ABSTRACT

NAIDU, SATISH N. A Distributed Network Switch Bus Architecture for Small Satellites. (Under the direction of Dr. William W. Edmonson).

Miniaturization in technology has led to the development of a class of satellites known as small satellites whose development time and cost has reduced drastically. This reduction in development time and cost is attracting the industry to fulfill a customer's need. There are a plenty of challenges faced during the design and development stage, and placing the satellite in an orbit. In addition to the development challenges encountered in all spacecraft missions, meeting their small sizes, relatively small power, and mass budget warrants to be a challenging task.

The Architecture of a system refers to the physical and logical framework used to interconnect different subsystems. This work presents a distributed architecture which supports computational decentralization of a task across a number of processors which are physically located in different subsystems. The data bus in the architecture is based on SpaceWire bus protocol containing a crossbar switch which connects each of the subsystems using Bus Interface Module (BIM). Each subsystem becomes an interchangeable module which corresponds to standard signaling and command interfaces. This architecture is proved to be modular, scalable and reconfigurable at the design stage for any given mission. This architecture supports shared memory module to store and access science data for the mission. It also supports additional processing elements and DSP farms as a backup for complex processing requirements. As a prototype of the proposed architecture, a 4-port crossbar switch with BIMs were implemented on a Xilinx Virtex-4 FPGA, which can be configured to accommodate any number of ports as needed during the design time. We also discuss a few fault tolerance techniques to prevent against radiation effects and to extend the duration of a mission.

A Distributed Network Switch Bus Architecture for Small Satellites

by
Satish Naidu

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2010

APPROVED BY:

Dr. William W. Edmonson
Chair of Advisor Committee

Dr. Winser E. Alexander

Dr. Alexander G. Dean

DEDICATION

To

my parents and friends

BIOGRAPHY

Satish Naidu was born on July 22, 1984 in Bangalore, India. He received his Bachelor of Engineering (B.E.) degree in Electronics and Communication from Sir M Visvesvaraya Institute of Technology (Visvesvaraya Technological University) in June 2006. He worked at Cognizant Technology Solutions for two years as a Technical Analyst and later joined the Graduate program at NC State University in Electrical and Computer Engineering in August 2008.

He has been part of HiperDSP group since Spring 2009, working under Dr. William Edmonson in the field of design and development of bus architectures for small satellites. He has also been a part of Advanced Space Technologies Research & Engineering Center (ASTREC), a collaborative space based project between the North Carolina State University, and the University of Florida, Gainesville.

His research interests include DSP, Computer Architecture, VLSI, ASIC Design and ASIC Verification. He has a keen interest in playing volleyball and badminton, and is a part of NCSU Badminton Club. He has represented NCSU in various badminton tournaments.

ACKNOWLEDGEMENTS

I thank my parents, Lalitha Rani and Narasimhalu Naidu, and sister, Sandhya, for the motivation, strength, courage and moral support they provided me with during my stay so far away from home.

I sincerely thank Dr. William Edmonson, my academic advisor, for all the motivation, support and providing guidance to successfully complete this thesis. He has given me the opportunity and freedom to explore my ideas in solving the problems and challenges faced during the course of my thesis. I am sincerely thankful to his patience and invaluable suggestions in making my thesis readable. I am also grateful to my other committee members, Dr. Winser Alexander and Dr. Alexander Dean for their valuable inputs during the course of my thesis. Their guidance and suggestions have helped me to be a successful student during my stay at NCSU. I would like to acknowledge all the faculty and staff members of Electrical and Computer Engineering Department at NCSU for sharing their ideas and help me conduct my research successfully.

I am also thankful to all the members of HiperDSP group for sharing their research ideas and providing useful suggestions during the weekly presentations. I would also like to thank my roommates for making my stay at NCSU memorable. I am also thankful to all my friends at Raleigh for their support and presence during my leisure. Above all, I thank God for helping me achieve success in every endeavor of my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Small Satellite Introduction

It was October 4, 1957, when the Soviet Union successfully launched Sputnik I, which marked the start of the space age. The world's first artificial satellite, about the size of a beach ball (58 cm. in diameter), weighed only 83.6 kg [1]. With the launch of Sputnik I, we saw the beginning of space age and so called space race between US and the Soviet Union. In reply, the United States successfully launched Explorer I on January 31, 1958. This satellite carried a small scientific payload that eventually discovered the magnetic radiation belts around the Earth, named after principal investigator James Van Allen. Since then, the number of satellites launched to the Earth's orbit has steadily increased.

Prior to 1990, the aerospace community was focusing on much larger spacecraft with long and expensive development and mission times. The small satellite renaissance began in the late 1980s and is changing the economics of space. The miniaturization of technology was the prime cause for these small satellites to accomplish missions with much lower development costs and development time. This reduction in development time and cost is attracting the industry to fulfill a customer's need.

Small satellites can be classified into four groups [2]. The first and largest being the Min-

isatellite, with a wet mass (satellite bus payload and fuel) in the range of 100 to 500 kg. These are simpler, but the underlying technology is similar to that of a large satellite. Second among the class is Microsatellite, with a wet mass between 10 and 100 kg. Sometimes, in generic terms the Minisatellites and Microsatellites are simply referred to as "small satellites". The next class, Nanosatellite has a wet mass in the range of 1 to 10 kg. Picosatellite being the last of the category has a wet mass of 0.1 to 1 kg. The focus of this thesis is on a particular standard of Nanosatellite called CubeSat, ranging between 1 to 2 kg which is discussed below.

1.2 CubeSats

With the aim of decreasing the development time and launch availability, in 1999 Professor Bob Twiggs of Stanford University and Dr. Jordi Puig-Suari of California Polytechnic (Cal Poly - SLO) University collaborated to create the CubeSat Standard as a means to standardize satellite buses, structures, and subsystems [15]. This standard is intended to provide access to space for small payloads. A CubeSat usually has a volume of exactly one liter (10 cm cube), weighs roughly about one kg, and typically uses commercial off-the-shelf (COTS) electronics components, usually referred to as a 1U form factor as shown in Figure 1.1 [15]. CubeSats do come with different extensions of form factors like 2U, being ten by ten by twenty cm, while weighing less than two kg and 3U as shown in Figure 1.2 [15], weighing less than three kg.

Despite their small size and limited mass, CubeSats can perform significant science missions and carry multiple payloads. University students developing CubeSats gain invaluable experience and are challenged with the same problems many aerospace engineers encounter in industry. In addition to the development challenges encountered in all spacecraft missions, meeting their small sizes, relatively small power, and mass budget warrants to be a challenging task.

Figure 1.1: 1U CubeSat form factor [15]

Figure 1.2: 3U CubeSat form factor [15]

1.3 Thesis Statement and Motivation

In a typical mission, each subsystem is designed and developed by different teams and each subsystem should be equipped with a suitable interface to connect to the system bus. Usually, the development teams are experts in their own area of R&D, but lack understanding of the data communication protocols between subsystems. This methodology of developing custom

computing systems and developing an interface in each subsystem to connect to the data bus was time consuming, complicated and had a significant need for redesign for each mission, which indeed increased the design cost in delivering each spacecraft to orbit. Also, such a design was not easily scalable and imposed difficulties to integrate and validate the tasks due to the variety of interfaces. So, to increase the scalability and flexibility in integrating the subsystems, which indeed reduces the subsystem design time and development cost, we made a transition from a mission specific centralized architecture to a mission independent distributed network switch bus architecture, as shown in Figure 1.3, with a standard Bus Interface Module (BIM). This is the proposed architecture of this thesis. This design is part of Command and Data Handling (CDH) subsystem, which is the core component of a satellite responsible for distributing commands to the whole spacecraft, data collection from different subsystems and payloads, controlling the crossbar switch and overall health maintenance of the spacecraft.

The Architecture consists of a configurable network switch to which all the subsystems are connected. The subsystems communicate with each other through the Bus Interface Module (BIM), which is designed based on SpaceWire bus protocol. The BIM also supports connecting a non-intelligent subsystem (consisting only sensors, motors, etc., without any processing elements) to the network. The interface as shown in Figure 1.4 has to be configured during the design stage for settings like bit rate, packet size, communication protocol, etc. Choosing a communication protocol which can deliver both commands and data on a single link would be an optimum solution. The crossbar switch and the BIM is implemented on an FPGA since it supports reconfiguration. This can be achieved using SpaceWire links because its communication controller can differentiate between data and commands. This interface can be implemented to support other bus protocols as well. The BIM is designed to have a standard interface to support different types of processing elements and logic components.

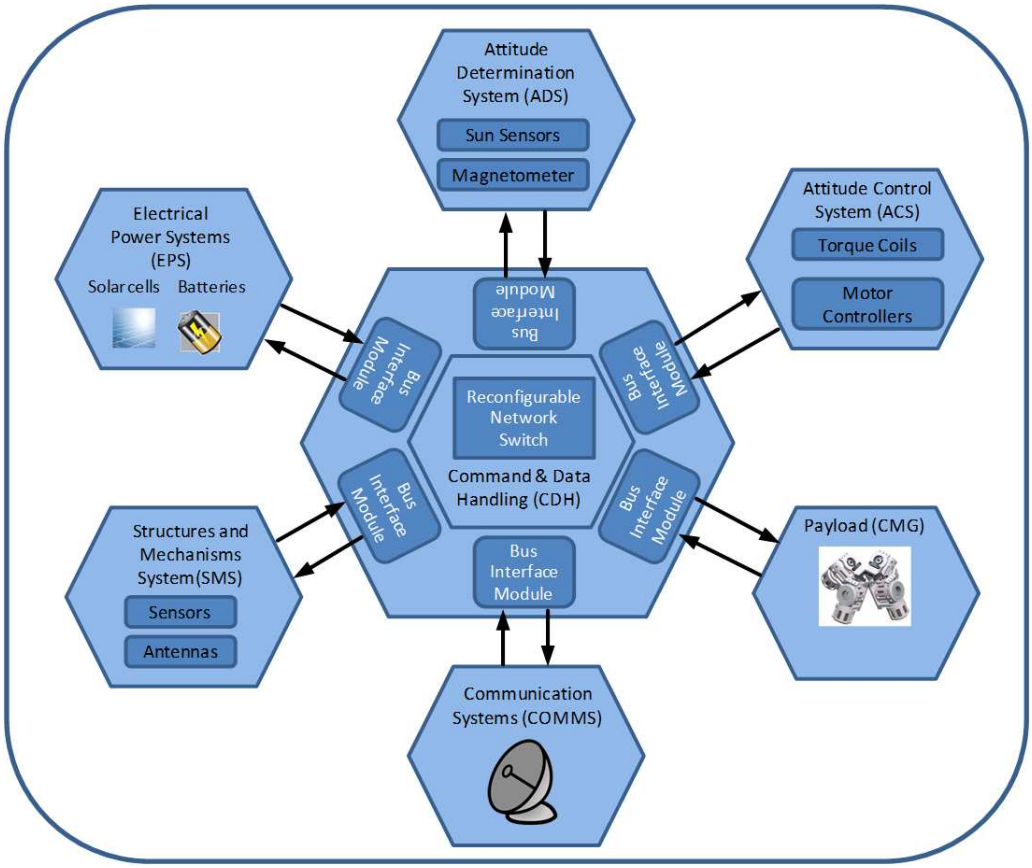

Figure 1.3: Top Level Architecture

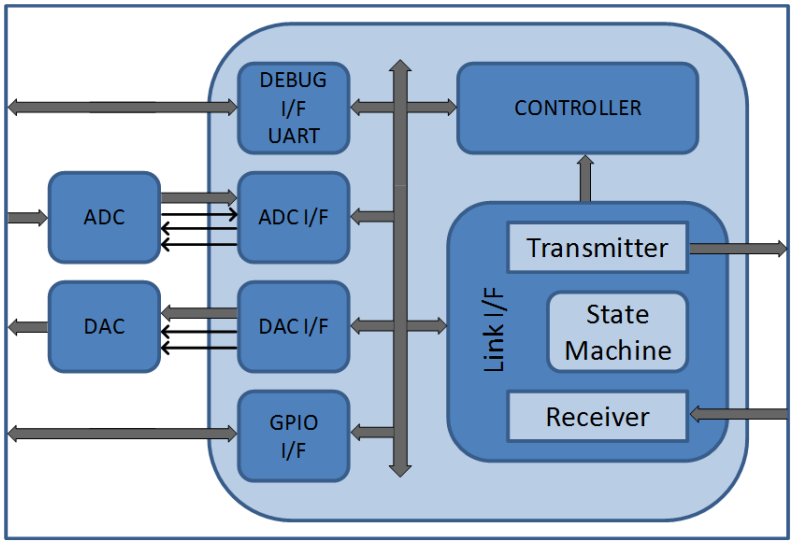

Figure 1.4: Bus Interface Module

1.4 Intellectual Merit and Broader Impact

In this thesis, we are proposing a new distributed network switch bus architecture consisting of bus interface modules and crossbar switch based on SpaceWire bus protocol. The motivation behind developing this architecture was to overcome the drawbacks of traditional centralized architectures which are discussed later in Chapter 2. This architecture decouples the individual subsystems from a centralized architecture to a distributed architecture supporting individual subsystems with their own processing elements based on the subsystem's needs. This reduces the burden of developing custom interfaces for each individual subsystem for a given mission. In the context of this thesis, distributed architecture refers to control and computational decentralization across a number of processors which may be physically located in different components or subsystems.

