RTL Design of SpaceWire Protocol and AMBA-Interface with LEON Processor

By

CHINTAN PATEL
(07MEC020)

Department of Electronics & Communication Engineering
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY,
AHMEDABAD 382481

RTL Design of SpaceWire Protocol and AMBA-Interface with LEON Processor

Major Project Report

Submitted In Partial Fulfillment of the Requirement

For

MASTER OF TECHNOLOGY
IN
ELECTRONICS & COMMUNICATION ENGG.
(VLSI DESIGN)

By

Chintan Patel (07MEC020)

Guided By:

Prof. N.P.Gajjar

Department of Electronics & Communication Engineering
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY,
AHMEDABAD 382481

CERTIFICATE

This is to certify that the Major Project Report entitled **"RTL Design of SpaceWire Protocol & AMBA Interface with LEON Processor"** submitted by Chintan Patel (07MEC020) towards the partial fulfillment of the requirement for the Master of Technology (Electronics & Communication Engineering) in the field of VLSI Design of Institute of Technology, Nirma University of Science and Technology, Ahmedabad is the record of the work carried our under our supervision and guidance. The work submitted has in our opinion reached a level required for being accepted for examination. The results embodied in this dissertation-project work to the best of our knowledge have not been submitted to any other University or Institute for the award of any degree or diploma.

Date:

Place: Ahmedabad

Project Guide	**P.G Co-ordinator**
Prof. N. P. Gajjar	**Dr.N.M.Devashrayee**
Institute of Technology	**VLSI Design**
Nirma University,	**Institute of Technology**
Ahmedabad	**Nirma University,**
	Ahmedabad

HOD	**Director**
Prof. A. S. Ranade	**Dr. K Kotecha**
Dept. of EE Engineering	**Institute of Technology**
Institute of Technology	**Nirma University,**
Nirma University,	**Ahmedabad**
Ahmedabad	

ACKNOWLEDGEMENT

A fruitful effort in a new work needs a direction and guiding hands that shows the way. It is proud privilege and pleasure to bring out indebt ness and warm gratitude to respect **Dr. N. M. Devashrayee**, P.G. Co-coordinator, M.Tech VLSI Design, Nirma University, Ahmedabad for their support during our project work.

We would like to express our profound gratitude to **Prof N. P. Gajjar** and all other team members for outstanding support and time to time guidance during our project work.

Finally we would like to thank Arunkumar, Umang Joshi for their continuous support and constant encouragement during the project work. We would like to thank each and everyone who directly or indirectly helped us in the accomplishment of the project.

Chintan Patel
(07MEC020)

ABSTRACT

With the progressive increase in the use of FPGAs for prototyping the embedded systems on a chip, the designing of IPs (Intellectual property) has also been increasing which are used to enhance the features of the designs. To attach an IP with one of the buses in the design, the designer has to create an interface for it. Some open source processors available like LEON and buses like AMBA bus .

This Project reports the designing of SpaceWire Protocol and AMBA interface with LEON3 processor used as a platform for application specification SoC. The European Space Agency (ESA) proposed the SpaceWire standards for reliable satellite on-board networking at high speed upto hundreds of Mbits/sec. LEON Processor supports radiation tolerant therefore used in most of the Space application. AMBA Bus is used to interface the on chip peripheral with processor. This project report covers the theory of the SpacWire Protocol, its simulation work, theory of LEON processor, implementation of LEON3 core and AMBA AHB interface on Virtex 4 FPGA board and testing of LEON3 based SoC design.

List of Figures

List of Tables

Abbreviation

GRLIB	Gaisler Research Library
AMBA	Advance Microcontroller Bus Controller
AHB	Advance High Speed Bus
APB	Advance Peripheral Bus
FPGA	Field Programmable Gate Array
SoC	System on Chip
SPARC	Scalable Processor Architecture
IP	Intellectual Property
LVDS	Low Voltage Differential Signaling
ESA	European Space Agency
FCT	Flow Control Token
EOP	End of Packet
EEP	Error end of Packet
ESC	Escape

Contents

Chapter 1

Introduction

To meet the growing needs of computing power, communication speed and performance requirements demanded by today's applications, the trend to wards *Systems on a Chip* (SoC) has arrived.These applications particularly in the domain of on board data handling in Space application. SpaceWire Protocols has grown organically from the needs of on-board processing applications. In principle a data-handling system developed for an optical instrument, for example, can be used for a radar instrument by unplugging the optical sensor and plugging in the radar one. Processing units, mass-memory units and down-link telemetry systems developed for one mission can be readily used on another mission, reducing the cost of development.

Motivation

ISRO Respond Project being done at Nirma Institute of Technology, Ahmedabad aims at developing a Design and Development of LEON3 based SoC design As a part of the this project, a design of SpaceWire Protocol and AMBA interface with LEON Processor is required to Design. This forms the motivation of the project.

Objective

This main objective of the project is to developing a SpaceWire Protocol IP and AMBA interface with LEON Processor customizable a platform for experimenting with application specific SoCs.

Overall Approach

This section briefly presents the overall approach. It describes the elements that are involved in the project and also the motive behind the choice of these elements.

SpaceWire Protocol

SpaceWire Protocol is a hi-directional, full-duplex, high-speed (2 to 200Mbits/s), serial data communication link. It was derived from the IEEE- 1355 terrestrial standard and is based on Low Voltage Differential Signaling (LVDS) physical layer resulting in a low-power high-speed link suitable for space applications.

Purpose of SpaceWire

The purpose of the SpaceWire standard is:

- to facilitate the construction of high-performance on-board data handling systems,
- to help reduce system integration costs,
- to promote compatibility between data handling equipment and subsystems,
- to encourage re-use of data handling equipment across several different missions.

LEON

The LEON3 core is an open source VHDL implementation of 32-bit processor conforming to the SPARC V8 architecture. LEON is chosen due to the following factors.

- Highly configurable
- Fully synthesizable over a variety of platforms
- VHDL code freely available under suitable license
- Reasonably good amount of documentation and active online help discussing the Problems and features of the processor
- SPARC compliant architecture which forms the base for number of successful Commercial architectures like the SUN-SPARC series of processors

LEON is designed for embedded applications with the following features on-chip Integer Unit, Floating-point and Co-processor, Cache sub-system with separate instruction and data caches, Debug support unit, flexible Memory interface and controller,Timers, Watchdog, UARTs,Interrupt controller, Parallel I/O port, AMBA on-chip buses, Boot loader and Watch point registers. LEON is explained in greater detail in Chapter 5.

AMBA AHB

The AMBA Advanced High-performance Bus (AHB) is a multi-master bus suitable to interconnect units that are capable of high data rates, and variable latency.

Outline of the report

Chapter 3 explains the general issues that need to be addressed in designing a Spacewire Protocol, which cover the SpaceWire Protocol Scope, description and design details

Chapter 4 explains the simulation result of the SpaceWire Protocol.

Chapter 5 elucidates the architecture of LEON, its compliance with SPARC, bus, external memory and cache sub-system, DSU, software considerations of LEON.
Chapter 6 gives the details of the implementation of the LEON3 core platform Gaisler research grlib and board AVNET EVAL XC4VLX60 step by step approach taken to test the LEON3 core using ‘c’application.
Chapter 7 gives the AMBA AHB interface implementation
Chapter 8 gives the Conclusion and Future work.

Chapter 2

Literature Survey

2.1. SpaceWire

The SpaceWire protocol [6] was first developed by the European Space Agency (ESA) in the late 1990's to provide a high-speed internal network for spacecraft. SpaceWire is intended as a common interface protocol designed to interface with not only spacecraft instruments and control systems, but also ground support and testing equipment. SpaceWire is a high performance serial bus, supporting data rates from 2Mbps to 400Mbps. Based initially on IEEE 1355-1995, the electrical interface has been optimized for the rigors of spacecraft operations and adopted as standard ECSS-E-50-12A by ESA. The intent of SpaceWire is to provide a unified, high performance data handling infrastructure, designed to meet the needs of future space miss

2.2. LEON3 Processor Core

LEON3 [4] is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture. It is designed for embedded applications, combining high performance with low complexity and low power consumption. The LEON3 core has the following main features:

- 7-stage pipeline with Harvard architecture,
- Separate instruction and data caches,
- Hardware multiplier and divider,
- On-chip debug support and multi-processor extensions.

A block diagram of the LEON3 core can be seen below.

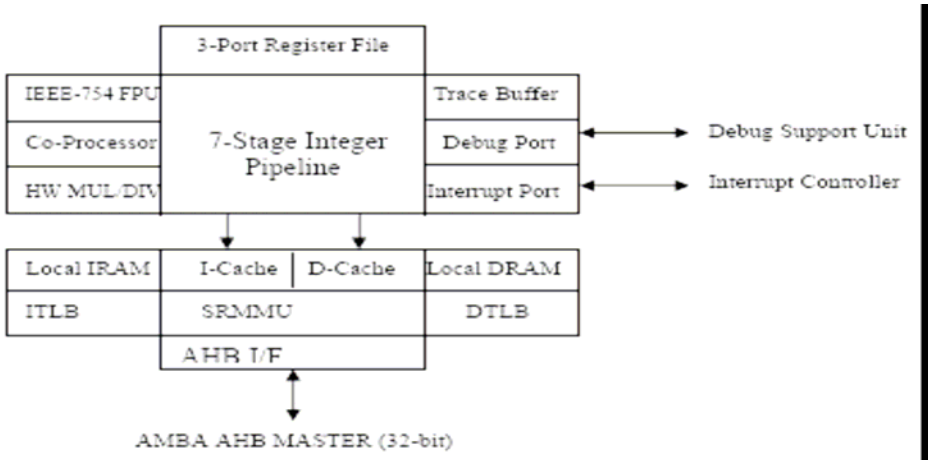


Figure 2.1 LEON3 processor core block diagram [4]

2.3. GRLIB IP Library

GRLIB [2] is a collection of reusable IP cores, divided on multiple VHDL libraries. Each library provides components from a particular vendor or a specific set of shared functions or interfaces. Data structures and component declarations to be used in a GRLIB-based design are exported through library specific VHDL packages.GRLIB is based on the AMBA AHB and APB on-chip buses, which is used as the standard interconnect interface. The implementation of the AHB/APB buses is compliant with the AMBA-2.0 specification, with additional 'sideband' signals for automatic address decoding, interrupt steering and device identification (a.k.a. plug play support). The AHB and APB signals are grouped according to functionality into VHDL records, declared in the GRLIB VHDL library. The GRLIB AMBA package source files are located in lib/grlib/amba.All GRLIB cores use the same data structures to declare the AMBA interfaces, and can then easily be connected together. An AHB bus controller and an AHB/APB bridge are also available in the GRLIB library, and allow to assemble quickly a full AHB/APB system. The figure 2.2 shows an example of a LEON3 system designed with GRLIB.

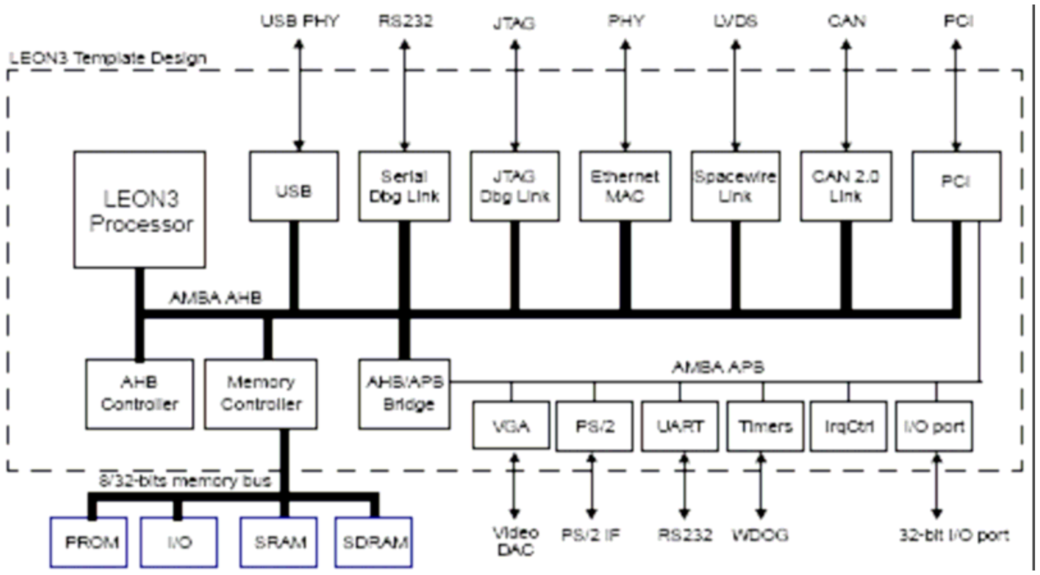


Figure 2.2 Grlib Library IPs [2]

2.4 AMBA AHB/APB buses

2.4.1 AMBA AHB on-chip bus

The AMBA Advanced High-performance Bus (AHB) [9] is a multi-master bus suitable to interconnect units that are capable of high data rates, and variable latency. A conceptual view is provided in the figure below. The attached units are divided into master and slaves, and controlled by a global bus arbiter.

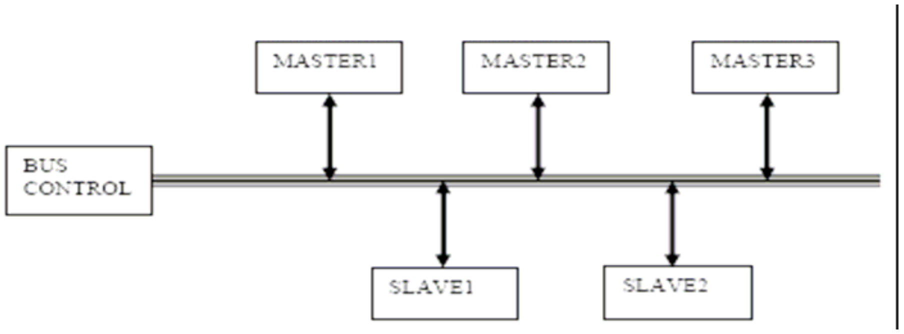


Figure 2.3 AHB - A conceptual view [2]

Since the AHB bus is multiplexed (no tristate signals), a more correct view of the bus and the attached units can be seen in figure 2.3. Each master drives a set of signals grouped into a VHDL record called HMSTO.The output record of the current bus master is selected by the bus multiplexers and sent to the input record (ahbsi) of all AHB slaves. The output record (ahbso) of the active slave is selected by the bus multiplexer and forwarded to all masters. A combined bus arbiter, address decoder and bus multiplexer controls which master and slave are currently selected

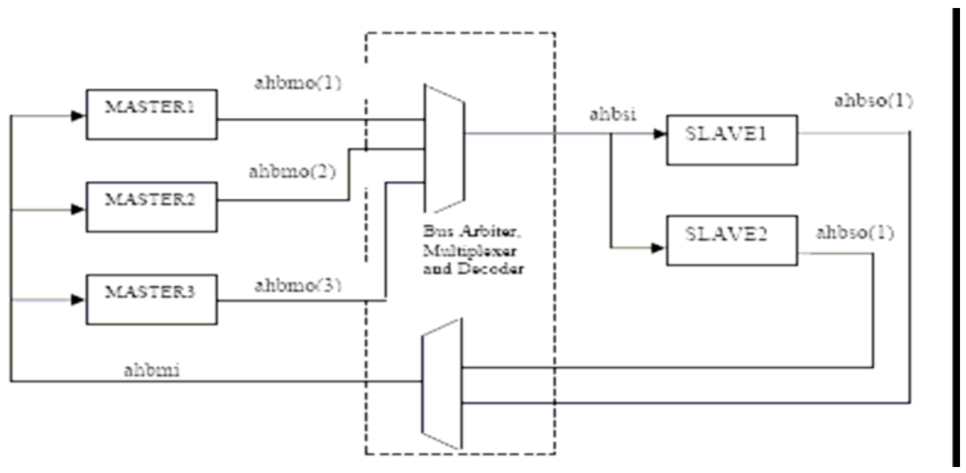


Figure 2.4 AHB- Detailed View [2]

2.4.2 AMBA APB on-chip bus

The AMBA Advanced Peripheral Bus (APB) is a single-master bus suitable to interconnect units of low complexity which require only low data rates. An APB bus is interfaced with an AHB bus by means of a single AHB slave implementing the AHB/APB Bridge. The AHB/APB Bridge is the only APB master on one specific APB bus. More than one APB bus can be connected to one AHB bus, by means of multiple AHB/APB bridges. A conceptual view is provided in figure 2.5.

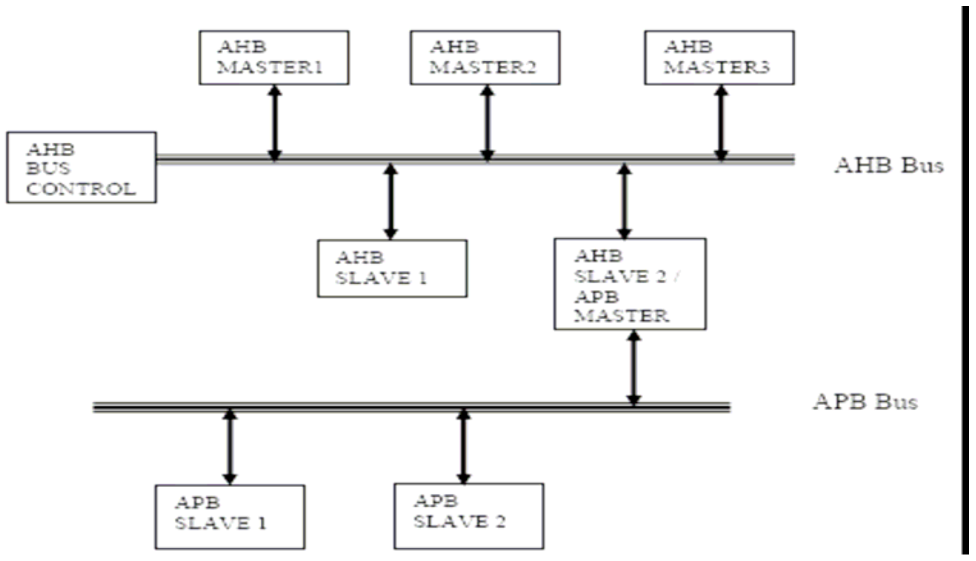


Figure 2.5 APB - A conceptual view [2]

Since the APB bus is multiplexed (no tristate signals), a more correct view of the bus and the attached units can be seen in figure 2.5.The access to the AHB slave input (AHBI) is decoded and an access is made on APB bus. The APB master drives a set of signals grouped into a VHDL record called APBI which is sent to all APB slaves. The combined address decoder and bus multiplexer controls which slave is currently selected. The output record (APBO) of the active APB slave is selected by the bus multiplexer and forwarded to AHB slave output (AHBO).

