Reconfigurable hardware programming in a protocol processor unit

Grammar based language framework
for hardware/software co-design

- SUNIL KALLUR RAMEGOWDA

ROYAL INSTITUTE
OF TECHNOLOGY

Master's Degree Project
School of Information and Communication Technology
Royal Institute of Technology(KTH)
Stockholm,Sweden 2015

Acknowledgement

First of all i would like to thank MR. Pierre Rohdin G for selecting and providing me an opportunity to pursue Master thesis at Ericsson AB. I wish to extend my sincere thanks to Mr. Tume Wihamre and Dr. Johny oberg for supporting me to accomplish

Reconfigurable hardware programming in a protocol processor unit

Sunil Kallur Ramegowda

July 31, 2015

Abstract

Reconfigurable hardware architectures have been the topic of research for many years. Programming such architectures requires manual coding or the design of custom compilers to generate the required files for the architecture.

A protocol processor,in general processes the packets according to the protocol. There are number of protocols like Ethernet,CPRI to define how the data has to be sent and received between the source and destination points. Data packets can be processed using a generic processors programmed in software,but hardware processing is always faster and energy efficient.

A compiler/mapper is investigated in this thesis work. The language application is developed using a parser generator tool called Antlr. The grammar is written in Extended Backus Naur Form (EBNF) and the corresponding language is used to describe the architecture and the protocols. The tool will generate the hardware model and their interconnection in SystemC based on the protocol description.

The complete system is verified by integrating the Ethernet protocol. Parts of the protocol implementation in SystemC is also considered in the work. The system is verified for different protocols. The framework works based on the user defined description of the protocol.

Future work involves the integration of further protocols into the system and then adapt the language to further involve all the future requirements. The concept of mapping can be used to design the hardware blocks and their interconnections in different languages.

Contents

Abstract i

Contents iii

Abbreviations iv

List of Figures v

List of Tables vi

Listings vii

1 Introduction 1
- 1.1 Background . . . 2
- 1.2 Purpose . . . 3
- 1.3 Problem Description . . . 3
- 1.4 Goals . . . 4
- 1.5 Limits on scope . . . 5
- 1.6 Structure of the thesis . . . 5

2 Background 7
- 2.1 Reconfigurable Systems . . . 7
  - 2.1.1 Granularity . . . 7
  - 2.1.2 Reconfiguration Models . . . 8
  - 2.1.3 Reconfiguration rate . . . 9
- 2.2 Protocols . . . 9
  - 2.2.1 Ethernet . . . 9
  - 2.2.2 Xio-s . . . 12
  - 2.2.3 CPRI . . . 14
- 2.3 Grammar and Language . . . 15
  - 2.3.1 Parser . . . 15
  - 2.3.2 Backus-Naur Form . . . 15
- 2.4 Environment and Tools . . . 16
  - 2.4.1 SystemC . . . 16
  - 2.4.2 TLM . . . 16
  - 2.4.3 UVM . . . 16

3 Methodology 17
- 3.1 Research process . . . 17
- 3.2 Freyja Architecture . . . 17

3.2.1 Switch port . . . 17
3.2.2 Operator units . . . 18
3.2.3 Switch wrapper . . . 19
3.2.4 Overall architecture . . . 19
3.3 High level description of the protocol . . . 22
3.3.1 Antlr . . . 22
3.4 Comparison of 3 Protocols . . . 23
3.5 Modelling and Testing . . . 25

4 Developed language **26**
4.1 Parser implementation . . . 26
4.2 Operator Instantiation . . . 27
4.3 Context switching . . . 28
4.4 Memory . . . 29
4.5 Transaction handling . . . 30
4.6 Error handling . . . 30

5 Tests **31**
5.1 Complete Test System . . . 31
5.2 Payload and blocking transport . . . 32
5.3 Input and output . . . 32
5.4 Blocking transport and timing annotation . . . 33

6 Analysis **34**
6.1 Ethernet . . . 34
6.2 Xio-s . . . 36

7 Conclusion and Future work **39**
7.1 Reconfigurable architecture . . . 39
7.2 Protocol sharing . . . 40
7.3 Language framework . . . 41
7.4 Limitations . . . 41
7.5 Future work . . . 42

References **45**

Appendix **46**

A Ethernet Protocol Description **46**

B Language Recognition terms **49**

Abbreviations

ASIC Application Specific Integrated Circuit.

ASIP Application Specific Instruction-set Processors.

CPRI Common Public Radio Interface.

CRC Cyclic Redundancy Check.

EBNF Extended Backus Naur Form.

EDA Electronic Design Automation.

FCS Frame Check Sequence.

FPGA Field Programmable Gate Arrays.

GPP General Purpose Processor.

HDLs Hardware Description Languages.

HLS High Level Synthesis.

IEEE Institute of Electrical and Electronics Engineers.

OSI Open Systems Interconnect model.

PLA Programmable Logic Array.

RTL Register Transfer Level.

TLM Transaction Level Modeling.

List of Figures

1 Hardware Software Design Gaps versus Time. 4
2 Flexibility Vs Performance of Hardware Classes 7
3 Reconfiguration Models 8
4 Ethernet Transmit 9
5 Ethernet Receive 10
6 Ethernet Encoder block 11
7 Ethernet Decoder block 12
8 Xio-s Transmitter 12
9 Xio-s Receiver 13
10 Xio-s Frame format Type 1 13
11 Xio-s Frame format Type 2 13
12 CPRI Transmitter 14
13 CPRI Receiver 14
14 Parser 15
15 Freyja Architecture 18
16 Switch port 18
17 Freyja operator 19
18 Freyja Switch Wrapper 20
19 Freyja Architecture for 4 operators 21
20 Antlr 22
21 Transmitter of all 3 protocols 24
22 Receiver of all 3 protocols 24
23 Antlr generated Files 26
24 CRC operator in Freyja Architecture 27
25 Complete Test System(*Multiport interconnections not shown) 31
26 FBI Interconnect 32
27 Input and Output of the system 32

List of Tables

1 OSI Model 1
2 Ethernet raw frame 10
3 Preamble and SFD 11
4 Control Word 11

Listings

1 Freyja Operator Instantiation 28
2 Freyja Memory content definition 29
3 Freyja Overall architecture definition 33
4 Freyja protocol Interconnection 34
5 Freyja CRC operator Control Block code segment 35
6 Freyja Reconfiguration constants 35
7 Freyja CRC Ctrl block with 2 protocols 36
8 Freyja One source to multiple destination interconnection 37
9 Freyja multiple source to one destination interconnection 37

1 Introduction

A set of rules define the communication strategy between digital systems. There are many rules which makes the communication possible between systems. Over the decades the rules have evolved into standards. Open Systems Interconnect model (OSI) is an international effort to facilitate communications among different manufacture and technology. OSI reference model partition the communication system into 7 abstraction layers. It address the interconnection requirement of an open systems environment.