The broader impact of this work is that, the subsystems can now communicate with each other using the same communication protocol. Since, the design is modular and scalable, the design time and cost for future missions will be reduced significantly. With the introduction of standard bus interface modules, the system integration cost and time is significantly reduced. The architecture supports multiple simultaneous communication links between subsystems, which lacked in previous architectures. The architecture supports time synchronization by distributing system time using Time-Codes, supported by the SpaceWire bus protocol, which is essential for mission critical applications. With the introduction of run time partial reconfiguration, we have also increased the fault tolerance of the system for missions with longer life time.

1.5 Thesis Organization

This thesis is structured in the following manner. Chapter 2 presents the related architectures which were used in the previous missions and the reasons for choosing a distributed

architecture. It also discusses the different bus protocols which support the proposed architecture. Chapter 3 describes the design and implementation of the proposed architecture, Command and Data Handling subsystem in particular. We also discuss the implementation of the Bus Interface Module, based on SpaceWire design document [4] and the crossbar switch on an FPGA, which supports the proposed architecture. Chapter 4 presents the results obtained by behavioral simulation and synthesis of our designs on an FPGA. Finally, Chapter 5 concludes this work and presents the possible future work in this field.

Chapter 2

Background

In this chapter we discuss the need for distributed architecture and the available bus architectures. In the end, we also discuss and compare the available serial bus protocols including the decision to choose the best candidate that supports the proposed architecture.

2.1 Distributed Architecture

The Architecture of a system refers to the physical and logical framework used to interconnect different systems. In the context of this thesis, distributed system refer to computational decentralization across a number of processors which may be physically located in different components or subsystems. These processors may be general-purpose microprocessors or microcontrollers with data/application sharing capabilities. The architecture enables subsystems to collaborate processing needs focused on a specific task and each may be optimized to efficiently execute particular tasks or control specific subsystems. In this context, we refer each subsystem as a node, where each node is optimized to efficiently execute control commands or particular task locally. The interconnection mechanism used to connect the nodes can have a major impact on wiring size, accessibility and modularity. The different architectures which could be considered are discussed below.

2.1.1 Ring Bus Architecture

In a ring architecture, as in Figure 2.1, each node is connected to two other nodes to form a closed loop network. Communication within the network is achieved by passing a token around the network. If the packet received at a node is not intended for it, then it passes the token to the neighboring node. This architecture results in small wire harness but needs a careful design in case of a subsystem failure, which may result in communication failure. Adding a new subsystem to the node or updating a node doesn't impose much burden on the other nodes. But still, this architecture doesn't meet the performance requirements of a satellite with respect to flexibility, latency and power. The Bus interface modules need to be constantly active to check for tokens, even if it is not the intended recipient.

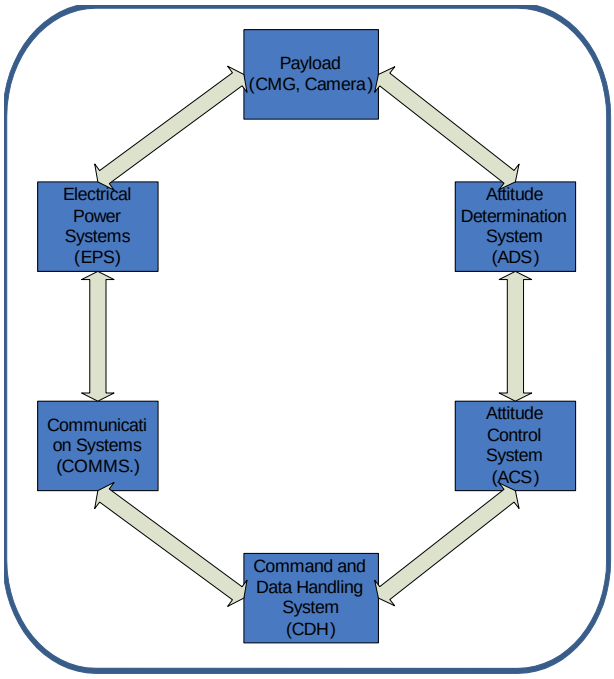

Figure 2.1: Ring Bus Architecture

2.1.2 Linear Bus Architecture

In a linear bus architecture, as in Figure 2.2, each node is connected to a common, shared data bus. This architecture usually results in a small wire harness. It is simple to add, remove or change a subsystem and doesn't have any impact on the other subsystems. A very robust communication protocol is necessary to govern the communication of data on the bus. Usually a bus arbiter is used to control the bus access requests from the subsystems. This architecture is simple, but gets hit with the performance bottleneck when there is a contention for the bus between two or more critical subsystems. This can also lead to mission failure.

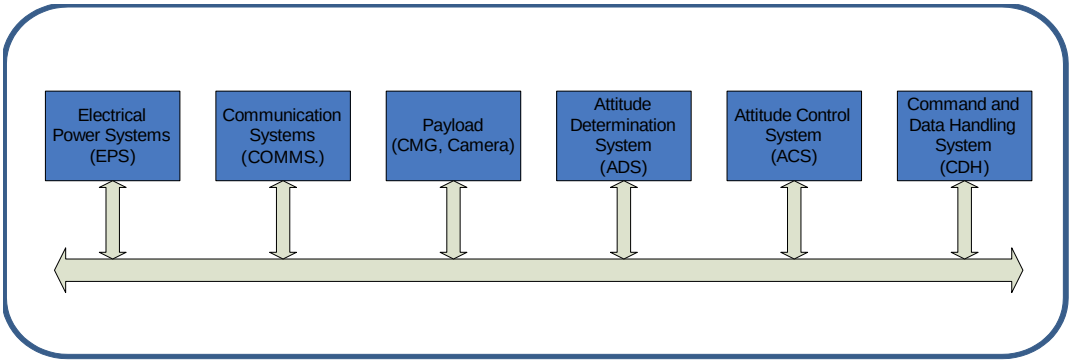

Figure 2.2: Linear Bus Architecture

2.1.3 Star Network Architecture

In a star network architecture, as in Figure 2.3, every node has a dedicated link to a central node or subsystem. Most of the traditional satellite systems used such an architecture. Each node is connected to a central node, which controls the communication between the subsystems. Each communication link between a peripheral node and the central node can use any bus protocol of their choice. So, there is a need for a large wire harness and significant software and hardware changes to the central node, in case of an addition or changes to a node in the network. However, such a network can be used in a distributed framework, if each node has its own computational capabilities.

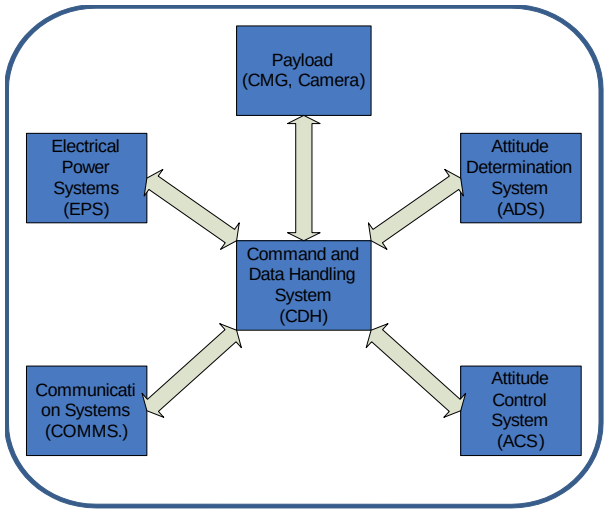

Figure 2.3: Star Network Architecture

2.1.4 Network Switch Bus Architecture

Network Switch Bus Architecture, as shown in Figure 2.4, is similar to a star network, but with less wire harness and highly modular and scalable. This architecture uses a crossbar switch as the communication bus to interconnect all of the subsystems. This architecture gives the best performance results because it can establish multiple simultaneous communication links between subsystems. Even if a communication link in the crossbar switch is broken, the packets can be routed through an alternate links. A communication link can be turned off if it is not in use, hence reducing the power utilization. The architecture is modular in the sense that the addition, removal or changes to one node doesn't have any impact on the other subsystems. This architecture can be extended to include more subsystems for future missions by simply adding more ports to the crossbar switch. This architecture best suits the distributed architecture of small satellites and hence considered in this thesis.

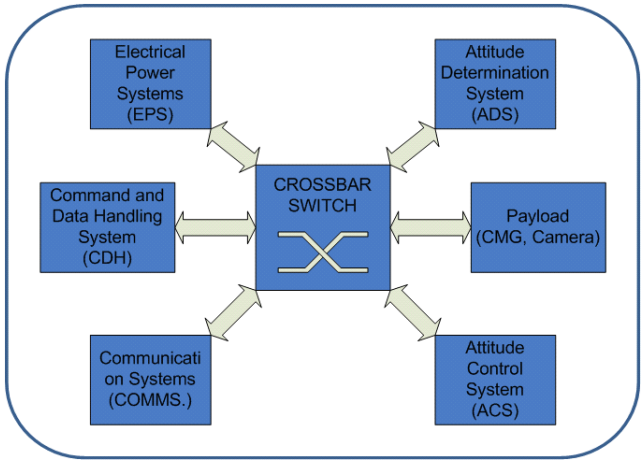

Figure 2.4: Network Switch Bus Architecture

2.2 Data Bus

The data bus is the physical infrastructure used for communication between the subsystems. Buses can be parallel buses, which carry data words in parallel on multiple wires (8-bit, 16-bit data buses), or serial buses which carry data in bit-serial form over the same link or wire. Parallel data transfers are generally faster than serial, but take up lot of real estate and pin count on a chip and needs more drivers to drive the bus, which increases the power utilization. Both require additional wiring for control mechanism and if the link is a full duplex communication, which establishes a simultaneous two way data transfer, then there is also a need for duplicating the bus. Thus, use of parallel data buses would be ruled out for intercommunication between subsystems in small satellites. However, they are used for on chip communication within the subsystems (for example, to communicate between a processing element and on chip memory or a peripheral I/O).

Typically, linear bus architecture will operate in a master-slave mode, where a single master node controls all the communications on the data bus, while the other nodes act as slaves. Contrary to this is the peer-to-peer network, where every node has equal access to the data bus. Between these two extremes lies the multi-master bus, where all the nodes have equal

rights to access the data bus unless the bus is being accessed by some other node in the network at that time. The architecture discussed in this thesis considers a peer-to-peer network, where any subsystem connected to a node of the crossbar switch can initiate a link to any other node simultaneously, thus achieving a higher data bandwidth. A comparison of bus protocols considered for on board serial communication is shown in Table 2.1. The bus protocols considered in this table are the ones which support network based bus architectures, in which communication between the nodes is based on packets containing a packet header, command or data bytes, followed by packet footer.

Table 2.1: Comparison of Serial Bus Protocols

Feature	SpaceWire	CAN	RapidIO
Description	SpaceWire is based on two existing commercial standards, IEEE–1355 and LVDS, which have been combined and adapted for use onboard spacecraft	A multi-master broadcast serial bus standard which supports distributed real time control with a very high level of security	High-performance packet-switched, interconnect technology for interconnecting chips on a circuit board, and also circuit boards to each other using a backplane
Application	Developed in Europe for use in satellites and space borne experiments	Developed by BOSCH for use in automotive electronics, engine control units, sensors, etc.	Embedded infrastructure applications like interconnecting microprocessors, memory, and memory mapped I/O devices
Maximum Data Rate	400 Mb/s	1 Mb/s	1- 2.5 Gb/s
Message Size	5 bytes overhead, data payload not limited by standard	Standard: 51 bits overhead Extended: 72 bits overhead; 0-8 bytes data payload	1 - 256 bytes of data payload with 12 - 16 bytes overhead

Table 2.1 (Continued)

Feature	SpaceWire	CAN	RapidIO
Duplex	Full	Half	Full
Message CRC (Yes/No)	No	Yes	Yes
Latency Jitter	Variable based on topology, estimated <=10 ms	<100 ms	35ns
Physical Layer Length & implementation	10 m, LVDS	30 m for 1 Mb/s, Twisted pair	30m
Physical Layer Independent (Yes/no)	No	No	Yes
COTS Test Equipment (Yes/No)	Yes	Yes	Yes
IP Available for FPGA/ASIC Implementation (Yes/No)	Yes	Yes	Yes
Software Design Tools	No	Yes	Yes
Open Specification	Yes	Yes	Yes

Comparing the bus protocols in the Table 2.1, we see that SpaceWire bus protocol provides a higher data rate with lesser packet overhead supporting full duplex communication and most important of all is the LVDS (Low Voltage Differential signaling) compatibility. Since the protocol is implemented on LVDS at the physical layer level, it provides higher noise immunity with lower voltage swings and hence, lower power consumption at high speeds. All these features makes SpaceWire the best suitable bus protocol to be considered for on-board data communication for a small satellite. We will discuss more about SpaceWire protocol and its implementation in the following sections.