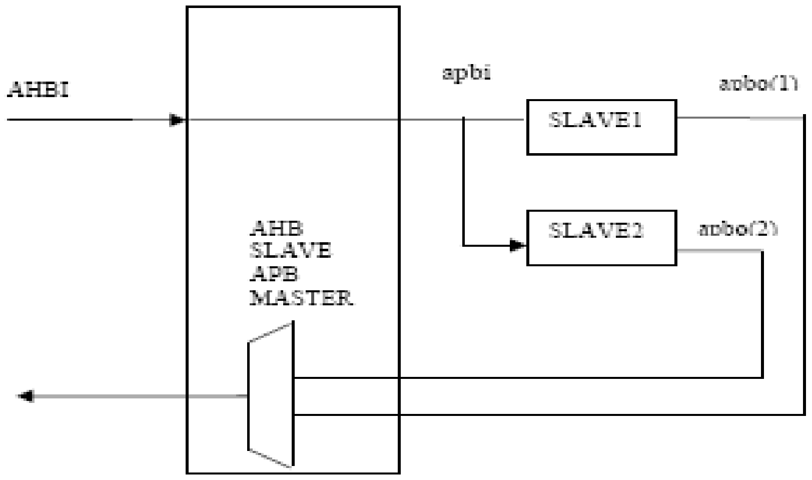


Figure 2.6 APB- Detailed View [2]

2.5 LEON flow

The following steps describe the flow for simulating and synthesizing a LEON based design.

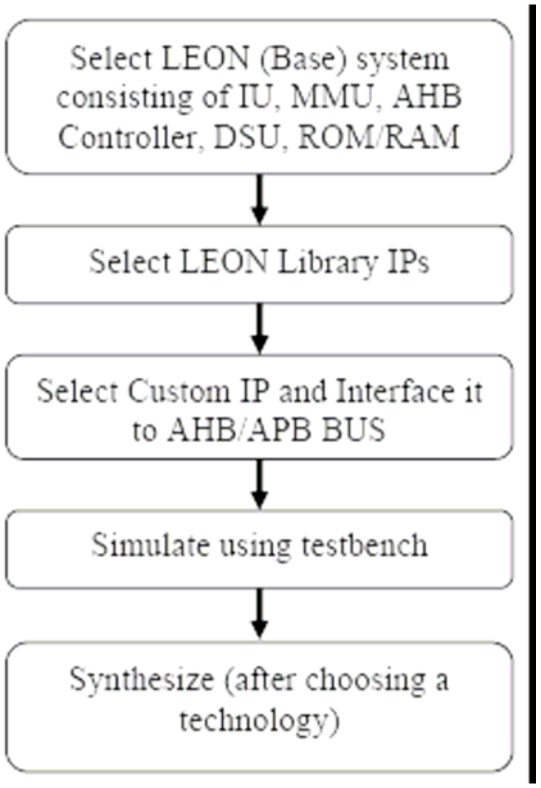

Select LEON (Base) system consisting of IU, MMU, AHB Controller, DSU, ROM/RAM
Select LEON Library IPs
Select Custom IP and Interface it to AHB/APB BUS
Simulate using testbench
Synthesize (after choosing a technology)

Figure 2.7 Leon Flow [10]

2.6 Environment

The following software and hardware are required to proceed further:

Software:

- Cygwin

It provides a unix-like environment for windows PC which is required to Execute the commands of the Leon based tools.

- Tcl/Tk (8.4+)

It is required to execute the tcl/tk scripts which open the GUI form of the

Leon based tools.

- Mentor ModelSim

It is used to simulate the sample designs to check their functionality before they are synthesized.

- Grmon debugger

It provides the debugging of the designs after they are downloaded into the board.

- GRTool

It is combined tool of sparc compiler, mysys.It compiles C application and converts into executable files.

- Xilinx ISE 9.2i(lower versons may also work) or above

To synthesize the designs and create their bit files.

Hardware:

- Windows-PC ,linux
- FPGA board (Avnet Virtex -4)
- Xilinx Parallel cable IV

Chapter 3

SpaceWire Protocol

3.1 Introduction.

The SpaceWire Protocol addresses the handling of payload data on-board a spacecraft. It is a standard for a high-speed data link, which is intended to meet the needs of future, high-capability, remote sensing instruments and other space missions.SpaceWire provides a unified high-speed data-handling infrastructure for connecting together sensors, processing elements, mass-memory units, and downlink telemetry sub-systems.
The purpose of SpaceWire is

- To facilitate the construction of high-performance on-board data-handling systems,
- To help reduce system integration costs,
- To promote compatibility between data-handling equipment and sub-systems,
- To encourage re-use of data-handling equipment across several different missions.

3.2 Scope.

The SpaceWire standard specifies the physical interconnection media and data communication Protocols to enable data to be sent reliably at high-speed (between 2 Mbps and 100 Mbps or more) from one unit to another. SpaceWire links are full-duplex, point-to-point, and serial data communication links. The scope of this standard is the physical connectors and cables, electrical properties, and logical protocols that comprise the SpaceWire data link. SpaceWire provides a means of sending packets of information from a source node to a required destination node. SpaceWire does not specify the contents of the packets of information.
The SpaceWire Protocol covers the following normative protocol levels

- **Physical Level:** Defines connectors and cables.
- **Signal Level:** Defines signal encoding, voltage levels, noise margins, EMC specifications and data signaling rates.

- **Character Level:** Defines the data and control characters used to manage the flow of data across a link.
- **Exchange Level:** Defines the protocol for link initialization, flow control, link error detection and link error recovery.
- **Packet Level:** Defines how data to be transmitted via a SpaceWire link is split up into packets
- **Network Level:** Defines the structure of a SpaceWire network and the way in which packets are transferred from a source node to a destination node across a network. Defines how link errors and network level errors are handled.

3.3 Description.

SpaceWire is a full-duplex, bi-directional, serial, point-to-point data link. It encodes data using two differential signal pairs in each direction. That is a total of eight signal wires, four in each direction. The various Protocols levels explain below.

3.3.1 Physical Level

The physical level of the SpaceWire standard covers cables, connectors and EMC specification.

- **Cables**

The SpaceWire cable comprises four twisted pair wires with a separate shield around each twisted pair and an overall shield. To achieve a high data signaling rate with SpaceWire over distances up to 10m the cable must have the following characteristics:-

- Low signal-signal skew between each signal in a differential pair and between Data and Strobe pairs
- Low signal attenuation
- Low cross-talk
- Good EMC performance
- Connectors

The SpaceWire connector is required to have eight signal contacts plus a screen termination Contact. A nine pin micro-miniature D-type is specified as the SpaceWire connector. This type of connector is available qualified for space use.

- **EMC Specification**

The EMC specifications for SpaceWire have been derived from the EMC specifications for the Rosetta Radiated emission, electric and magnetic fields,

- Radiated susceptibility, electric and magnetic fields,
- Conducted emission,
- Signaling rate
- Fault isolation, and

3.3.2 SIGNAL LEVEL

The signal level part of the SpaceWire standard covers signal voltage levels, noise margins and signal encoding.

Signal Level and Noise Margins

Low Voltage Differential Signaling or LVDS is specified as the signaling technique to be used in SpaceWire. LVDS uses balanced signals to provide very high-speed interconnection using a low voltage swing (350 mV typical). The balanced or differential signaling provides adequate noise margin to enable low voltages to be used in practical systems. Low voltage swing means low power consumption at high speed. LVDS is appropriate for connections between boards in a unit, and unit to unit interconnections over distances of 10m or more. A typical LVDS driver and receiver are shown in Figure 3.1 connected by cable with 100 ohm differential impedance.

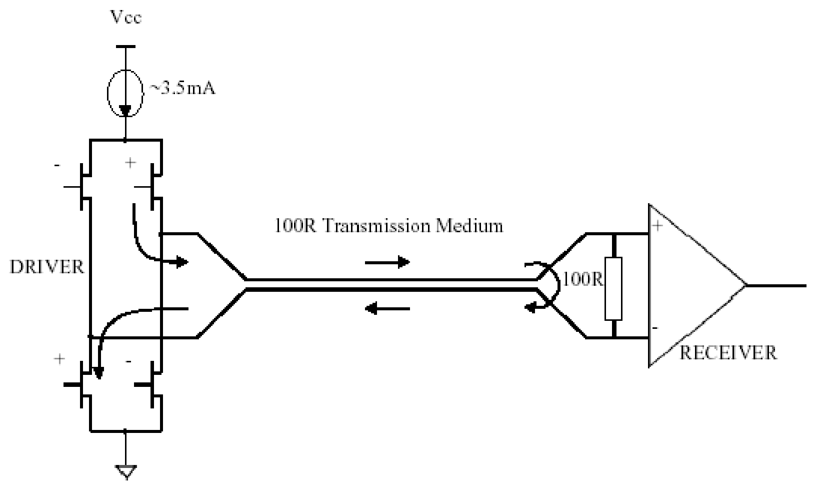


Fig 3.1 LVDS Operation

The LVDS driver uses current mode logic. A constant current source of around 3.5mA provides the current that flows out of the driver, along the transmission medium, through the 100-ohm termination resistance and back to the driver via the transmission medium. Two pairs of transistor switches in the driver control the direction of the current flow through the termination resistor. When the driver transistors marked "+" in Figure 3.1 are turned on and those marked "-" are turned off, current flows as indicated by the arrows on the diagram creating a positive voltage across the termination resistor. LVDS receivers are specified to have high input impedance so that most of the current will flow through the termination resistor to generate around ±350mV with the nominal 3.5mA current source.

Data Encoding

SpaceWire uses Data-Strobe (DS) encoding. This is a coding scheme which encodes the transmission clock with the data into data and strobe so that the clock can be recovered by simply XORing the data and strobe lines together. The data values are transmitted directly and the strobe signal changes state whenever the data remains constant from one data bit interval to the next. This coding scheme is illustrated below in Figure 3.2.The reason for using DS encoding is to improve the skew tolerance to almost 1-bit time, compared to 0.5 bit time for simple data and clock encoding.

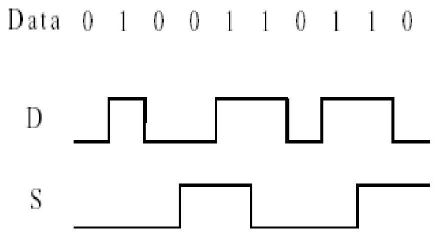


Figure 3.2 Data-Strobe (DS) Encoding

A SpaceWire link comprises two pairs of differential signals, one pair transmitting the D and S signals in one direction and the other pair transmitting D and S in the opposite direction. That is a total of eight wires for each bi-directional link.

3.3.3 Character Level

There are two types of characters:-

- Data characters which hold an eight-bit data value transmitted least-significant bit first. Each data character contains a parity-bit, a data-control flag and the eight-bits of data. The parity-bit covers the previous eight-bits of data, the current parity-bit and the current data-control flag. It is set to produce odd parity so that the total number of 1's in the field covered is an odd number. The data control flag is set to zero to indicate that the current character is a data character.
- Control characters which hold a two-bit control code. Each control character is formed from a parity-bit, a data-control flag and the two-bit control code. The data-control flag is set to one to indicate that the current character is a control character. Parity coverage is similar to that for a data character. One of the four possible control characters is the escape code (ESC). This can be used to form longer control codes. One longer control code is specified which is the NULL code. NULL is formed from ESC followed by the flow control token (FCT). NULL is transmitted whenever a link is not sending data or control tokens to keep the link active and to support link disconnect detection.

The data and control characters are illustrated in Figure 3.3.

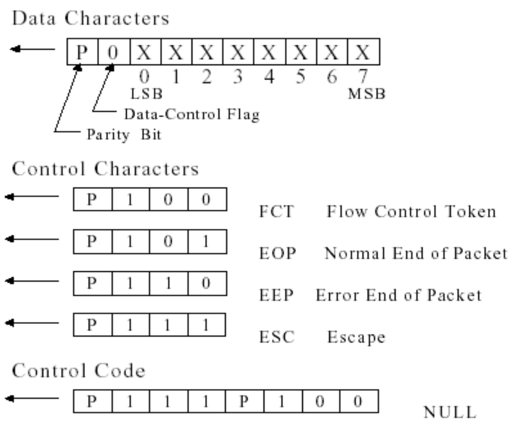


Figure 3.3 Data and Control Characters

3.3.4 Exchange Level

The Exchange Level shows the following Processes.

Initialization: Following reset the link output is held in the reset state until it is instructed to start and attempt to make a connection with the link interface at the other end of the link. A connection is made following a handshake that ensures both ends of the link are able to send and receive characters successfully. Each end of the link sends NULLs, waits to receive a NULL, then sends FCTs and waits to receive an FCT. Since a link interface cannot send FCTs until it has received a NULL, receipt of one or more NULLs followed by receipt of an FCT means that the other end of the link has received NULLs successfully and that full connection has been achieved.

Flow Control: A transmitter is only allowed to transmit data characters if there is space in the host system receive buffer for them. The host system indicates that there is space for eight more data characters by requesting the link transmitter to send a flow control token (FCT). The FCT is received at the other end of the link (end B) enabling the transmitter at end B to send up to eight more FCTs. If there is more room in the host receive buffer then multiple FCTs may be sent, one for every eight spaces in the receive buffer. Correspondingly, if multiple FCTs are received then it means that there is a corresponding amount of space available in the receiver buffer e.g. four FCTs means that there is room for 32 data characters.

Detection of Disconnect Errors: Link disconnection is detected when following reception of a Data bit no new data bit is received within a link disconnect timeout window (850 nsec). Once a Disconnection error has been detected the link attempts to recover from the error (see in Figure 3.4).

Detection of Parity Errors: Parity errors occurring within a data or control character are detected when the next character is sent, since the parity bit for a data or control token is contained in the next character. Once a parity error has been detected the link will attempt to recover from the error (see in Figure 3.4).

Link Error Recovery: Following an error or reset the link attempts to re-synchronise and restart using an “exchange of silence” protocol (see Figure 3.4). The end of the link that is either reset or that finds an error ceases transmission. This is detected at the other end of the link as a link disconnects and that end stops transmitting too. The first link resets its input and output for 6.4 us to ensure that the other end will detect the disconnect. The other end will also wait for 6.4 us after ceasing transmission. Each link then waits a further 12.8 us before starting to transmit These periods of time are sufficient to ensure that the receivers at both ends of the link are ready to receive characters before either end starts transmission.The two ends of the link go through the NULL/FCT handshake to re-establish a connection and ensure proper character synchronization.

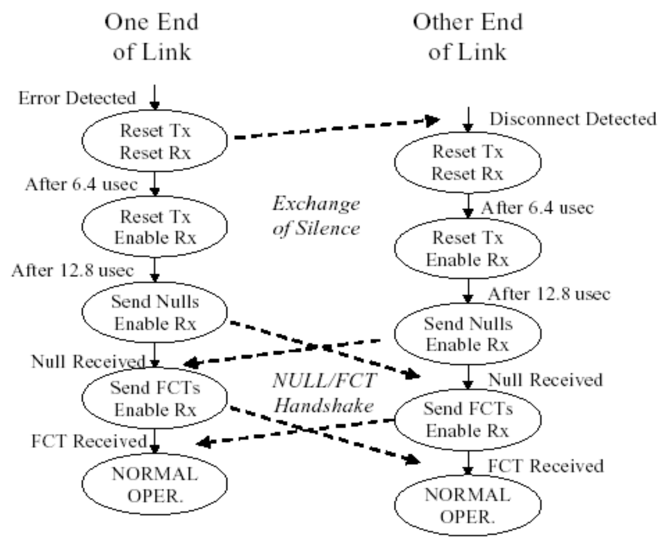


Fig 3.4 Link Restart

3.3.5 PACKET LEVEL

The packet level protocol follows the packet level protocol defined in IEEE-1355. It defines how data is encapsulated in packets for transfer from source to destination. The format of a packet is illustrated in Figure 3.5.

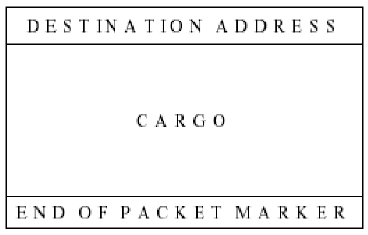


Figure 3.5 Packet Format

The “Destination Address” is a list of one or more data characters that represent the destination identity. This list of data characters represents either the identity code of the destination node or the path that the packet will take to get to the destination node. The “Cargo” is the data to be transferred from source to destination.The “End of Packet Marker” is used to indicate the end of a packet. Two ends of packet markers are defined.

- EOP Normal end_of_packet marker - indicates end of packet
- EEP Error End_of_packet marker - indicates that the packet has been terminated prematurely due to a link error. Since there is no start of packet marker, the first data character following an end_of_packet marker (either EOP or EEP) is regarded as the start of the next packet.

3.3.6 NETWORK LEVEL

The network level defines what a SpaceWire network is, describes the components that make up a SpaceWire network explains how packets are transferred across a SpaceWire network and details the manner in which the SpaceWire network recovers from errors. A SpaceWire network is made up of a number of SpaceWire nodes interconnected by SpaceWire routing switches. SpaceWire nodes are the sources and destination of packets and provide the interface to the application system(s). SpaceWire nodes may be directly connected together using SpaceWire links or they may be interconnected via SpaceWire routing switches using SpaceWire links to make the connection between node and routing switch. A SpaceWire routing switch has several links interfaces which are connected together inside the routing switch by a switch matrix which allows any link input to pass the packets that it receives on to any link output for re-transmission.

3.4 ENCODER/DECODER BLOCK DIAGRAM (INFORMATIVE)

An example block diagram of a SpaceWire Encoder/Decoder is illustrated in Figure 3.6 below.

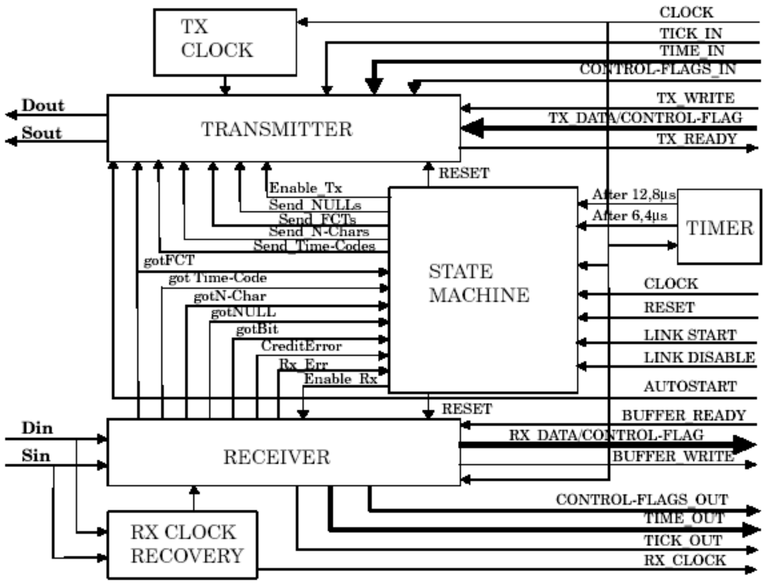


Figure 3.6 SpaceWire Link Interface Block Diagram

3.4.1 Transmitter

The Transmitter is responsible for encoding data and transmitting it using the DS encoding technique. It receives its data from the Transmit Host Interface. If there is no data to transmit the Transmitter will send Nulls. The Transmitter is only allowed to send data if the host system at the other end of the link (end B) has room in its host receive buffer. This is indicated by the link interface at end B sending an FCT, indicating that it is ready to accept other 8 data characters. The Transmitter is responsible for keeping track of the FCTs received and the number of data characters sent to avoid input buffer overflow at the other end of the link. To do this the Transmitter holds a credit count of the number of characters it has been given permission to send. The transmitter is also responsible for sending FCTs whenever the local Receiver has space for eight more data characters.