	Layers	**Functions**	**Examples**
7	Application	High level APIs	Mail,IE,Firefox
6	Presentation	Character code translation, Data conversion, Data compression etc	ASCII,JPEG
5	Session	Session establishment between processes running on different systems.	HTTP,SMTP
4	Transport	Acknowledgement, Segmentation, Multiplexing etc	TCP,UDP
3	Network	Addressing,Routing, Traffic control etc	IPv4,IPv6
2	Data Link	Error free data transfer from one node to another	PPP,IEEE 802.2
1	Physical	Transmission and reception in physical medium.	DSL

Table 1: OSI Model

The rules are called as protocols in communication systems. The software and/or hardware changes based on the protocol chosen to process the message and extract the relevant information at each layer of abstraction. The hardware solutions based on a General Purpose Processor (GPP) or an Application Specific Integrated Circuit (ASIC) exists [1][2]. GPP will have more flexibility but are less energy efficient when compared to ASIC which are less flexible and most energy efficient. ASIP or domain specific processors are more suitable for the protocol processing task and depending on their architectural characteristics they allow varying degrees of trade-off between

flexibility and energy-efficiency[3]. Resource and performance varies depending on the reconfigurable architecture and its level of abstraction[4].

The design of reconfigurable hardware architecture requires the compiler to produce the configuration or the hardware compatible code[5]. These files can be produced on run time when the application is running or in a static way before execution. The complexity of the system depends on the selected design. The hardware for processing the different protocols can be made reconfigurable. Investigation and design of such a concept is performed in this thesis work.

The reconfigurable hardware is modeled in SystemC language using TLM. The files required for the reconfiguration is obtained by parsing the description of protocols using the language defined by the grammar. Antlr tool is used for building the base parser file for the defined grammar and then the required functions are implemented to output the complete system and configuration files.

1.1 Background

In 1960 Gerald Estrin, proposed the idea of a fixed plus variable structure computer [6]. It consisted of a fixed processor and an array of reconfigurable hardware which was controlled by the fixed processor. Even though the idea was demonstrated with a proof,the industry did not consider to further innovate in these field and till 1980's there were no significant developments. In 1985,the reconfigurable Programmable Logic Array (PLA) was patented[7]. Innovation in PLA's further continued with the commercially available Field Programmable Gate Arrays (FPGA) in today's market.

FPGA introduction led to the research of optimizing and reusing the reconfigurable logic blocks. The Electronic Design Automation (EDA) tools compile and synthesize Hardware Description Languages (HDLs) to create a physical design in terms of the FPGA's resources. There were many academic and industry projects showing the concept of High Level Synthesis (HLS)[8] [9]. They operate on internal models known as control/data flow graphs (CDFG) and produces a Register Transfer Level (RTL) model of the hardware implementation[10].Increasing design complexity also created the necessity for the innovation of new EDA tools.

Ericsson AB [11] is a market leader in the radio base station equipments. There are different protocols being used for communication in the Radio Base Station (RBS) units. Ethernet explained by IEEE in 802.3 standard defines the protocol for 10Gbit transfer which is mainly used for communication between the silicon chips. Other protocols include Common Public Radio Interface (CPRI),Serial Rapid IO(SRIO),Xio-s(Ericsson Specific protocol) for reliable communication be-

tween chips at high data rate. Most of these protocols in MAC layer of abstraction share common functions. Ericsson design and manufacture custom ASIC chips for these protocols. The common functions which can be used by different protocols for designing a reconfigurable architecture which will minimize the hardware cost and will provide more flexibility compared to ASIC chips. More details about the protocols and reconfigurable architecture is explained in Chapter 3.

The reconfigurable architecture requires a new hardware and software co-design. The reconfiguration details are extracted based on the hardware design and the compiler/mapper should be able to produce such reconfiguration. This is accomplished by using Grammar based technique i.e by defining a language based on EBNF grammar and then describing the protocols using this language. The overall architecture and working principle will be explained in further chapters.

The thesis deals with understanding the reconfigurable architecture and identifying the configuration details to make the system work for different protocols. The description in high level language is used to extract these configuration details and to verify the complete system using a test bench.

1.2 Purpose

The thesis purpose is to investigate an approach of a compiler or mapper to describe the protocols in high level language and then map it to hardware blocks and their interconnection. This involves showing the proof of concept by SystemC TLM simulation models. The Individual hardware blocks are modeled in SystemC and can vary from a simple block to complex functions of the protocols. This thesis work serves as a proof for the project in Ericsson AB to further investigate the feasibility of developing such architectures.

1.3 Problem Description

As explained in the previous section reconfigurable architecture with lowest granularity are available for different applications.They still face the challenges of lower speed,high energy consumption and the compatibility of tool chains between different vendors. The lowest reconfigurable granularity can be designed at bit,block or function level. When targeting hybrid architectures to improve either performance,cost or speed, the application must be partitioned in such a way that certain repetitive or computation intensive functions are mapped on a reconfigurable hardware. Such mapping is not simple as it requires deep understanding of both hardware and software design [12].

Reference for below figure [13]

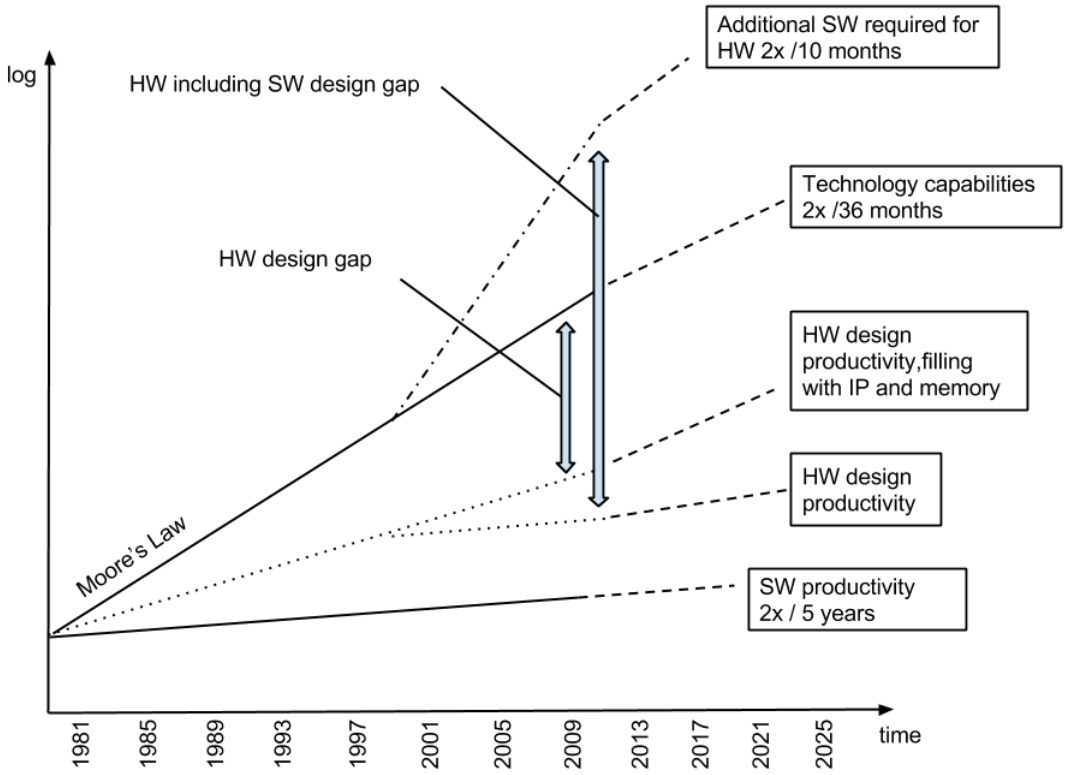

Figure 1: Hardware Software Design Gaps versus Time.

