

2.1 UART Interface

For the communication between the FPGA and the Host a UART interface can be used. Often this UART interface is done via an USB UART. The following parameters are used:

- 1 Start bit
- 8 Data bits
- 1 Stop bit
- No Parity bit
- Baudrate 115200 (default)

2.2 ETHERNET Interface

For the communication between the FPGA and the Host also an 100Mbit Ethernet interface can be used. The following parameters are used:

- 100Mbit only
- Access via Broadcast or Unicast MAC and IP
- IPv4
- TTL: 128
- UDP Port: 0xBEEF

Data is encapsulated into a UDP/IPv4 frame as one command per frame. It also expects ASCII character and does the padding and cut off of the padding.

The Conv Slave will either react and send a response to Broadcast or it will send a response to the requesting MAC and IP, based on a generic.

In addition it has to be configured a MAC and IP address so the node knows when to react on requests and uses them also for responses.

2.3 Protocol

The protocol run on the UART and Ethernet is a proprietary protocol defined by NetTimeLogic.

It is a simple protocol with no retransmission and therefore also not failsafe. The protocol uses ASCII characters, so it can also be entered directly from a terminal. A couple of extra characters are used in the Data stream to allow synchronization of start and end of the commands as well as separation of the individual fields. The command always starts with a '$' character followed by a two-character command code. Then individual fields can follow, each field is separated by a ',' character. After the fields a '*' character indicates the end of the command and that a checksum is followed, the two characters of checksum are followed. The command is ended with a <CR><LF> (carriage return and line feed) combination. The checksum is optional for the host and can be left away, in this case the '*' character is also left away. The checksum XOR combines all received bytes between the '$' and '*' characters (not including) starting with 0x00 as starting value. If a checksum is present, the checksum is checked and an error is signaled by the FPGA to the host if the checksum is not correct and the command ignored.
The protocol engine in the FPGA allows empty lines and comments be transferred also via UART. A comment line starts with "--" characters. This functionality, and the fact that the checksum is optional can be used if the whole content of a file containing not only commands but also comments is copied to a terminal. The Universal Configuration Manager will always send only commands from the host to the FPGA and always with a checksum.

2.3.1 Write Command and Write Response

The two messages described here are used for writing a register.

The format of the write command looks the following:

$WC,<ADDRESS>,<DATA>*<CHECKSUM><CR><LF>

e.g. : $WC,0x50000000,0x40000001*14

A write command is always issued by the host.
The write command starts with the command identifier of the two characters "WC". Following the identifier, the 32bit AXI address to be written in hexadecimal format is added. Following the address, 32bit of write data in hexadecimal format is added. Both address and data have to start with "0x" followed by 8 hexadecimal characters.

A write command will always trigger a write response in the FPGA. If something goes wrong an error response is sent containing an error code.

The format of the write response looks the following:

$WR,<ADDRESS>*<CHECKSUM><CR><LF>

e.g. : $WR,0x50000000*64

A write response is always issued by the FPGA.
The write response starts with the command identifier of the two characters “WR”. Following the identifier, the 32bit AXI address written in hexadecimal format is added which is the address which was written in the FPGA (as in the examples, 0x50000000). The address has to start with “0x” followed by 8 hexadecimal characters.

2.3.2 Read Command and Write Response

The two messages described here are used for reading a register.

The format of the read command looks the following:

$RC,<ADDRESS>*<CHECKSUM><CR><LF>

e.g. : $RC,0x50000000*70

A read command is always issued by the host.
The read command starts with the command identifier of the two characters “RC”. Following the identifier, the 32bit AXI address to be read in hexadecimal format is added. The address has to start with “0x” followed by 8 hexadecimal characters.

A read command will always trigger a read response in the FPGA. If something goes wrong an error response is sent containing an error code.

The format of the read response looks the following:

$RR,<ADDRESS>,<DATA>*<CHECKSUM><CR><LF>

e.g. : $RR,0x50000000,0x00000001*04

A read response is always issued by the FPGA.
The read response starts with the command identifier of the two characters “RR”. Following the identifier, the 32bit AXI address read in hexadecimal format is added which is the address which was read in the FPGA (as in the examples, 0x50000000). Following the address, 32bit of read data read in hexadecimal format is added. Both address and data have to start with “0x” followed by 8 hexadecimal characters.

2.3.3 Connect Command and Connect Response

The two messages described here are used for testing the connection, for e.g. to figure out if a system is connected that supports this protocol.

The format of the connect command looks the following:

$CC*<CHECKSUM><CR><LF>

e.g. : $CC*00

A connect command is always issued by the host.
The connect command starts with the command identifier of the two characters “CC”.

A connect command will always trigger a connect response in the FPGA. If something goes wrong an error response is sent containing an error code.

The format of the connect response looks the following:

$CR*<CHECKSUM><CR><LF>

e.g. : $CR*11

A connect response is always issued by the FPGA.
The connect response starts with the command identifier of the two characters "CR".

2.3.4 Error Response

The error messages described here is used when something goes wrong. It is always issued as reaction to another command.

The format of the error response looks the following:

$ER,<ERROR CODE>*<CHECKSUM><CR><LF>

e.g. : $ER,0x00000003*70

An error response is always issued by the FPGA.
The error response starts with the command identifier of the two characters "ER".
Following the identifier, a 32bit error code in hexadecimal format is added.
The enumeration of the errors as of today is as following:

- 0x00000000: Checksum error
- 0x00000001: Unknown command (or error in command)
- 0x00000002: Read error on AXI
- 0x00000003: Write error on AXI
- 0x00000004: Access timeout error on AXI (illegal address, no answer)

3 Register Set

This is the register set of the Conf Slave. It is accessible via AXI4 Light Memory Mapped. All registers are 32bit wide, no burst access, no unaligned access, no byte enables, no timeouts are supported. Register addresses are only offsets in the memory area where the core is mapped in the AXI inter connects. Non existing register access in the mapped memory area is answered with a slave decoding error.

The registers described below represent an entry in the ROM and repeats every 16bytes:
Entry 0: 0x00000000 – 0x0000000F (always Conf Slave core)
Entry 1: 0x00000010 – 0x0000001F
Entry 2: 0x00000020 – 0x0000002F
Entry 3: 0x00000030 – 0x0000003F
..
Max 255 entries

3.1 Register Overview

Registerset Overview			
Name	Description	Offset	Access
Conf Entry CoreTypeAndInstance Reg	Conf Entry Core Type and Instance Number Register	0x00000000	RO
Conf Entry CoreAddrLow Reg	Conf Entry Core Address Range Low Register	0x00000004	RO
Conf Entry CoreAddrHigh Reg	Conf Entry Core Address Range High Register	0x00000008	RO
Conf Entry CoreIrqMask Reg	Conf Entry Core Interrupt Mask Register	0x0000000C	RO

3.2 Register Descriptions

3.2.1 General

3.2.1.1 CONF Entry CoreType And Instance Number Register

Contains the Core Type and Instance Number of the Slave connected. First 32bit register in a 16byte config block

Conf Entry CoreTypeAndInstance Reg																															
Reg Description																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CORE_TYPE-																INSTANCE_NR															
RO																RO															
Reset: 0xXXXXXXXX																															
Offset: 0x0000, 0x0010, 0x0020 ...																															