Transmit Clock

The Transmit Clock is responsible for producing the clock signals needed by the transmitter. The Transmit-clock signals are typically derived from the local system clock or from a special transmit clock circuit.

Transmit Host Interface

The Transmit Host Interface provides the interface between the Transmitter and the local system source of data. The local system writes data into the Transmit Host Interface at any time provided the host interface is ready to receive data.

3.4.2 Receiver

The Receiver is responsible for decoding the DS signals (Din and Sin) to produce a sequence of data characters that are passed on to the host system via the Receive Host Interface. It also receives NULLs, FCTs and other control characters (EOP, EEP). NULLs represent an active link. They are flagged to the exchange-level state machine but are ignored otherwise. When an FCT is received the Receiver must inform the Transmitter so that it can update its credit count accordingly. All other control characters received are flagged to the host system. The receiver will ignore any NChars, L-Chars, parity errors or escape errors until the first NULL has been received. The disconnection detection mechanism with the receiver will be enabled as soon as the first bit arrives (i.e. first transition detected on D or S inputs to receiver).

Receive Clock Recovery

The receive-clock is recovered by simply XORing the received data and strobe signals together. The Receive Clock Recovery circuit provides all the clock signals needed by the receiver.

Receive Host Interface

The Receive Host Interface provides the interface between Receiver and the local host system. As data is received by the Receiver it is written into the Receive Host Interface and passed on to the local host system. The local host system is responsible for informing the link interface whenever it is ready to receive eight more data characters from the Receive Host Interface so that the Transmitter can send an FCT to the interface at the other end of the link (see Fig 3.6).

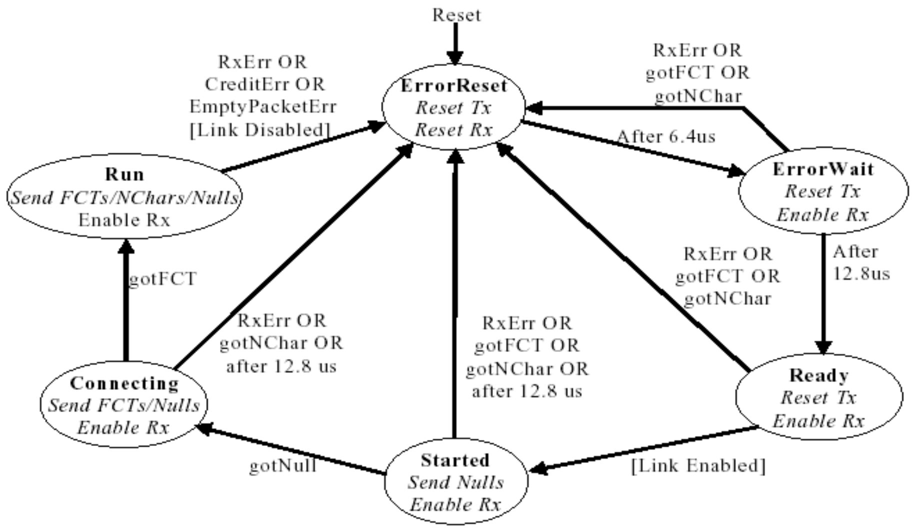
Receive Buffer Data Management

The host system is responsible for data buffer management. This makes the SpaceWire interface more versatile and eases partitioning of the error recovery mechanism across the various levels of the SpaceWire standard. Several different types of host receiver buffering may be implemented:-

- FIFO buffering – where the size of the FIFO buffer depends upon the particular application.
- Memory buffering – where direct memory access (DMA) is used to transfer data to host system memory. As soon as the DMA channel has been set up, several FCTs can be requested immediately to allow the data to be transferred as fast as possible.
- No buffering – where the host system is able to accept data at the highest rate that the link interface can provide it. In this case several FCTs can be sent initially, followed by one more every time eight normal characters are received.

3.4.3 STATE MACHINE (NORMATIVE)

The complete state transition diagram for the SpaceWire link interface is illustrated in Figure 3.7 below.



RxErr = Disconnect Error OR Parity Error OR Escape Error (ESC not followed by FCT)
Note: Disconnect Error only enabled after first bit received.
Parity Error, Escape Error,, gotFCT, gotNChar only enabled after first Null received (i.e. gotNull asserted).

Figure 3.7 State Diagram for SpaceWire Link Interface

3.5 Definition of States

In this section the states represented in figure 3.7 are described.

ErrorReset

The *ErrorReset* state shall be entered after a system reset, after link operation has been terminated for any reason or if there is an error during link initialisation. In the *ErrorReset* state the Transmitter, Receiver,Transmit Host Interface and Receive Host Interface shall all be reset. When the reset signal is de-asserted the *ErrorReset* state shall be left unconditionally after a delay of 6.4 us (nominal) and the state machine shall move to the *ErrorWait* state. Whenever the reset signal is asserted the state machine shall move immediately to the ErrorReset state and remain there until the reset signal is deasserted.

ErrorWait

The *ErrorWait* state shall be entered only from the *ErrorReset* state.In the *ErrorWait* state the Receiver shall be enabled and the Transmitter shall be held reset. This allows the Receiver to start the disconnection detection mechanism (after registering a transition on the D or S line) and to begin looking for the arrival of a NULL.If a NULL is received then the gotNULL condition shall be set. This condition will be acted upon in the *Started* state. The *ErrorWait* state shall be left unconditionally after a delay of 12.8 us (nominal) and the state machine shall move to the *Ready* state.If, while in the *ErrorWait* state, a disconnection error is detected, or if after the first NULL has been received, a parity error or escape error occurs, or any character other than a NULL is received, then the state machine shall move back to the *ErrorReset* state.The ErrorReset and ErrorWait state with their 6.4 us and 12.8 us delays ensure that the receivers at both ends of a link are enabled before either end begins transmission.

Ready

The *Ready* state shall be entered only from the *ErrorWait* state.In the *Ready* state the link interface isready to initialise as soon as it is allowed to do so.The Receiver shall be enabled and the Transmittershall be held reset.If a NULL is received then the gotNULL condition shall be set.This condition will be acted upon in the *Started* state.The state machine shall wait in the *Ready* state until the [Link Enabled] guard becomes true and then it shall move on into the *Started* state. If, while in the *Ready* state, a disconnection error is detected, or if after the first NULL has been received, a parity error or escape error occurs, or any character other than a NULL is received, then the state machine shall move to the *ErrorReset* state. In the Ready state the two receivers are enabled and the state machine is waiting for the local host system to command the link to start.

Started

The *Started* state shall be entered from the *Ready* state when the link interface is enabled. In the *Started* state the state machine begins making a connection with the link interface at the other end of the link by sending NULLs. When the Started state is entered a 12.8 us (nominal) timeout timer shall be started. In *Started* state the Receiver shall be enabled and the Transmitter shall send NULLs.If a NULL is received then the gotNULL condition shall be set.The state machine shall move to the *Connecting* state if the gotNULL condition is set. The NULL that set the gotNULL condition may have been received in the *ErrorWait*, *Ready* or *Started* states. In the *Started* state a least one NULL must be requested to be sent from the transmitter before moving to the *Connecting* state. If, while in the *Started* state, a disconnection error is detected, or if after the first NULL has been received, a parity error or escape error occurs, or any other character other than a NULL is received, then the state machine shall move to the *Error Reset* state.If the 12.8 us timeout timer expires (i.e. no NULL received since leaving the *Error Reset* state) then the state machine shall move to the *Error Reset* state. In the *Started* state the attempt to make a connection across the link is started. NULLs are transmitted and the receiver is waiting to receive a NULL.

Connecting

The *Connecting* state shall be entered from the *Started* state after a NULL has been received (gotNULL condition set). On entering the *Connecting* state a 12.8 us timeout timer shall be started. In the *Connecting* state the Receiver shall be enabled and the Transmitter shall be enabled to send FCTs and NULLs. If an FCT is received the state machine shall move to the *Run* state. If a disconnect error, parity error or escape error is detected, or if an N-Char is received while in the *Connecting* state then the state machine shall move to the *ErrorReset* state. If the 12.8 us timeout occurs then the state machine shall move to the *ErrorReset* state. The *Connecting* state is entered when the link interface (end A) has received a NULL. It now has to wait for an FCT to be received indicating that the other end of the link (end B) has also received a NULL. When the link interface has received a NULL and an FCT it means that communication is established in both directions. If an FCT fails to arrive within 12.8 us then something is wrong with the link connection so the link interface is reset once more (*ErrorReset* state) and connection is attempted once again.

Run

The *Run* state shall be entered from the *Connecting* state. In the *Run* state the Receiver is enabled and the Transmitter is enabled to send N-Chars, FCTs and NULLs. If the link interface is disabled, or if a disconnect error, parity error, escape error, credit error or empty packet error is detected , while in the *Run* state then the state machine shall move to the *Error Reset* state. The *Run* state is the state for normal operation. Link connection has been made. L-Chars and N-Chars can flow freely in both directions across the link. The link remains in the *Run* state until an error occurs or until the link is disabled.

Definition of Transitions

Reset

Reset represents power on reset, other hardware reset or software commanded reset.

After T us

After 6.4 us or after 12.8 us represents a delay of the specified time measured from when the current state is entered. The actual time intervals are nominal delays (see Fig 3.7).

[Link Enabled]

[Link Enabled] is a condition that must be met for the transition to occur (i.e. a guard). [Link Enabled] can be set true by software or hardware (see section 3.7).

gotNull

gotNull means that a NULL has been received. NULL detection is enabled whenever the Receiver is enabled. Any sequence of bits encountered prior to the first NULL being received shall be ignored. NULL detection shall include the parity bit within the NULL i.e. the parity bit that covers the ESC character within the NULL control code. The second parity bit associated with the NULL, that covers the FCT character shall not be included in the NULL detection. Hence the NULL shall be detected and gotNull asserted, when the 1110100 sequence of bits is received as illustrated in figure 3.8. If a parity error occurs with the first parity bit (for the ESC character) then the NULL will not be detected. If a parity error occurs with the second parity bit, then this error will be picked up immediately since parity error detection is enabled within the receiver after a NULL has been received.

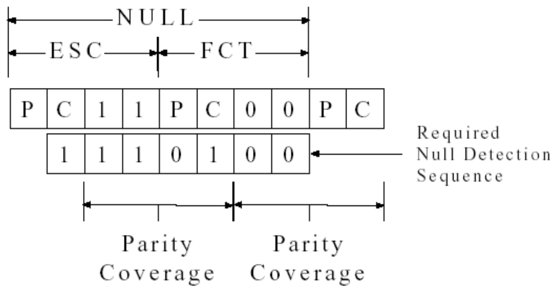


Fig 3.8 Null Detection

gotFCT

gotFCT means that an FCT has been received. FCTs are only valid when received in the *Connecting* and *Run* states. If received in any other state they represent an error.

gotNChar

gotNChar means that an N-Char has been received.An N-Char received when the exchange-level state machine is not in the *Run* state is an error.

[Link Disabled]

[Link Disabled] is a condition set by external hardware or software in order to disable and stop the link interface.

RxErr

RxErr or Receiver Error is shorthand for Disconnect Error, Parity Error or Escape Error.

Disconnect Error

Disconnect Error is an error condition asserted when the length of time since the last transition on the D or S lines was longer ago than 850 ns nominal. The disconnect detection mechanism is activated after leaving the *ErrorReset* state as soon as the first edge is detected on the D or S line.

Parity Error

The parity error event occurs if a parity error is detected. Parity detection is enabled whenever the receiver is enabled after the first NULL has been received.

Escape Error

The escape error event occurs if an ESC character is followed by any character other than an FCT (ESC followed by FCT is a NULL. Escape error detection is enabled whenever the receiver is enabled after the first NULL has been received.

Character Sequence Error

Any characters received before a NULL has been received are ignored. Once a NULL has been received an FCT received before a NULL has been sent indicates an error (i.e. FCT received in *ErrorWait*, *Ready* or *Started* state). An N-Char should only be received after both a NULL and an FCT have been received otherwise an error has occurred (i.e. N-Char can only be received in the *Run*state).Note: In the state diagram of figure 3-7, the invalid gotFCT or gotNChar events are shown explicitly, rather than as a general character sequence error event.

Credit Error

Credit error occurs if data is received when the host system is not expecting any more data, i.e. when all the N-Chars expected, according to the requested “8 more” N-Chars and subsequent transmitted FCTs, have been received. A credit error ought never to occur and indicates that some undetected error has occurred on the link affecting the transfer of FCTs.

Empty Packet Error

Empty packets are not permitted (see section 8). If the next N-Char received after an EOP or EEP is another EOP or EEP then an empty packet error has occurred. An empty packet error ought never to occur and indicated that some undetected error has occurred on the link producing a spurious EOP or EEP.

3.6 LINK INITIALISATION (INFORMATIVE)

This section explains how the state diagram given in section 3.4 handles link initialization, going from the reset of one end of a link through to the link operating normally sending data in both directions. The basic state diagram with the receiver error conditions removed is illustrated in Figure 3.8 .After a link interface (one end of a link) has been reset, it enters the *ErrorReset* state where the transmitter and receiver are reset. The transmitter reset is a controlled reset, resulting first in the transmitter stopping transmission followed by resetting of the strobe signal and then the data signal. This sequence avoids the simultaneous transition of both data and strobe signals. The link interface will remain in the *ErrorReset* state for approximately 6.4 us nd then move to the *ErrorWait* state. In the *ErrorWait* state the transmitter remains disabled, but the receiver is enabled so that it can begin searching for NULLs. The link interface remains in the *ErrorWait* state for 12.8 us and then moves into the *Ready* state.The 6.4 us from *ErrorReset* to *ErrorWait* and the 12.8 us delay from *ErrorWait* to *Ready* make sure that the receivers at both ends of a link are ready to receive characters before either end starts transmission. The link interface may be enabled in many possible ways, for example, by software command, automatically when the receiver detects a NULL, or the link may be permanently enabled. When a link interface is enabled the [Link Enabled] condition becomes true. The link interface will move from the *Ready* state to the *Started* state as soon as the link is enabled. In the *Started* state the link interface instructs the transmitter to start sending NULLs. It will remain in this state until the receiver detects that a NULL has been received over the link or until a connection timeout has expired. The connection timeout is set to a nominal 12.8 us since this period has to be generated for the *ErrorReset* state timeout. If a NULL is received then the link interface will move tothe *Connecting* state. If no NULL is received within 12.8 us it will move to the *ErrorReset* state. In the latter case the link interface will go through the reset sequence (*ErrorReset, ErrorWait, Ready*) and attempt to make a connection again a short time later. In the *Connecting* state the link interface will send some FCTs (and NULLs) and will wait for an FCT to be received. If an FCT is received the link interface will move on to the *Run* state. If an FCT has not been received within 12.8 us then link connection has not been made properly, so the link interface moves back to the *ErrorReset* state. The link interface will then go through the reset sequence (*ErrorReset, ErrorWait, Ready*) and attempt to make a connection again a short time later. When the link enters the *Run* state it starts normal operation, sending and receiving data and control characters. It remains in the *Run* state until the link is disabled. The link interface then moves through the reset sequence (*ErrorReset, ErrorWait, Ready*) and stays in the ready state until the link is enabled once more. A link can only send FCTs once it has received a NULL. So, when a link has received an FCT it knows that the link is connected in both directions.NULL correlation in the *ErrorWait*, *Ready* and *Started* states ensures proper character synchronization. The NULL/FCT handshake sequence ensures that the link is connected in both directions before normal link operation begins.The time taken from a link being enabled in the *Starting* state to normal operation in the *Run* state can be as little as the time taken to transfer two NULLs and an FCT. End A is enabled and sends a NULL. End B is autostart enabled when it receives the NULL from end A and sends a NULL followed by an FCT. End A receives the NULL from end B and sends an FCT.Both ends receive FCTs and move to the *Run* state. At a link data signaling rate of 10 Mbps this could take just 2 us.

NORMAL OPERATION (INFORMATIVE)

In normal operation both ends of the link are in the *Run* state and will be sending and receiving NChars, FCTs and NULLs. Consider a host system with buffer space sufficient to hold 16 normal-characters. This host system at one end of a link (end A) will indicate that it is ready to receive normal-characters by twice flagging that it has room for 8 more characters to the link interface.The link interface will send two FCTs to the other end of the link (end B) which will increment its credit count accordingly (from zero to 16). The link interface at end B indicates to its host system that it is ready to transmit data (normal-characters) when the host system at end B has data to transfer, it will pass it to the link interface, which will send it across the link to end A.As each character is transmitted by the link interface (end B) it will decrement its credit count until it reaches zero, at which point the link interface (end B) will indicate to its host system that it is not ready to transfer any more data.The data received at end A will be passed on to its host system which will place it in its 16 character buffer. As the host system uses the data out of this buffer it makes space for more data to be received. As soon as there is space for another 8 more characters it flags this to the link interface, which will then send out another FCT informing end B that 8 more normal characters may be sent.

ERROR DETECTION (NORMATIVE)

There are six forms of receiver error that can be detected and acted upon at the exchange level –disconnect errors, parity errors, escape errors, credit errors, character sequence errors and empty packet errors. Whenever one of these errors occurs both characters synchronisation and flow-control status cease to be valid. Both ends of the link must be reset and re-initialized to recover character synchronisation and flow control status. An error can occur in the transmitter if it is given an invalid character to transmit. In this event the transmitter shall ignore the invalid character, cease N-Char transmission and report the error to the network level.

Disconnect Error

An operational link interface sends normal-characters, FCTs or NULLs continuously, thus the data and/or strobe signals are always changing. The receiver shall detect a disconnection when the time interval from the last transition on either the data or strobe signal exceeds the disconnect-detection time. The disconnect-detection time shall be 850 nsec nominal. Before being able to detect a disconnect error the receiver must have received at least one bit. A disconnect error can either be caused when one end of the link is disabled or when the link is physically disconnected (intentionally or unintentionally). If a physical disconnection is the cause of the disconnect error then both ends of the link will try repeatedly to make a connection until the link is reconnected or until the link interfaces are disabled. If a disconnect error is detected then the link interface shall follow the exchange of silence error recovery procedure. If the disconnect error occurs in the *Run* state then the disconnect error shall be flagged up to the network level as a link error.

Parity Error

When a parity bit is received it shall be checked. If a parity error occurs after the first NULL has been received, then the link interface shall follow the error recovery procedure. If the parity error occurs in the *Run* state then the parity error shall be flagged up to the network level as a link error.

Escape Error

An ESC character shall only be used to form the NULL (ESC followed by FCT, see Fig. 3.8). If a ESC character is received followed by any character other than an FCT then the link interface shall follow the error recovery procedure. If the escape error occurs in the *Run* state then the escape error shall be flagged up to the network level as a link error.