2.3 SpaceWire Bus Protocol

SpaceWire is a spacecraft communication network bus protocol based on the IEEE 1355 standard of communications. It is coordinated by the European Space Agency (ESA) in collaboration with international space agencies including NASA [21], JAXA [22] and Roscosmos [23]. Within a SpaceWire network the nodes are connected through low-cost, low-latency, full-duplex, point-to-point serial links and packet switching routers. SpaceWire covers two (physical and data-link) of the seven layers of the OSI model for communications. One of the principal aims of SpaceWire is the support of equipment compatibility and reuse at both the component and subsystem levels. SpaceWire provides a unified high speed data handling infrastructure for connecting together sensors, processing elements, memory units and down link telemetry systems. Some of the vendors who supply the SpaceWire IP are SpaceWire-UK [16], DSI Technology [17], 4Links [18] and STAR-Dundee [19]. The purpose of this 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, and
- To encourage reuse of data handling equipment across several different missions.

This Standard [4] specifies the physical interconnection media and data communication protocols to enable the reliable sending of data at high-speed (between 2 Mb/s and 400 Mb/s) from one unit to another. This Standard covers the following protocol levels:

- Physical level: Defines connectors, cables, cable assemblies and printed circuit board tracks.
- Signal level: Defines signal encoding, voltage levels, noise margins, 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 for transmission over 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. It also defines how link errors and network level errors are handled.

2.3.1 Physical Level

The physical level of the standard covers cables, connectors, cable assemblies and printed circuit board (PCB) tracks. The SpaceWire cable comprises four twisted pair wires with a separate shield around each twisted pair and an overall shield. The SpaceWire connector has eight signal contacts plus a screen termination contact. SpaceWire includes specifications for running SpaceWire signals over printed circuit boards including backplanes using pairs of tracks with 100 Ω differential impedance. More details about the physical level characteristics of the SpaceWire is discussed in [4].

2.3.2 Signal Level

The signal level part of this Standard covers signal voltage levels, noise margins and signal encoding. Low voltage differential signaling or LVDS is specified as the signaling technique to use 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 the use of low voltages in practical systems. Low voltage swing means

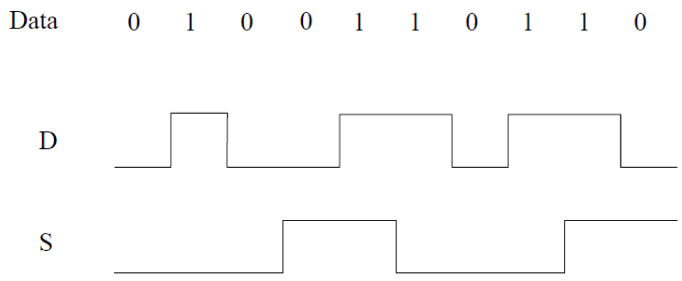

Figure 2.5: Data-Strobe (DS) encoding [4]

low power consumption at high speed. LVDS is appropriate for connections between boards in a unit, and unit to unit interconnections.

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 in Figure 2.5 [4]. 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 bidirectional link.

2.3.3 Character Level

SpaceWire takes into consideration the character level protocol defined in IEEE Standard 1355-1995 [5], but it additionally includes Time-Codes to support the distribution of system time. Time-Codes are used to synchronize all the subsystems to have the same system time which helps in performing mission critical operations. This clock synchronization service enables the synchronization of the transmitting actions of the SpaceWire nodes and ensures that communication conflicts are avoided. 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 a data character or two bits of a control character, 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 two control bits. Each control character is formed from a parity bit, a data-control flag and two control bits. 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 control codes. Two control codes are specified and valid which are the NULL code and the Time-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. A Time-Code is formed by ESC followed by a single data-character. The data and control characters are illustrated in Figure 2.6 [4].

2.3.4 Exchange Level

The exchange level protocol is a significantly more capable version than that defined in IEEE Standard 1355-1995 [5] and provides the following services:

- Initialization: Following a reset, the link output is held in the reset state until it is instructed to start and attempts 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

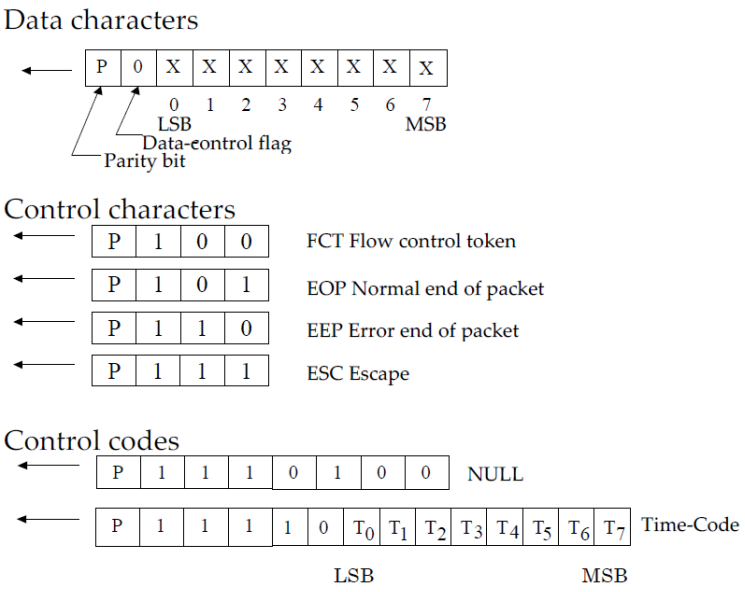

Figure 2.6: Data and control characters [4]

NULLs followed by receipt of an FCT means that the other end of the link has received NULLs successfully and that full connection is achieved.

- Flow control: A transmitter can only transmit N-Chars (normal characters, which are data characters, EOP or EEP) if there is space for them in the host system receive buffer at the other end of the link. The host system indicates that there is space for eight more N-Chars 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 N-Chars. If there is more room in the host receive buffer then multiple FCTs can 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 N-Chars.

- Detection of disconnect errors: Link disconnection is detected when no new data bit is received within a link disconnect timeout window (850 ns) following reception of a data bit. Once a disconnection error is detected, the link attempts to recover from the error

as shown in Figure 2.7 [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 character is contained in the next character. Once a parity error is detected, the link attempts to recover from the error as shown in Figure 2.7 [4].

- Link error recovery: Following an error or reset the link attempts to re-synchronize and restart using an "exchange of silence" protocol as shown in Figure 2.7 [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 disconnect and that end stops transmitting too. The first link resets its input and output for 6.4 μs to ensure that the other end detects the disconnect. The other end also waits for 6.4 μs after ceasing transmission. Each link then waits a further 12.8 μs 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 or FCT handshake to re-establish a connection and ensure proper character synchronization.

2.3.5 Packet Level

The packet level protocol follows the IEEE Standard 1355-1995 [5]. It defines how data is encapsulated in packets for transfer from source to destination. The format of a packet is illustrated in Figure 2.8 [4].

The "destination address" is a m bit data character that represent the destination identity, where $m = \log_2 n$, n being the number of ports on the crossbar switch. This list of data characters represents either the identity code of the destination node or the path that the packet takes to get to the destination node. The "cargo" is the data to transfer from source to destination. The "end of packet marker" is used to indicate the end of a packet. Two end of packet markers are defined:

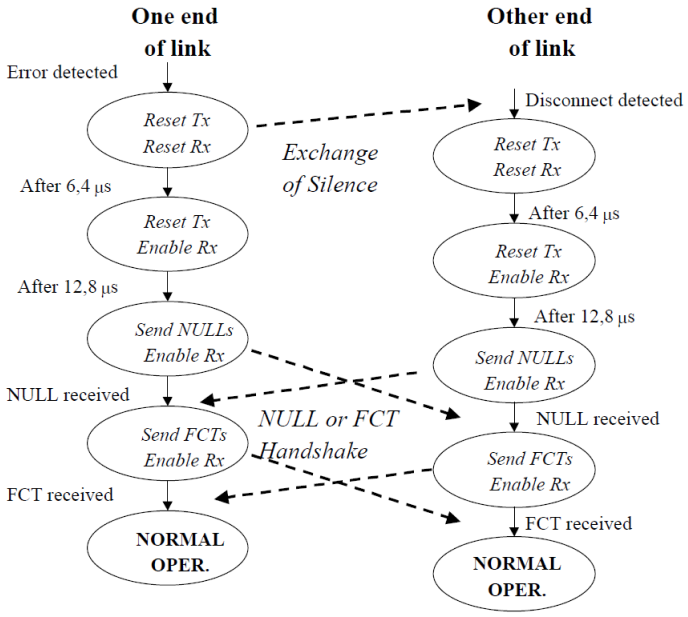

Figure 2.7: Link Restart [4]

1. EOP normal end_of_packet marker - indicates end of packet,

2. EEP error end_of_packet marker - indicates that the packet is 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. The packet level protocol provides support for packet routing via wormhole routing switches.

2.3.6 Network Level

The network level defines the components of a SpaceWire network and how the packets are transferred across it. A SpaceWire network is made up of a number of SpaceWire nodes interconnected by SpaceWire routing switches. SpaceWire nodes are the sources and destinations of packets and provide the interface to the application systems. SpaceWire nodes can be directly connected together using SpaceWire links or they can be interconnected via SpaceWire routing

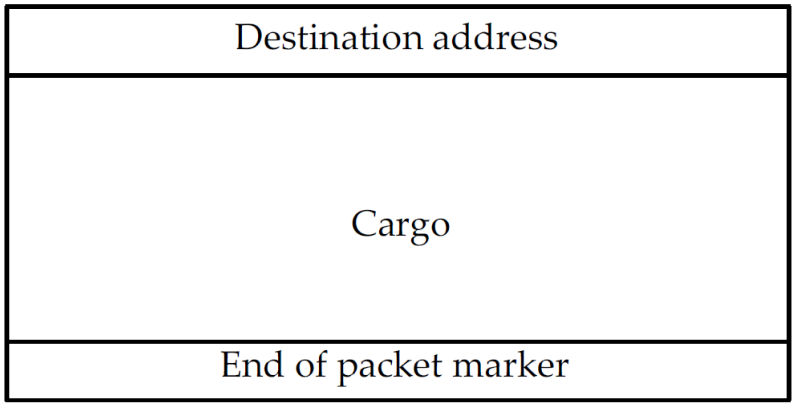

Figure 2.8: Packet format [4]

switches using SpaceWire links to make the connection between node and routing switch. A SpaceWire routing switch has several link interfaces connected together by a switch matrix, which allows any link input to pass the packets that it receives on to any link output for retransmission.

We will further discuss how this SpaceWire bus protocol is used in our architecture and the implementation details in the next chapter.

Chapter 3

Architecture Design and Implementation

The Command and Data handling (CDH) subsystem is the core component of a satellite. The main responsibility of CDH is to distribute commands to the whole spacecraft, data collection from different subsystems and payloads, and overall health maintenance of the spacecraft. The CDH subsystem mainly consists of a processing element (usually a microprocessor/microcontroller), bootstrap loader, data bus, crossbar switch, peripheral I/Os, and bus interface modules; each of which are discussed in detail in the following sections. The processing element acts as the bus master which controls the data bus with respect to individual subsystems. In critical cases, it can also perform processing tasks such as attitude and orbit control. This makes the CDH subsystem a highly complicated system with the need for high performance, while operating in a harsh environment. Traditional satellite architectures are centralized, in which every subsystem is connected to the CDH subsystem and all transactions had to pass through the CDH subsystem, which forced the designers to develop custom, often centralized architectures and hardware for satellite CDH subsystem. The architecture proposed in this thesis represents a distributed bus architecture in which the data bus contains a crossbar

switch which connects each of the subsystems using a SpaceWire Bus Interface Module (BIM). This architecture is proved to be modular, scalable and reconfigurable at the design stage for any given mission. Each subsystem becomes an interchangeable module which corresponds to standard signaling and command interfaces. The only differences between the subsystems are the commands and the internal actions performed by the subsystem. The subsystems can now communicate with each other using the same communication protocol. This architecture supports shared memory module to store and access science data for the mission. It also supports additional processing elements and DSP farms as a backup for complex processing requirements. A communication link between any two subsystems is established by initiating a link through the switch. Once the link is successfully established with appropriate handshakes between the subsystems, data transfers are initiated until the link disconnect request is received. The architecture supports multiple simultaneous communication links between different subsystems. Partial dynamic reconfiguration techniques can be implemented to reconfigure the network switch as a technique to mitigate radiation effects. The BIM designed in this thesis, which currently supports only SpaceWire communication links could be extended to support other network bus protocols (CAN, TTCAN, FlexRay, etc.) for future missions if needed.

3.1 Architecture Implementation

An example of the proposed distributed architecture is shown in Figure 3.1. The architecture was designed to support six subsystems, which can contain the subsystems of a typical small satellite, such as Guidance and Navigation (GNC), Electrical Power Systems (EPS), Communication Systems (Comms.), Command and Data Handling System (CDH), Attitude Determination System (ADS) and Payload. The architecture is based on Network Switch Bus Architecture discussed in section 2.1.4. SpaceWire serial data bus protocol is used for communication between subsystems with high data bandwidth requirement such as payloads, GNC, CDH and ADS. The I^2C protocol is used for other subsystems which need lesser data bandwidth

such as EPS and Communication systems. In this thesis, we concentrate only on CDH subsystem with SpaceWire data bus protocol which supports the network switch. To understand the implementation of the architecture, we can consider the subsystems to be a black-box with a Bus Interface Module (BIM) connected to the processing element of the subsystem. Whenever a subsystem needs to communicate to another subsystem, it tries to establish a link between the two subsystems using the BIM. For all practical purposes, we can consider that the communication link is always between the BIMs, since the underlying communication protocol is the same and independent of the subsystems. This makes the architecture modular and scalable, because interchanging the subsystems will not affect the implementation of the architecture.