Name	Description	Bits	Access
CORE_TYPE	Enumeration of which NetTimeLogic (or 3rd party) core is connected as Slave	Bit: 31:16	RO

	NetTimeLogic Conf Slave Core Type = 1 NetTimeLogic Clk Clock Core Type = 2 NetTimeLogic Clk Signal Generator Core Type = 3 NetTimeLogic Clk Signal Timestamper Core Type = 4 NetTimeLogic Irig Slave Core Type = 5 NetTimeLogic Irig Master Core Type = 6 NetTimeLogic Pps Slave Core Type = 7 NetTimeLogic Pps Master Core Type = 8 NetTimeLogic Ptp Ordinary Clock Core Type = 9 NetTimeLogic Ptp Transparent Clock Core Type = 10 NetTimeLogic Ptp Hybrid Clock Core Type = 11 NetTimeLogic Red Hsr Prp Core Type = 12 NetTimeLogic Rtc Slave Core Type = 13 NetTimeLogic Rtc Master Core Type = 14 NetTimeLogic Tod Slave Core Type = 15 NetTimeLogic Tod Master Core Type = 16 NetTimeLogic Tap Slave Core Type = 17 NetTimeLogic Dcf Slave Core Type = 18 NetTimeLogic Dcf Master Core Type = 19 NetTimeLogic Red Tsn Core Type = 20 3rd Party cores = 20 Empty (end of table) = 0 Reserved unknown = 0xFFFF		
INSTANCE_NR	Instance number per core type	Bit: 15:0	RO

3.2.1.2 CONF Entry Core Address Range Low Register

Contains the Lowest Address of the address range of the Slave connected. Must be 4byte aligned. Second 32bit register in a 16byte config block

Conf Entry CoreAddrLow Reg																													
Reg Description																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDR_LOW																													
RO																													
Reset: 0x00000000																													
Offset: 0x0004, 0x0014, 0x0024 ...																													

Name	Description	Bits	Access
ADDR_LOW	Lowest Address of the address range of the Slave connected.	Bit:31:0	RO

3.2.1.3 CONF Entry Core Address Range High Register

Contains the Highest Address of the address range of the Slave connected. Must be 4byte aligned. Third 32bit register in a 16byte config block

Conf Entry CoreAddrHigh Reg																													
Reg Description																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDR_HIGH																													
RO																													
Reset: 0x00000000																													
Offset: 0x0008, 0x0018, 0x0028 ...																													

Name	Description	Bits	Access
ADDR_LOW	Highest Address of the address range of the Slave connected.	Bit:31:0	RO

3.2.1.4 CONF Entry Core Interrupt Mask Register

Contains the Interrupt Mask showing which IRQs are used by this core of the Slave connected. Must be 4byte aligned. Fourth 32bit register in a 16byte config block

Conf Entry CoreAddrHigh Reg																														
Reg Description																														
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ_31	IRQ_30	IRQ_29	IRQ_28	IRQ_27	IRQ_26	IRQ_25	IRQ_24	IRQ_23	IRQ_22	IRQ_21	IRQ_20	IRQ_19	IRQ_18	IRQ_17	IRQ_16	IRQ_15	IRQ_14	IRQ_13	IRQ_12	IRQ_11	IRQ_10	IRQ_9	IRQ_8	IRQ_7	IRQ_6	IRQ_5	IRQ_4	IRQ_3	IRQ_2	IRQ_1	IRQ_0
RO																														
Reset: 0x00000000																														
Offset: 0x000C, 0x001C, 0x002C ...																														

Name	Description	Bits	Access
IRQ_[X]	Interrupts used by the Slave connected: 0 = unused, 1 = used	Bit:31:0	RO

4 Design Description

The following chapters describe the internals of the Conf Slave: starting with the Top Level, which is a collection of subcores, followed by the description of all subcores.

4.1 Top Level – Conf Slave

4.1.1.1 Parameters

The core must be parametrized at synthesis time. There are a couple of parameters which define the final behavior and resource usage of the core.

Name	Type	Size	Description
Configs_Gen	Conf_Config_Type	256	Configuration Array
NumberOfConfigs_Gen	natural	1	Number of Configurations valid (1-256)
AxiTimeoutNanosecond_Gen	natural	1	AXI timeout in Nanoseconds 0 means no timeout (wait forever)
UartBaudRate_Gen	natural	1	Baudrate (UART only).
ClockClkPeriodNanosecond_Gen	natural	1	Clock Period in Nanosecond: Default for 50 MHz = 20 ns
AxiAddressRangeLow_Gen	std_logic_vector	32	AXI Base Address for the Configuration ROM
AxiAddressRangeHigh_Gen	std_logic_vector	32	AXI Base Address plus Register-set Size Default plus 0xFFFF
IoFf_Gen	boolean	1	If an IO flip flop shall be used (Ethernet only)
Broadcast_Gen	boolean	1	If the core shall answer on the broadcast address (Ethernet only)

Table 4: Parameters

4.1.1.2 Structured Types

4.1.1.2.1 Conf_ConfigEntry_Type

Defined in Conf_Package.h of library ConfLib
Type represents a single configuration entry. An array of this type is defined as Conf_Config_Type

Field Name	Type	Size	Description
CoreType	natural	1	Core Type (will be mapped to 16bit) See register description for enumeration
CoreInstanceNr	natural	1	Core Instance Number (will be mapped to 16bit) starting with 1
AddressRangeLow	std_logic_vector	32	AXI Address Range low
AddressRangeHigh	std_logic_vector	32	AXI Address Range high
InterruptMask	std_logic_vector	32	Interrupt Mask which interrupts are used by this core

Table 5: Conf_ConfigEntry_Type

4.1.1.3 Entity Block Diagram

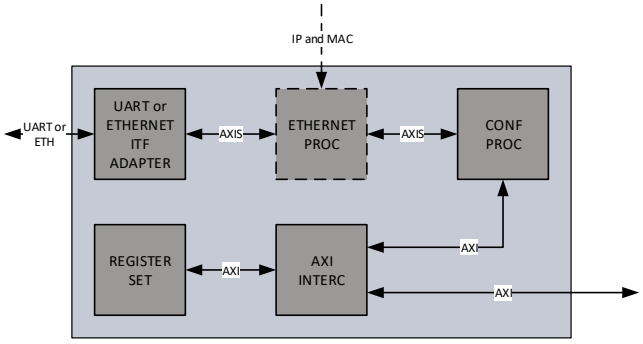

Figure 3: Conf Slave

4.1.1.4 Entity Description

UART or Ethernet Interface Adapter

This block converts the UART or Ethernet data stream into AXIS and vice versa. See 4.2.1 for more details on the UART.

See 4.2.2 for more details on the UART.

Ethernet Processor

This block unpacks and packs the user data from/into UDP/IP Ethernet packets and answers ARP requests.
See 4.2.3 for more details.

Conf Processor

This block parses the protocol data received from the UART/ETH block, converts it into AXI access and generates responses towards the host via the UART/ETH block.
See 4.2.4 for more details.