Credit Error

In the *Run* state if a normal character is received when the host system is not expecting any N-Chars then a credit error has occurred.A credit error may be caused if an error occurs undetected by the parity bit (e.g. two bits in error) which results in one or more spurious FCTs. In the event of a credit error the link interface shall follow the error recovery procedure described .If the credit error occurs in the *Run* state then the credit error shall be flagged up to the network level as a link error.

Character Sequence Error

During initialization it is possible for a link interface to receive FCTs or normal-characters when they are not expected. Any unexpected characters are caught by the exchange-level state machine resulting the link being reset and re-initialized (see figure 3.8). A character sequence error shall not be flagged up to the network level as a link error because it can only occur during link initialization.

Empty Packet Error

An EOP or EEP followed immediately by another EOP or EEP represents an empty packet, which is not permitted. In the *Run* state, if the next N-Char received after an EOP or EEP has been received is another EOP or EEP, then there has been an error on the link. If the empty packet error occurs in the *Run* state then the empty packet error shall be flagged up to the network level as a link error.

Exchange of Silence Error Recovery Procedure

When one end of the link (end A) is disabled or detects an error, it will cease transmission. This will cause a disconnect error at the other end of the link (end B). End B will then cease transmission resulting in a disconnect error at end A. This procedure is known as an “exchange of silence”.Both ends of the link will cycle through the reset sequence (*ErrorReset, ErrorWait, Ready*) ending up in the *Ready* state ready to begin operation once enabled. If both ends are enabled then they will move to the *Started* state and re-initialise. If one end (end A) is disabled and the other end (end B) is enabled then end B will move from the *Ready* state to the *Started* state and will send NULLs for 12.8 us. Since end A is disabled it cannot respond. End A will, however, have started its disconnect timer and will also have registered that a NULL has been received. When end

B completes the 12.8 us timeout it will move to the *ErrorReset* state and disconnect (stop its output). End A is able to detect the disconnection so will also move to the *ErrorReset* state. Both ends will once again move through the reset sequence. This series of events will continue until either end A is enabled or end B is disabled.

Link Error

During initialization, receiver errors (disconnect error, parity error, escape sequence error, character sequence error, credit error and empty packet error) are likely to occur and are part of the natural initialization sequence. These errors shall not be reported to the network level when they occur during link initialization (*ErrorReady*, *ErrorWait*, *Ready*, *Started* and *Connecting* states). Once a link connection has been established (*Run* state) then a receiver error represents a failure of the link connection and must be reported to the network level so that appropriate action for error recovery and/or reporting can be taken. A link error is reported to the network level whenever any of the following errors occur while a link interface is in the *Run* state: disconnect error, parity error, escape sequence error, credit error, empty packet error. Note the exclusion of character sequence error from this list. A character sequence error is only possible during initialisation.

EXCEPTION CONDITIONS (INFORMATIVE)

Several exception conditions have been identified where things, for one reason or another, do not follow the usual sequence of events. These exceptions are considered in this section.

Disconnect error while waiting to start

"Waiting to start" means that a link interface is in either the *ErrorReset*, *ErrorWait*, *Ready* or possibly the *Started* state. For a disconnect error to be detected while waiting to start, the other end of the link (end B say) must have sent at least one bit, so that the disconnect detect mechanism at end A can be activated. End B must have then given up waiting for end A to send a NULL and moved to the *ErrorReset* state and stopped its transmitter – thus causing the disconnect. An alternative possibility is that the link became physically disconnected. The following tables illustrate the various sequences of events starting from when end B has just moved to the *ErrorReset* state. If a physical disconnection has occurred then both ends of the link will continue to try to make a connection, cy0+cling around the reset sequence, until they are disabled or until the connection is reestablished.

Link connected in one direction but not in the other

A link may be connected in one direction and not in the other while a link is in the process of being plugged in (contact bounce time may be significantly larger than tens of us) or if there is a break in the link cable. In this case the sequence of events listed in the table below will be followed. Consider for convenience that both links are in the *started* state and that end A is connected to end B, but end B is not connected to end A.

Chapter 4

SpaceWire Protocol Simulation

SpaceWire protocol composed of various components .The components is as below

TX : transmitter.
RX : receptor.
TX fifo : first in first out buffer for the transmission.
RX fifo: first in first out buffer for the reception.
Two domain clock : interface the signal between different clock.
Time-id buffer : buffer for the time id seeded.
FSM: state machine manage IP.

During simulation, I have applied various tests and got result described below

4.1 SpaceWire in loop back mode

In this mode transmitter txd and receiver rxd shorted. The simulation result is in Figure 4.1.

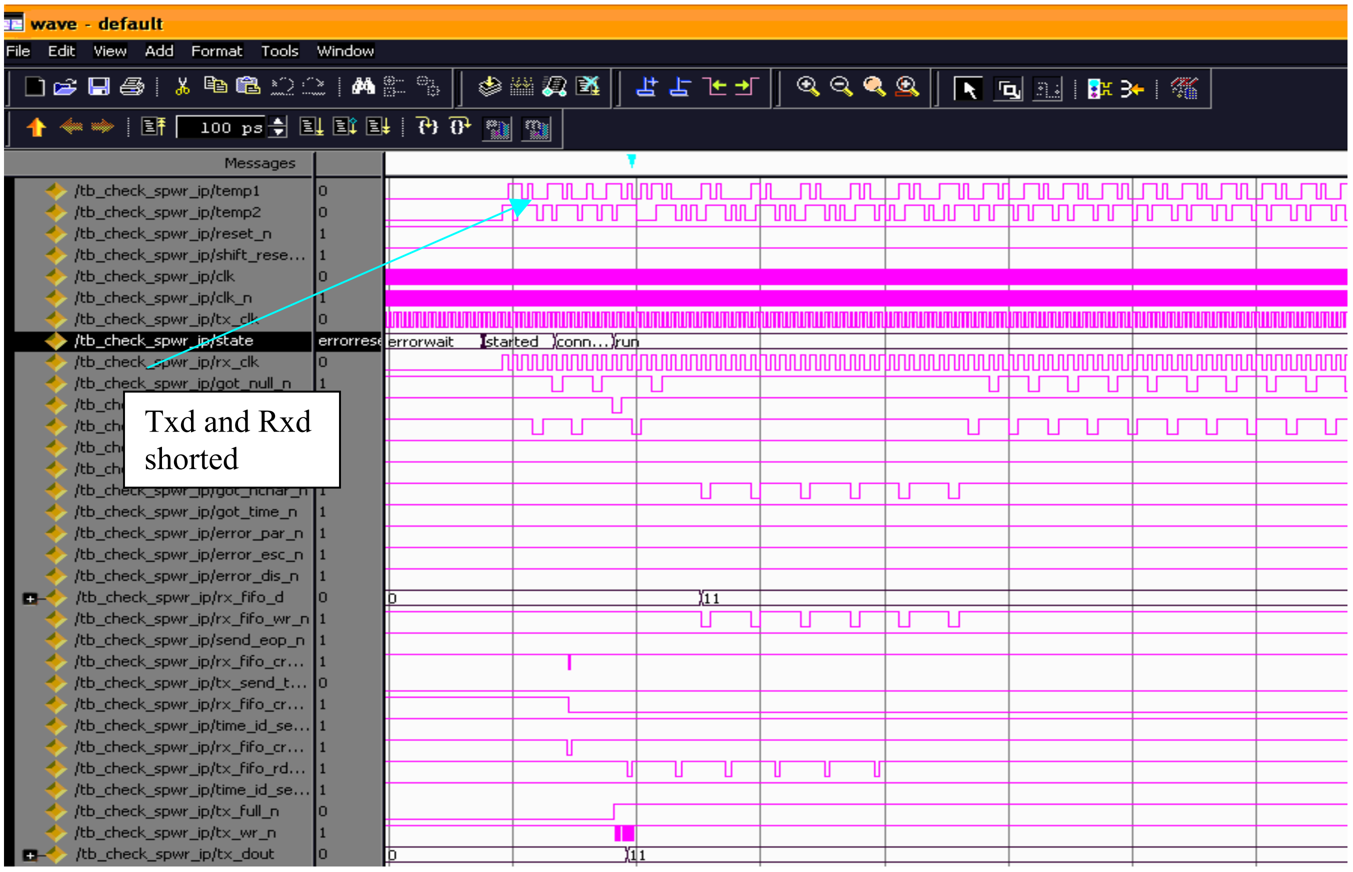
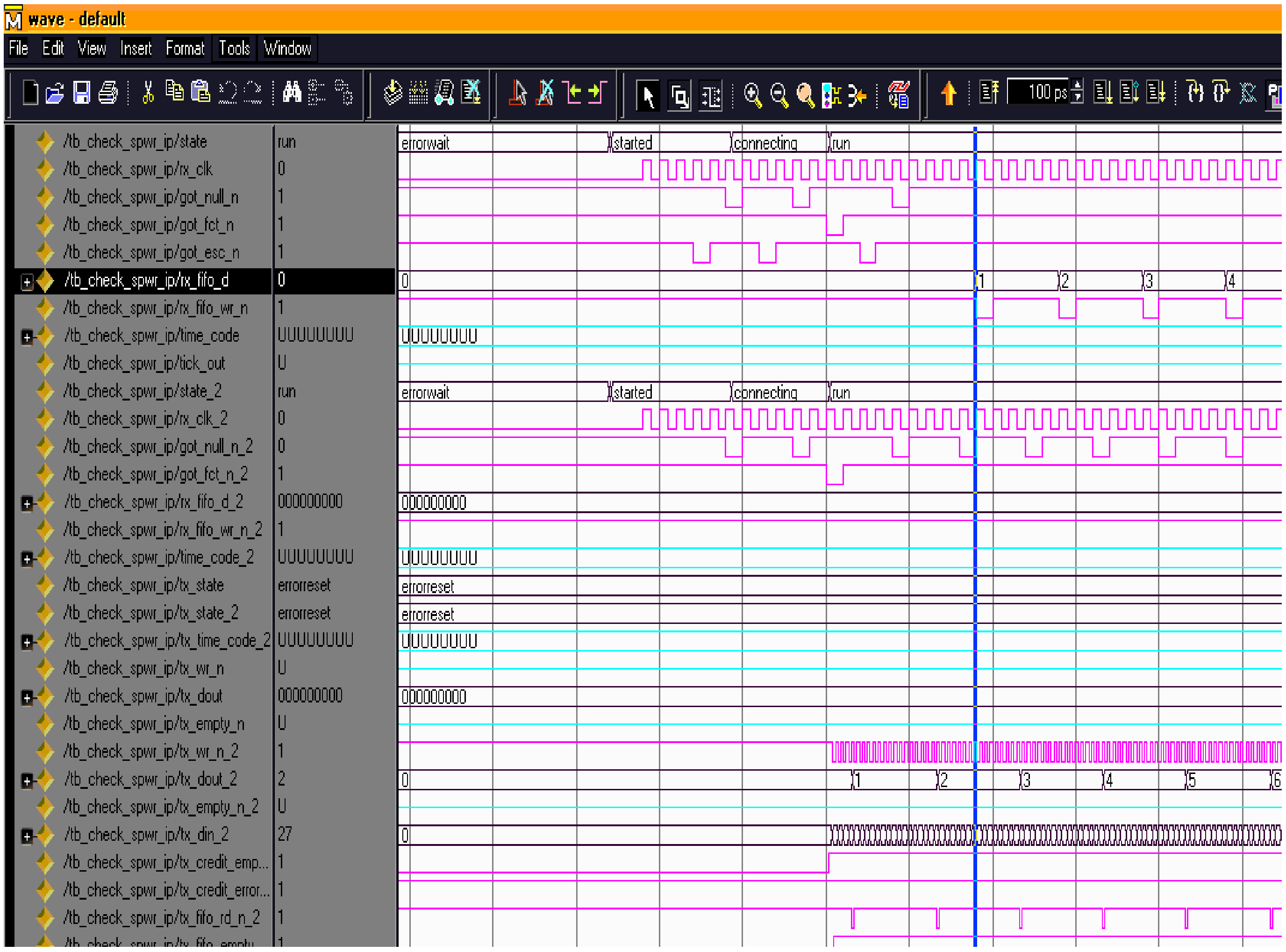

wave - default
File Edit View Add Format Tools Window
100 ps
Messages
/tb_check_spwr_ip/temp1 0
/tb_check_spwr_ip/temp2 0
/tb_check_spwr_ip/reset_n 1
/tb_check_spwr_ip/shift_rese... 1
/tb_check_spwr_ip/clk 0
/tb_check_spwr_ip/clk_n 1
/tb_check_spwr_ip/tx_clk 0
/tb_check_spwr_ip/state errorrese
errorwait started conn... run
/tb_check_spwr_ip/rx_clk 0
/tb_check_spwr_ip/got_null_n 1
/tb_check_spwr_ip/got_fchar_n 1
/tb_check_spwr_ip/got_time_n 1
/tb_check_spwr_ip/error_par_n 1
/tb_check_spwr_ip/error_esc_n 1
/tb_check_spwr_ip/error_dis_n 1
/tb_check_spwr_ip/rx_fifo_d 0
0 11
/tb_check_spwr_ip/rx_fifo_wr_n 1
/tb_check_spwr_ip/send_eop_n 1
/tb_check_spwr_ip/rx_fifo_cr... 1
/tb_check_spwr_ip/tx_send_t... 0
/tb_check_spwr_ip/rx_fifo_cr... 1
/tb_check_spwr_ip/time_id_se... 1
/tb_check_spwr_ip/rx_fifo_cr... 1
/tb_check_spwr_ip/tx_fifo_rd... 1
/tb_check_spwr_ip/time_id_se... 1
/tb_check_spwr_ip/tx_full_n 0
/tb_check_spwr_ip/tx_wr_n 1
/tb_check_spwr_ip/tx_dout 0
0 11
Txd and Rxd shorted

Figure-4.1 SpaceWire in loop back mode

4.2 SpaceWire IP1 to IP2 data transfer

I have taken two SpaceWire protocols says IP1 and IP2.The Transmitter of IP1 sends data to SpaceWire Link and this data receives at the receiver of the IP2.The simulation result is in Figure 4.2.

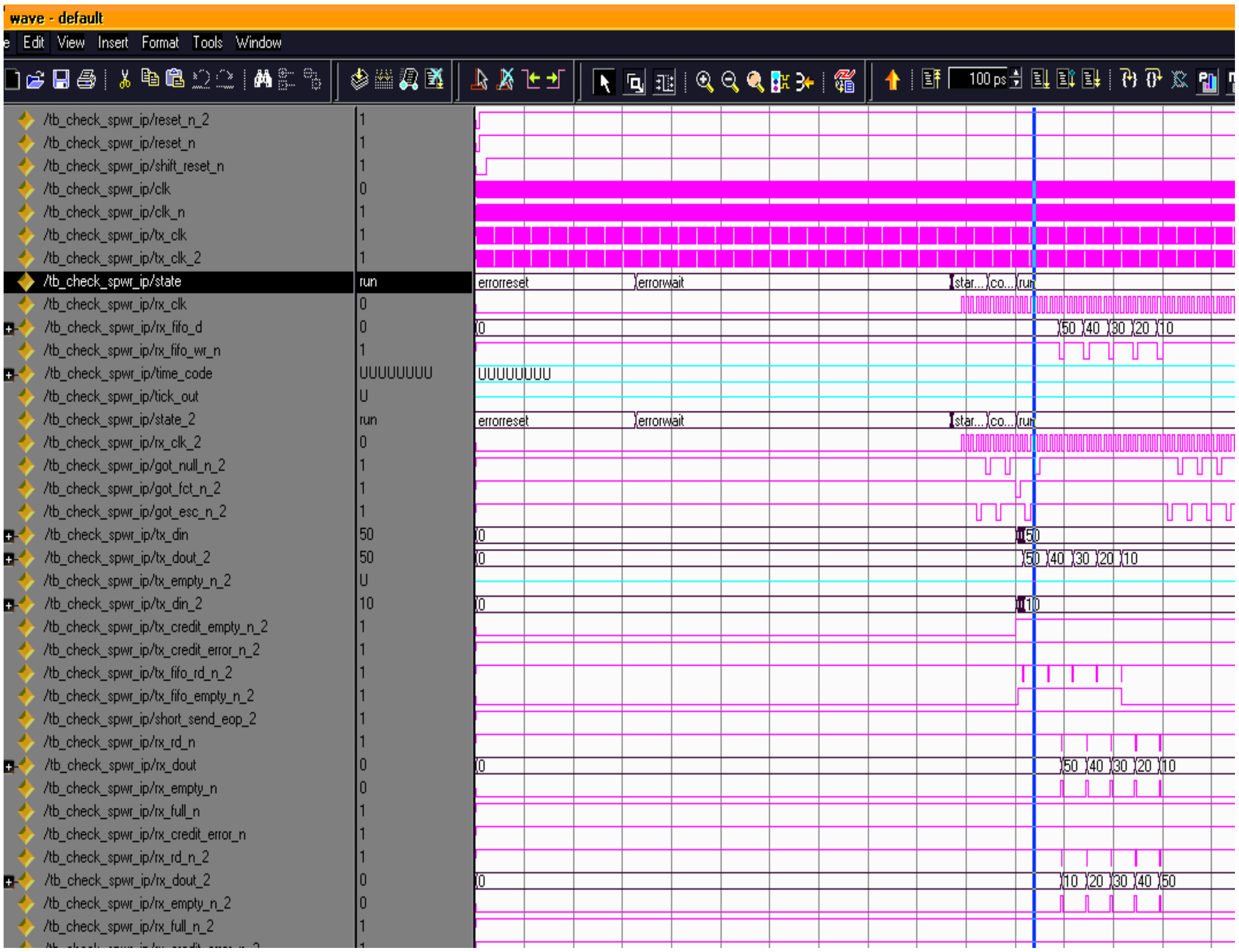


```
/tb_check_spwr_ip/state            run
/tb_check_spwr_ip/rx_clk           0
/tb_check_spwr_ip/got_null_n       1
/tb_check_spwr_ip/got_fct_n        1
/tb_check_spwr_ip/got_esc_n        1
/tb_check_spwr_ip/rx_fifo_d        0
/tb_check_spwr_ip/rx_fifo_wr_n     1
/tb_check_spwr_ip/time_code        UUUUUUUU
/tb_check_spwr_ip/tick_out         U
/tb_check_spwr_ip/state_2          run
/tb_check_spwr_ip/rx_clk_2         0
/tb_check_spwr_ip/got_null_n_2     0
/tb_check_spwr_ip/got_fct_n_2      1
/tb_check_spwr_ip/rx_fifo_d_2      000000000
/tb_check_spwr_ip/rx_fifo_wr_n_2   1
/tb_check_spwr_ip/time_code_2      UUUUUUUU
/tb_check_spwr_ip/tx_state         erroreset
/tb_check_spwr_ip/tx_state_2       erroreset
/tb_check_spwr_ip/tx_time_code_2   UUUUUUUU
/tb_check_spwr_ip/tx_wr_n          U
/tb_check_spwr_ip/tx_dout          000000000
/tb_check_spwr_ip/tx_empty_n       U
/tb_check_spwr_ip/tx_wr_n_2        1
/tb_check_spwr_ip/tx_dout_2        2
/tb_check_spwr_ip/tx_empty_n_2     U
/tb_check_spwr_ip/tx_din_2         27
/tb_check_spwr_ip/tx_credit_emp... 1
/tb_check_spwr_ip/tx_credit_error... 1
/tb_check_spwr_ip/tx_fifo_rd_n_2   1
/tb_check_spwr_ip/tx_fifo_empty    1
```