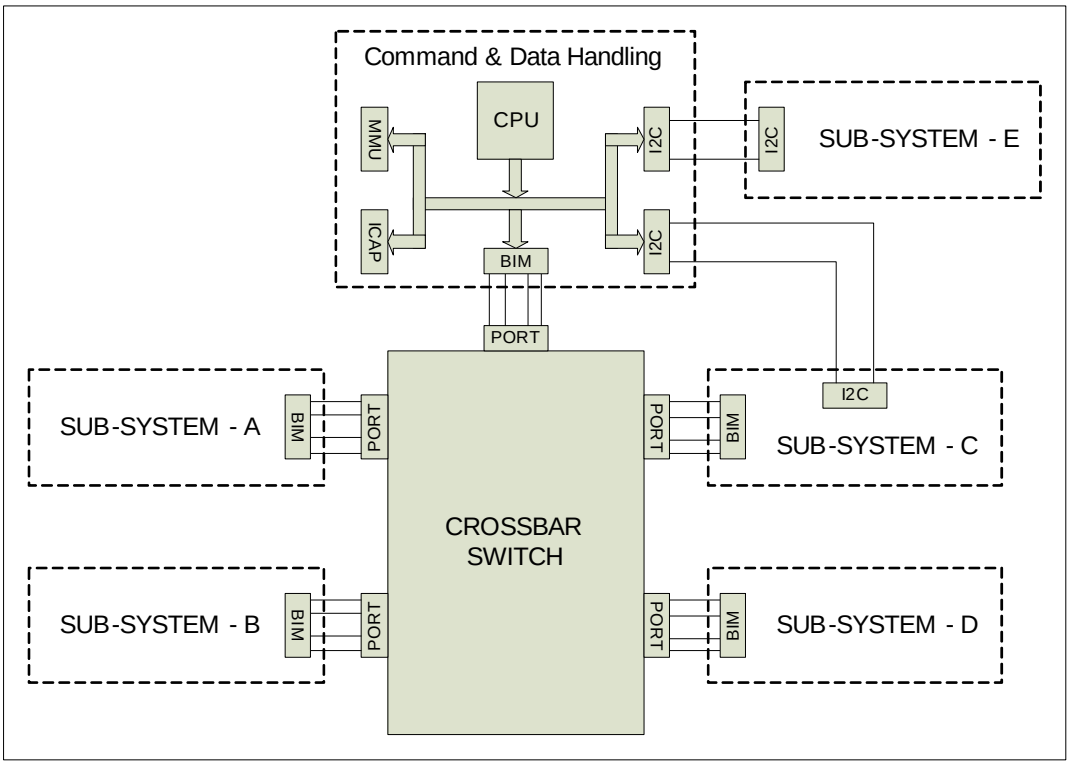

Figure 3.1: Example Implementation of Top Level Architecture

Let us now understand the steps involved in establishing a communication link between any two subsystems. For example, let's assume that subsystem A is establishing a link with

subsystem B.

1. The Processing Element (PE) of subsystem A asserts the link enable signal on the BIM and places the address to which it is trying to communicate (subsystem B in this example) on the bus connecting the BIM.

2. The BIM on the side A sees the link enable signal and hence enables the port on the Crossbar switch to which it is connected. Once the port is enabled, the BIM serializes the address value (seen on the parallel bus, input side of A) based on SpaceWire bus protocol standard, and sends it to the port on the crossbar switch.

3. The port on the crossbar switch reads the address and registers it into its configuration register file, which then enables the destination port and hence the BIM (subsystem B, in this case). At this stage, a link is said to be established between the two subsystems.

4. Now, to check the correctness of the link established, the BIM on the source side (A), sends a NULL character and waits until it receives a NULL back from the destination (B), as an acknowledgement. Meanwhile a similar NULL character will be sent from the destination side (B), as soon as its BIM was enabled by the crossbar switch.

5. Upon successfully exchanging NULLs between the two subsystems, both ends exchange a sequence of flow control tokens (FCTs), the count of which indicates the number of data characters a BIM can accept.

After successfully completing all the above steps, the PEs on either side can now transmit either data or time codes based on the requirement. Once the required operation is completed, the source can disable the link by asserting an active low signal on link enable which would disable the link between the two ports and also clears the configuration register file in the crossbar switch to make sure that other links could be established in future by different ports.

If there are n ports on the crossbar switch, a maximum of $n/2$ simultaneous communication links can be established between different subsystems. Each link is mutually exclusive, which

means that all the links are independent of each other. Now, let's look at some of the issues which were looked upon when designing the bus architecture. If two ports try to establish a link simultaneously to the same destination port, then the one with higher priority is given preference to establish the link. The source which was denied the link, will store the data to be transferred on a queue at the BIM of the source side. Once the link gets freed and established, it will transfer the data to the destination BIM. If a link gets disconnected in the middle of a data transmission, then a new link is re-established following the steps mentioned above. The next section investigates further on the design and implementation of the BIM.

3.2 Bus Interface Module Implementation

This section describes the implementation of a link interface, part of Bus Interface Module which connects a host data bus on one end and the SpaceWire crossbar switch on the other end. The link interface is made up of an Encoder-Decoder and State Machine as shown in Figure 3.2. All these specifications abide by the SpaceWire design document [4].

3.2.1 Encoder-Decoder

The Encoder consists of transmitter and transmit clock generator, while the decoder is made up of receiver and receive clock recovery as shown in Figure 3.2.

3.2.1.1 Transmitter

The transmitter is responsible for encoding and transmitting data using the Data-Strobe (DS) encoding technique. It receives N-Chars (data, EOP or EEP) or Time-codes for transmission from the host system. If there is neither a Time-Code, FCT nor an N-Char to transmit, the transmitter sends NULL. The transmitter sends N-Chars only 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, showing that it is ready to accept another 8 N-Chars. The

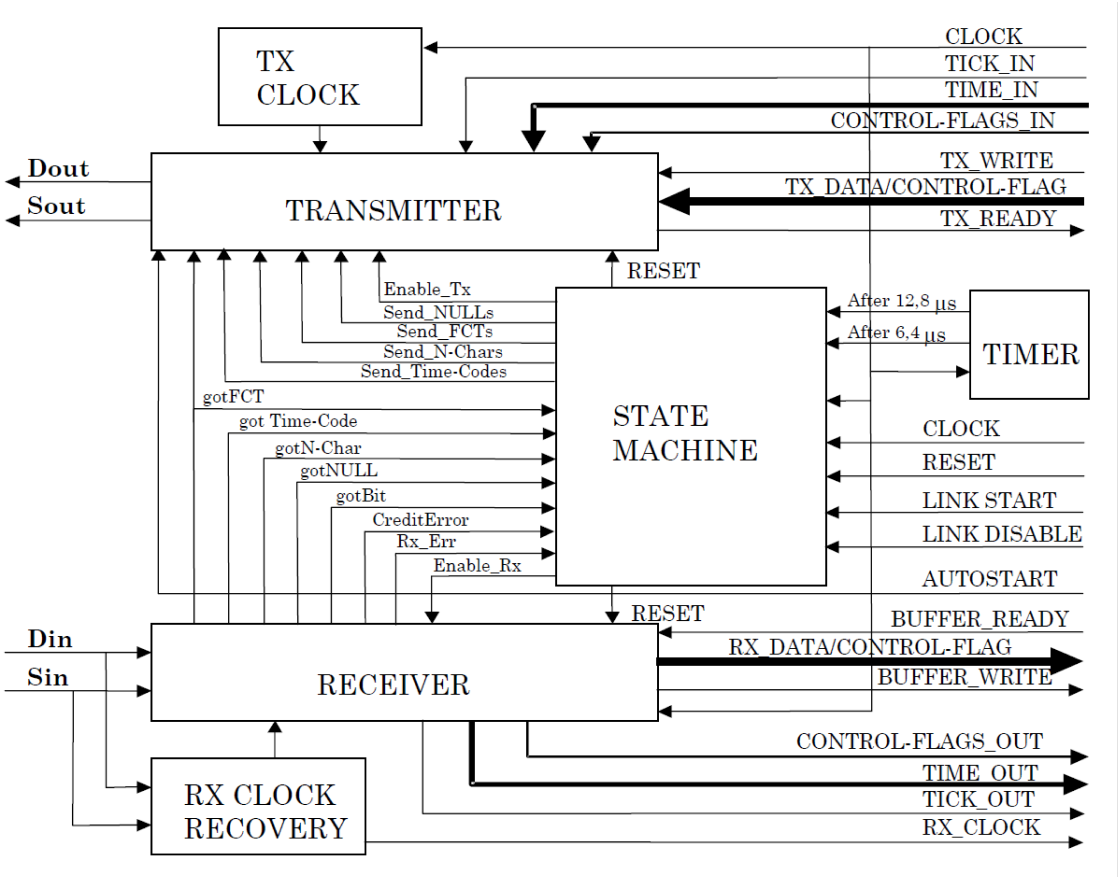

Figure 3.2: SpaceWire link interface block diagram [4]

transmitter is responsible for keeping track of the FCTs received and the number of N-Chars 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 can be in one of four possible states:

1. Reset: The transmitter does nothing.

2. Send NULLs: It can only send NULL on the link. It does not read N-Chars from the transmit host interface. It does not accept an order to send FCT from the host system. It does not send Time-Codes.

3. Send FCTs or NULLs: It sends FCT or NULL but still does not read N-Chars from the transmit host interface. It does not send Time-Codes.

4. Send Time-Codes, FCTs, N-Chars or NULLs: Normal behavior, sending NULLs, FCTs, Time-Codes and N-Chars.

The change of state is controlled by a state machine.

The transmitter is also responsible for sending FCTs whenever the local host system has space to receive eight more N-Chars. The host system requests the transmitter to send FCTs when it has room for at least eight more N-Chars that have not already been reserved for data by a request from a previously sent FCT. The transmitter can only send an FCT when it is in state “Send FCTs or NULLs” or “Send Time-Codes, FCTs, N-Chars or NULLs”.

The transmitter can only send Time-Codes in state 4, when the TICK_IN signal is asserted and it has finished sending the current character or control code. The value of the Time-Code is the value of the TIME_IN and CONTROL-FLAGS_IN signals at the point in time when TICK_IN is asserted.

A typical interface between the host system and the transmitter comprises TX_READY, TX_WRITE and TX_DATA as shown in Figure 3.2. When the transmitter is ready to receive another N-Char from the host system, it asserts the TX_READY signal. When the host system

has an N-Char to transmit and the TX_READY signal is asserted it may put the N-Char onto the TX_DATA lines and assert the TX_WRITE signal. When the transmitter has registered the N-Char data it de-asserts the TX_READY signal.

3.2.1.2 Transmit Clock

The transmitter can operate at any data signaling rate from the minimum (2 Mb/s) to the maximum possible (400 Mb/s). The transmit clock is responsible for producing the variable data signaling clock signals used by the transmitter. The transmit clock signals are typically derived by dividing down the local system clock or a phase locked loop multiple of the local system clock.

3.2.1.3 Receiver

The receiver is responsible for decoding the DS signals (Din and Sin) to produce a sequence of N-Chars (data, EOP, EEP) that are passed on to the host system. It also receives NULLs, FCTs and Time-Codes. 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 informs the transmitter so that it can update its credit count accordingly. All other control characters received are flagged to the state machine. The receiver ignores any N-Chars, L-Chars, parity errors or escape errors until the first NULL is received. The disconnection detection mechanism within the receiver is enabled as soon as the first bit arrives (i.e. first transition detected on D or S inputs to receiver).

Time-Codes hold system time information. A valid Time-Code causes the assertion of the TICK_OUT signal from the receiver. The value of the Time-Code is placed on the TIME_OUT and CONTROL_FLAGS_OUT outputs when the TICK_OUT signal is asserted. These signals are used by the host system to update or regulate its system clock. The receiver can be in one of four states:

1. Reset: The receiver does nothing.

2. Enabled: The receiver is enabled and is waiting for the first bit to arrive.

3. GotBit: The receiver has received the first bit (First Bit Received) and disconnect error detection is enabled. The receiver is enabled to listen for NULLs only.

4. GotNULL: The receiver has received a NULL and is enabled to receive NULLs, FCTs, Time-Codes and N-Chars. Disconnect, parity and escape error detection is enabled.

The change of state from Reset to Enabled is controlled by the state machine. The receiver is responsible for receiving FCTs from the other end of the link and for passing these FCTs on to the credit counter in the transmitter. A typical interface between the receiver and the host system comprises BUFFER_READY, BUFFER_WRITE and RX_DATA. When the host system is ready to receive another N-Char from the receiver it asserts the BUFFER_READY signal. When the receiver has received an N-Char and the BUFFER_READY signal is asserted it puts the N-Char onto the RX_DATA lines and assert the BUFFER_WRITE signal. When the host system has registered the N-Char data it de-asserts the BUFFER_READY signal. If an N-Char is received and the BUFFER_READY signal is not asserted then a credit error has occurred.

3.2.1.4 Receive Clock Recovery

The receive clock (RX_CLOCK) is recovered by simply XORing the received Data and Strobe signals together. The receive clock recovery circuit provides all the clock signals used by the receiver with the exception of the local clock signal use for disconnect timeout.