AXI interconnect

This block connects the internal Registerset with the AXI Master in the Conf Processor and connects to an external AXI interconnect for accessing all other registers.
See 4.2.5 for more details.

Register Set

This block contains a Config ROM and allows reading the configuration from the Config ROM via AXI. The Config ROM stores all the information about the instantiated Slaves. The configuration has to be passed to the Conf Slave core via a structure via generics.
See 4.2.6 for more details.

4.1.1.5 Entity Declaration

Name	Dir	Type	Size	Description
Generics				
General				
Configs_Gen	-	Conf_Config_Type	256	Configuration Array
NumberOf Configs_Gen	-	natural	1	Number of Configurations valid (1-256)
AxiTimeout Nanosecond_Gen	-	natural	1	AXI timeout in Nanoseconds 0 means no timeout (wait forever)

UartBaudRate_Gen	-	natural	1	Baudrate (UART only).
ClockClkPeriod Nanosecond_Gen	-	natural	1	Clock Period in Nanosecond: Default for 50 MHz = 20 ns
AxiAddress RangeLow_Gen	-	std_logic_vector	32	AXI Base Address for the Configura-tion ROM
AxiAddress RangeHigh_Gen	-	std_logic_vector	32	AXI Base Address plus Registerset Size Default plus 0xFFFF
IoFf_Gen	-	boolean	1	If an IO flip flop shall be used (Ethernet only)
Broadcast_Gen		boolean	1	If the core shall answer on the broadcast address (Ethernet only)
Ports				
System				
SysClk_ClkIn	in	std_logic	1	System Clock
SysRstN_RstIn	in	std_logic	1	System Reset
Config				
Uart (UART only)				
Uart_DatIn	in	std_logic	1	Uart Input
Uart_DatOut	out	std_logic	1	Uart output
Ethernet (ETHERNET only)				
(R)(G)MiiRxClk _ClkIn	in	std_logic	1	RX Clock
(R)(G)MiiRxRstN _RstIn	in	std_logic	1	Reset aligned with RX Clock
(R)(G)MiiTxClk _ClkIn	in	std_logic		TX Clock
(R)(G)MiiTxRstN _RstIn	in	std_logic		Reset aligned with TX Clock
(R)(G)MiiRxDv _EnaIn	in	std_logic		RX Data valid
(R)(G)MiiRxErr _EnaIn	in	std_logic		RX Error

(R)(G)MiiRxData_DatIn	in	std_logic_vector	2-8	RX Data MII:4, RMII:2, GMII:8, RGMII:4
(R)(G)MiiCol_DatIn	in	std_logic		Collision
(R)(G)MiiCrs_DatIn	in	std_logic		Carrier Sense
(R)(G)MiiTxEn_EnaOut	out	std_logic		TX Data valid
(R)(G)MiiTxErr_ErrOut	out	std_logic		TX Error
(R)(G)MiiTxData_DatOut	out	std_logic_vector	2-8	TX Data MII:4, RMII:2, GMII:8, RGMII:4
AXI4 Light Master				
AxiWriteAddrValid_ValOut	out	std_logic	1	Write Address Valid
AxiWriteAddrReady_RdyIn	in	std_logic	1	Write Address Ready
AxiWriteAddrAddress_AdrOut	out	std_logic_vector	32	Write Address
AxiWriteAddrProt_DatOut	out	std_logic_vector	3	Write Address Protocol
AxiWriteDataValid_ValOut	out	std_logic	1	Write Data Valid
AxiWriteDataReady_RdyIn	in	std_logic	1	Write Data Ready
AxiWriteDataData_DatOut	out	std_logic_vector	32	Write Data
AxiWriteDataStrobe_DatOut	out	std_logic_vector	4	Write Data Strobe
AxiWriteRespValid_ValIn	in	std_logic	1	Write Response Valid
AxiWriteRespReady_RdyOut	out	std_logic	1	Write Response Ready
AxiWriteRespResponse_DatIn	in	std_logic_vector	2	Write Response
AxiReadAddrValid_ValOut	out	std_logic	1	Read Address Valid
AxiReadAddrReady_RdyIn	in	std_logic	1	Read Address Ready

AxiReadAddr Address_AdrOut	out	std_logic_vector	32	Read Address
AxiReadAddr Prot_DatOut	out	std_logic_vector	3	Read Address Protocol
AxiReadData Valid_ValIn	in	std_logic	1	Read Data Valid
AxiReadData Ready_RdyOut	out	std_logic	1	Read Data Ready
AxiReadData Response_DatIn	in	std_logic_vector	2	Read Data Re-sponse
AxiReadData Data_DatIn	in	std_logic_vector	32	Read Data

Table 6: Signal Generator

4.2 Design Parts

The Conf Slave core consists of a couple of subcores. Each of the subcores itself consist again of smaller function block. The following chapters describe these subcores and their functionality.

4.2.1 UART Interface Adapter

4.2.1.1 Entity Block Diagram

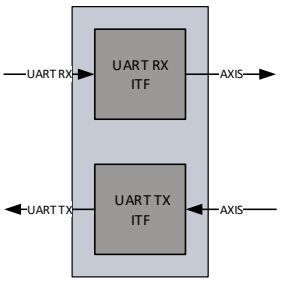

Figure 4: UART Interface Adapter

4.2.1.2 Entity Description

UART RX Interface

This module receives an UART stream and converts it into an 8bit wide AXIS stream, signaling valid every time a complete byte was received and expects that the next module is always able to receive data. This module uses the UART in 8n1 configuration without handshaking.

UART TX Interface

This module receives an 8bit wide AXIS stream and converts it into an UART stream, signaling ready on AXI only when sending is not in progress and has completed. This module uses the UART in 8n1 configuration without handshaking.

4.2.1.3 Entity Declaration

Name	Dir	Type	Size	Description
Generics				
General				
ClockClkPeriod Nanosecond_Gen	-	natural	1	Integer Clock Period
UartBaudRate_Gen	-	natural	1	Baudrate
Ports				
System				
SysClk_ClkIn	in	std_logic	1	System Clock
SysRstN_RstIn	in	std_logic	1	System Reset
Uart				
Uart_DatIn	in	std_logic	1	Uart Input
Uart_DatOut	out	std_logic	1	Uart output
Error Output				
Uart_ErrOut	out	std_logic_vector	2	An error occurred Bit 0 = RX Bit 1 = TX
Axi Output				
AxisValid_ValOut	out	std_logic	1	AXI Stream output
AxisReady_ValIn	in	std_logic	1	
AxisData_DatOut	out	std_logic_vector	8	
AxisStrobe_ValOut	out	std_logic_vector	1	
AxisKeep_ValOut	out	std_logic_vector	1	
AxisLast_ValOut	out	std_logic	1	
AxisUser_DatOut	out	std_logic_vector	1	
Axi Input				
AxisValid_ValIn	in	std_logic	1	AXI Stream input
AxisReady_ValOut	out	std_logic	1	
AxisData_DatIn	in	std_logic_vector	8	
AxisStrobe_ValIn	in	std_logic_vector	1	
AxisKeep_ValIn	in	std_logic_vector	1	
AxisLast_ValIn	in	std_logic	1	
AxisUser_DatIn	in	std_logic_vector	1	

Table 7: UART Interface Adapter

4.2.2 Ethernet Interface Adapter

4.2.2.1 Entity Block Diagram

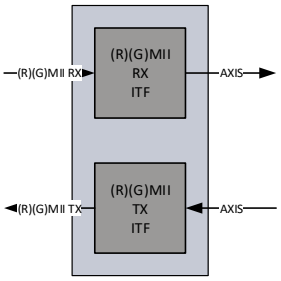

Figure 5: Ethernet Interface Adapter

4.2.2.2 Entity Description

(R)(G)MII RX Interface

This module receives an Ethernet stream over (R)(G)MII and converts it into an 8bit wide AXIS stream, signaling valid every time a complete byte was received and expects that the next module is always able to receive data. This module uses the UART in 8n1 configuration without handshaking.