Figure-4.2 IP1 to IP2 data transfer

4.3 SpaceWire IP1 to IP2 data transfer in duplex mode:

In duplex mode IP1 send data {10,20,30,40,50} to IP2 and IP2 send data {50,40,30,20,10} to IP1.



```
/tb_check_spwr_ip/reset_n_2            1
/tb_check_spwr_ip/reset_n              1
/tb_check_spwr_ip/shift_reset_n        1
/tb_check_spwr_ip/clk                  0
/tb_check_spwr_ip/clk_n                1
/tb_check_spwr_ip/tx_clk               1
/tb_check_spwr_ip/tx_clk_2             1
/tb_check_spwr_ip/state                run
/tb_check_spwr_ip/rx_clk               0
/tb_check_spwr_ip/rx_fifo_d            0
/tb_check_spwr_ip/rx_fifo_wr_n         1
/tb_check_spwr_ip/time_code            UUUUUUUU
/tb_check_spwr_ip/tick_out             U
/tb_check_spwr_ip/state_2              run
/tb_check_spwr_ip/rx_clk_2             0
/tb_check_spwr_ip/got_null_n_2         1
/tb_check_spwr_ip/got_fct_n_2          1
/tb_check_spwr_ip/got_esc_n_2          1
/tb_check_spwr_ip/tx_din               50
/tb_check_spwr_ip/tx_dout_2            50
/tb_check_spwr_ip/tx_empty_n_2         U
/tb_check_spwr_ip/tx_din_2             10
/tb_check_spwr_ip/tx_credit_empty_n_2  1
/tb_check_spwr_ip/tx_credit_error_n_2  1
/tb_check_spwr_ip/tx_fifo_rd_n_2       1
/tb_check_spwr_ip/tx_fifo_empty_n_2    1
/tb_check_spwr_ip/short_send_eop_2     1
/tb_check_spwr_ip/rx_rd_n              1
/tb_check_spwr_ip/rx_dout              0
/tb_check_spwr_ip/rx_empty_n           0
/tb_check_spwr_ip/rx_full_n            1
/tb_check_spwr_ip/rx_credit_error_n    1
/tb_check_spwr_ip/rx_rd_n_2            1
/tb_check_spwr_ip/rx_dout_2            0
/tb_check_spwr_ip/rx_empty_n_2         0
/tb_check_spwr_ip/rx_full_n_2          1
```

Figure 4.3 Data Transfer in duplex mode

Chapter 5

LEON3 Processor: An Embedded Core

5.1 Introduction

LEON3 is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture. It is designed for embedded applications, combining high performance with low complexity and low power consumption. The LEON3 processor is a synthesizable VHDL model. To enable the development of SoC devices using the LEON core, the full source code is freely available. LEON was initially developed by Jiri Gailser while working for the **E**uropean **S**pace **A**gency (ESA) and Gailser Research is now maintaining and further enhancing the model.

New modules can easily be added using the on-chip AMBA AHB/APB buses. The VHDL model is fully synthesizable with most synthesis tools and can be implemented On both FPGAs and Asics. Salient features of LEON are given below. A block diagram of the LEON3 core can be seen below:

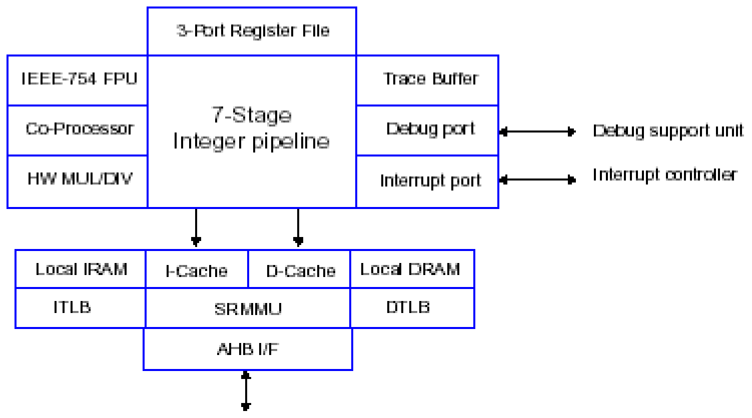


Fig 5.1 LEON3 processor core block diagram

5.2 Salient features of LEON are given below

- Integer Unit
- Floating-point and co-processor
- Cache sub-system with separate instruction and data caches
- Debug support unit
- Flexible Memory interface and controller
- Timers
- Watchdog
- UARTs
- Interrupt controller
- Parallel I/O port
- AMBA on-chip buses
- Boot loader
- Watch point registers

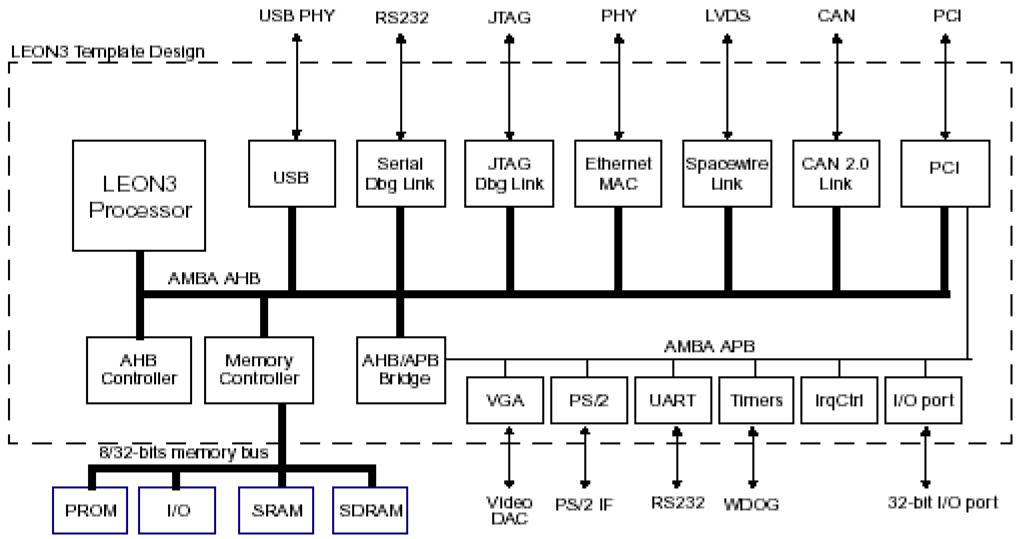


Fig 5.2 LEON3 based SoC

5.3 Compliance with SPARC

The LEON IU (Integer Unit) implements the SPARC instructions as defined by the SPARC Architecture Manual. **S**calable **P**rocessor **Arc**hitecture is a CPU instruction set architecture (ISA), derived from a reduced instruction set computer (RISC) lineage. SPARC was designed as a target for optimizing compilers and easily pipelined hardware implementations. SPARC implementations provide exceptionally high execution rates and short time-to market development schedules.It is a model which specifies unambiguously the behavior observed by software on SPARC systems. Therefore, it does not necessarily describe the operation of the hardware in any actual implementation. Any implementation is not required to execute every instruction in hardware. An attempt to execute a SPARC instruction that is not implemented in hardware generates a trap.If the unimplemented instruction is non-privileged, then it must be possible to emulate it in software.If it is a privileged instruction, whether it is emulated by software is implementation-dependent.

5.3.1 SPARC System Components

The architecture allows for a spectrum of input/output (I/O), memory management unit (MMU), and cache system sub-architectures. SPARC assumes that these elements are Optimally defined by the specific requirements of particular systems. They are invisible to nearly all user application programs and the interfaces to them can be limited to localized modules in an associated operating system.

Reference MMU

The SPARC ISA does not mandate that a single MMU design be used for all system Implementations. Rather, designers are free to use the MMU that is most appropriate for their application or no MMU at all, if they wish. The memory bus in LEON provides a direct interface to PROM, memory mapped I/O devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM). Chip-select decoding is done for two PROM banks, one I/O bank, five SRAM banks and two SDRAM banks.

Supervisor Software

SPARC does not assume all implementations must execute identical supervisor software.

Thus, certain supervisor-visible traits of an implementation can be tailored to the requirements of the system. For example, SPARC allows for implementations with different instruction concurrency and different exception trap hardware.

Register File

A large windowed register file — at any one instant, a program sees 8 global integers registers plus a 24 – register window into a larger register file. The Windowed registers can be described as a cache of procedure arguments, local values, and return addresses. A separate floating-point register file configurable by software into 32 single-precision (32-bit), 16 double-precision (64-bit), 8 quad-precision registers (128-bit), or a mixture ther eof. LEON has 8 register windows by default and can be configurable.

5.4 Bus

The open standard from ARM, **A**dvanced **M**icroprocessor **B**us **A**rchitecture (AMBA) [6] Was implemented within LEON. The AMBA specification defines an on-chip communications standard for designing high-performance embedded microcontrollers. Three distinct buses are defined within theAMBA specification: Advanced High-performance Bus (AHB), Advanced System Bus (ASB), Advanced Peripheral Bus (APB).

5.4.1 AHB

AHB is intended to address the requirements of high-performance synthesizable designs. It is a high-performance system bus that supports multiple bus masters and provides high-bandwidth operation. AMBA AHB implements the features required for high-performance, high clock frequency systems including burst transfers, split transactions, single-cycle bus master handover, single-clock edge operation, wider data bus configurations (64/128 bits). An AMBA AHB design may contain one or more bus masters, typically a system would contain at least the processor and test interface. The external memory interface, APB bridge and any internal memory are the most common AHB slaves. Any other peripheral in the system could also be included as an AHB slave. However, low bandwidth peripherals typically reside on the APB. A typical AMBA AHB system design contains the following components:

AHB master A bus master is able to initiate read and write operations by providing

An address and control information. Only one bus master is allowed to actively use the bus at any one time.

AHB slave A bus slave responds to a read or writes operation within a given address space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.

AHB arbiter The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as highest priority or fair access can be implemented depending on the application requirements.

AHB decoder The AHB decoder is used to decode the address of each transfer and Provide a select signal for the slave that is involved in the transfer. A single centralized decoder is required in all AHB implementations.

5.4.2 APB

The Advanced Peripheral Bus (APB) is part of the Advanced Microcontroller Bus Architecture (AMBA) hierarchy of buses and is optimized for minimal power consumption and reduced interface complexity. The AMBA APB is used to interface to any peripherals which are low-bandwidth and do not require the high performance of a Pipelined bus interface.

APB Bridge The APB bridge is the only bus master on the AMBA APB. In addition, the APB bridge is also a slave on the higher-level system bus. The bridge unit converts system bus transfers into APB transfers and performs the following functions – latches the address and holds it valid throughout the transfer, decodes the address and generates a peripheral select (only one select signal can be active during a transfer), drives the data onto the APB for a write transfer, drives the APB data onto the system bus for a read transfer, generates a timing strobe for the transfer. **APB slave** APB slaves have a simple, yet flexible, interface description. The select signal, the address and the write signal can be combined to determine which register should be updated by the write operation. For read transfers the data can be driven on to the data bus when write signal is low and both select and enable are high. Address is used to determine which register should be read.

5.4.3 LEON's AMBA Bus

LEON implements AHB and APB bus. The processors sit as masters over the AHB bus. The memory controller, APB bridge, DSU and PCI initiator are the AHB slaves.APB bridge is the only master on the APB bus. The memory controller, AHB status register, cache controller, write protection register, configuration register, timers, UART1 and UART2, interrupt controller, I/O port, 2nd interrupt controller, DSU UART and PCI arbiter are the APB slaves.

5.5 Memory

The caches, register windows and on-chip registers are mapped to the Block RAM inside the FPGA

5.5.1 External Memory access

The memory bus provides a direct interface to PROM, memory mapped I/O devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM). The external memory bus is controlled by a programmable memory controller. The controller acts as a slave on the AHB bus. The function of the memory controller is programmed through memory configuration registers 1, 2 & 3 (MCR1, MCR2 & MCR3) through the APB bus. The memory bus supports four types of devices: PROM, SRAM, SDRAM and local I/O.

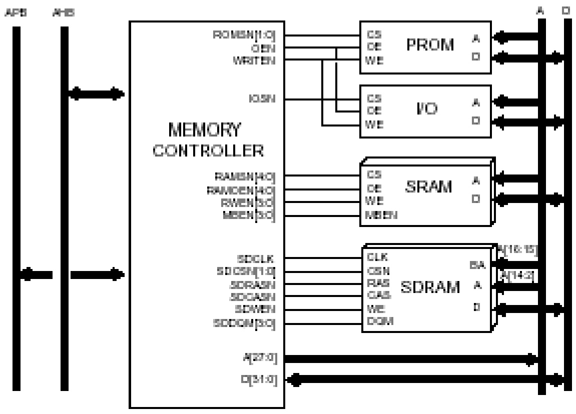


Fig 5.3 PROM/IO/SRAM/SDRAM Memory controller

5.5.2 Cache sub-system

The LEON processor implements a Harvard Architecture with separate instruction and data buses, connected to two independent cache controllers. The LEON instruction/data is a direct-mapped cache configurable to 1 – 64 kbyte. The instruction/data cache is divided into cache lines with 8 –32 bytes of data. Each line has a cache tag associated with it consisting of a tag field and one valid bit for each 4-byte sub-block. On an instruction/data cache miss to a cacheable location, the instruction/data is fetched and the corresponding tag and data line updated.

5.6 Debug Support Unit

5.6.1 DSU

The (optional) debug support unit (DSU) allows non-intrusive debugging on target hardware. The DSU allows a user to insert instruction and data watch-points, and access to all on-chip registers from a remote debugger. A trace buffer is provided to trace the executed instruction flow and/ or AHB bus traffic. The DSU has no impact on performance and has low area complexity. Communication to an outside debugger (e.g. gdb) is done using a dedicated UART (RS232).

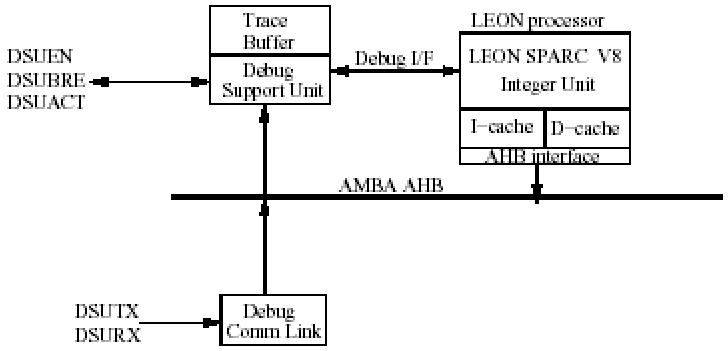


Fig. 5.4: Debug Support Unit and communication link

The debug support unit as shown in Fig 5.4 is used to control the trace buffer and the processor debug mode. The DSU is attached to the AHB bus as slave, occupying a 2-Mbyte address space. Through this address space, any AHB master can access the processor registers and the contents of the trace buffer. The DSU control registers can be accessed at any time, while the processor registers and caches can only be accessed when the processor has entered debug mode. The trace buffer can be accessed only when tracing is disabled/completed. In debug mode, the processor pipeline is held and the processor is controlled by the DSU. Entering the debug mode can occur on the following events:

- executing a breakpoint instruction
- integer unit hardware breakpoint/watchpoint hit
- rising edge of the external break signal (DSUBRE)
- setting the break-now (BN) bit in the DSU control register
- a trap that would cause the processor to enter error mode
- occurrence of any, or a selection of traps as defined in the DSU control register
- after a single-step operation
- DSU breakpoint hit

The debug mode can only be entered when the debug support unit is enabled
through an external pin (DSUEN). When the debug mode is entered, the following
actions are taken:

- PC and nPC(next PC) are saved in temporary registers (accessible by the debug unit)
- an output signal (DSUACT) is asserted to indicate the debug state
- the timer unit is (optionally) stopped to freeze the LEON timers and watchdog

The insruction that caused the processor to enter debug mode is not executed, and the processor state is kept unmodified. Execution is resumed by clearing the BN bit in the DSU control register or by de-asserting DSUEN. The timer unit will be re-enabled and execution will continue from the saved PC and nPC. Debug mode can also be entered after the processor has entered error mode, for instance when an applicationhas terminated and halted the processor. The error mode can be reset and the processor restarted at any address.

5.6.2 Trace buffer

The trace buffer consists of a circular buffer that stores executed instructions or AHB data transfers. A 30-bit counter is also provided and stored in the trace as time tag. The trace buffer operation is controlled through the DSU control register and the Trace buffer control register. When the processor enters debug mode, tracing is suspended. The size of the trace buffer is by default 128 words (2 Kbytes), but can be configured to any size through the VHDL model configuration record.

5.6.3 DSU Monitor

DSUMON is a debug monitor for the LEON processor debug support unit. It
includes the following functions:

- Read/write access to all LEON registers and memory
- Built-in dis-assembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Remote connection to GNU debugger (gdb)
- Auto-probing and initialization of LEON peripherals and memory settings

DSUMON can operate in two modes: stand-alone and attached to gdb. In standalone mode, LEON applications can be loaded and debugged using a command line interface. A number of commands are available to examine data, insert breakpoints and advance execution. When attached to gdb, DSUMON acts as a remote gdb target, and applications are loaded and debugged through gdb (or a gdb front-end such as ddd). The LEON DSU uses a dedicated UART to communicate with an outside monitor. The UART uses automatic baud-rate detection. To successfully attach DSUMON, first a serial cable between the target board and the host system is attached. Then it is powered on and the target board is reset, and finally the DSUMON software is started bythe user. The DSUEN signal on the LEON processor has to be asserted for the DSU tooperate. The DSUEN can be hardwired to '1' before synthesis(DSU always enabled)or can be set through a switch(DSU can be optionally enabled if needed).When DSUMON first connects to the target, a check is made to see if the system has been initialized with respect to memory, UART and timer settings. If no initialization has been made (debug mode entered directly after reset), the system first has to be initialized before any application can run. This is performed automatically by probing for available memory banks, and detecting the system frequency.

5.7 Software Considerations

LECCS(LEON/ERC32 Cross Compilation System) is a GNU-based free C/C++ cross-compilation system for both ERC32 and LEON processors. The following componentsare included –

- GNU C/C++ compiler
- Linker, assembler, archiver etc
- Standalone C-library
- RTEMS real-time kernel
- Boot-prom utility
- GNU debugger with Tk front-end
- graphical user interface for gdb
- Remote target monitor
- DSU monitor

LECCS allows cross-compilation of single or multi-threaded C and C++ applications for both LEON and ERC32. Using the gdb debugger, it is possible to perform sourcelevel symbolic debugging, either on a simulator or using real target hardware. GaislerResearch also provides TSIM, a high-performance LEON simulator which seamlessly can be attached to gdb and emulate a LEON system at more than 10 MIPS.RTEMS (**R**eal-**T**ime **E**xecutive for **M**ultiprocessor **S**ystems) , is a real-time executive (kernel) ported to the LEON architecture. As of now, it is the only OS that is ported over the LEON. It has also been ported to the following processor families – Intel i80386 and above, Intel i80960, Motorola MC68xxx, Motorola MC683xx, MIPS PowerPC, SPARC, Hewlett Packard PA-RISC, Hitachi SH, AMD A29K. It provides a high performance environment for embedded applications including the following features:

- multitasking capabilities
- homogeneous and heterogeneous multiprocessor systems
- event-driven, priority-based, preemptive scheduling
- optional rate monotonic scheduling
- intertask communication and synchronization

- priority inheritance
- responsive interrupt management
- dynamic memory allocation
- high level of user configurability

Chapter 6

Implementation

6.1 Introduction

This chapter will guide how to implement a leon3 based SoC design, and how to download and run software on the target system.