The know how of this process to build the complete system which solves the practical problems are of great importance. The MAC layer shares the common functions among different protocols. How the reconfigurable hardware can be programmed to accomplish the protocol processing in an efficient way by designing the framework using high level description needs to be explored. This allows the user with minimal know how about hardware to reconfigure and show result in short time.

1.4 Goals

The thesis goal is to achieve the below milestones:

- Understand the reconfigurable hardware architecture designed at Ericsson AB
- Understand Ethernet,Xio-s and CPRI protocols

- Define a language to describe the protocols in high level description
- Identifying how to represent the reconfiguration information
- Mapping the description of language to hardware and interconnections
- Integrating Ethernet protocol for the complete system
- Verifying the system by simulation

1.5 Limits on scope

The thesis focus more on showing the proof of concept considering one to two protocols i.e Ethernet and Xio-s. The language will be designed such that it is easy with minor modification to extend for other protocols like CPRI. Developing and integrating the TLM models for all the protocol's will not be feasible in this time line. The architectural changes required are suggested but not changed as the focus is more on describing the protocol in high level description.

1.6 Structure of the thesis

The thesis is organized to provide required details for understanding the overall work. The first chapter gives a brief introduction to the reader about the topic of investigation,limitations and goals.The rest of the thesis is structured as follows.

chapter 2 This chapter will describe the background about the topic. It is summarized in three sections starting with Reconfigurable hardware architectures and their terminologies,different protocols and their functions and grammar introduction. It also introduces the simulation environment and languages used to model the hardware.

chapter 3 This chapter describes the research methodology inculcated in carrying out this thesis work. It starts with the research process and then the introduction of the reconfigurable hardware architecture under study i.e Freyja architecture details. Then the process of building a language application is explained using Antlr tool. The comparison of three protocols under study and their common functions are also discussed.

chapter 4 The chapter starts with the details of building the framework. It also discuss how the architectural requirements are mapped using the high level description. Mapping of Freyja operators,their memory contents,interconnections and

error handling are discussed.

chapter 5 This chapter explains the details of complete test system. How the overall system can be represented using the input file and how the data is processed and handled within the system.

chapter 6 This chapter explains the integration of Ethernet protocol functions into the system of files auto generated from the language application. The challenges and the mapping of input description are discussed in detail. The Xio-s protocol is also considered but part of the functions are integrated to show the context switching and highlighting the control block changes.

chapter 7 This chapter explains the conclusion and future work.

2 Background

To understand the reconfigurable hardware and its terminologies, this chapter explains in detail about the architecture and its meaning. Protocols and their common functions are also explained which helps in designing the reconfigurable protocol processors. The further sections explains the meaning of grammar and language and its terms.

2.1 Reconfigurable Systems

In the field of computer architecture, designers make decisions based on flexibility and performance requirement[14]. ASIC are the least flexible in terms of adapting for any change in the application and GPP are the most flexible as they are independent of the application and the core can be programmed to make the required algorithm work at the cost of higher power and lower efficiency. ASIC and GPP lies in extreme corners of the graph between Flexibility Vs Performance as in Fig. 2. Reconfigurable architectures are intended to fill the gap and provide more flexibility in terms of hardware and potentially higher performance than software[14].

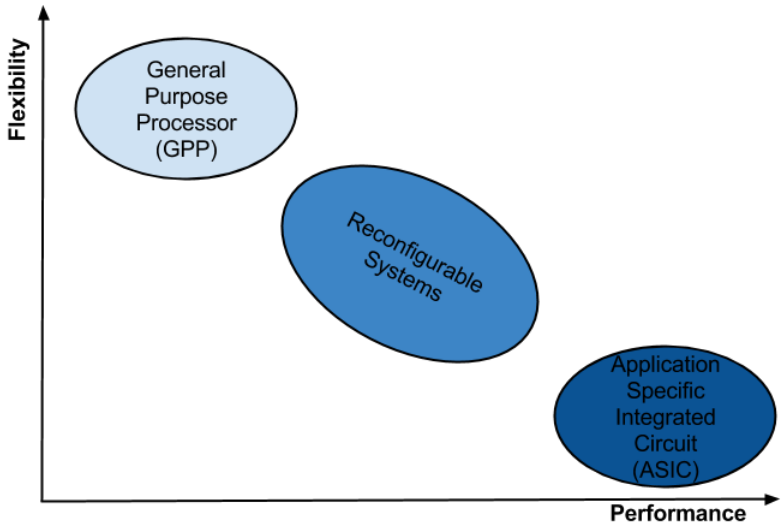

Figure 2: Flexibility Vs Performance of Hardware Classes

2.1.1 Granularity

Reconfigurable devices like FPGA have the configurable logic blocks(CLB) which can be configured to map the required functionality. The complexity of the function

is not a concern but the number of inputs and output of the function has to be considered based on the FPGA architecture. This level of granularity in implementing the functions is called as Fine grained Reconfigurable architecture as it provides the reconfigurable granularity till lowest possible level. These reconfigurable devices are not energy efficient and the execution speed is too less than the ASIC counterpart. Another type of reconfigurable devices are the coarse grained reconfigurable architectures. These devices have the granularity at function levels. They will configure the function blocks to achieve the efficient algorithm implementation.The function blocks can vary from constant block to complex functions which are commonly used based by the application.

2.1.2 Reconfiguration Models

The reconfigurable architectures need configuration of hardware. This can be at compile time or at runtime of an application as in Fig. 3.

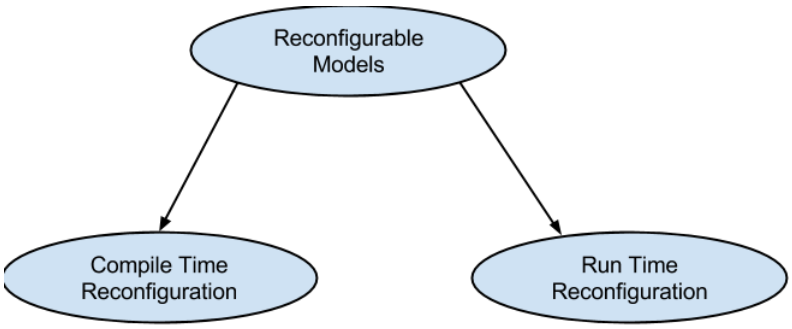

Figure 3: Reconfiguration Models

In compile time reconfiguration model, the reconfigurable hardware system is configured at compile time and will be static during the application run time. In this model the programmable logic can be configured to perform some specific task like hardware accelerators to achieve high performance. FPGA configured to perform floating point multiplication together with a GPP will accelerate the performance of the application if the GPP doesn't have a Floating Point Unit. In run time reconfiguration model, the reconfiguration hardware is configured at run time and will be dynamically programmed to perform different tasks. The decision for making such dynamic reconfiguration has to be embedded coupled with the application and hence it increase the overhead. The Dynamic Reprogrammable Resource Array(DRRA) fabric developed at KTH Electronic System Dept is an example of this model[15].