(R)(G)MII TX Interface

This module receives an 8bit wide AXIS stream and converts it into an an Ethernet stream over (R)(G)MII, signaling ready on AXI only when the adapter is ready to receive more data. This module uses the UART in 8n1 configuration without hand-shaking.

4.2.2.3 Entity Declaration

Name	Dir	Type	Size	Description
Generics				
General				
ClockClkPeriod Nanosecond_Gen	-	natural	1	Integer Clock Period
IoFf_Gen	-	boolean	1	If an IO flip flop shall be used
Ports				
System				
SysClk_ClkIn	in	std_logic	1	System Clock
SysRstN_RstIn	in	std_logic	1	System Reset
Ethernet (ETHERNET only)				
(R)(G)MiiRxClk _ClkIn	in	std_logic	1	RX Clock
(R)(G)MiiRxRstN _RstIn	in	std_logic	1	Reset aligned with RX Clock
(R)(G)MiiTxClk _ClkIn	in	std_logic		TX Clock
(R)(G)MiiTxRstN _RstIn	in	std_logic		Reset aligned with TX Clock
(R)(G)MiiRxDv _EnaIn	in	std_logic		RX Data valid
(R)(G)MiiRxErr _EnaIn	in	std_logic		RX Error
(R)(G)MiiRxData _DatIn	in	std_logic_vector	2-8	RX Data MII:4, RMII:2, GMII:8, RGMII:4
(R)(G)MiiCol_DatIn	in	std_logic		Collision
(R)(G)MiiCrs_DatIn	in	std_logic		Carrier Sense
(R)(G)MiiTxEn_ EnaOut	out	std_logic		TX Data valid
(R)(G)MiiTxErr _ErrOut	out	std_logic		TX Error
(R)(G)MiiTxData _DatOut	out	std_logic_vector	2-8	TX Data MII:4, RMII:2, GMII:8, RGMII:4
Axi Output				
AxisValid_ValOut	out	std_logic	1	AXI Stream output
AxisReady_ValIn	in	std_logic	1	
AxisData_DatOut	out	std_logic_vector	8	
AxisStrobe_ValOut	out	std_logic_vector	1	

AxisKeep_ValOut	out	std_logic_vector	1	
AxisLast_ValOut	out	std_logic	1	
AxisUser_DatOut	out	std_logic_vector	1	
Axi Input				
AxisValid_ValIn	in	std_logic	1	AXI Stream input
AxisReady_ValOut	out	std_logic	1	
AxisData_DatIn	in	std_logic_vector	8	
AxisStrobe_ValIn	in	std_logic_vector	1	
AxisKeep_ValIn	in	std_logic_vector	1	
AxisLast_ValIn	in	std_logic	1	
AxisUser_DatIn	in	std_logic_vector	1	

Table 8: Ethernet Interface Adapter

4.2.3 Ethernet Processor

The Ethernet Processor needs the information of the MAC and IP Address of the module, so it will answer requests only when addressed. This has to be set by the user from outside the core.

4.2.3.1 Entity Block Diagram

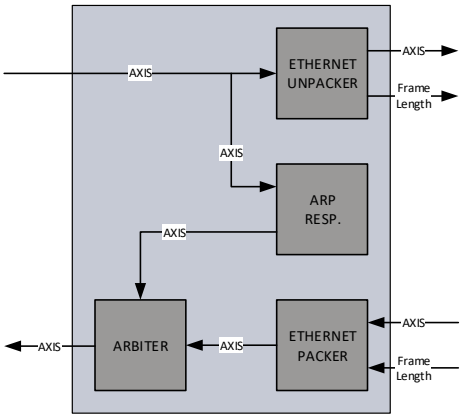

Figure 6: Ethernet Processor

4.2.3.2 Entity Description

Arbiter

This module arbitrates two AXIS interfaces to one, allowing the ARP Responder to answer ARP requests and also the Conf Processor to send responses.

ARP Responder

This module answers ARP requests with ARP responses.

Ethernet Unpacker

This module extracts the user data from the UDP/IP Ethernet frame. For this it checks the MAC, IP and UDP port, only the user data is forwarded to the Conf Processor, removing the Ethernet, IP and UDP header as well as padding. It does not make a CRC check, it relies on the checksum in the user data

Ethernet Packer

This module encapsulates the user data into UDP/IP Ethernet frames. For this it needs to know the user data length. Once the Conf Processor signals that it wants to send data it will start to send an Ethernet Frame with the Ethernet, IP and UDP header and the forward the user data and pads the frame to 64byte.

4.2.3.3 Entity Declaration

Name	Dir	Type	Size	Description
Generics				
General				
ClockClkPeriod Nanosecond_Gen	-	natural	1	Integer Clock Period
UartBaudRate_Gen	-	natural	1	Baudrate
Ports				
System				
SysClk_ClkIn	in	std_logic	1	System Clock
SysRstN_RstIn	in	std_logic	1	System Reset
MAC and IP Input				
OwnMac_DatIn	in	Common_Byte_Type	6	Our MAC
OwnIp_DatIn	in	Common_Byte_Type	4	Our IP
Frame Length Output				
FrameLength_DatOut	out	std_logic_vector	12	Length of the command in bytes
Axi Rx Output				
AxisRxValid_ValOut	out	std_logic	1	AXI RX Stream output
AxisRxReady_ValIn	in	std_logic	1	
AxisRxData_DatOut	out	std_logic_vector	8	
AxisRxStrobe_ValOut	out	std_logic_vector	1	
AxisRxKeep_ValOut	out	std_logic_vector	1	
AxisRxLast_ValOut	out	std_logic	1	
AxisRxUser_DatOut	out	std_logic_vector	1	
Axi Rx Input				
AxisRxValid_ValIn	in	std_logic	1	AXI RX Stream input
AxisRxReady_ValOut	out	std_logic	1	
AxisRxData_DatIn	in	std_logic_vector	8	
AxisRxStrobe_ValIn	in	std_logic_vector	1	
AxisRxKeep_ValIn	in	std_logic_vector	1	
AxisRxLast_ValIn	in	std_logic	1	
AxisRxUser_DatIn	in	std_logic_vector	1	
Frame Length Input				
FrameLength_DatIn	in	std_logic_vector	12	Length of the command in bytes
Axi Tx Output				
AxisTxValid_ValOut	out	std_logic	1	AXI TX Stream output
AxisTxReady_ValIn	in	std_logic	1	