6.2 Overview

Implementing a leon3 system is typically done using one of the template designs on the GRLIB designs directory. Configuration of the design is done using xconfig.

- Simulation of design and test bench
- Synthesis and place route

The template design is leon3-avnet-eval-xc4vlx60, and is based on three files:

- config.vhd*:* a VHDL package containing design configuration parameters. Automatically generated by the xconfig GUI tool.
- leon3mp.vhd *:* contains the top level entity and instantiates all on-chip IP cores. It uses config.vhd to con-figure the instantiated IP cores.
- Testbench.vhd*:* test bench with external memory, emulating the leon3-avnet-eval-xc4vlx60 board.

Each core in the template design is configurable using VHDL generics. The value of these generics is assigned from the constants declared in config.vhd, created with the xconfig GUI tool.

6.3 Configuration

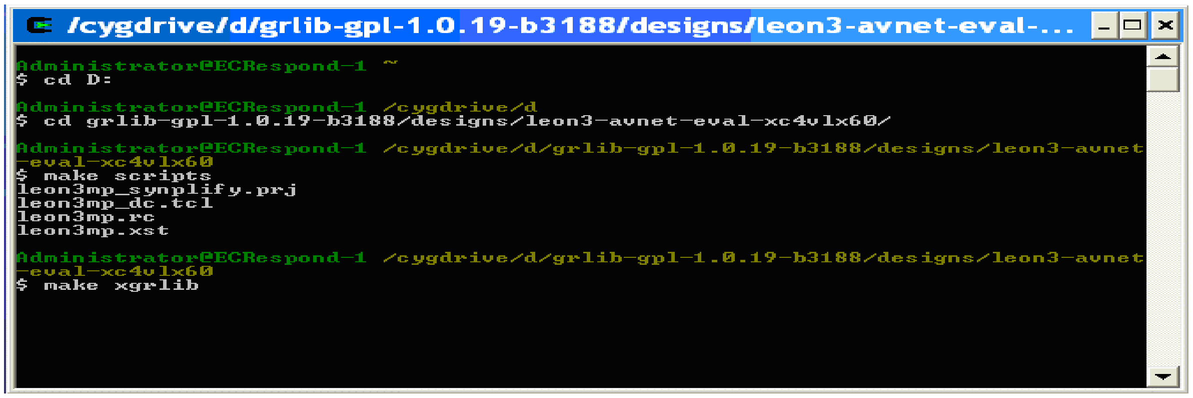
Change directory to designs/ leon3-avnet-eval-xc4vlx60, and issue the command 'make xconfig' in a bash shell (Linux) or cygwin shell (windows). This will launch the xconfig GUI tool that can be used to modify the leon3 template design. When the configuration is saved and xconfig is exited, the config. is automatically updated with the selected configuration.

Commands to be applied For LEON-3 configuration in CgWin:

1) First run Cgwin software and select the design according to our FPGA board.   .

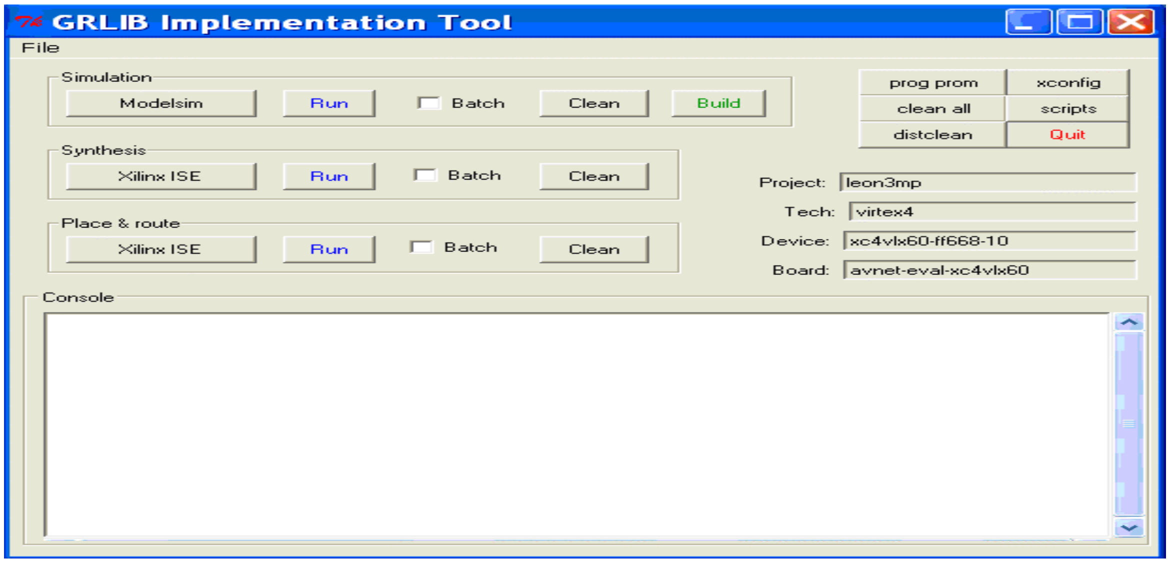
2) Give the following Command in Cgwin to prepare a script and for configuration window:

1. **make scripts**
2. **make xgrlib**



```
/cygdrive/d/grlib-gpl-1.0.19-b3188/designs/leon3-avnet-eval-...

Administrator@ECRespond-1 ~
$ cd D:

Administrator@ECRespond-1 /cygdrive/d
$ cd grlib-gpl-1.0.19-b3188/designs/leon3-avnet-eval-xc4vlx60/

Administrator@ECRespond-1 /cygdrive/d/grlib-gpl-1.0.19-b3188/designs/leon3-avnet
-eval-xc4vlx60
$ make scripts
leon3mp_synplify.prj
leon3mp_dc.tcl
leon3mp.rc
leon3mp.xst

Administrator@ECRespond-1 /cygdrive/d/grlib-gpl-1.0.19-b3188/designs/leon3-avnet
-eval-xc4vlx60
$ make xgrlib
```

Figure 6.1Cygwin Process Wizard



Project: leon3mp
Tech: virtex4
Device: xc4vlx60-ff668-10
Board: avnet-eval-xc4vlx60

Figure 6.2 GRLIB TOOL

3) Now in the configuration window the options for various tools for Synthesis, Simulation and Post & Route are provided. So choose a suitable one.

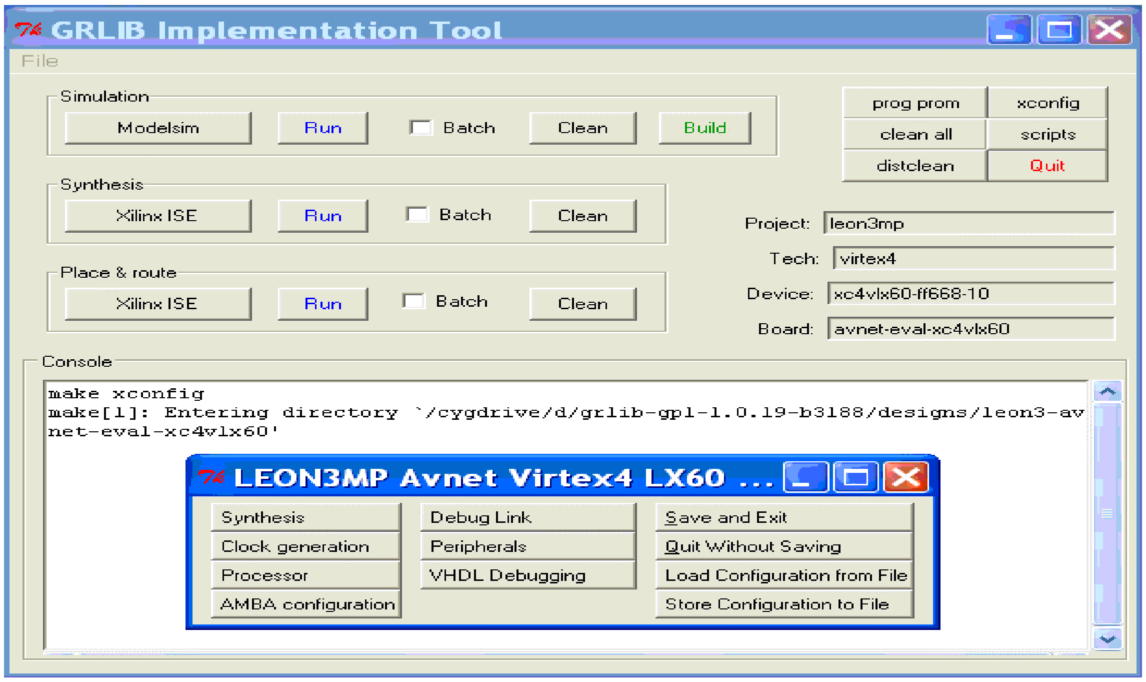
- Simulation

  - Modelsim
- Sinthesis

  - Xilinx ISE
- Post & Route

  - Xilinx ISE

4). After selecting various tools select 'Xconfig' for various parameters configuration for LEON-3 processor as shown in fig.



Project: leon3mp
Tech: virtex4
Device: xc4vlx60-ff668-10
Board: avnet-eval-xc4vlx60

```
make xconfig
make[1]: Entering directory `/cygdrive/d/grlib-gpl-1.0.19-b3188/designs/leon3-av
net-eval-xc4vlx60'
```

Figure 6.3 GRLIB Avnet Virtex LX60 Design

6.4 LEON3 Processor Simulation

The template design can be simulated in a test bench that emulates the prototype board. The test Bench includes external PROM and SDRAM which are pre-loaded with a test program.The test program will execute on the LEON3 processor, and test various functionality in the design.The test program will print diagnostics on the simulator console during the execution. The following command should be give to compile and simulate the template design and testbench:

```
make vsim
vsim testbench
```

A typical simulation log can be seen below.

```
$ vsim testbench
VSIM 1> run -a
# leon3-avnet-eval-xc4vlx60 Demonstration design
# GRLIB Version 1.0.15, build 2183
# Target technology: spartan3, memory library: spartan3
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 4, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Gaisler Research Leon3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research JTAG Debug Link
# ahbctrl: mst2: Gaisler Research SpaceWire Serial Link
# ahbctrl: mst3: Gaisler Research SpaceWire Serial Link
# ahbctrl: slv0: European Space Agency Leon2 Memory Controller
# ahbctrl: memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl: memory at 0x20000000, size 512 Mbyte
# ahbctrl: memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Gaisler Research AHB/APB Bridge
# ahbctrl: memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv2: Gaisler Research Leon3 Debug Support Unit
# ahbctrl: memory at 0x90000000, size 256 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency Leon2 Memory Controller
# apbctrl: I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Gaisler Research Generic UART
# apbctrl: I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Gaisler Research Multi-processor Interrupt Ctrl.
# apbctrl: I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Gaisler Research Modular Timer Unit
# apbctrl: I/O ports at 0x80000300, size 256 byte
# apbctrl: slv8: Gaisler Research General Purpose I/O port
# apbctrl: I/O ports at 0x80000800, size 256 byte
# apbctrl: slv12: Gaisler Research SpaceWire Serial Link
# apbctrl: I/O ports at 0x80000c00, size 256 byte
# apbctrl: slv13: Gaisler Research SpaceWire Serial Link
# apbctrl: I/O ports at 0x80000d00, size 256 byte
# grspw13: Spacewire link rev 0, AHB fifos 2x64 bytes, rx fifo 16 bytes, irq 11
# grspw12: Spacewire link rev 0, AHB fifos 2x64 bytes, rx fifo 16 bytes, irq 10
# grgpio8: 18-bit GPIO Unit rev 0
# gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8 10
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1
# apbuart1: Generic UART rev 1, fifo 1, irq 2
# ahbjtag AHB Debug JTAG rev 0
# dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 2 kbytes
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 1*8 kbyte, dcache 1*4 kbyte
# clkgen_spartan3e: spartan3/e sdram/pci clock generator, version 1
# clkgen_spartan3e: Frequency 50000 KHz, DCM divisor 4/5
# **** GRLIB system test starting ****
# Leon3 SPARC V8 Processor
# CPU#0 register file
# CPU#0 multiplier
# CPU#0 radix-2 divider
# CPU#0 floating-point unit
# CPU#0 cache system
# Multi-processor Interrupt Ctrl.
# Generic UART
# Modular Timer Unit
# timer 1
# timer 2
# chain mode
# Test passed, halting with IU error mode
# ** Failure: *** IU in error mode, simulation halted ***
# Time: 1104788 ns Iteration: 0 Process: /testbench/iuerr File: testbench.vhd
# stopped at testbench.vhd line 338
VSIM 2>
```

The test program executed by the test bench consists of two parts, a simple prom boot loader(prom.S) and the test program itself (systest.c). Both parts can be re-compiled using the ‘make soft’ command. This requires that the BCC tool-chain is installed on the host computer. Note that the simulation is terminated by generating a VHDL failure, which is the only way of stopping the simulation from inside the model. An error message is then printed:

```
# Test passed, halting with IU error mode
# ** Failure: *** IU in error mode, simulation halted ***
# Time: 1104788 ns Iteration: 0 Process: /testbench/iuerr File: testbench.vhd
# Stopped at testbench.vhd line 338
```

This error can be neglected.

Synthesis and place route

The template design can be synthesized with either Synplify-8.9 or ISE-9.2. Synthesis can be done in batch or interactively.

To use ISE interactively, use:

make ise-map

or

make scripts

ise leon3mp.npl

To perform place&route for a netlist generated with XST

make ise

The final programming file will be called ‘leon3mp.bit’.

Synthesis Reports of leon3-avnet-eval-xc4vlx60 board are as below are as below.

```
==========================================================================
HDL Synthesis Report
Macro Statistics
# RAMs                                  : 3
 16x32-bit dual-port RAM                : 1
 8x30-bit dual-port RAM                 : 2
# ROMs                                  : 7
 16x3-bit ROM                           : 2
 4x64-bit ROM                           : 2
# Multipliers                           : 1
 33x33-bit multiplier                   : 1
# Adders/Subtractors                    : 108
 10-bit subtractor                      : 1
 11-bit adder                           : 6
 11-bit subtractor                      : 2
 12-bit subtractor                      : 3
 15-bit subtractor                      : 1
 16-bit adder                           : 4
 18-bit adder                           : 3
 Timing Summary:
Speed Grade: -4
   Minimum period: 17.814ns (Maximum Frequency: 56.134MHz)
   Minimum input arrival time before clock: 2.061ns
   Maximum output required time after clock: 10.027ns
   Maximum combinational path delay: No path found
Process "Synthesize" completed successfully
```

6.5 LEON3 SoC design Testing

After Implementation we need to test LEON3 Core by some Soft program. The ‘C’code compilation can be done using gcc compiler. By using command - make soft, generate two files prom.srec and sdram.srac .This two file have been executed by the test bench and give result on the Modelsim console. The ‘C’ program and result are shown below.

LEON3 Core Integer Unit Testing

LEON core Integer unit has been tested by applying factorial program.

‘C ’code:

```
#include<stdio.h>
void main ()
{   report_start ();
    Printf ("factorial test started\n");
    int i,n=5,ans=1;
    for(i=n;i>0;i--)
    {
       ans=ans*i;
    }
    Printf ("Factorial of 5 is: %d", ans);
    Printf ("\factorial test completed\n");
        reported ();
}
```

Result on ModelSim Transcript:

```
# **** GRLIB system test starting ****
# Factorial test started
# Factorial of 5 is: 120
# Factorial test completed
```

LEON3 Core GPIO Port Testing

```
#include<stdio.h>
void main()
{
    report_start();

    int *data = (int *) 0x80000800;
```

```
   int *output = (int *) 0x80000804;
   int *direction= (int *) 0x8000808;

   *data=10;

   *output= *data * 20;

    printf("\nvalue of Data Resister:%d",*data);

    printf("\naddress of Data Resister:%x",data);

    printf("\nresult in Output Resister:%d",*output);

    printf("\naddress in Output Resister:%x\n",output);

  //  printf("address of Output resister:%x",output);

   *direction=~0;

    report_end();
}
```

Result on ModelSim Transcript:

```
**** GRLIB system test starting ****
#
# value of Data Resister:10
#
# address of Data Resister:80000800
#
# result in Output Resister:200
#
# address in Output Resister:80000804
```

6.6 Implementation on FPGA

For Implementation of LEON-3 Core on FPGA , create bit file by Xilinx ISE 9.2i tool. This Bit file of LEON-3 Core is also available at www.gaisler.com. After generation of the Bit File implement on FPGA board using 'Xilinx IMPACT' software.

Steps for Implementing test application for LEON core

1). To apply tests for LEON-3 core. Apply following command

make soft

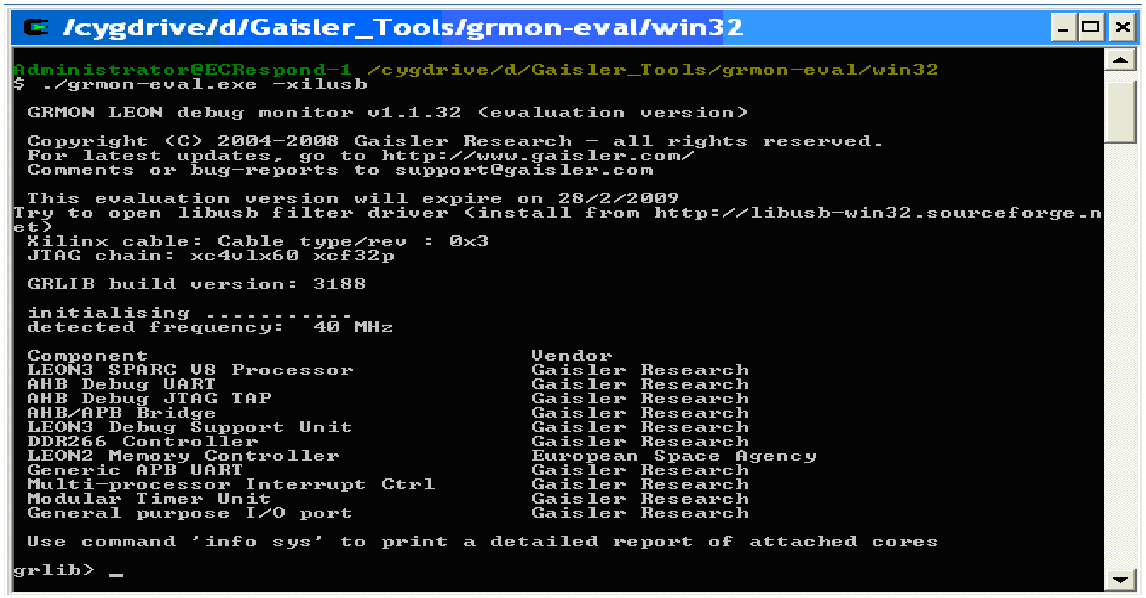
2). After applying the above command sram.srec, sdram.srec files will be generated.