2.1.3 Reconfiguration rate

The Fine grain Systems will have more reconfiguration data(In FPGAs it is in term of bit streams) which leads to more time and the Coarse Grained Reconfigurable systems will have comparatively less blocks as they have higher granularity and will contain less reconfiguration data. Hence the Coarse Grain architecture will take less time to re configure. This depends on the dynamic reconfiguration architecture whether the complete fabric is reconfigured or partially reconfigured during runtime.

2.2 Protocols

The communication between chips in Radio Base station equipments has many protocols to fulfill the requirements of the specification. The protocols differ by standards. The different protocols being used at Ericsson AB and related to this thesis work are described below:

2.2.1 Ethernet

Ethernet is a widely used protocol for data communication. It is typically used in Local Area Network (LAN) applications. IEEE organization has standardized the protocol and revises it according to the technological advancement. The recent standard available is from 2012 [16] and it defines the protocol for different applications.

Ethernet Transmit

The Ethernet transmit sequence is shown in Fig. 4.

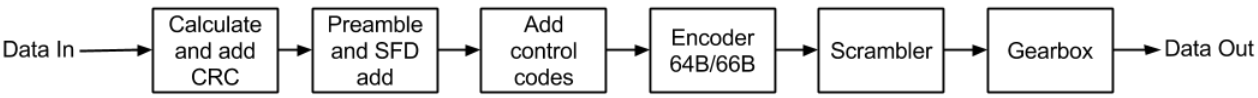

Figure 4: Ethernet Transmit

The *DataIn* is from the higher layers of the protocols which contains the data to be transmitted using MAC layer protocol. Physical layer protocols are out of this thesis scope and not explained.

Ethernet Receive

The Ethernet receive sequence is shown in Fig. 5. *DataIn* is from the physical transmission layer and *Dataout* is to the higher protocol layers.

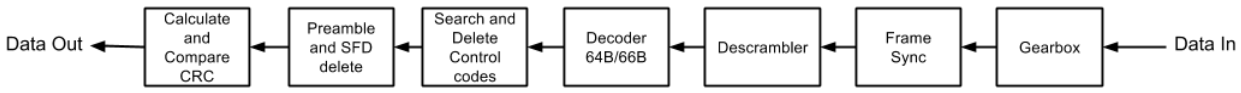

Figure 5: Ethernet Receive

Ethernet Raw frame

The Ethernet raw frame format is represented as in Table. 2. Each of these frames enter the transmitter *Datain* as in Fig. 4 and in the receiver *Dataout* as shown in Fig. 5.

Mac Destination Address	**Mac Source Address**	**802.1Q VLAN tag**	**Ethertype/length**	**Payload**
6 octets	6 octets	4 octets	2 octets	42-1500 octet

Table 2: Ethernet raw frame

A brief functional description of each blocks in the transmitter and receiver section is explained below.

CRC

Cyclic Redundancy Check (CRC) is used to detect errors incurred during the physical transmission. The CRC value is computed by dividing the data to be transmitted with the pre-defined CRC polynomial stored in the memory[17]. The remainder of the division is known as the Frame Check Sequence (FCS).

In transmitter side the FCS is computed for the incoming data and appended as last 4 bytes (32bits). In the Ethernet receive, the FCS is again computed for the incoming data and is compared with the FCS field for any errors. This block will not change the incoming data apart from appending the FCS field.

Preamble and SFD

Preamble is added at the start of the frame to indicate the new Ethernet frame. This block will not change the incoming data apart from adding the Preamble(7bytes) and Start of Frame Delimiter (SFD) at the beginning of the frame(1byte).

In the receiver, the Preamble and SFD are identified and deleted.

Preamble	10101010 10101010 10101010 10101010 10101010 10101010 10101010
SFD	10101011

Table 3: Preamble and SFD

Control codes

The Add control codes block will add the control codes for the incoming data such that Encoder block can use the 8 octets to encode the data based on this control codes. Idles are added to the data if the length of the data is not equal to 8 octets. Each bit in the control word represents whether the octet is data, terminate or an idle octet.

Type	Idel	Idel	Terminate	Data 4	Data 3	Data 2	Data 1	Data 0
Control word	1	1	1	0	0	0	0	0

Table 4: Control Word

In the receiver, the complete process is reversed. It will search for the control word and delete before forwarding to the next block.

Encoder 64/66B and Decoder 66B/64B

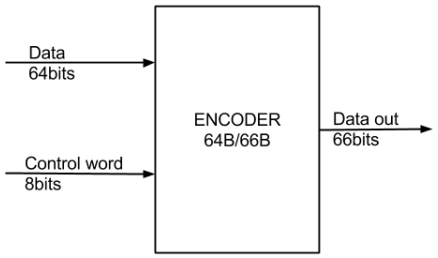

Figure 6: Ethernet Encoder block

The Encoder block is represented as show in Fig. 6. The 8 octet data is encoded using the control word into 66 bit output. The first 2 bits of the output are called sync header which is used for the synchronization from the receiver. The sync header "10" correspond to data and "01" corresponds to control codes

In the decoder the sync header is used to synchronize the 66bit data. The process of decoder is the reverse interpretation of the encoder module.

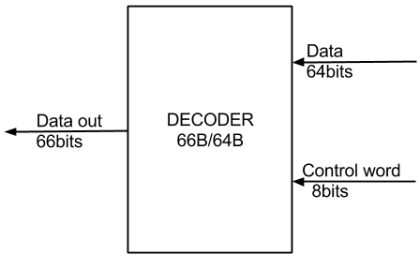

Figure 7: Ethernet Decoder block

Scrambler and Descrambler

This block is used to randomize the signal so that long sequence of 1's and 0's are eliminated. This is performed using the Scrambler polynomial.

The De-scrambler will take the scrambled input and will output the unscrambled data.

Gearbox

This block is used to switch output rates. The incoming data is transmitted at different rates based on the clock frequency.

2.2.2 Xio-s

Xio-s is Ericsson proprietary protocol used for communication between chips.