AxisTxData_DatOut	out	std_logic_vector	8	
AxisTxStrobe_ValOut	out	std_logic_vector	1	
AxisTxKeep_ValOut	out	std_logic_vector	1	
AxisTxLast_ValOut	out	std_logic	1	
AxisTxUser_DatOut	out	std_logic_vector	1	
Axi Tx Input				
AxisTxValid_ValIn	in	std_logic	1	AXI TX Stream input
AxisTxReady_ValOut	out	std_logic	1	
AxisTxData_DatIn	in	std_logic_vector	8	
AxisTxStrobe_ValIn	in	std_logic_vector	1	
AxisTxKeep_ValIn	in	std_logic_vector	1	
AxisTxLast_ValIn	in	std_logic	1	
AxisTxUser_DatIn	in	std_logic_vector	1	

Table 9: Ethernet Processor

4.2.4 Conf Processor

4.2.4.1 Entity Block Diagram

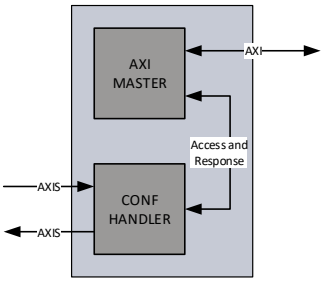

Figure 7: Conf Processor

4.2.4.2 Entity Description

Conf Handler

This module parses the incoming commands and sends responses according to the command, AXI status or error. Once a command is correctly and completely received it will either directly respond to the command or if it is an AXI access command will trigger the AXI Master to make an AXI access according to the address and data from the command. When the AXI access has completed the AXI Master will signal this to the Conf Handler which will then initiate a response.

AXI Master

This module will do an AXI access as AXI4Lite Master when triggered by the Conf Handler and will return the result of the access and also will start a timeout on every start of an AXI access for the case where a non-existing address shall be accessed since no response will ever come. When this timeout (configurable via generic) occurs it will abort the AXI transfer. This is not according to the AXI specification but is the only solution to avoid a deadlock. Also aborting will not cause any problem with NetTimeLogic IP cores. With a timeout value of 0 it will wait forever.

4.2.4.3 Entity Declaration

Name	Dir	Type	Size	Description
Generics				
General				
ClockClkPeriod Nanosecond_Gen	-	natural	1	Integer Clock Period

AxiTimeout Nanosecond_Gen	-	natural	1	AXI timeout in Nanoseconds 0 means no timeout (wait forever)
Ports				
System				
SysClk_ClkIn	in	std_logic	1	System Clock
SysRstN_RstIn	in	std_logic	1	System Reset
Frame Length Output				
FrameLength_DatOut	out	std_logic_vector	12	Length of the command in bytes
Axi Output				
AxisValid_ValOut	out	std_logic	1	AXI Stream output
AxisReady_ValIn	in	std_logic	1	
AxisData_DatOut	out	std_logic_vector	8	
AxisStrobe_ValOut	out	std_logic_vector	1	
AxisKeep_ValOut	out	std_logic_vector	1	
AxisLast_ValOut	out	std_logic	1	
AxisUser_DatOut	out	std_logic_vector	1	
Frame Length Input				
FrameLength_DatIn	in	std_logic_vector	12	Length of the command in bytes
Axi Input				
AxisValid_ValIn	in	std_logic	1	AXI Stream input
AxisReady_ValOut	out	std_logic	1	
AxisData_DatIn	in	std_logic_vector	8	
AxisStrobe_ValIn	in	std_logic_vector	1	
AxisKeep_ValIn	in	std_logic_vector	1	
AxisLast_ValIn	in	std_logic	1	
AxisUser_DatIn	in	std_logic_vector	1	
AXI4 Light Master				
AxiWriteAddr Valid_ValOut	out	std_logic	1	Write Address Valid
AxiWriteAddr Ready_RdyIn	in	std_logic	1	Write Address Ready
AxiWriteAddr Address_AdrOut	out	std_logic_vector	32	Write Address
AxiWriteAddr Prot_DatOut	out	std_logic_vector	3	Write Address Protocol

AxiWriteData Valid_ValOut	out	std_logic	1	Write Data Valid
AxiWriteData Ready_RdyIn	in	std_logic	1	Write Data Ready
AxiWriteData Data_DatOut	out	std_logic_vector	32	Write Data
AxiWriteData Strobe_DatOut	out	std_logic_vector	4	Write Data Strobe
AxiWriteResp Valid_ValIn	in	std_logic	1	Write Response Valid
AxiWriteResp Ready_RdyOut	out	std_logic	1	Write Response Ready
AxiWriteResp Response_DatIn	in	std_logic_vector	2	Write Response
AxiReadAddr Valid_ValOut	out	std_logic	1	Read Address Valid
AxiReadAddr Ready_RdyIn	in	std_logic	1	Read Address Ready
AxiReadAddr Address_AdrOut	out	std_logic_vector	32	Read Address
AxiReadAddr Prot_DatOut	out	std_logic_vector	3	Read Address Protocol
AxiReadData Valid_ValIn	in	std_logic	1	Read Data Valid
AxiReadData Ready_RdyOut	out	std_logic	1	Read Data Ready
AxiReadData Response_DatIn	in	std_logic_vector	2	Read Data Re-sponse
AxiReadData Data_DatIn	in	std_logic_vector	32	Read Data

Table 10: Conf Processor

4.2.5 AXI Interconnect

4.2.5.1 Entity Block Diagram

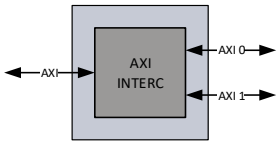

Figure 8: Signal Generator

4.2.5.2 Entity Description

AXI Interconnect

This module multiplexes the AXI4Lite interface between multiple AXI4Lite Slaves and one AXI4Lite Master. The interconnect does not support width conversion or clock domain crossing, it is just a simple interconnect for two Slaves (but the number of Slaves is configurable)

4.2.5.3 Entity Declaration

Name	Dir	Type	Size	Description
Generics				
General				
AxiNrOfSlaves_Gen	-	natural	1	Number of AXI Slaves
Ports				
System				
SysClk_ClkIn	in	std_logic	1	System Clock
SysRstN_RstIn	in	std_logic	1	System Reset
AXI4 Light Slave				
AxiSWriteAddrValid_ValIn	in	std_logic	1	Write Address Valid
AxiSWriteAddrReady_RdyOut	out	std_logic	1	Write Address Ready
AxiSWriteAddrAddress_AdrIn	in	std_logic_vector	32	Write Address
AxiSWriteAddrProt_DatIn	in	std_logic_vector	3	Write Address Protocol
AxiSWriteDataValid_ValIn	in	std_logic	1	Write Data Valid
AxiSWriteDataReady_RdyOut	out	std_logic	1	Write Data Ready