3). To load this Executable file on board software named 'GRMON' is required.

4). Now load this application on FPGA board, apply following command

./grmon-eval.exe –xilusb

As we apply above command 'GRMON' will be connected with our FPGA board and it will show the connection as well as the information about LEON-3 system which can be seen in the figure 6.4.



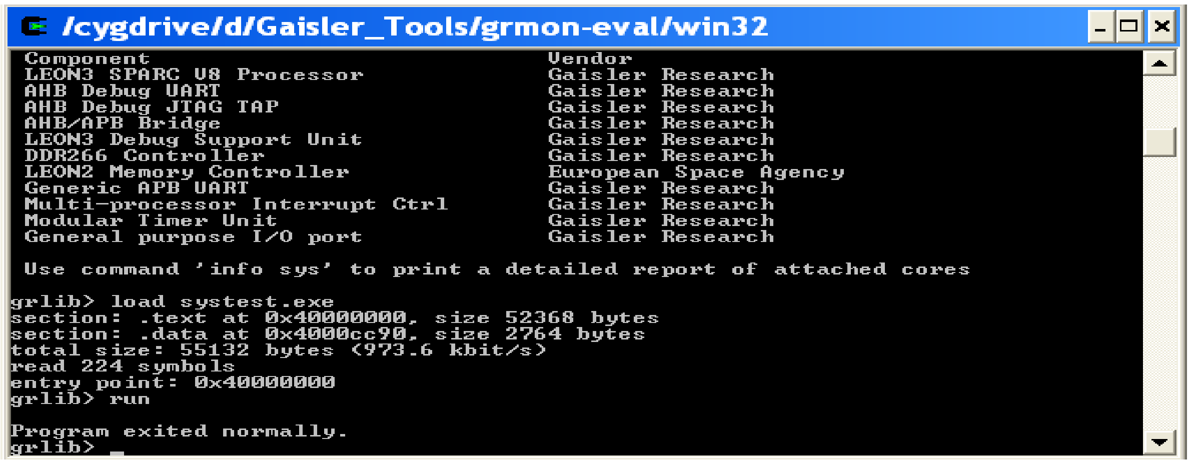
```
Administrator@ECRespond-1 /cygdrive/d/Gaisler_Tools/grmon-eval/win32
$ ./grmon-eval.exe -xilusb

 GRMON LEON debug monitor v1.1.32 (evaluation version)

 Copyright (C) 2004-2008 Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to support@gaisler.com

 This evaluation version will expire on 28/2/2009
Try to open libusb filter driver (install from http://libusb-win32.sourceforge.n
et)
 Xilinx cable: Cable type/rev : 0x3
 JTAG chain: xc4vlx60 xcf32p

 GRLIB build version: 3188

 initialising ..........
 detected frequency:  40 MHz

 Component                            Vendor
 LEON3 SPARC V8 Processor             Gaisler Research
 AHB Debug UART                       Gaisler Research
 AHB Debug JTAG TAP                   Gaisler Research
 AHB/APB Bridge                       Gaisler Research
 LEON3 Debug Support Unit             Gaisler Research
 DDR266 Controller                    Gaisler Research
 LEON2 Memory Controller              European Space Agency
 Generic APB UART                     Gaisler Research
 Multi-processor Interrupt Ctrl       Gaisler Research
 Modular Timer Unit                   Gaisler Research
 General purpose I/O port             Gaisler Research

 Use command 'info sys' to print a detailed report of attached cores

grlib> _
```

Fig. 6.4 GRMON result

5). To see the detail of LEON system apply command "info sys" on consol of 'GRMON'

To load the application, apply the following command in GRMON window

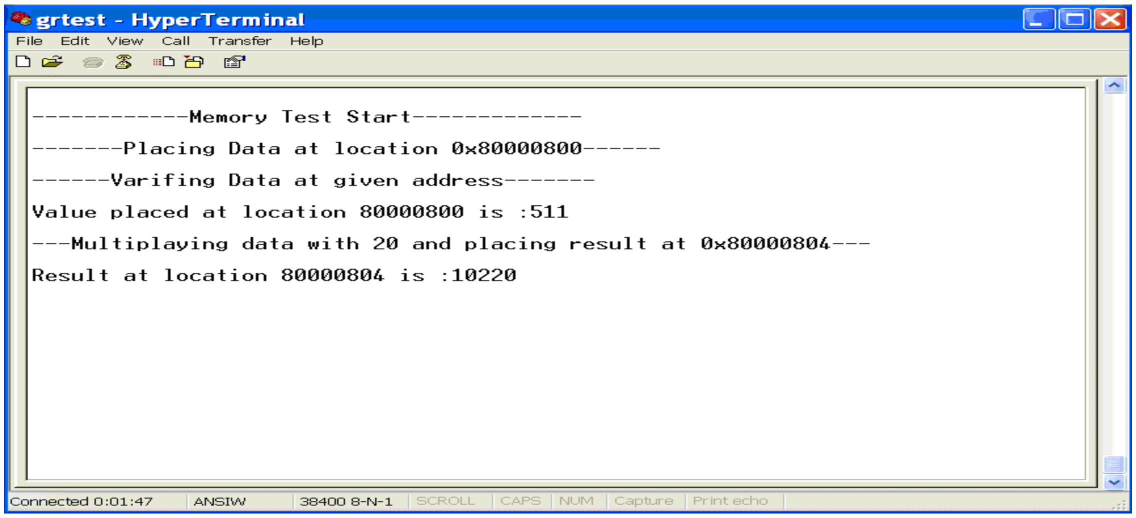
load systest.exe



```
 Component                            Vendor
 LEON3 SPARC V8 Processor             Gaisler Research
 AHB Debug UART                       Gaisler Research
 AHB Debug JTAG TAP                   Gaisler Research
 AHB/APB Bridge                       Gaisler Research
 LEON3 Debug Support Unit             Gaisler Research
 DDR266 Controller                    Gaisler Research
 LEON2 Memory Controller              European Space Agency
 Generic APB UART                     Gaisler Research
 Multi-processor Interrupt Ctrl       Gaisler Research
 Modular Timer Unit                   Gaisler Research
 General purpose I/O port             Gaisler Research

 Use command 'info sys' to print a detailed report of attached cores

grlib> load systest.exe
section: .text at 0x40000000, size 52368 bytes
section: .data at 0x4000cc90, size 2764 bytes
total size: 55132 bytes (973.6 kbit/s)
read 224 symbols
entry point: 0x40000000
grlib> run

Program exited normally.
grlib> _
```

Fig. 6.5 Loading application on LEON system

6). To check the result of application connect the serial cable RS232 with the UART of board and serial COM port of computer and Open HyperTerminal window. Boud rate set to 38400 .



```
------------Memory Test Start-------------
-------Placing Data at location 0x80000800------
------Varifing Data at given address-------
Value placed at location 80000800 is :511
---Multiplaying data with 20 and placing result at 0x80000804---
Result at location 80000804 is :10220
```

Connected 0:01:47 | ANSIW | 38400 8-N-1 | SCROLL | CAPS | NUM | Capture | Print echo

Fig. 6.6 HyperTerminal Result

Chapter 7

AMBA AHB BUS

7.1 Introduction

GRLIB is based on the AMBA AHB and APB on-chip buses, which is used as the standard interconnect interface. The implementation of the AHB/APB buses is compliant with the AMBA-2.0 specification, with additional 'sideband' signals for automatic address decoding, interrupt steering and device identification (a.k.a. plug &play support). The AHB and APB signals are grouped according to functionality into VHDL records, declared in the GRLIB VHDL library. The GRLIB AMBA package source files are located in lib/grlib/amba. All GRLIB cores use the same data structures to declare the AMBA interfaces, and can then easily be connected together. An AHB bus controller and an AHB/APB bridge are also available in the GRLIB library, and allow to assemble quickly a full AHB/APB system. The following sections will describe how the AMBA buses are implemented and how to develop a SOC design using GRLIB.

7.2 AMBA AHB On chip bus

7.2.1 AHB master interface

The AHB master inputs and outputs are defined as VHDL record types, and are exported through the TYPES package in the GRLIB AMBA library:

```
-- AHB master inputs
type ahb_mst_in_type is record
hgrant : std_logic_vector(0 to NAHBMST-1); -- bus grant
hready : std_ulogic; -- transfer done
hresp : std_logic_vector(1 downto 0); -- response type
hrdata : std_logic_vector(31 downto 0); -- read data bus
hcache : std_ulogic; -- cacheable
hirq : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt result bus
```

```
end record;
-- AHB master outputs
type ahb_mst_out_type is record
hbusreq : std_ulogic; -- bus request
hlock : std_ulogic; -- lock request
htrans : std_logic_vector(1 downto 0); -- transfer type
haddr : std_logic_vector(31 downto 0); -- address bus (byte)
hwrite : std_ulogic; -- read/write
hsize : std_logic_vector(2 downto 0); -- transfer size
hburst : std_logic_vector(2 downto 0); -- burst type
hprot : std_logic_vector(3 downto 0); -- protection control
hwdata : std_logic_vector(31 downto 0); -- write data bus
hirq : std_logic_vector(NAHBIRQ-1 downto 0);-- interrupt bus
hconfig : ahb_config_type; -- memory access reg.
hindex : integer range 0 to NAHBMST-1; -- diagnostic use only
end record;
```

The elements in the record types correspond to the AHB master signals as defined in the AMBA 2.0 specification, with the addition of four sideband signals: HCACHE, HIRQ, HCONFIG and HINDEX. A typical AHB master in GRLIB has the following definition:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;
entity ahbmaster is
generic (
hindex : integer := 0); -- master bus index
port (
reset : in std_ulogic;
clk : in std_ulogic;
hmsti : in ahb_mst_in_type; -- AHB master inputs
hmsto : out ahb_mst_out_type -- AHB master outputs
);
end entity;
```

The input record (HMSTI) is routed to all masters, and includes the bus grant signals for all masters in the vector HMSTI.HGRANT. An AHB master must therefore use a generic that specifies which HGRANT element to use. This generic is of type integer, and typically called HINDEX (see example above).

7.2.2 AHB slave interface

Similar to the AHB master interface, the inputs and outputs of AHB slaves are defined as two VHDL records types:

```
-- AHB slave inputs
type ahb_slv_in_type is record
hsel : std_logic_vector(0 to NAHBSLV-1); -- slave select
haddr : std_logic_vector(31 downto 0); -- address bus (byte)
hwrite : std_ulogic; -- read/write
htrans : std_logic_vector(1 downto 0); -- transfer type
hsize : std_logic_vector(2 downto 0); -- transfer size
hburst : std_logic_vector(2 downto 0); -- burst type
hwdata : std_logic_vector(31 downto 0); -- write data bus
hprot : std_logic_vector(3 downto 0); -- protection control
hready : std_ulogic; -- transfer done
hmaster : std_logic_vector(3 downto 0); -- current master
hmastlock : std_ulogic; -- locked access
hbsel : std_logic_vector(0 to NAHBCFG-1); -- bank select
hcache : std_ulogic; -- cacheable
hirq : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt result bus
end record;
-- AHB slave outputs
type ahb_slv_out_type is record
hready : std_ulogic; -- transfer done
hresp : std_logic_vector(1 downto 0); -- response type
```

```
hrdata : std_logic_vector(31 downto 0); -- read data bus
hsplit : std_logic_vector(15 downto 0); -- split completion
hcache : std_ulogic; -- cacheable
hirq : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt bus
hconfig : ahb_config_type; -- memory access reg.
hindex : integer range 0 to NAHBSLV-1; -- diagnostic use only
end record;
```

The elements in the record types correspond to the AHB slaves signals as defined in the AMBA 2.0 specification, with the addition of five sideband signals: HBSEL, HCACHE, HIRQ, HCONFIG and HINDEX. A typical AHB slave in GRLIB has the following definition:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;
entity ahbslave is
generic (
hindex : integer := 0); -- slave bus index
port (
reset : in std_ulogic;
clk : in std_ulogic;
hslvi : in ahb_slv_in_type; -- AHB slave inputs
hslvo : out ahb_slv_out_type -- AHB slave outputs
);
end entity;
```

The input record (ahbsi) is routed to all slaves, and include the select signals for all slaves in the vector ahbsi.hsel. An AHB slave must therefore use a generic that specifies which hsel element to use. This generic is of type integer, and typically called HINDEX (see example above).

7.2.3 AHB bus control

GRLIB AMBA package provides a combined AHB bus arbiter (ahbctrl), address decoder and bus multiplexer. It receives the ahbmo and ahbso records from the AHB units, and generates ahbmi and ahbsi as indicated in figure 2.4. The bus arbitration function will generate which of the ahbmi.hgrant elements will be driven to indicate the next bus master. The address decoding function will drive one of the ahbsi.hsel elements to indicate the selected slave. The bus multiplexer function will select which master will drive the ahbsi signal, and which slave will drive the ahbmo signal.

7.2.4 AHB bus index control

The AHB master and slave output records contain the sideband signal HINDEX. This signal is used to verify that the master or slave is driving the correct element of the ahbso/ahbmo buses. The generic HINDEX that is used to select the appropriate hgrant and hsel is driven back on ahbmo.hindex and ahbso.hindex. The AHB controller then checks that the value of the received HINDEX is equal to the bus index. An error is issued during simulation if a mismatch is detected.

7.3 AHB plug & play configuration

The GRLIB implementation of the AHB bus includes a mechanism to provide plug&play support. The plug&play support consists of three parts: identification of attached units (masters and slaves), address mapping of slaves, and interrupt routing. The plug&play information for each AHB unit consists of a configuration record containing eight 32-bit words. The first word is called the identi- fication register and contains information on the device type and interrupt routing. The last four words are called bank address registers, and contain address mapping information for AHB slaves. The remaining three words are currently not assigned and could be used to provide core-specific configuration information.

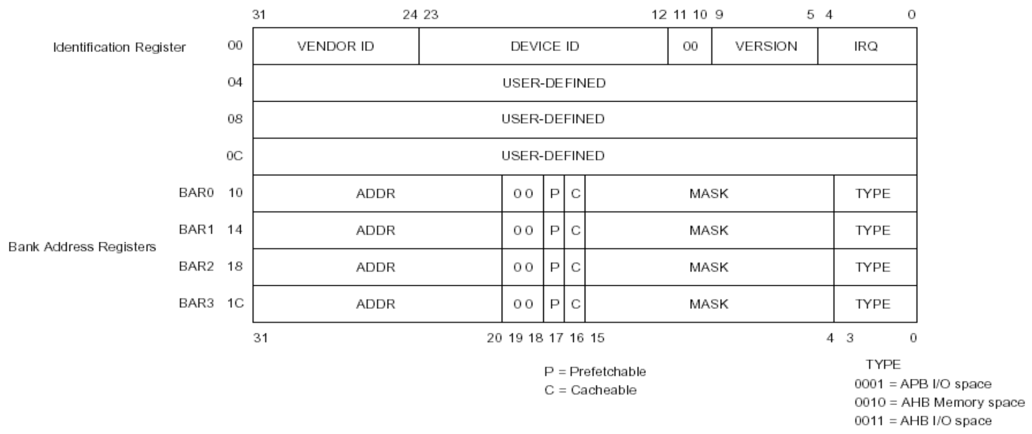


Fig.7.1 AHB plug & play configuration layout

The plug&play information for all attached AHB units appear as a read-only table mapped on affixed address of the AHB, typically at 0xFFFFF000. The configuration records of the AHB masters appear in 0xFFFFF000 - 0xFFFFF800, while the configuration records for the slaves appear in 0xFFFFF800 - 0xFFFFFFFC. Since each record is 8 words (32 bytes), the table has space for 64 masters and 64 slaves. A plug&play operating system (or any other application) can scan the con- figuration table and automatically detect which units are present on the AHB bus, how they are configured, and where they are located (slaves). The configuration record from each AHB unit is sent to the AHB bus controller via the HCONFIG signal. The bus controller creates the configuration table automatically, and creates a read-only memory area at the desired address (default 0xFFFFF000). Since the configuration information is fixed, it can be efficiently implemented as a small ROM or with relatively few gates. A debug module (ahbreport) in the WORK.DEBUG package can be used to print the configuration table to the console during simulation, which is useful for debugging. A typical example is provided below:

7.4 SpaceWire Interface with AMBA AHB Bus

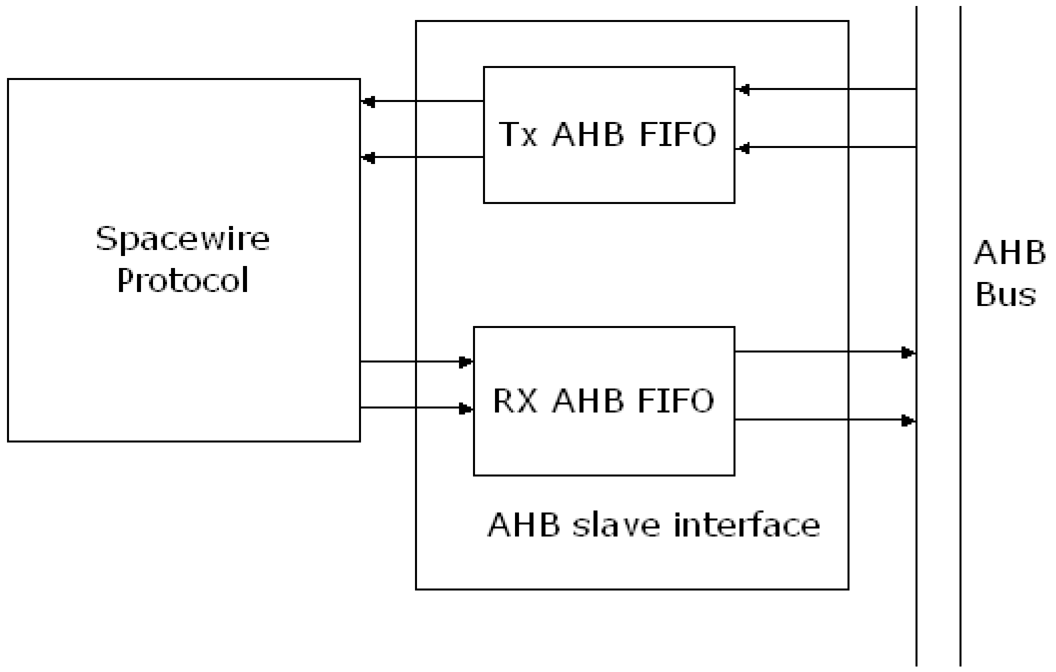


Fig. 7.2 AHB slave interface

AMBA AHB interface with SpaceWire Protocol having two types of interface.

- Direct
- Indirect

In direct interface, signals interfaced directly to the AMBA AHB signals. In indirect interface data transfer controlled through FSM. Interface module composed by AHB TX FIFO, AHB Rx FIFO and AMBA signals.