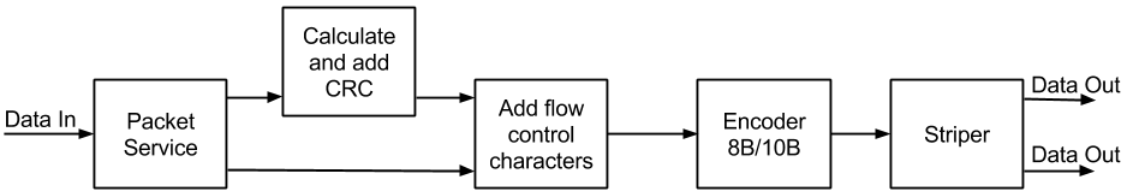

Figure 8: Xio-s Transmitter

There are 7 different types of packet services for Xio-s protocol. This assigns the packet to proper channel based on the service type. As an example, if the service type format 1 then it uses only CRC16 channel or if the Service type format II then it uses both CRC16 and CRC32 channels.

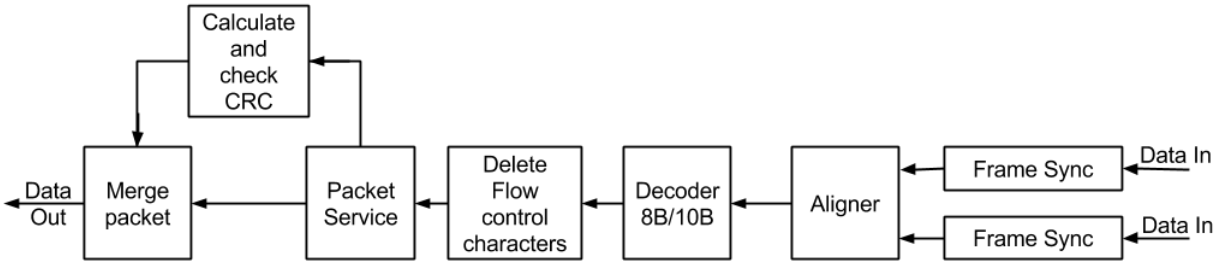

Figure 9: Xio-s Receiver

	Service type [4:0]	Data	CRC [15:0]

Figure 10: Xio-s Frame format Type 1

	Service type [4:0]	Data 1	CRC 1 [15:0]	Data 2	CRC 2 [31:0]

Figure 11: Xio-s Frame format Type 2

CRC

There are 16bit and 32bit CRC calculations required in Xio-s protocol. So the polynomials for CRC16 and CRC32 are stored in the memory and the function is used according to the service type.

In the receiver, the CRC is again computed and compared with the received bytes.

Flow control characters

The flow control character will add the control word similar to Ethernet protocol in each channel. These are used for indication of start and end of frames.

In the receiver the flow and control characters are identified and deleted.

Encoder/Decoder

The encoder module will encode one octet at a time to 10bits. So for 8 octects it outputs 80bits The encoder 8B/10B is invented by IBM and famous for short run length and DC balance.

The decoder module does the reverse of encoder and thus the output of decoder will be the same as the input of encoder.

Striper

It is used to split the 80bits incoming data into 40bits of 2 physical channels to increase the data transfer rate.

Frame Sync

This block is used to synchronize the receiving data.It is performed using the special character in the transmitted data called as k28.5 character.

Aligner

This block is used in the receiver if striper is used in the transmitter side.It aligns the two incoming 40bits channels into one 80bits channel.

2.2.3 CPRI

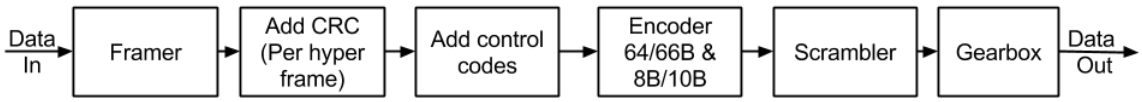

Figure 12: CPRI Transmitter

CPRI is an industry cooperation aimed at defining a publicly available specification for the key internal interface of radio base stations between Radio Equipment Control(REC) and the Radio Equipment(RE) [18]. It is the co operating work of Ericsson AB,Huawei Technologies Co.Ltd,NEC corporation,Nortel Networks SA and Siemens AG. All the blocks in this protocol are similar to the blocks in Ethernet protocol. The frame structure is similar to the Xio-s protocol.

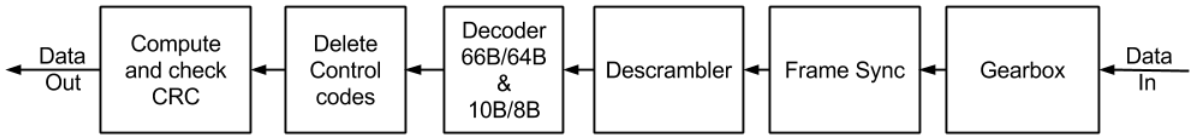

Figure 13: CPRI Receiver

2.3 Grammar and Language

A Grammar is used to describe the syntax of a language, that is, all possible legal sentences or combination of words that make up the language. More formally, Grammar G describes all allowed legal sequences of strings. This is called the language(G) of the grammar. The language in turn is made of sequence of elements which can be letters, numbers or special symbols. For example, In the word "KTH" the capital letters K,T,H are to be recognized and then the word needs to be formed. This is performed by the Lexer which recognizes the letter and forms the token. To be able to recognize words, Lexer need some special constructs. These special constructs makeup the language that can be recognized by regular expressions. For example, regular expression [0-9] recognizes a single letter in the range from 0 to 9.

2.3.1 Parser

The Lexer scans the input character streams and forms the valid tokens. The Parser takes tokens as inputs and then based on the parsing rules in the grammar, decides the parsing strategy. The parser output can be used either to create an interpreter or a compiler.

Figure 14: Parser

Here, a language application is built to output the reconfigurable hardware architecture. Hence the parser output is used to translate it to the required output.

2.3.2 Backus-Naur Form

In computer science world, Backus Naur Form (BNF) is the notational technique for context free grammars. It is a set of derivation rules to define the language.

For example,

⟨ int⟩ ::= ⟨ DIGIT ⟩ | ⟨ int ⟩ ⟨ DIGIT ⟩

⟨ DIGIT ⟩ ::= [0-9]

In the above Grammar, ⟨ int ⟩ on the left hand side is called as non terminal and the ⟨ DIGIT ⟩ is called as terminal. So the sequences of digits like 9999... can be parsed by representing grammar as above BNF code. An extension to BNF grammar with more operators to write the syntax is called as EBNF. The grammar above can be rewritten in EBNF as below

⟨ int ⟩::= ⟨ DIGIT ⟩ *

Here * means one or more occurrences of digits. Similarly "+" operator means 0 or more occurrences.

2.4 Environment and Tools

The EBNF description of the grammar is used by the Antlr4 tool to parse the input file and produce the source and configuration files of the architecture. The hardware modeling is achieved by using SystemC and the verification by Universal Verification Methodology (UVM).

2.4.1 SystemC

SystemC is an ANSI standard C++ class library for system and hardware design for use by designers and architects who need to address complex systems that are hybrid between hardware and software [19]. It helps in modeling the concurrent processes. This makes it possible to model the hardware which are concurrent systems by nature.

2.4.2 TLM

Transaction level Modeling is the approach of abstracting the lower implementation details of the function units and representing the overall system architecture. TLM helps in modeling the communication and the function unit implementation separately [20]. The Transaction refers to the set of data being exchanged. TLM speeds up the simulation by replacing a set of pin level events with a single function call.TLM 2.0 standard is used for the implementation of the bus system.