3.2.2 State Machine

The state machine controls the overall operation of the link interface. It provides link initialization, normal operation and error recovery services. The operation of the state machine

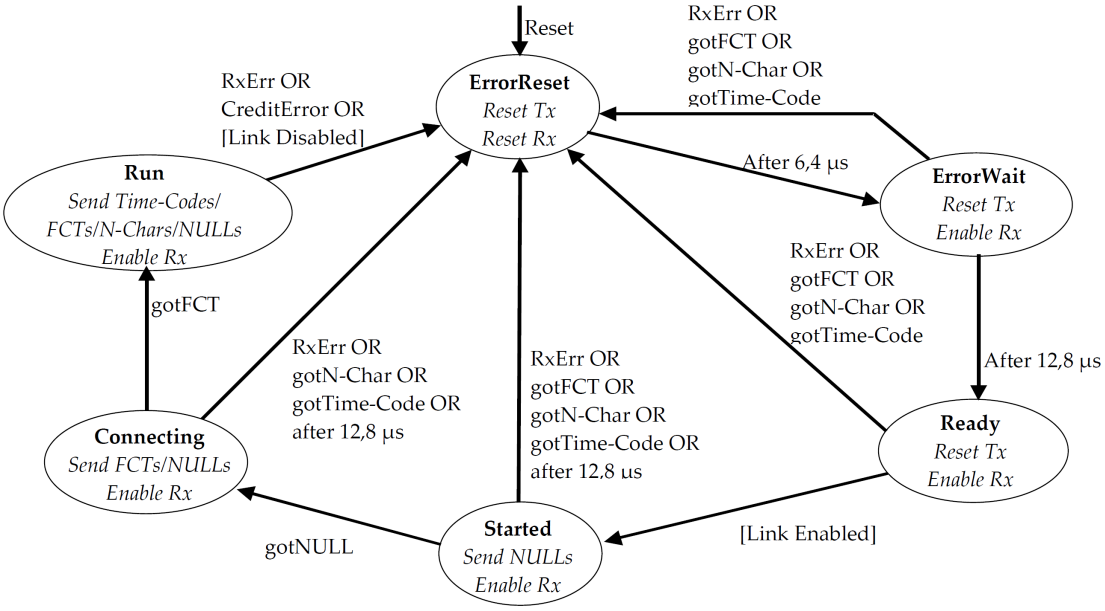

Figure 3.3: State diagram for SpaceWire link interface

is described in the form of a state diagram as shown in Figure 3.3.

The different states in the state diagram are explained below. The state transitions happen only when the conditions governing them are met.

1. ErrorReset

   - The ErrorReset state shall be entered after a system reset, after link operation is terminated for any reason or if there is an error during link initialization.
   - In the ErrorReset state the Transmitter and Receiver shall all be reset.
   - When the reset signal is de-asserted the ErrorReset state shall be left unconditionally after a delay of 6.4 µs (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 de-asserted.

2. 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 reset.
   - If a NULL is received then the gotNULL condition shall be set.
   - The ErrorWait state shall be left unconditionally after a delay of 12.8 μs (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 gotNULL condition is set, 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.

3. Ready

   - The Ready state shall be entered only from the ErrorWait state.
   - In the Ready state the link interface is ready to initialize as soon as it is allowed to do so. The receiver shall be enabled and the transmitter shall be reset.
   - If a NULL is received then the gotNULL condition shall be set.
   - 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 a parity error or escape error occurs, or any character other than a NULL is received after the gotNULL condition is set, , then the state machine shall move to the ErrorReset state.

4. Started

- The Started state shall be entered from the Ready state when the link interface is enabled.
- When the Started state is entered a 12.8 μs (nominal) timeout timer shall be started.
- In the 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.
- In the Started state the sending from the transmitter of at least one NULL shall be requested before moving to the Connecting state.
- If, while in the Started state, a disconnection error is detected, or if a parity error or escape error occurs, or any character other than a NULL is received after the gotNULL condition is set, , then the state machine shall move to the ErrorReset state.
- If the 12.8 μs timeout expires (i.e. no NULL received since leaving the ErrorReset state) then the state machine shall move to the ErrorReset state.

5. Connecting

- The Connecting state shall be entered from the Started state after a NULL is received (gotNULL condition set).
- On entering the Connecting state a 12.8 μs 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 (gotFCT condition true) the state machine shall move to the Run state.

- If, while in the Connecting state, a disconnect error, parity error or escape error is detected, or if any character other than NULL or FCT is received, then the state machine shall move to the ErrorReset state.
- If the 12.8 μs timeout occurs then the state machine shall move to the ErrorReset state.

6. Run

- The Run state shall be entered from the Connecting state.
- If the link interface is disabled, or if a disconnect error, parity error, escape error or credit error is detected, while in the Run state, then the state machine shall move to the ErrorReset state.

3.3 Processing Elements

Processing Elements are the core components of any subsystem. The main role of a processing element in a CDH subsystem is to distribute commands to different subsystems, health monitoring of the subsystems and the data bus (crossbar switch in our architecture) and also to support data processing. Processing Elements can be custom designed or can be chosen from COTS, which can be implemented as an ASIC or an FPGA. The advantages of choosing Field Programmable Gate Arrays (FPGAs) for use in small satellite on-board systems are the flexibility of design and debug, shorter time-to-market, lower cost, reconfigurability, etc. High-density FPGAs are becoming the preferred implementation platform in a number of terrestrial applications previously dominated by application specific integrated circuits (ASICs). So far, high-density FPGAs have mostly been used in payload systems of small satellites, however introduction of radiation hardened versions and on board partial dynamic reconfiguration techniques offered by leading manufacturers such as Xilinx and Actel paves the way for their use as the main computing device. Soft intellectual property (IP) cores written in the hardware

description language (VHDL and Verilog) are used to build the on-board computer. A few general design metrics to be considered in choosing a particular IP core for a small satellite are listed below

1. Low power Consumption
2. High Performance
3. Small size
4. High reliability
5. Reconfigurability
6. Low cost
7. Radiation tolerance

There are many vendors who provide IPs which meets the above mentioned design metrics. The decision to choose a particular IP is mission dependent. In this thesis we decided to choose the MicroBlaze IP core provided by Xilinx [6]. The MicroBlaze core is a 32-bit RISC Harvard architecture soft processor core with a rich instruction set optimized for embedded applications. A rich assortment of IP cores for MicroBlaze gives us an unprecedented amount of flexibility in building the subsystem. The MicroBlaze IP core is fully supported by the Embedded Development Kit (EDK) including a GNU compiler provided by Xilinx. The MicroBlaze Configuration Wizard guides us through selecting configuration parameters and provides estimates instantly on frequency, performance and area. Another salient feature in choosing the Xilinx FPGA is the availability of partial dynamic reconfiguration of individual components of a subsystem on the fly.

3.4 Memory

Program code must be stored in some kind of non-volatile memory that is very unlikely to have SEU problems. Two schemes for program code storage can be considered:

- All program code is stored in the same non-volatile memory
- A special boot-loader program is stored in a more SEU-secure memory. This program then loads user program code from another (possibly less SEU-secure) memory, from another satellite unit or from ground control.

The first alternative has the advantage that only one memory chip is needed. The second has the advantage that the boot code can be stored in one (small) highly reliable radiation tolerant memory such as a PROM, and the rest of the code in cheaper (large) memory such as flash memory.

3.5 Radiation Effects

A brief discussion of sources and problems caused by radiation are discussed in this section. The main source of radiation for small satellites in space is the trapped radiation, which is mainly electrons and protons trapped in the Earth's magnetic field. The Earth's magnetic field is not uniformly distributed. It is split into two distinct belts, with energetic electrons forming the outer belt and a combination of protons and electrons creating the inner belt. The energy levels of these particles are up to 0.1–10 MeV for electrons on the outer belt and 50-100 MeV for protons in the inner belt. The inner Van Allen Belt extends from an altitude of 100–10,000 km. Since, the small satellites are in the LEO orbit (160 - 2,000 km), we need to address these issues if the mission duration is long. More detailed analysis on these radiation effects can be found in the [7].

3.5.1 Total Ionizing Dose (TID)

Since some radiation effects are slowly accumulating (e.g. charge trapping) the semiconductor device will perform well enough to allow the system to work, until a certain amount of radiation has been absorbed. In the case of charge trapping in the gate oxide of MOS transistors, the transistor threshold voltage changes, resulting in higher leakage currents and overall higher power consumption. TID is something that has to be destructively tested for each device model and manufacturer. Testing chips for TID is an expensive and time-consuming process. Fortunately NASA and a few space-related companies occasionally tests new chips and their test reports are public [8].

3.5.2 Single Event Latch-up (SEL)

Parasitic PNP and NPN transistors in a CMOS chip form thyristors (PNPN junctions) in the substrate, as shown in Figure 3.4. Normally these thyristors are switched off, but they may get triggered and start conducting (latch up) by ionizing radiation. More specifically it is the inadvertent creation of a low-impedance path between the power supply rails of a MOSFET circuit, triggering a parasitic structure which disrupts proper functioning of the part and possibly even leading to its destruction due to overcurrent. A power cycle is required to correct this situation.

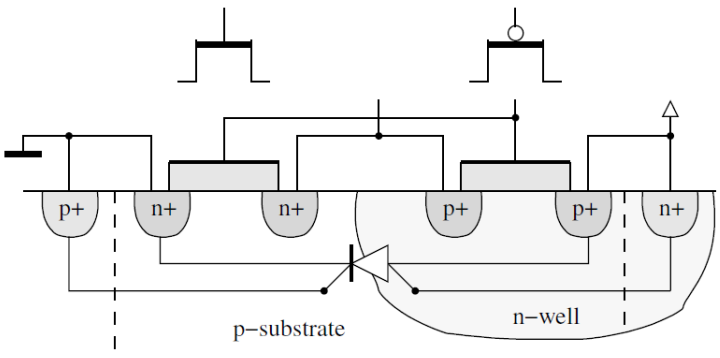

Figure 3.4: Single Event Latchup [9]

3.5.3 Single Event Upset (SEU)

A single event upset (SEU) is a change of state caused by ions or electro-magnetic radiation striking a sensitive node in a micro-electronic device. The state change is a result of the free charge created by ionization in to an important node of a logic element (e.g. memory cell). Figure 3.5 shows the nodes in an SRAM cell, where charge injection can lead to SEU. This is a non-destructive phenomenon, and the device can be reset to its previous state (e.g. the original value can be written back to a memory cell).

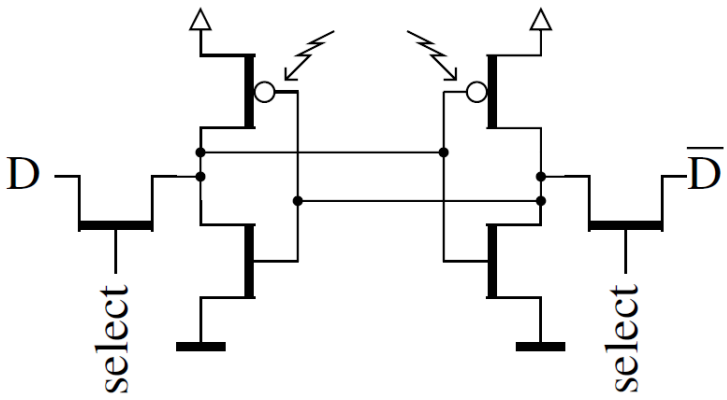

Figure 3.5: Single Event Upset in SRAM Cell [9]

The charge needed to switch the logic state is dependent on the physical dimensions of the transistors involved in the bi-stable device. As the memory chips get more and more dense, the individual transistors becomes smaller and smaller and consequently the critical charge gets smaller, resulting in higher single event upset risk. In combinatorial logic SEUs take the form of single event transients (SET), where the output for a short period of time shows the wrong value, but later returns to the normal state.

The ideal FPGA is a high-density SRAM-based that is not susceptible to SEU. As for TID, 300 krad (Si) is sufficient for the great majority of missions. High SEU tolerance is essential for FPGAs used in critical applications: a threshold of at least 37MeVcm^2/mg for upsets in both

configuration memory (SRAM - based FPGAs) and user flip-flops and registers.

3.6 Radiation Mitigation Techniques

Three different radiation mitigation techniques are discussed in this section. Xilinx chips are chosen based on the radiation tolerance, high gate density, on-board RAM, and large I/O count offered by the Virtex family. On-board reprogrammability of the FPGAs allows for design changes and updates right up to launch time, allowing the design team to meet demanding schedules.

3.6.1 Redundancy

The required redundancy can be of different levels, for example:

- Triple Modular Redundancy (TMR) hardware: One method of increasing a design's resistance to SEU effects is by implementing the design using TMR, which may have up to 3.2× the gate count and a performance cost of approximately 10%. Xilinx SRAM-based FPGAs require the designer to implement TMR for user logic. Xilinx has recently released their TMR tool (XTMR). TMR is implemented by creating three identical copies of a module and feeding their outputs into a majority voter, which simply outputs the most popular of the three outputs. Thus if one of the three modules fails and produces an incorrect result, the majority voter will still output the correct result since the other two modules' outputs agree. TMR will not fail unless two of the modules have failed.

- Spare unit: Two (or more) identical units are available. In case one breaks, the other unit can be powered up and perform the task, after initialization.