The Tx Data AHB FIFO block is a FIFO containing the data to be transmitted

The Rx Data FIFO block is a FIFO containing the data to be the host memory.

The Tx AHB slave interface is used when the data transmission is in charge of the host.

7.5 Interface signals

Signals	I/O	Description

ahb_slv_in. HSEL
ahb_slv_in.HWRITE
ahb_slv_in.HADDR(31-0)
ahb_slv_in.HTRANS(1-0)
ahb_slv_in.HWDATA(31-0)
ahb_slv_in.HREADY
ahb_slv_in.HSIZE(2-0)
ahb_slv_in.HBURST(2-0)
ahb_slv_in.HPROT(3-0)
	Input	AMBA AHB Slave Bus in for the Tx host Interface

ahb_slv_out.HREADY
ahb_slv_out.HRESP(1-0)
ahb_slv_out.HRDATA(31-0)
ahb_slv_out.HSPLIT(15-0)
	Output	AMBA AHB Slave Bus out for the host Interface

Table 7.1 AHB signals

7.6 Interface Read/write logic

The interface module basically read and writes data from the host memory to the AHB FIFO. The FSM for the read/write is in Figure 7.3.

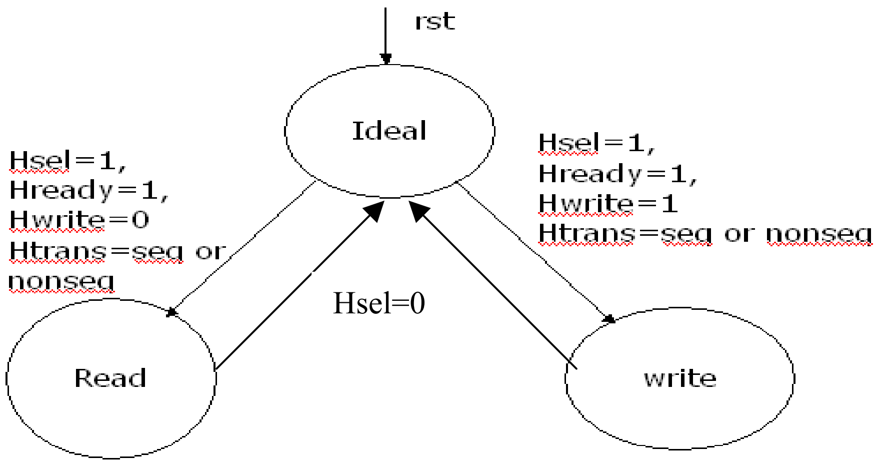


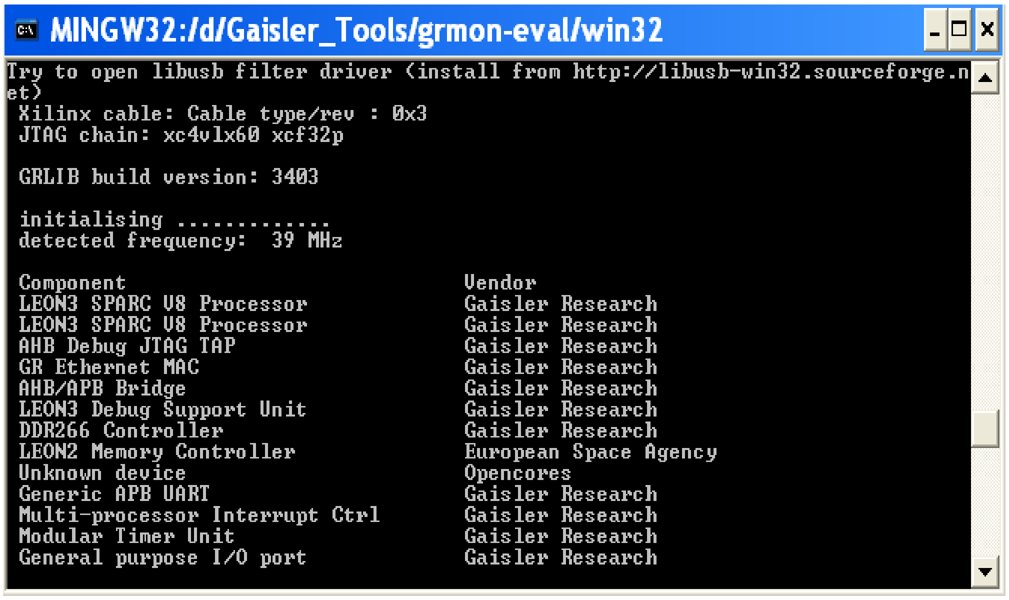
Fig. 7.3 Read/Write Logic

7.7 Impimetation of AHB Interface on FPGA

To test the AHB interface with LEON3 Processor, C code for reading and writing AHB FIFO.

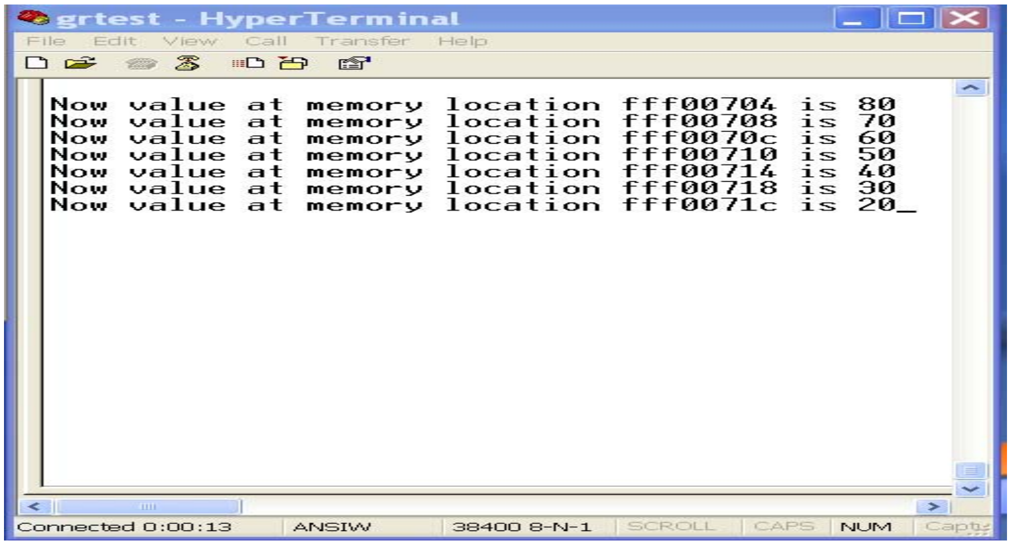
C programme to test AHB interface with LEON3 Processor.

```
#include<stdio.h>
 main() {
         report_start();
         volatile int *data = (volatile int*) 0xfff00700;
         *data=0x80;
         printf("\nNow value at memory location %x is %x",data,*data);
         data=data+1;
        *data=0x70;
        printf("\nNow value at memory location %x is %x",data,*data);
        data=data+1;
        *data=0x60;
        printf("\nNow value at memory location %x is %x",data,*data);
        data=data+1;
        *data=0x50;
        printf("\nNow value at memory location %x is %x",data,*data);
        data=data+1;
        *data=0x50;
        printf("\nNow value at memory location %x is %x",data,*data);
        data=data+1;
        *data=0x40;
        printf("\nNow value at memory location %x is %x",data,*data);
        data=data+1;
        *data=0x30;
        printf("\nNow value at memory location %x is %x",data,*data);
        data=data+1;
        *data=0x20;
        printf("\nNow value at memory location %x is %x",data,*data);
        report_end()};
```



```
Try to open libusb filter driver (install from http://libusb-win32.sourceforge.n
et)
 Xilinx cable: Cable type/rev : 0x3
 JTAG chain: xc4vlx60 xcf32p

 GRLIB build version: 3403

 initialising .............
 detected frequency:  39 MHz

 Component                             Vendor
 LEON3 SPARC V8 Processor              Gaisler Research
 LEON3 SPARC V8 Processor              Gaisler Research
 AHB Debug JTAG TAP                    Gaisler Research
 GR Ethernet MAC                       Gaisler Research
 AHB/APB Bridge                        Gaisler Research
 LEON3 Debug Support Unit              Gaisler Research
 DDR266 Controller                     Gaisler Research
 LEON2 Memory Controller               European Space Agency
 Unknown device                        Opencores
 Generic APB UART                      Gaisler Research
 Multi-processor Interrupt Ctrl        Gaisler Research
 Modular Timer Unit                    Gaisler Research
 General purpose I/O port              Gaisler Research
```

Fig 7.4 AHBFIFO interface with LEON



```
Now value at memory location fff00704 is 80
Now value at memory location fff00708 is 70
Now value at memory location fff0070c is 60
Now value at memory location fff00710 is 50
Now value at memory location fff00714 is 40
Now value at memory location fff00718 is 30
Now value at memory location fff0071c is 20_
```

Fig 7.5 AHBFIFO read/write result on HyperTerminal

Chapter 8

Conclusion and Future Scope

Conclusion

The project work entitled as **RTL Design of SpaceWire Protocol and AMBA Interface with LEON3 Processor,** during project work literature survey is carried out to understand the SpaceWire Protocol, LEON3 Processor, GRLIBL IP library this has provided conceptual understandings.I have implemented Open source LEON3 Processor based SoC design and AMBA AHB interface with AHBFIFO on Avnet-virtex-4 FPGA board. I have also done RTL design and simulation of SpaceWire Protocol.

Future Scope

Following work is to be carried out during remaining part of Project period.

- Interfacing SpaceWire Protocol with LEON3 Processor through AMBA AHB Bus
- Plugging SpaceWire Protocol with LEON3 based SoC design.

REFERENCES

[1] Sergio Sayonara, Luca Fanucci,” Radiation Tolerant SpaceWire Router for Satellite On-Board Networking”.

[2] GRLIB IP Library User’s Manual, Version 1.0.19.

[3] A. Sai Pramod Kumar, thesis report “A Prototype and Validation Platform for LEON based Multiprocessor SoCs” Department of Computer Science and Engineering, ITDelhi, 2002.

[4] Jiri Gaisler. The LEON-2 Processor User’s Manual, Version 2-1.0.4, 2002. http://www.gaisler.com.

[5] Gaisler Research. TSIM Simulator User’s Manual, Version 1.0.11, 2001. http://www.gaisler.com/tsim.html.

[6] ECSS-E-50-12A Standards, Space engineering

[7] S.M. Parkes et al, “SpaceWire – Links, Nodes, Routers and Networks”, European Cooperation for Space Standardization, Standard No. ECSS-E50-12A, Issue1, January 2003.

[8] Chris McClements, Steve Parkes, Agustin Leon “The SpaceWire CODEC International SpaceWire Seminar (ISWS 2003)”.

[9]AMBA 2.0 reference manual.

[10] Kritikal Solutions pvt Ltd,”Generic Embedded Design Kit”, May 2006

Appendix-A

Installation of Tools

Grlib Installation

Grlib can be downloaded from the gaisler research site having the following link location

http://www.gaisler.com/products/grlib/ grlib-gpl-1.0.19-b3188

GRLIB is distributed as a gzipped tar-file and can be installed in any location on the host system:

```
tar xzf grlib-gpl-<version>.tar.gz
for example
tar xzf grlib-gpl-1.0.19-b3188.tar
```

the above command will produce the directory grlib-gpl-1.0.19-b3188 .The distribution of the glib directory has the following file hierarchy:

Bin	various scripts and tool support files
Boards	support files for FPGA prototyping boards
Designs	template designs
Doc	infra-structure documentation
Grlib.html	Grlib IP library html page
Lib	IP library
Software	VHDL libraries and documentation

GRLIB uses the GNU 'make' utility to generate scripts and to compile and synthesis designs. It must therefore be installed on a unix system or in a 'unix-like' environment. Tested hosts systems are Linux and Windows with Cygwin.

Grlib IP cores

GRLIB is organized around VHDL libraries, where each IP vendor is assigned a unique library name. Each vendor is also assigned a unique subdirectory under grlib/lib in which all vendor-specific source files and scripts are contained. The vendor-specific directory can contain subdirectories, to allow for further partitioning between IP cores etc.

The basic directories delivered with GRLIB under grlib-gpl-1.0.19-b3188/lib are:

grlib	packages with common data types and functions
gaisler	Gaisler Research's components and utilities

tech/* target technology libraries for gate level simulation
work components and packages in the VHDL work library

Other vendor-specific directories are also delivered with GRLIB like contrib., esa , micron, open cores, techmap , cypress, gleichmann, open chip but are not necessary for the understanding of the design concept. Libraries and IP cores are described in detail in separate documentation.

GRMON Installation

GRMON can be downloaded from the gaisler research site having the following link location

ftp://gaisler.com/gaisler.com/grmon/grmon-eval-1.1.19.tar.gz

GRMON is currently available for three platforms: linux, windows and solaris. GRMON can be installed anywhere on the host computer - for convenience the installation directory should be added to the search path. For example to install the grmon in cygwin environment use the following commands (in cygwin)

a) to untar the file in /opt (can be any other directory) directory
tar xzf grmon-eval-1.1.16.tar.gz
b) to set the path of the grmon , append the following line in the. bashrc file in thehome directory

```
export PATH=/opt/ grmon-eval-1.1.16/cygwin:$PATH
```

BCC Installation

BCC can be downloaded from the gaisler research site having the following Link location

ftp://gaisler.com/gaisler.com/bcc/bin/windows/sparc-elf-3.2.3-1.0.24-cygwin.tar.bz2

BCC is provided as a bzipped tar-file. It should be unpacked in the /opt Directory of the host using the following commands:

a) mkdir /opt -- to make directory opt in / (if it is not present already)
b) cd /opt -- change directory to /opt
c) tar -xjf sparc-elf-3.2.3-1.0.24-cygwin.tar.bz2
d) After installation, add /opt/sparc-elf-3.2.3/bin (or /opt/sparc-elf-3.4.4/bin) to the PATH variable by appending the following line in the.bashrc file in the home directory .

```
export PATH=/opt/sparc-elf-3.2.3/bin:$PATH
```

Appendix-B

SpaceWire Component

The IP is composed of height component:

TX: transmitter.

RX: receptor.

TX fifo: first in first out buffer for the transmission.

RX fifo: first in first out buffer for the reception.

Two domain clock: interface the signal between different clock.

Time-id buffer: buffer for the time id seeded.

FSM: state machine manage IP.

Transmitter: tx.vhd

```
Component TX is
port (
     Reset_n: in std_logic;
     Tx_Clk: in std_logic;
     -- Main FSM interface
     State: in FSM_State;
     -- Tx Fifo interface
     Tx_FIFO_Din: in std_logic_vector (8 downto 0);
     Tx_FIFO_Rd_n: inout std_logic;
     Tx_FIFO_Empty_n: in std_logic;
     -- Credit
     Rx_FIFO_Credit_Rd_n: out std_logic;    -- Read one
                            more FCT from  The FIFO
     Rx_FIFO_Credit_Empty_n: in std_logic;  -- true when
                   there is no  more FCT in the FIFO
     Tx_Credit_Empty_n: in std_logic;
      -- Time code
     Send_Time_n: in std_logic;
     Time Code: in std_logic_vector (7 downto 0);
     time_id_sended_n: out std_logic;
     send_esc: in std_logic;
     send_eop_n: out std_logic;
     -- Link
     Dout: out std_logic;
     Sout: out std_logic
);
end component;
```

Receiver: Rx.vhd

```
Component Rx is
   Port (

     Reset_n : in std_logic;
     Clk : in std_logic;
     Rx_Clk : inout std_logic;
     -- Main FSM interface
     State : in FSM_State;
     -- Got out
     got_NULL_n  : out std_logic;
     got_ESC_n  : out std_logic;
     got_FCT_n   : out std_logic;
     got_EOP_n   : out std_logic;
     got_EEP_n   : out std_logic;
     got_NChar_n :out std_logic;
     -- error
     Error_Par_n :out std_logic;-- Parity error
     Error_ESC_n :out std_logic;-- ESC followed by ESC,EOP
     Error_Dis_n : out std_logic;  -- Disconnected
     -- Rx Fifo interface
     Rx_FIFO_D : out std_logic_vector(8 downto 0);
     Rx_FIFO_Wr_n : out std_logic;
     -- Time Code interface
     got_Time_n  : out std_logic;
     -- Link
     Din : in std_logic;
     Sin : in std_logic
      );

end Rx;
```

fsm: fsm.vhd

```
component fsm is
     port(
          Reset_n : in std_logic;
          Clk : in std_logic;
          State : out FSM_State;
          linkEnabled : in std_logic;
          -- input
          short_got_fct_n : in std_logic;
          short_got_null_n : in std_logic;
          short_got_NChar_n : in std_logic;
          short_got_Time_n : in std_logic;
          -- input error
          Rx_credit_error_n : in std_logic;
          Tx_credit_error_n : in std_logic;
          short_Error_Dis_n : in std_logic;
          short_Error_Par_n : in std_logic;
          short_Error_ESC_n : in std_logic;
          view_fsm : out std_logic_vector(3 downto 0)
     );
end component;
```

Receiver fifo: rx_fifo.vhd

```
component Rx_Fifo is
     generic(
          WIDTH : integer; --:= 8;
          LENGTH : integer; --:= 128;
          MAX_CREDIT : integer --:= 7*8
             );
     port(
          Reset_n : in std_logic;
          Clk : in std_logic;
          State : in FSM_State;
          -- Credit
          Credit_Rd_n : in std_logic;  -- allow 8 writes in
                                            the  fifo
          Credit_Empty_n : out std_logic; -- true when all
                     the FIFOhas been allowed to be written
          credit_error_n : out std_logic;
          -- Data Input
          Din : in std_logic_vector(WIDTH-1 downto 0);
          Wr_n : in std_logic;
          Full_n : out std_logic;
          short_got_EOP_n : in std_logic;
          -- Data Output
          Dout : out std_logic_vector(WIDTH-1 downto 0);
          Rd_n : in std_logic;
          Empty_n : out std_logic;
          Credit : inout integer range 0 to MAX_CREDIT);
     end component;
```

Two domain clock:

```
component twodomainclock is
     generic( N_short_to_width : integer;
              use_short_to_width : integer;
              N_width_to_short : integer;
              use_width_to_short : integer;
              N_short_to_width_n : integer;
              use_short_to_width_n : integer;
              N_width_to_short_n : integer;
              use_width_to_short_n : integer
              );
 port (
     Rst : in std_logic;
     Clk_speed : in std_logic;
     Clk_slow : in std_logic;
in_short_pulse : in std_logic_vector(N_short_to_width - 1
downto 0);
in_width_pulse : in std_logic_vector(N_width_to_short - 1
downto 0);
out_width_pulse : inout std_logic_vector(N_short_to_width -
1 downto0);
out_short_pulse : out std_logic_vector(N_width_to_short - 1
downto 0);
in_short_pulse_n    : in
std_logic_vector(N_short_to_width_n - 1downto0);
in_width_pulse_n : in std_logic_vector(N_width_to_short_n -
1 downto0);
out_width_pulse_n :inout
std_logic_vector(N_short_to_width_n-1downto0);
out_short_pulse_n :out std_logic_vector(N_width_to_short_n-
1 downto 0) );
end component;
```