2.4.3 UVM

Universal Verification Methodology(UVM) is a verification methodology based on the best features of OVM(Open verification Methodology) and Verification Methodology Manual(VMM) [21]. A Base version of UVM is used to verify the functionality of the architecture.

3 Methodology

This chapter explains the process applied in describing the protocol for the hardware architecture. It also explains the details of reconfigurable architecture under study, the common protocol functions and the methodology used to verify the result.

3.1 Research process

The hardware architecture and its interconnections need to be understood to design the protocol processor. The reconfigurable architecture developed at Ericsson AB is called as Freyja architecture. The hardware architecture details has to be abstracted out to define it in high level description. Similarly the protocols details has to be abstracted using the same description to reconfigure the hardware for different protocols. The mapping between them requires a custom set of files to define and reconfigure the architecture according to the requirement. To accomplish this,the research involves the following major steps:

- Study the Reconfigurable hardware architecture
- Identify the common protocol functions
- Develop a grammar to describe the details in high level language
- Test the developed system

The study of the architecture is explained in detail in the next section. The further sections explains the process involved in achieving the remaining steps.

3.2 Freyja Architecture

As in Figure 15 , Freyja architecture is a reconfigurable protocol processor. It consists of different protocol operators which are connected through the central switch. This switch based network topology can be configured to process the data based on protocols. The Ring bus (RB) interface the Freyja with the higher layers of protocol and it issues the data frames of different protocols as tokens. The Common Memory Interface(CMI) is used to fetch the data from memory. The physical interface is represented by the Serializer/Deserializer block i.e SERDES. The details of each of the Freyja architecture blocks are discussed below.

3.2.1 Switch port

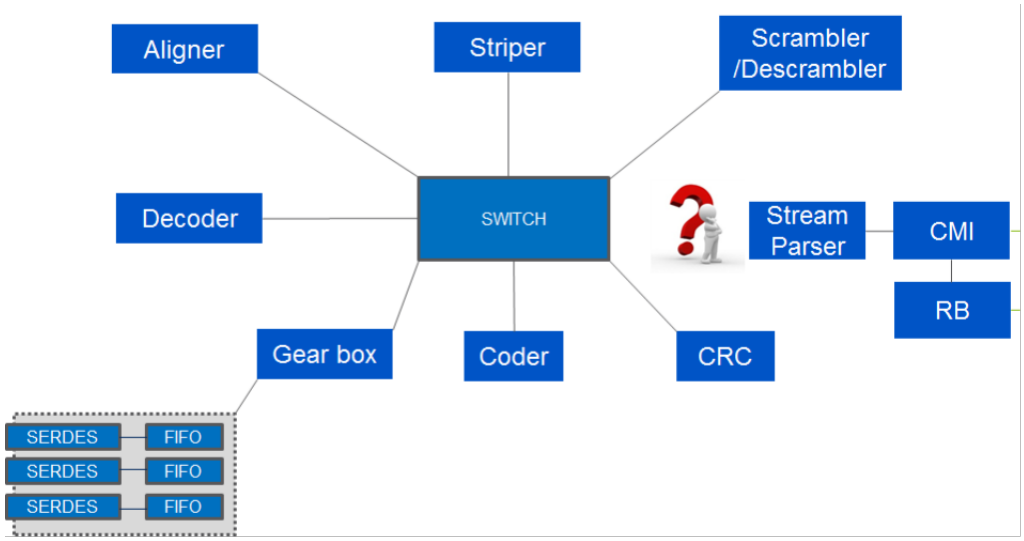

Figure 15: Freyja Architecture

Freyja Switch port is the smallest unit in the architecture which receives the transactions from different functions and then forwards it to the internally connected next switch port based on the destination of the transaction. It consists of a simple initiator socket,simple target socket,Multipass through initiator socket and Multi pass through target socket. It is represented as in Figure 16

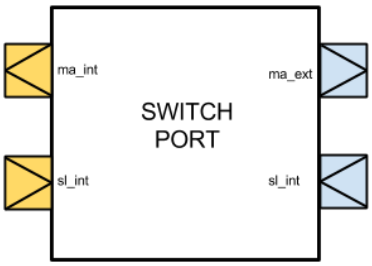

Figure 16: Switch port

3.2.2 Operator units

Freyja Operator units are designed to contain a control, process and memory blocks. Each operator function is implemented in the process block and the transaction routing and context switching is performed in the control block. The memory block stores the constants required for the process block. It can also be used to store the result and then the control block can access the results. As in Figure 17 , the 3 components of each operator is encapsulated with one simple initiator socket and one target socket which initiates the transaction and receives the transaction from the switch respectively. The transaction received in routed to the control block where it sends the transaction to memory block about the received transaction. Based on the required operator, the memory block will send the transaction to the process block. Meanwhile the received transaction is sent to the process block from the control block to perform the required operation.

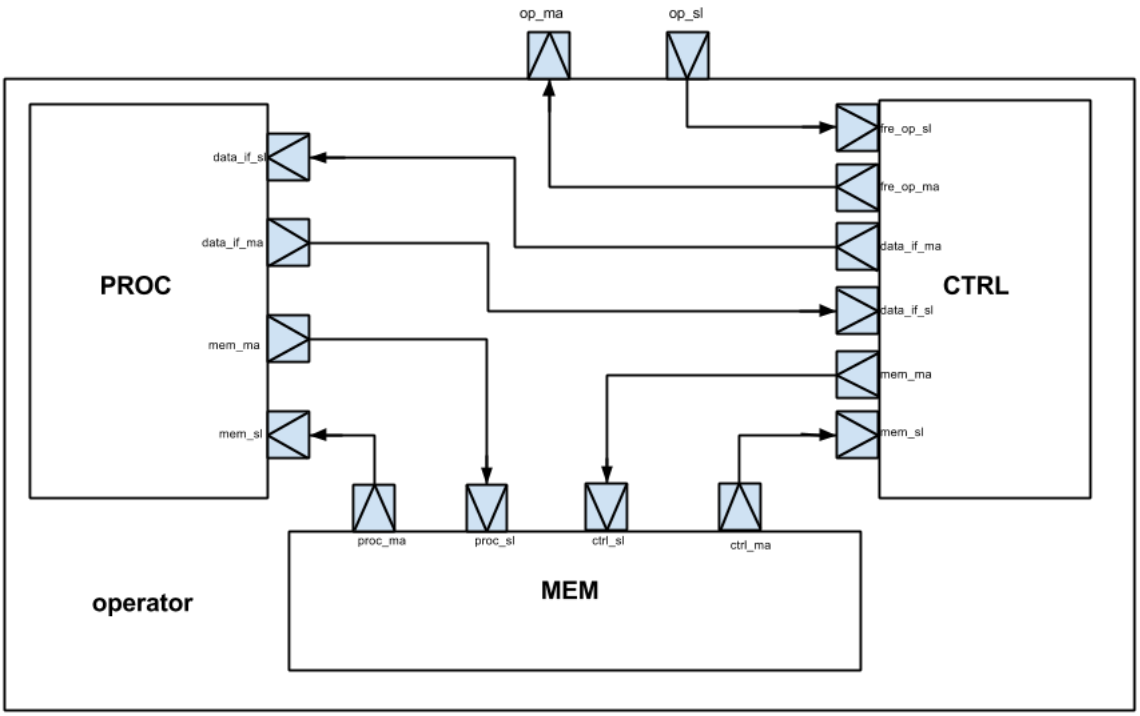

Figure 17: Freyja operator

3.2.3 Switch wrapper

The switch wrapper instantiates the switch ports and their interconnections. Based on the number of operators the required number of switch ports is instantiated.