AxiSWriteDataData_DatIn	in	std_logic_vector	32	Write Data
AxiSWriteDataStrobe_DatIn	in	std_logic_vector	4	Write Data Strobe
AxiSWriteRespValid_ValOut	out	std_logic	1	Write Response Valid
AxiSWriteRespReady_RdyIn	in	std_logic	1	Write Response Ready
AxiSWriteResp Response_DatOut	out	std_logic_vector	2	Write Response
AxiSReadAddrValid_ValIn	in	std_logic	1	Read Address Valid
AxiSReadAddrReady_RdyOut	out	std_logic	1	Read Address Ready
AxiSReadAddr Address_AdrIn	in	std_logic_vector	32	Read Address
AxiSReadAddrProt_DatIn	in	std_logic_vector	3	Read Address Protocol
AxiSReadDataValid_ValOut	out	std_logic	1	Read Data Valid
AxiSReadDataReady_RdyIn	in	std_logic	1	Read Data Ready
AxiSReadData Response_DatOut	out	std_logic_vector	2	Read Data
AxiSReadDataData_DatOut	out	std_logic_vector	32	Read Data Response
AXI4 Light Master				
AxiMWriteAddr Valid_ValOut	out	Axi_ItfValid_Type	AxiNrOf Slaves_Gen	Write Address Valid
AxiMWriteAddr Ready_RdyIn	in	Axi_ItfReady_Type	AxiNrOf Slaves_Gen	Write Address Ready
AxiMWriteAddr Address_AdrOut	out	Axi_ItfAddress_Type	AxiNrOf Slaves_Gen	Write Address
AxiMWriteAddr Prot_DatOut	out	Axi_ItfProt_Type	AxiNrOf Slaves_Gen	Write Address Protocol
AxiMWriteData Valid_ValOut	out	Axi_ItfValid_Type	AxiNrOf Slaves_Gen	Write Data Valid
AxiMWriteData Ready_RdyIn	in	Axi_ItfReady_Type	AxiNrOf Slaves_Gen	Write Data Ready
AxiMWriteData Data_DatOut	out	Axi_ItfData_Type	AxiNrOf Slaves_Gen	Write Data
AxiMWriteData Strobe_DatOut	out	Axi_ItfStrobe_Type	AxiNrOf Slaves_Gen	Write Data Strobe

AxiMWriteResp Valid_ValIn	in	Axi_ItfValid_Type	AxiNrOf Slaves_Gen	Write Response Valid
AxiMWriteResp Ready_RdyOut	out	Axi_ItfReady_Type	AxiNrOf Slaves_Gen	Write Response Ready
AxiMWriteResp Response_DatIn	in	Axi_ItfResponse_ Type	AxiNrOf Slaves_Gen	Write Response
AxiMReadAddr Valid_ValOut	out	Axi_ItfValid_Type	AxiNrOf Slaves_Gen	Read Address Valid
AxiMReadAddr Ready_RdyIn	in	Axi_ItfReady_Type	AxiNrOf Slaves_Gen	Read Address Ready
AxiMReadAddr Address_AdrOut	out	Axi_ItfAddress_Type	AxiNrOf Slaves_Gen	Read Address
AxiMReadAddr Prot_DatOut	out	Axi_ItfProt_Type	AxiNrOf Slaves_Gen	Read Address Protocol
AxiMReadData Valid_ValIn	in	Axi_ItfValid_Type	AxiNrOf Slaves_Gen	Read Data Valid
AxiMReadData Ready_RdyOut	out	Axi_ItfReady_Type	AxiNrOf Slaves_Gen	Read Data Ready
AxiMReadData Response_DatIn	in	Axi_ItfResponse_ Type	AxiNrOf Slaves_Gen	Read Data Re-sponse
AxiMReadData Data_DatIn	in	Axi_ItfData_Type	AxiNrOf Slaves_Gen	Read Data

Table 11: Signal Generator

4.2.6 Registerset

4.2.6.1 Entity Block Diagram

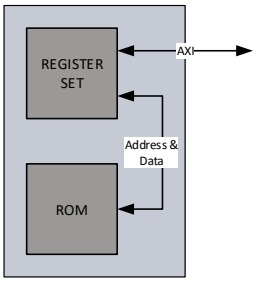

Figure 9: Registerset

4.2.6.2 Entity Description

Registerset

This module is an AXI4 Light Memory Mapped Slave. It provides access to the Configuration ROM. AXI4 Light only supports 32 bit wide data access, no byte enables, no burst, no simultaneous read and writes and no unaligned access. It will forward the address from AXI to the ROM, will wait for the data and respond on AXI. The Registerset only supports reads.
It also converts the configuration generic into a ROM Initialization which is assigned to the ROM at synthesis time.
Access outside the ROM size but within the Address space is answered with 0.

ROM

This module is a simple ROM which contains the core configurations of the Slaves connected to AXI.

4.2.6.3 Entity Declaration

Name	Dir	Type	Size	Description
Generics				
Register Set				
Configs_Gen	-	Conf_Config_Type	256	Configuration Array
NumberOf Configs_Gen	-	natural	1	Number of Configurations valid (1-256)
AxiAddressRange Low_Gen	-	std_logic_vector	32	AXI Base Address for the Configura-

				tion ROM
AxiAddressRange High_Gen	-	std_logic_vector	32	AXI Base Address plus Registerset Size
Ports				
System				
SysClk_ClkIn	in	std_logic	1	System Clock
SysRstN_RstIn	in	std_logic	1	System Reset
AXI4 Light Slave				
AxiWriteAddrValid _ValIn	in	std_logic	1	Write Address Valid
AxiWriteAddrReady _RdyOut	out	std_logic	1	Write Address Ready
AxiWriteAddrAddress _AdrIn	in	std_logic_vector	32	Write Address
AxiWriteAddrProt _DatIn	in	std_logic_vector	3	Write Address Protocol
AxiWriteDataValid _ValIn	in	std_logic	1	Write Data Valid
AxiWriteDataReady _RdyOut	out	std_logic	1	Write Data Ready
AxiWriteDataData _DatIn	in	std_logic_vector	32	Write Data
AxiWriteDataStrobe _DatIn	in	std_logic_vector	4	Write Data Strobe
AxiWriteRespValid _ValOut	out	std_logic	1	Write Response Valid
AxiWriteRespReady _RdyIn	in	std_logic	1	Write Response Ready
AxiWriteResp Response_DatOut	out	std_logic_vector	2	Write Response
AxiReadAddrValid _ValIn	in	std_logic	1	Read Address Valid
AxiReadAddrReady _RdyOut	out	std_logic	1	Read Address Ready
AxiReadAddrAddress _AdrIn	in	std_logic_vector	32	Read Address
AxiReadAddrProt _DatIn	in	std_logic_vector	3	Read Address Protocol
AxiReadDataValid _ValOut	out	std_logic	1	Read Data Valid
AxiReadDataReady _RdyIn	in	std_logic	1	Read Data Ready
AxiReadData Response_DatOut	out	std_logic_vector	2	Read Data
AxiReadDataData _DatOut	out	std_logic_vector	32	Read Data Re-