Having a spare unit is often a good choice, as that doesn't add much design complexity and the two units can be tested independent of each other. TMR can protect against failures in the user-defined memory (flip-flops, etc.), which scrubbing alone is unable to do.

3.6.2 Scrubbing

The periodic refresh of the FPGA configuration memory is called configuration scrubbing (or simply, scrubbing). Scrubbing can be performed periodically to ensure that a single upset is present, no longer than the time it takes to refresh the entire FPGA configuration memory. Alternatively, the configuration bitstream may be read and compared to a golden copy and the configuration refresh only done when an error in the bitstream is detected. This is the preferred method as reading the configuration memory is faster than writing to it. This procedure is called readback with compare, and is also often referred to as scrubbing.

Although scrubbing ensures that the configuration bitstream is free of errors, there is a period of time between the moment the upset occurs and the moment when it is repaired in which the FPGA configuration is incorrect. Thus the design may not function correctly during that time. To completely mitigate the errors caused by SEUs, scrubbing must be used in conjunction with another form of mitigation that masks the faults in the bitstream. The most popular of these techniques is triple modular redundancy (TMR). TMR is a well-known technique that masks any single-bit fault in the configuration bitstream. Combined with scrubbing, which is used to ensure that no more than one configuration bit is upset at any point in time, TMR can completely mask the effects of SEUs.

3.6.3 Partial Dynamic Reconfiguration

A fairly new concept developed by Xilinx [10], a process of configuring a portion of a FPGA while the other part is still running/operating can be used to reconfigure only the part which is affected by SEUs. This technique is similar to scrubbing, in which an error is detected by reading the contents of the configuration bitstream and compared to a golden copy of the configuration and reconfigure only the modules whose configuration bits are in error. This technique has an advantage in which reconfiguration of a portion of the FPGA will not affect the currently operating modules on the same FPGA. Module-based partial reconfiguration permits

to reconfigure distinct modular parts of the design. To ensure the communication across the reconfigurable module boundaries, special bus macros ought to be prepared. It works as a fixed routing bridge that connects the reconfigurable module with the rest part of the design. Finally for each reconfigurable module of the design, separate bit-stream is created. Such a bit-stream is used to perform the partial reconfiguration of an FPGA.

The obvious downside to TMR is the area cost. A design with full TMR applied will consume at least three times the area of the original circuit. TMR may cause a triplication of the total power consumed by the design. Partial dynamic reconfiguration implementation technique neither increases area nor power of the original design compared to TMR.

Chapter 4

Results and Analysis

In this chapter we present the results obtained by synthesizing the implementations presented in Chapter 3. The BIM and the crossbar switch was designed using Verilog and implemented on Xilinx Virtex 4 FPGA XC4VSX35. This particular FPGA is suitable for high performance solutions for DSP applications. Virtex-4 devices are produced on a state of the art 90-nm copper process, using 300 mm (12 inch) wafer technology which can operate at 1.2V. It also supports readback capability of bitstreams, which is useful for partial dynamic reconfiguration. Combining a wide variety of flexible features, the Virtex-4 family enhances programmable logic design capabilities and is a powerful alternative to ASIC technology. A complete behavioral simulation of the design was performed using Xilinx ISE 11.1 suite. The design is modular, in which a BIM can be instantiated to represent each subsystem as required for a mission, and the crossbar switch can be configured to have the required number of ports by setting the parameters during the design stage. For simulation purposes, we designed a four port crossbar switch connecting four different BIMs representing four subsystems in a network, shown in Figure 4.1. All possible communication links between the nodes were simulated and verified. A detailed analysis of each of them is discussed below.

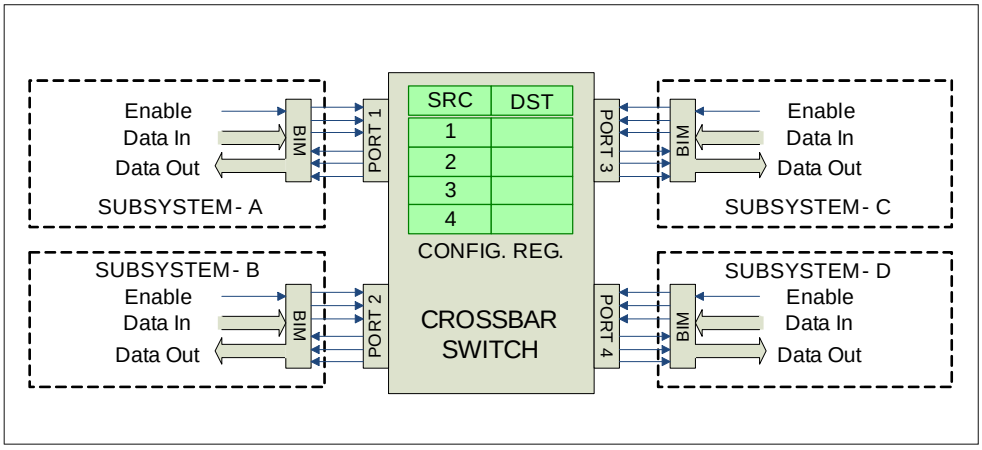

Figure 4.1: A switch network example

4.1 FPGA Implementation and Design Flow

The ISE Design Suite features include design entry and synthesis supporting Verilog or VHDL, place-and-route (PAR), completed verification and debug using ChipScope Pro tools, and creation of the bit files that are used to configure the chip. We have used the latest Xilinx ISE 11.1 suite for this work. The different stages of ISE design flow are:

- Design Entry

  The design was implemented in Verilog using the ISE Text editor after creating a new project and choosing a right FPGA device and package.

- Behavioral Simulation

  We then perform a behavioral simulation of our Verilog hardware code to confirm the functional correctness of our design. The waveforms can be viewed in the graphical waveform viewer. The design can be simulated at any time during the design process to verify design functionality.

- Design Synthesis

  Xilinx ISE suite comes with XST synthesis tool. The synthesis process checks for syntax

errors in the code and analyzes the hierarchy of the design. It then creates a netlist (.ngc) of primitive gates that contains the logical data and constraints. This could be implemented for any Xilinx FPGA device. We can specify the timing constraints; namely clock period, clock-to-pad and pad-to-clock, in the User Constraints file (.ucf file) for our design.

- Design Implementation

  The design is implemented using the following steps:

  - Translate: The translate process runs NGDBuild that merges all the input netlists (the logical design data) and the design constraints information to create the ngd design file.
  - Map: The map process maps the logic defined in the ngd file on the target FPGA. The output ncd file contains the information of the resources such as the CLBs, registers and IOBs consumed on the target FPGA.
  - Place and Route: Once the design is mapped, the design is then placed and routed (PAR) on the FPGA. This means that the resources described in the .ncd file are assigned specific locations on the FPGA. The connections between the resources are then mapped onto the FPGA's programmable interconnect network. The PAR process has a large impact on the speed of the design and therefore it generates an updated ncd file with the static timing report. This report gives us the information for the clock frequency, the slack, whether there are any timing violations and the clock-to-pad and pad-to-clock delays.

The process of hardware implementation of the design on FPGA is completed if there are no errors and the timing constraints are met. The design summary generated after the complete process contains information of the resource utilization for the target FPGA. We will discuss about the simulation results in the following sections.

4.2 Functional Verification

Verification is a process used to demonstrate the functional correctness of a design. The main purpose of "functional" verification is to ensure that a design implements intended functionality. The testbench environment was developed using Verilog and simulations were done on Xilinx ISE simulator. A verification plan was developed with a list of features which need to be verified as per the design document. Verification was performed at block level and at system level.

A list of functions which were verified at the top level of the design are:

1. Reset logic for the BIM.
2. Link enabling of the BIM.
3. Crossbar switch configuration.
4. Exchange of Nulls and FCTs.
5. Data and Time code transfers.
6. Link Disconnection.

At the block level/unit level each block of the BIM was verified to meet the design specifications. The different test scenarios considered to verify the blocks are listed below.

1. Transmitter Block:
   (a) Check the functionality of Enable_Tx signal.
   (b) Check the response of the transmitter for different inputs from the FSM like Reset, Send_NULLs, Send_FCTs, Send_N-Chars, Send_Time-Codes.
   (c) Check the response for gotFCT input from Receiver.
   (d) Check the functionality of the outputs Dout and Sout for different input scenarios mentioned above.

(e) Check for data consistency between the input and output of the block.

2. Receiver Block:

   (a) Check for correctness in the clock recovery block.

   (b) Check the response for Enable_Rx and Reset inputs from the State Machine.

   (c) Check the outputs of the receiver block to verify whether it is decoding the character received and informing the state machine with appropriate enable signals.

   (d) Check for correctness of the Data/Time-Codes sent to the host with respect to the data received.

3. State Machine:

   (a) Check the response for the global Reset and Link Enable signals.

   (b) Drive appropriate inputs to check if all the states are covered.

   (c) Check the correctness of state transitions based on appropriate conditions for transition.

   (d) Check for different input combinations which can drive the state machine into a wrong state.

   (e) There are 6 states in the state machine and 6 different inputs to the state machine from the receiver, which accounts for 36 scenarios to be checked.

4.3 Behavioral Simulation

4.3.1 Link Establishment

A link is established by first enabling the BIM on the source side, followed by placing the destination address on the input bus of BIM. The BIM then enables the port on the switch to which it is connected and configures the crossbar switch based on the destination address

received. It then enables the BIM on the destination port. Once the physical link is established, a series of handshakes between the two BIMs is initiated to check the correctness of the link. A snapshot of the simulation results is shown in Figure 4.2. We can see that, when the link_start signal is high and reset is low, the BIM sends a link_enable signal to the ports connected (link_en_1 and link_en_3 in the Figure), which then writes the crossbar switch register file, reg_sw with the address presented on data_i bus, respectively. In this example, port 1 is connected to port 4 and port 2 is connected to port 3. We can also observe that once the switch is configured it enables the destination ports by driving the corresponding signals (T2 and T4 in this example).

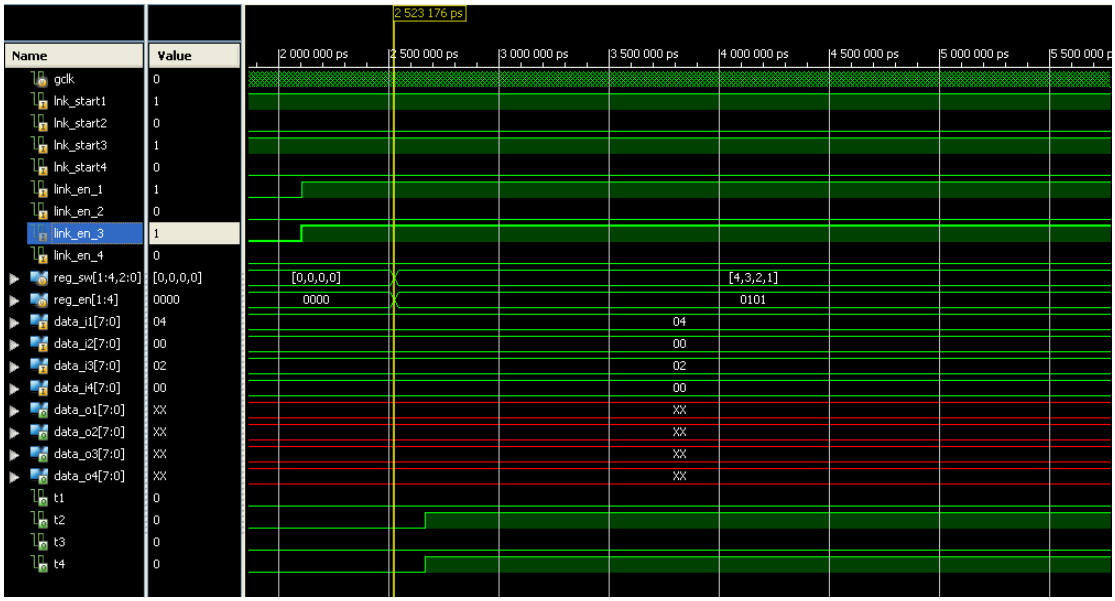

Figure 4.2: Link Establishment

Once the physical link is established, the state machines in the corresponding BIMs transitions from reset state (000001) to Started state (001000) through ErrorWait and Ready state as shown in Figure 4.3, which indicates that the link is enabled between the two interfaces. Now both ends will start sending Nulls. On successful reception of Nulls at both the ends, a complete link is said to be established.