As in Figure 18, each switch port can send the transaction to any of the other switch port through the internal multi pass through socket. The other switch ports can receive the transaction using the multi pass through receive socket. A transaction source and destination cannot be the same operator as there is no such interconnection.

3.2.4 Overall architecture

Assuming there are 4 operators in the Freyja architecture, an illustration of the system is shown in Figure 19

The overall Frejya architecture consists of switch wrapper instantiating switch ports and the operator blocks for each of the operator functions. A transaction originating from operator 1 as shown in fig with orange box can be routed to any of the other operators as destination based on the address of the payload. This will be assigned in the control block of the operator where the transaction is originated.

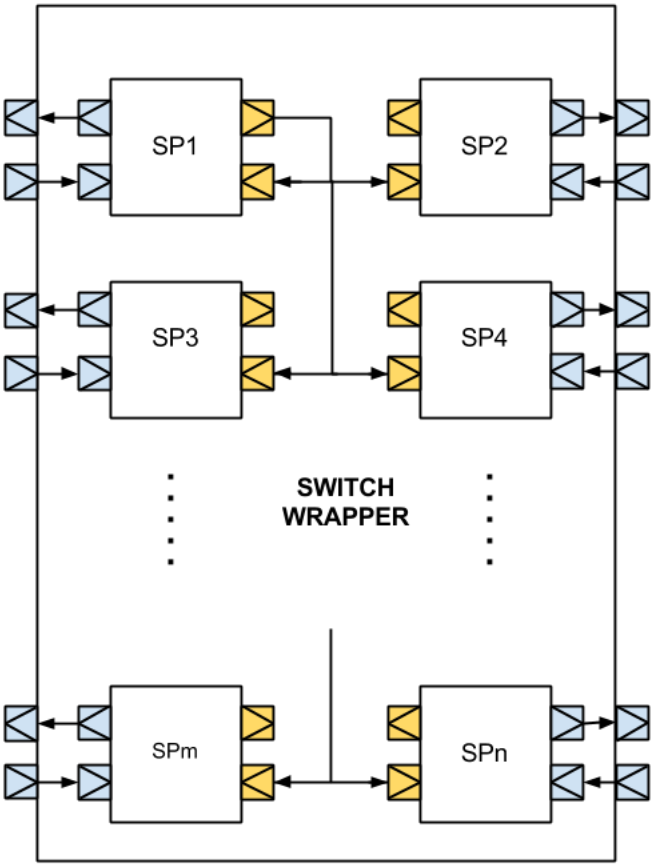

Figure 18: Freyja Switch Wrapper

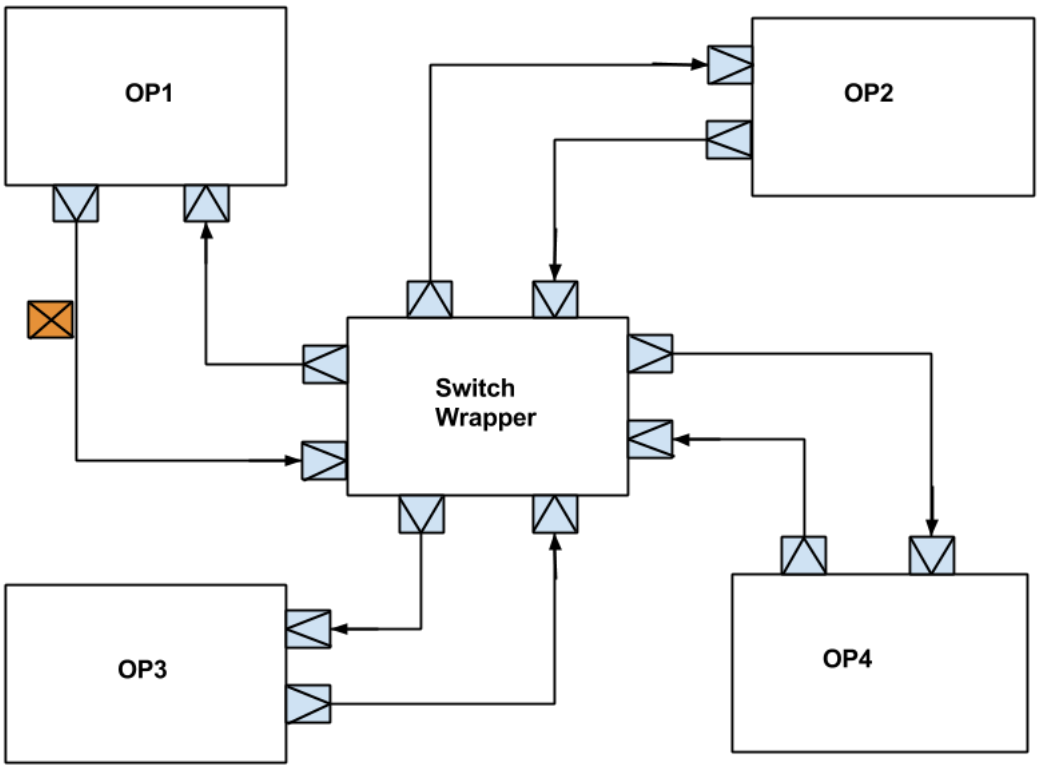

Figure 19: Freyja Architecture for 4 operators

3.3 High level description of the protocol

High level language description requires the design of Lexer and parser as explained in Section 2.3. They can be designed using languages like C,C++,java,C# and more. Based on the application requirement, a parsing strategy needs to be decided. As the machine resources grew in today's world, researchers have developed more complex and powerful Non Deterministic parsing strategies. Today both "bottom-up" and "top-down" approaches exist. Debugging a "bottom-up" parsers are hard to understand and debug compared to the "top-down" parsers.

There are many tools like ANTLR4,APG,AXE,YACC etc which helps to build a language application. They provide the user with Lexer and Parser implementations for the matched grammar defined by the user. This helps in reducing time for building the language in a short time. Antlr is one such tool which helps in designing the recursive descent top-down parser and a clear error recovery mechanism.

3.3.1 Antlr

ANTLR4 accepts as inputs any context-free grammar that does not contain indirect or hidden left-recursion.[22]. Antlr 4 generates a recursive descent top down parser. It uses ALL(*) production prediction function. ALL(*) prediction mechanism launches sub parsers at decision point and they operate in pseudo parallel to explore all possibilities of input combinations. Antlr 4 currently generates parsers in Java or c# and the previous version supports even C and C++.