				sponse

Table 12: Registerset

4.3 Configuration example

4.3.1 Sample Configuration

The configuration below shows a sample configuration for multiple cores

```
constant Configs_Con          : Conf_Config_Type(255 downto 0) := (
                                    0 => (
                                        CoreType      => Conf_RedHsrPrpCoreType_Con,
                                        CoreInstanceNr    => 1,
                                        AddressRangeLow   => x"10000000",
                                        AddressRangeHigh  => x"1000FFFF",
                                        InterruptMask     => x"00000001"),
                                    1 => (
                                        CoreType      => Conf_ClkClockCoreType_Con,
                                        CoreInstanceNr    => 1,
                                        AddressRangeLow   => x"20000000",
                                        AddressRangeHigh  => x"2000FFFF",
                                        InterruptMask     => x"00000000"),
                                    2 => (
                                        CoreType      => Conf_PpsMasterCoreType_Con,
                                        CoreInstanceNr    => 1,
                                        AddressRangeLow   => x"30000000",
                                        AddressRangeHigh  => x"3000FFFF",
                                        InterruptMask     => x"00000000"),
                                    3 => (
                                        CoreType      => Conf_PtpTransparentClockCoreType_Con,
                                        CoreInstanceNr    => 1,
                                        AddressRangeLow   => x"40000000",
                                        AddressRangeHigh  => x"4000FFFF",
                                        InterruptMask     => x"00000010"),
                                    4 => (
                                        CoreType      => Conf_PtpOrdinaryClockCoreType_Con,
                                        CoreInstanceNr    => 1,
                                        AddressRangeLow   => x"50000000",
                                        AddressRangeHigh  => x"5000FFFF",
                                        InterruptMask     => x"00000020"),
                                    others => Conf_ConfigEntry_Type_Rst_Con);

constant NumberOfConfigs_Con  : natural := 5;
```

Figure 10: SampleConfiguration

The signal generation pattern values can be changed while Signal_Val is set to ‘0’.

4.4 Clocking and Reset Concept

4.4.1 Clocking

To keep the design as robust and simple as possible, the whole ConvSlave and all other cores from NetTimeLogic are run in one clock domain. This is considered to be the system clock. Per default this clock is 50MHz. Where possible also the interfaces are run synchronous to this clock. For clock domain crossing asynchronous fifos with gray counters or message patterns with meat stability flip-flops are used. Clock domain crossings for the AXI interface is moved from the AXI slave to the AXI interconnect.

Clock	Frequency	Description
System		
System Clock	50MHz (Default)	System clock where the CC runs on as well as the counter clock etc.
System		
UART	Baudrate up to 2MHz	Asynchronous interface without clock, clock is encoded in data as baudrate
(R)(G)MII Interface		
PHY (R)(G)MII RX Clock	2.5/25/125MHz	Asynchronous, external receive clock from the PHY. Depending on the interface not all frequencies apply.
PHY (R)(G)MII TX Clock	2.5/25/125MHz	Asynchronous, external transmit clock to/from the PHY. Depending on the interface not all frequencies apply.
AXI Interface		
AXI Clock	50MHz (Default)	Internal AXI bus clock, same as the system clock

Table 13: Clocks

4.4.2Reset

In connection with the clocks, there is a reset signal for each clock domain. All resets are active low. All resets can be asynchronously set and shall be synchronously released with the corresponding clock domain. All resets shall be asserted for the first couple (around 8) clock cycles. All resets shall be set simultaneously and released simultaneously to avoid overflow conditions in the core. See the reference designs top file for an example of how the reset shall be handled.

Reset	Polarity	Description
System		
System Reset	Active low	Asynchronous set, synchronous release with the system clock
(R)(G)MII Interface		
PHY (R)(G)MII RX Reset	Active low	Asynchronous set, synchronous release with the (R)(G)MII RX clock
PHY (R)(G)MII TX Reset	Active low	Asynchronous set, synchronous release with the (R)(G)MII TX clock
AXI Interface		
AXI Reset	Active low	Asynchronous set, synchronous release with the AXI clock, which is the same as the system clock

Table 14: Resets

5 Resource Usage

Since the FPGA Architecture between vendors and FPGA families differ there is a split up into the two major FPGA vendors.

5.1 Altera (Cyclone V)

Configuration	FFs	LUTs	BRAMs	DSPs
UART (2 Slaves)	483	588	0	0
Ethernet (2 Slaves)	1115	2507	2	0

Table 15: Resource Usage Altera

5.2 Xilinx (Artix 7)

Configuration	FFs	LUTs	BRAMs	DSPs
UART (2 Slaves)	451	579	0	0
Ethernet (2 Slaves)	899	1418	2	0

Table 16: Resource Usage Xilinx

6 Delivery Structure

```
AXI                                 -- AXI library folder
 |-Library                          -- AXI library component sources
 |-Package                          -- AXI library package sources

CLK                                 -- CLK library folder
 |-Package                          -- CLK library package sources

COMMON                              -- COMMON library folder
 |-Library                          -- COMMON library component sources
 |-Package                          -- COMMON library package sources

CONF                                -- CONF library folder
 |-Core                             -- CONF library cores
 |-Doc                              -- CONF library cores documentations
 |-Library                          -- CONF library component sources
 |-Package                          -- CONF library package sources
 |-Refdesign                        -- CONF library cores reference designs
 |-Testbench                        -- CONF library cores testbench sources and sim/log

SIM                                 -- SIM library folder
 |-Doc                              -- SIM library command documentation
 |-Package                          -- SIM library package sources
 |-Testbench                        -- SIM library testbench template sources
 |-Tools                            -- SIM simulation tools
```

7 Testbench

The Conf Slave testbench consist of 2 parse/port types: UART and CONF.
The CONF port allows to easily read and write registers, it will handle the protocol and do UART access. The UART ports allow to generate error conditions which are not possible with the CONF port.
Two Dummy AXI Slaves are instantiate and connected cia an AXI Interconnect to the DUT, so the CONF port can read and write registers in the two Dummy Slaves via the DUT

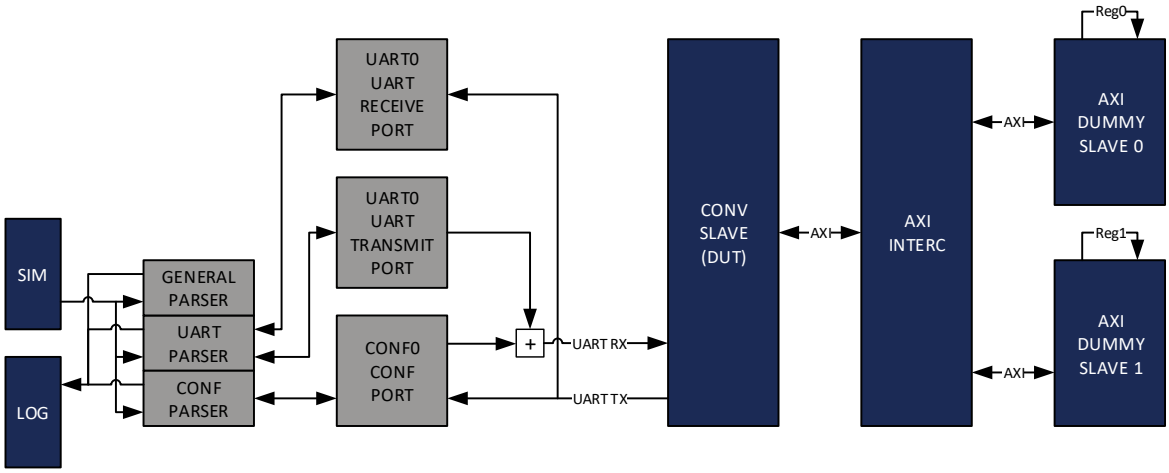

Figure 11: Testbench Framework

For more information on the testbench framework check the Sim_ReferenceManual documentation.