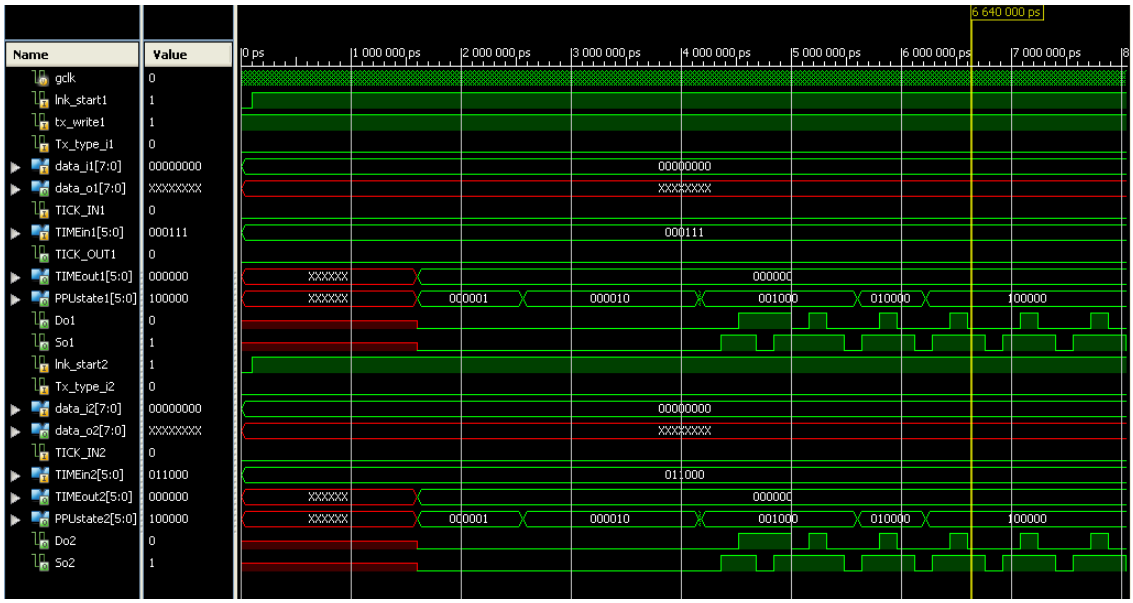

Figure 4.3: Link correctness check

4.3.2 NULLs and FCT Transfer

After successful link establishment, Nulls are sent until the other end received a Null. Once a Null is received, the transmitters stop sending Nulls and the two ends start sending FCTs to indicate each other about the space available in their buffer to receive N-Chars or Time-codes. Figure 4.4 shows the complete transaction of Null and FCTs between the two interfaces. The number of FCTs sent depends on the capacity at the receiver end. In this design each receiver has the capacity to receive 7 FCTs, which account for 56 N-Chars. On receiving a FCT, the receiver indicates the transmitter and the FSM about it and also updates the outstanding counter which indicates the number of N-Chars it is expecting to receive.

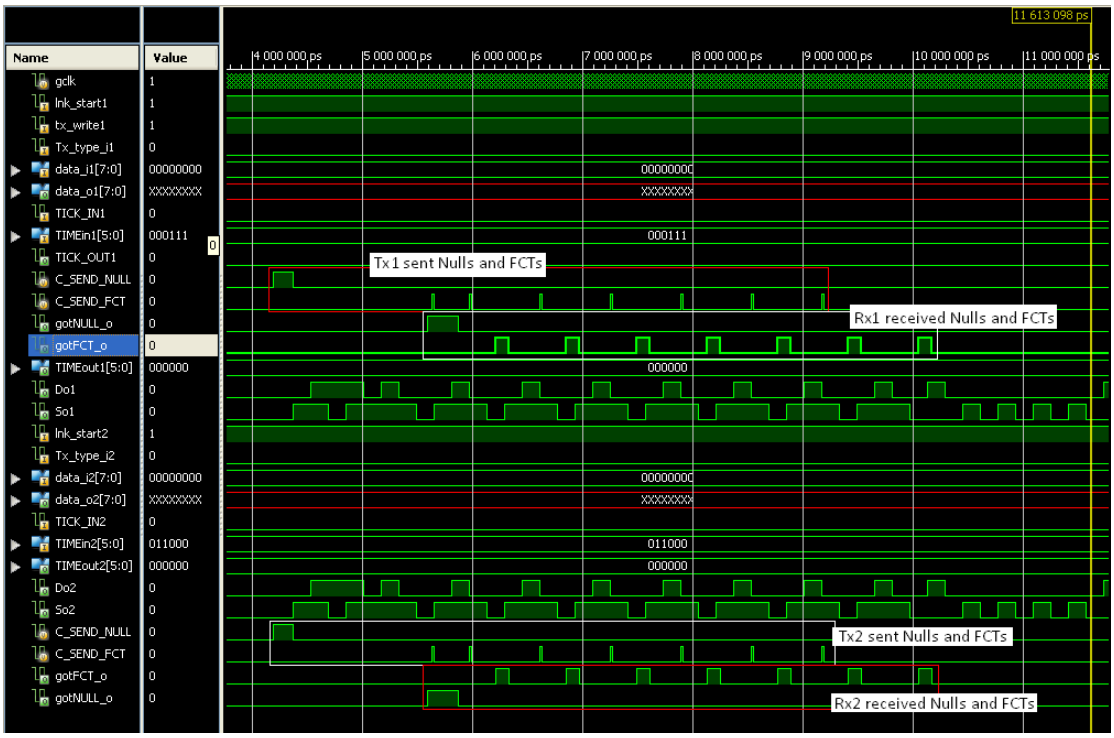

Figure 4.4: Nulls and FCT Transfer

4.3.3 N-Char or Data Transfer

Once all the Nulls and FCTs are received by the receiver at both ends, either ends can transmit data (N-Chars) simultaneously. Figure 4.5 shows the simulation results for data transfer between the two interfaces. It can be seen that, whenever data is ready to be sent by one end, it generates an impulse to send a N-Char. If the N-Char is received by the receiver on the other end without any error, then the data is said to be successfully transmitted. The receiver indicates the FSM about the receipt of the N-Char. It also keeps a count of the outstanding number of N-Chars it is expecting to receive. Once the outstanding counter reaches zero, it indicates that the expected number of N-Chars have been received and informs the transmitter to send more FCTs to receive more data, if any. If not then Nulls are exchanged between the two interfaces to keep the link alive. Link can be disconnected or disabled at any point of time after the transfer is complete.

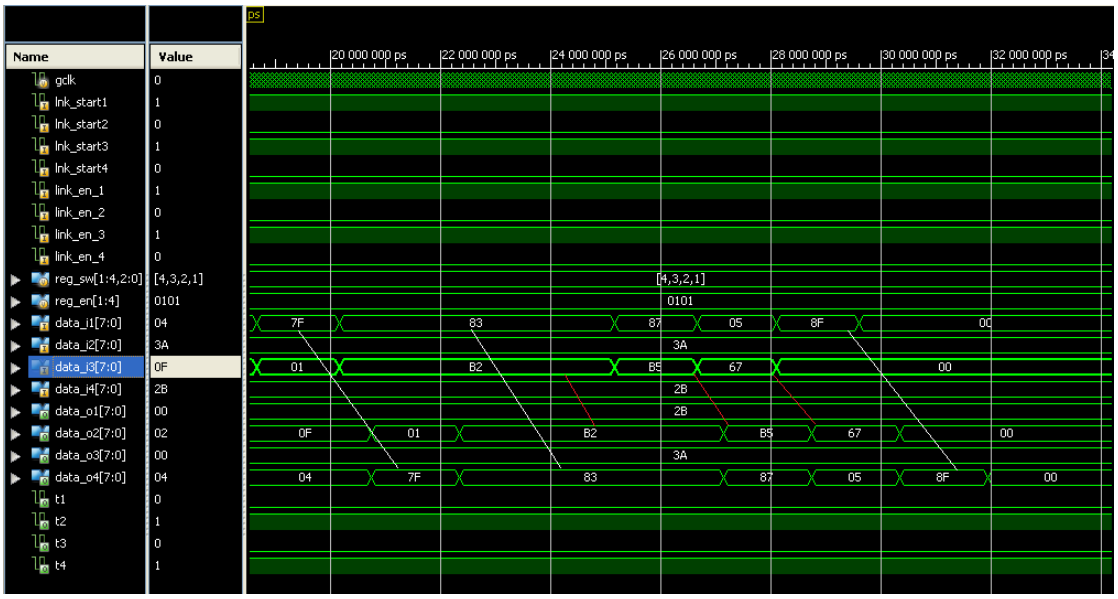

Figure 4.6: Two simultaneous links with Data transfer

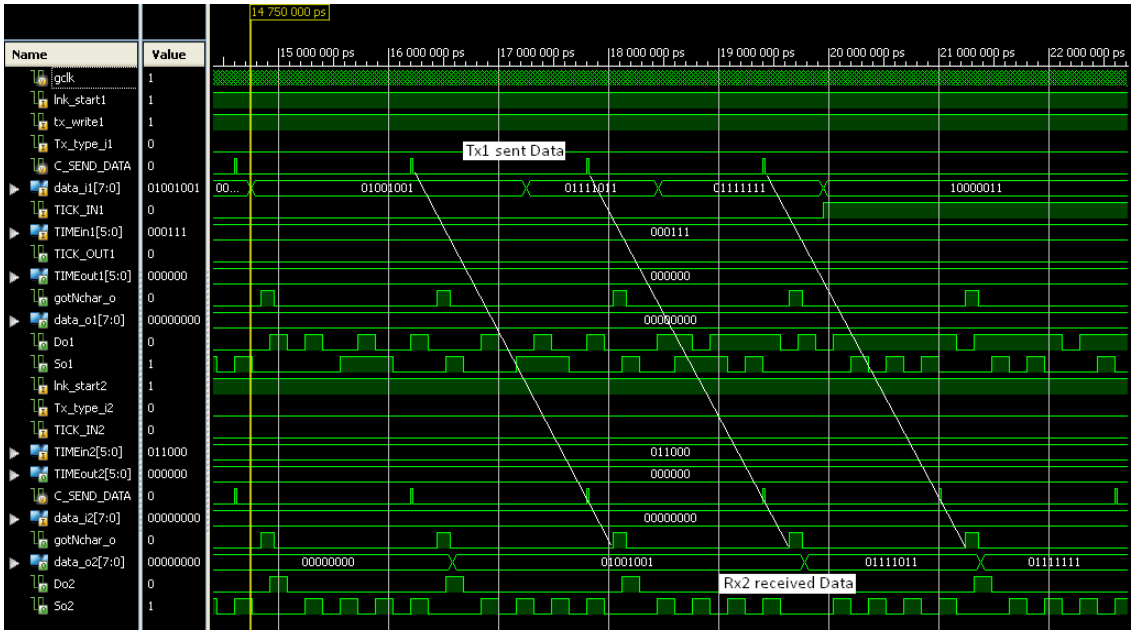

Figure 4.5: N-Char or Data Transfer

Figure 4.6 shows the top level transaction between the BIMs where two simultaneous links were established between ports 1 and 4, and ports 2 and 3.

4.3.4 Time Code Transfer

An important feature of SpaceWire is the process of exchanging time-codes. Time-codes provide a mechanism for supporting distributed system synchronization. Whenever a system needs to send a Time-code to the other end. It enables TICK_IN, which transmits the TIME_IN across the network as shown in the Figure 4.7. When the time-code reaches the other end of the link, it flags out saying that it received a time-code and updates its TIME_OUT signal appropriately.

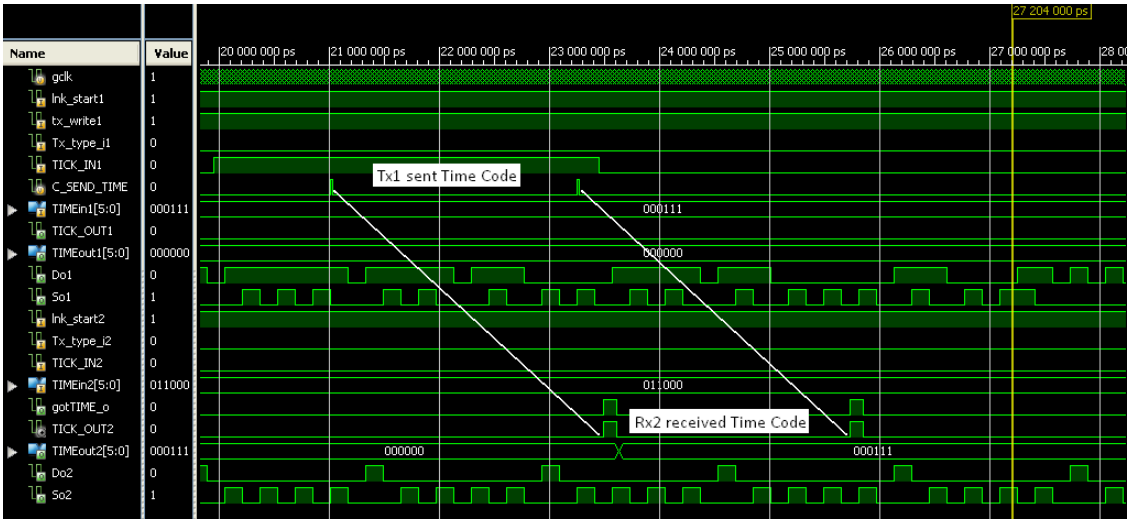

Figure 4.7: Time Code Transfer

4.3.5 Link Disconnection

A link gets disconnected when an active low signal is asserted on the link_start signal of the source which initiated the link. This will disconnect the corresponding link and frees the port connection in the crossbar switch. Figure 4.8 shows the process involved in a link disconnection. The link disconnection mechanism is mutually independent of other active links in the crossbar switch.