The Baud rate is set to 2 Megabaud to speed up simulation time.

7.1 Run Testbench

1. Run the general script first

```
source XXX/SIM/Tools/source_with_args.tcl
```

2. Start the testbench with all test cases

```
src XXX/CONF/Testbench/Core/ConfSlave/Script/run_Conf_SlaveUart_Tb.tcl
```

3. Check the log file LogFile1.txt in the XXX/CONF/Testbench/Core/ConfSlave/Log/ folder for simulation results.

8 Reference Designs

The Conf Slave reference design contains a PLL to generate all necessary clocks (cores are run at 50 MHz) and an instance of the Conv Slave IP core (either in the UART or Ethernet version), two instances of an AXI Dummy Slave.
The Reference Design is intended to just standalone and using the Universal Configuration Manager to access the Registers
All generics can be adapted to the specific needs.

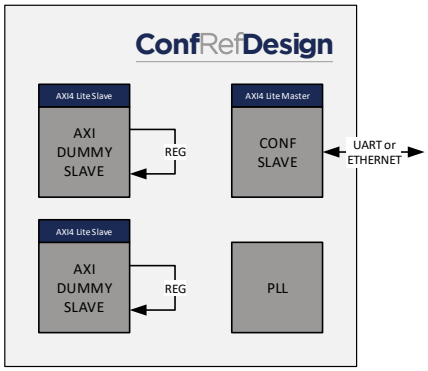

Figure 12: Reference Design

8.1 Altera: Terasic SocKit

The SocKit board is an FPGA board from Terasic Inc. with a Cyclone V SoC FPGA from Altera. (http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=816)

1. Open Quartus 16.x
2. Open Project /CONF/Refdesign/Altera/SocKit/ConfSlaveMii/ConfSlave.qpf
3. Rerun implementation
4. Download to FPGA via JTAG

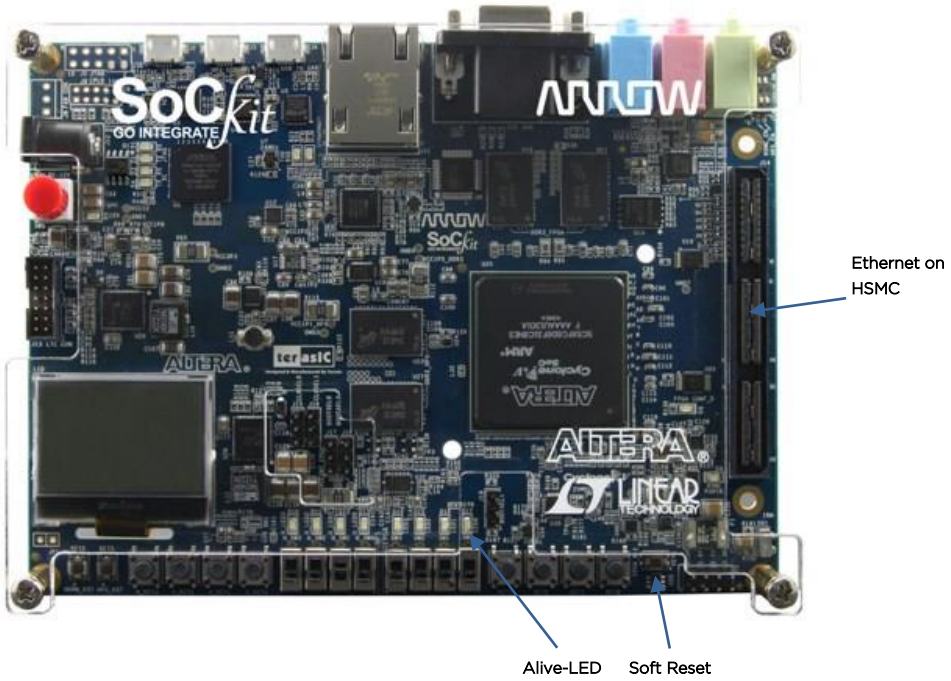

Figure 13: SocKit (source Terasic Inc)

For the ports on the HSMC connector the ETHERNET to HSMC adapter from Terasic Inc. was used.

8.2 Xilinx: Digilent Arty

The Arty board is an FPGA board from Digilent Inc. with an Artix7 FPGA from Xilinx. (http://store.digilentinc.com/arty-board-artix-7-fpga-development-board-for-makers-and-hobbyists/

1. Open Vivado 2017.4
2. Run TCL script /CONF/Refdesign/Xilinx/Arty/ConfSlaveUart/ConfSlave.tcl for UART or /CONF/Refdesign/Xilinx/Arty/ConfSlaveMii/ConfSlave.tcl for Ethernet
    a. This has to be run only the first time and will create a new Vivado Project
3. If the project has been created before open the project and do not rerun the project TCL
4. Rerun implementation
5. Download to FPGA via JTAG

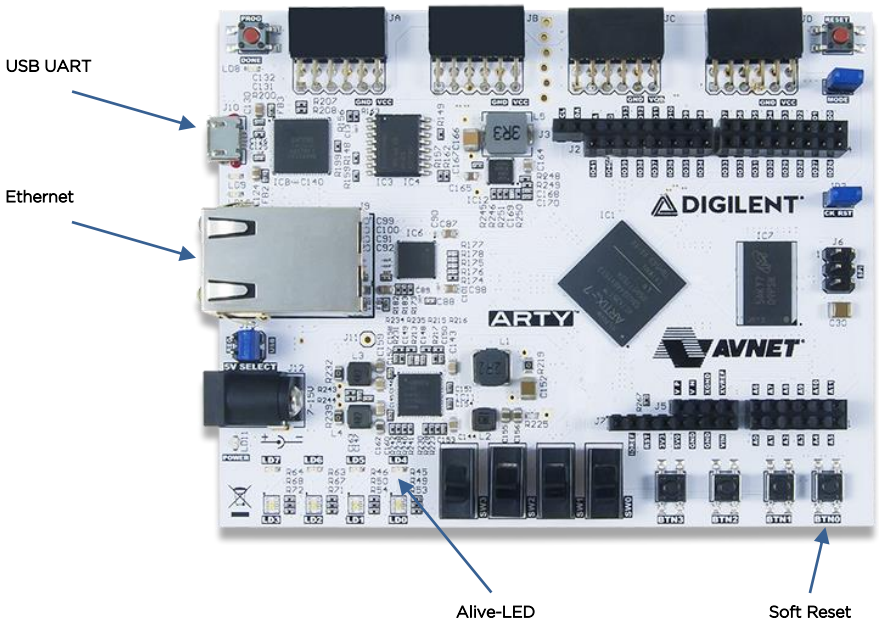

Figure 14: Arty (source Digilent Inc)

A List of tables

B List of figures