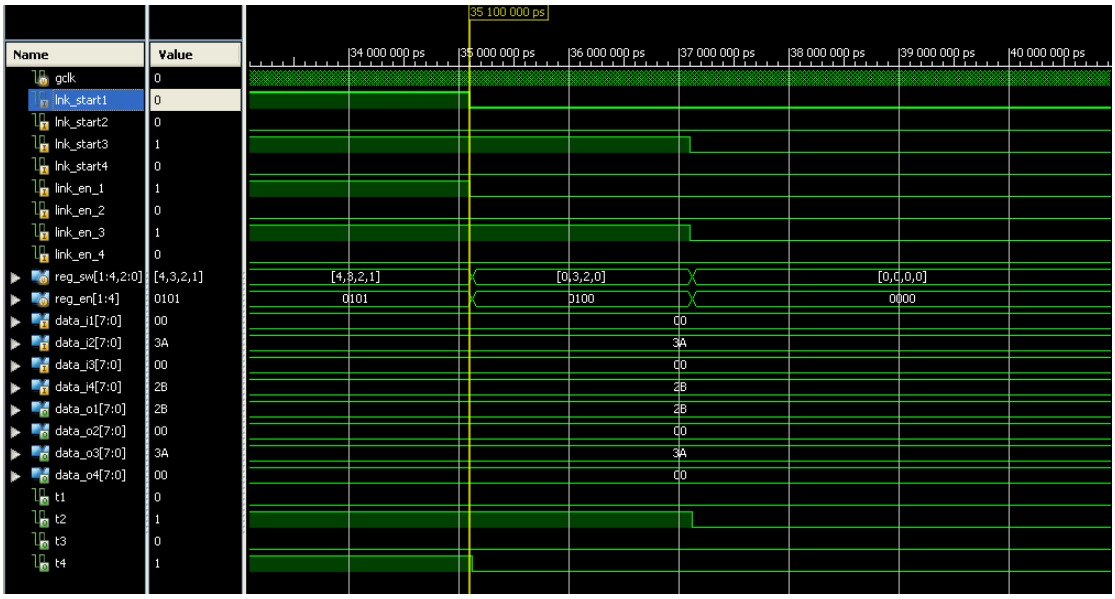

Figure 4.8: Link Disconnection

4.4 Resource Utilization and Analysis

This design was implemented as a prototype to demonstrate the behavior and hence we used the Xilinx Virtex 4 (XC4VSX35) board available. But, if the design has to be implemented for a particular mission, then the choice of FPGA is very important. The type of FPGA technology and device family used in a design is a key factor for system reliability. There are basically three types of programming technologies currently in existence: antifuse-based, flash-based, and SRAM-based. Each type has advantages and disadvantages associated with its use in satellite applications. For each type of programming technology there are multiple device families. A detailed analysis on choosing the best FPGA for a mission is discussed in more detail in [11]. Tables 4.1 and 4.2 contains the resource utilization for a single BIM and a 4-port crossbar switch implemented on XC4VSX35 respectively. Each slice in an FPGA contains 2 Flip Flops and 2 LUTs. Therefore, percentage of slice utilization gives us the estimate of the total number of modules that can be implemented on a given FPGA. From the Table 4.2, we can estimate that each port on a crossbar switch takes up 50 slices. So, a BIM together with a port on the

crossbar switch occupies 300 slices on an FPGA.

Table 4.1: FPGA Resource Utilization of a single BIM

Logic Utilization	Used	Available	Utilization
Number of Slices	251	15360	1.6%
Number of Slice Flip Flops	162	30720	0.5%
Number of 4 input LUTs	478	30720	1.5%
Number of bonded IOBs	59	448	13.1%

Table 4.2: FPGA Resource Utilization of a 4-port Crossbar switch

Logic Utilization	Used	Available	Utilization
Number of Slices	180	15360	1.1%
Number of Slice Flip Flops	72	30720	0.2%
Number of 4 input LUTs	348	30720	1.1%
Number of bonded IOBs	26	448	5.8%

Based on the simulation and implementation results, let us analyze the resource utilization for a small satellite with six subsystems. To implement this system, we need six BIMs with a six port crossbar switch. Since, the crossbar switch is controlled by the CDH subsystem, we need to look at the resource utilization to implement the CDH subsystem as well. Table 4.3 shows an estimate of the resource utilization for implementing such a system on Virtex FPGA.

Table 4.3: Resource Estimation Example

Module	LUT usage	Slice Usage	Slice Available	Slice Utilization
Microblaze Soft Core	6000	3000	15360	19.5%
Six BIMs	3000	1500	15360	9.7%
Six-port Crossbar Switch	600	300	15360	1.9%
Glue Logic	5000	2500	15360	16.2%
Total	14600	7300	15360	47.5%

4.5 Power Analysis

XPower Analyzer is a Xilinx tool [12] dedicated for power analysis of post-implemented place and routed designs. It provides a comprehensive graphical user interface (GUI) that allows a detailed analysis of the power consumed and also offers thermal information under operating conditions as well. XPower Analyzer leverages device knowledge and design data to deliver accurate estimate of device power and individual net power utilization. The power consumption consists of two components: the static power and the dynamic power. The static power, known as the quiescent power in the tool, is the power consumed when there is no switching activity in the design. This depends primarily on the FPGA device selected. The quiescent power for our Virtex 4 FPGA is 0.6W. The dynamic power is the power consumed by the load capacitance during switching activity. The dynamic power is calculated based on the equation 4.1, where P_{dyn} is the dynamic power in W, C is the load capacitance in pF, V is the operating voltage in V, f is the frequency of operation in Hz, and α is the switching activity factor.

$$P_{dyn} = C \cdot V^2 \cdot f \cdot \alpha \tag{4.1}$$

Using the XPower Analyzer, we did a power estimation for our implementation on Virtex 4 FPGA. We did a similar power estimation for the design on an Actel radiation tolerant RTSX-S FPGA (RT54SX32S) [24] using Actel's power estimation tool. Tables 4.4 and 4.5 contains the detailed information about total power (static and dynamic) estimation for a single BIM and a 4-port crossbar switch implemented on Xilinx Virtex 4 and Actel's RTSX-S FPGAs, respectively. The dynamic power consumed by the design is in mW, while the major power is dissipated as static power. If the design is implemented on a Flash based FPGA, then the static power can be drastically reduced [13]. Another solution to reduce static power is to use Xilinx Virtex 6 FPGAs, which use latest technologies to significantly reduce both static and dynamic power [20]. On an average, there is a 30% reduction in static and dynamic power in

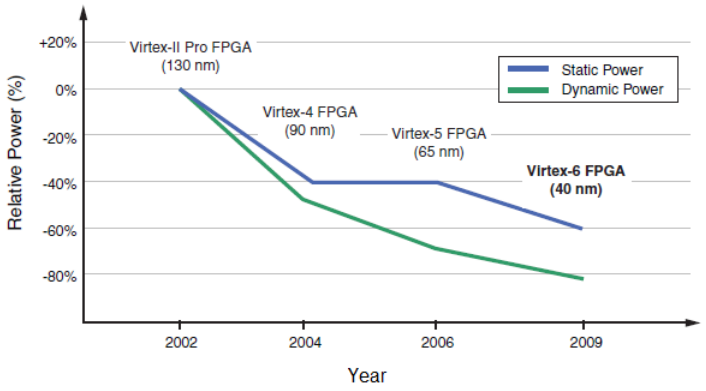

Figure 4.9: Relative Power Consumption of Virtex FPGAs [20]

Virtex 6 FPGAs compared to Virtex 4 FPGAs, solely due to process enhancements as shown in figure 4.9 [20]. A power comparison between 3 major FPGA vendors is presented by Actel [13] and is reproduced in figures 4.10 and 4.11.

Table 4.4: Power Estimation for a BIM

Name	Xilinx Virtex 4	Actel RTSX-S
Total Quiescent Power	0.6001W	0.0625 W
Total Dynamic Power	0.0382W	0.028 W
Total Power	0.6383W	0.0905 W

Table 4.5: Power Estimation for a 4-port crossbar switch

Name	Xilinx Virtex 4	Actel RTSX-S
Total Quiescent Power	0.5993W	0.0625 W
Total Dynamic Power	0.0177W	0.023 W
Total Power	0.6170W	0.0859 W

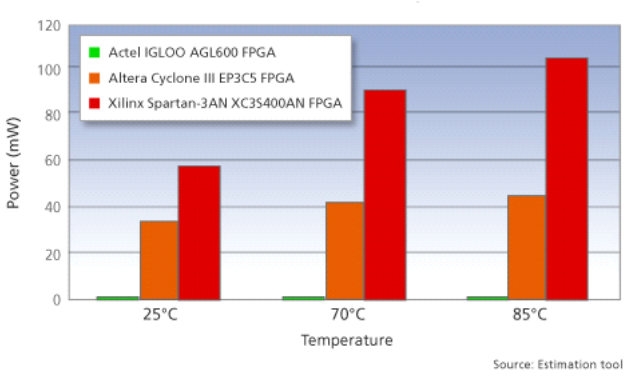

Figure 4.10: Static Power Comparison of different FPGAs [13]

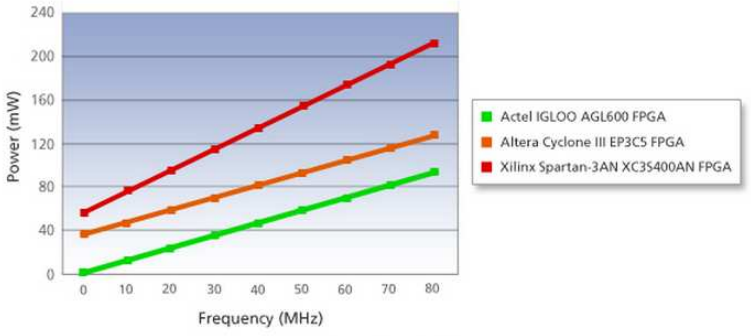

Figure 4.11: Dynamic Power Comparison of different FPGAs [13]

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This work presents a new distributed network switch bus architecture for small satellites based on SpaceWire bus protocol. We understood the need for distributed bus architecture in small satellites and discussed about different bus architectures implemented as spacecraft data bus and showed that the network switch bus architecture is more appropriate for mission critical applications and low power requirements of a small satellite. A comparison of different serial bus protocols showed the merits and de-merits of the bus protocols. The proposed architecture was flexible, modular and configurable, hence supports different subsystems based on the mission needs. The Bus Interface Module and the crossbar switch designed proves to be compact, modular and subsystem independent. In this thesis, we have implemented the design to setup a communication link between the Bus Interface modules of the subsystems through the network switch, also demonstrating the ability to control and configure the switch. Every subsystem will now communicate using the same bus protocol. This proposed architecture will help reduce the design time and cost for future missions.

5.2 Future Work

As a continuation to this work, we can write a software program which controls the link establishment, sends appropriate commands to the BIMs to initiate data/time-code transfer and disables the link after the communication. We can also develop a design tool to configure the architecture, which gives more clarity and understanding to the user. This design tool should also include the design methodology in implementing the architecture. Also, we can extend the design to implement other serial communication ports which can interact with the SpaceWire ports. With the introduction of run time partial reconfiguration, we can increase the fault tolerance of the system. But, to implement this we need to use SRAM based FPGAs and the appropriate tools. So, the choice of the FPGA is critical to make such decisions. We were not able to implement the Triple Modular Redundancy (TMR) for the current design due to the lack of design tools, which could be an extension to this work.

Bibliography

[1] NASA History, http://history.nasa.gov/sputnik/, March 23, 2010.

[2] Miniaturized satellite, Wikipedia, http://en.wikipedia.org/wiki/Miniaturized_satellite

[3] CubeSat, Wikipedia, http://en.wikipedia.org/wiki/CubeSat

[4] SpaceWire - Links, nodes, routers and networks, ECSS-E-ST-50-12C, 31st July 2008

[5] IEEE Computer Society, "IEEE Standard for Heterogeneous Interconnect (HIC) (Low-Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction)", IEEE Standard 1355-1995, IEEE, June 1996.

[6] MicroBlaze Soft Processor Core, http://www.xilinx.com/tools/microblaze.htm

[7] Holmes-Siedle, A and Adams, L (1993), "Handbook of radiation effects". ISBN 0- 19-856347-7

[8] "Total ionizing dose effects", http://radhome.gsfc.nasa.gov/radhome/tid.htm

[9] LaBel, K A et al. (1996), "Single Event Effect Criticality Analysis", http:// radhome.gsfc.nasa.gov/radhome/papers/seecai.htm

[10] Partial Reconfiguration Design with PlanAhead, Xilinx, March 25, 2008.

[11] A Comparison of Radiation-Hard and Radiation- Tolerant FPGAs for Space Applications, NASA, JPL, December 30, 2004.

[12] Xilinx Power Analyzer,
http://www.xilinx.com/products/design_tools/logic_design/verification/xpower_an.htm

[13] Power Comparison, http://www.actel.com/products/solutions/power/comparison.aspx

[14] Open Cores, http://opencores.org/

[15] CubeSat, http://www.cubesat.org/

[16] SpaceWire-UK, http://www.spacewire.co.uk/

[17] DSI Technologies, http://www.dsi-it.de/cms/website.php

[18] 4Links, http://www.4links.co.uk/

[19] STAR-Dundee, http://www.star-dundee.com/

[20] Xilinx Virtex 6 Power Consumption,
http://www.xilinx.com/support/documentation/white_papers/wp298.pdf

[21] NASA, http://www.nasa.gov/

[22] JAXA, http://www.jaxa.jp/index_e.html

[23] Russian Federal Space Agency - Roscosmos,
http://www.federalspace.ru/main.php?lang=en

[24] Actel's Radiation tolerant FPGA for space,
http://www.actel.com/products/milaero/rtsxsu/default.aspx