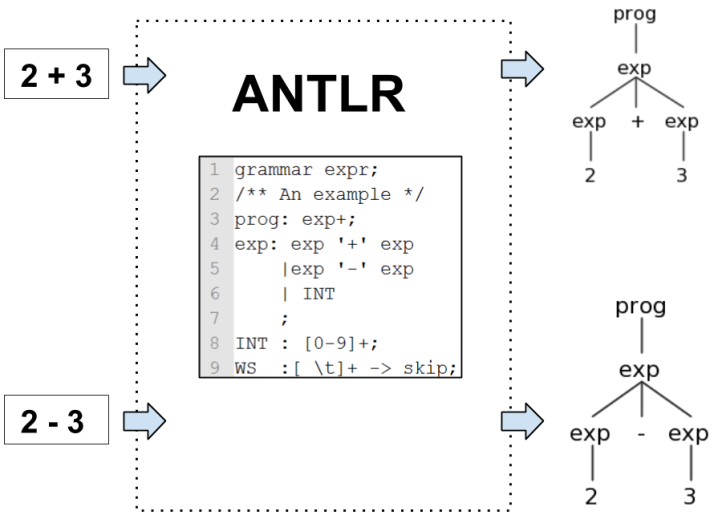

Figure 20: Antlr

Antlr 4 grammar use yacc-like syntax with EBNF operators like Kleene star(*) and token literals in single quotes. Both Lexical and syntactic rules are specified in the same grammar file. The Lexical rules are specified in capital letters which distinguishes them from others.

Figure 20 illustrates ANTLRs yacc-like metalanguage. Antlr 4 automatically rewrites the rule to be non left recursive and unambiguous. The grammar analysis is performed at parse time and caching results in lookahead DFA for efficiency.[22].

Parse tree listners and Visitors

Antlr provides 2 tree walking mechanisms in its runtime library. In parse tree Listners, ANTLR generates a parsetreeListner subclass specific to each grammar with entry and exit methods for each rule. This is suitable for applications wherein the complete tree need to be invoked from root till the last leaf node in-order. The parse tree Visitor mechanism is used when the tree walking needs to be controlled.

3.4 Comparison of 3 Protocols

The parser generator tools need to be used to describe the protocols and produce the reconfigurable files for the hardware architecture. The protocols of the MAC layer considered in this work are explained in section 2.2.

The common functions in each protocol are highlighted with the same color for the boundary line in Figure 21 & Figure 22. A common hardware architecture can be designed with minimal reconfiguration to perform the protocol processing of different protocols stated above.

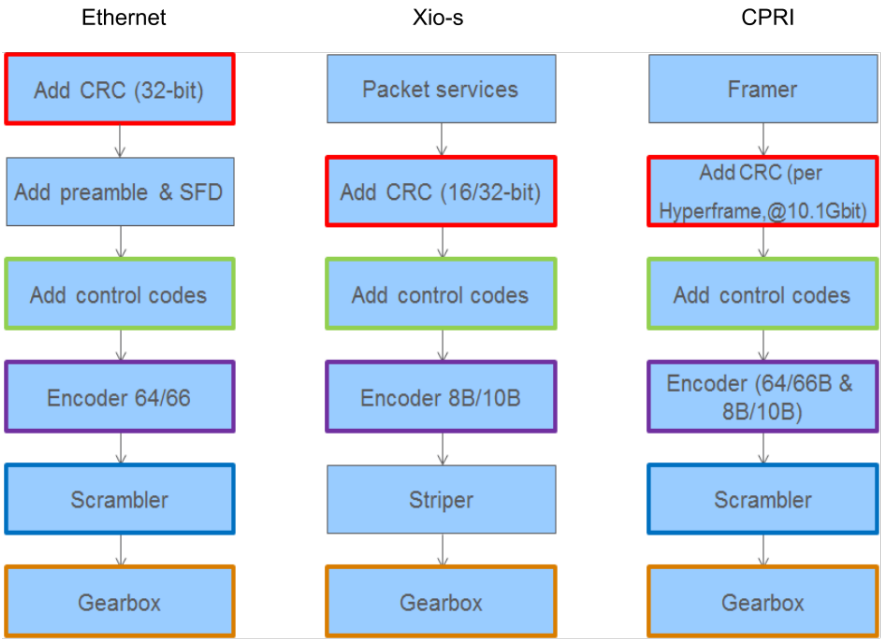

Figure 21: Transmitter of all 3 protocols

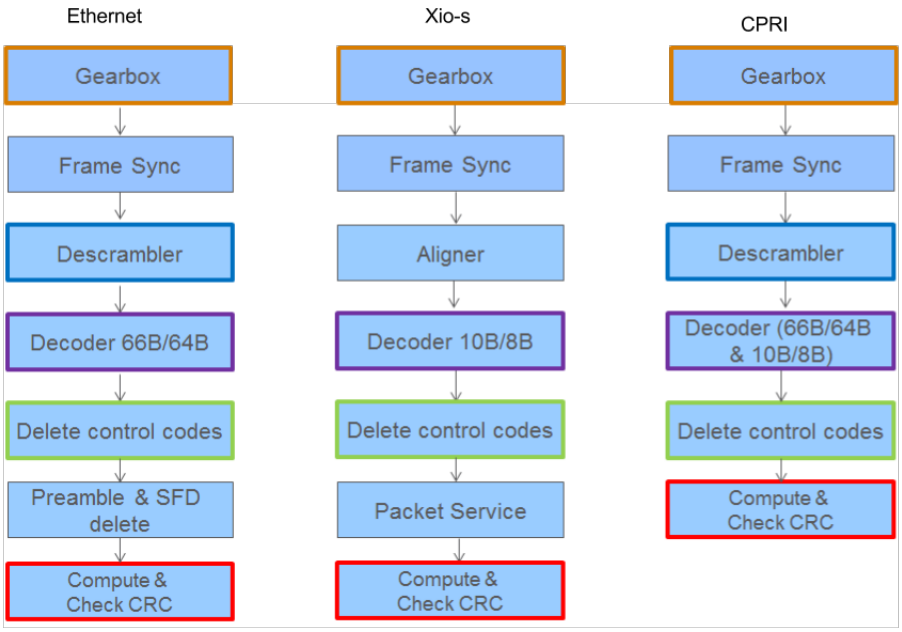

Figure 22: Receiver of all 3 protocols

3.5 Modelling and Testing

4 Developed language

A language description is developed to define the Freyja architecture which maps different protocol operators and their interconnections. The high level description accounts for different operator?s instantiation, transaction routing, constants in memory and error handling. Each of these is explained in detail in this chapter.

4.1 Parser implementation

Antlr tool is used as a parser generator to develop the language. The grammar is defined using EBNF. Antlr parser use a new parsing technology called Adaptive LL(*). This parsing strategy combines the simplicity, efficiency and predictability of conventional top-down LL(k) parser with the power of GLR like mechanism to make parsing decision. It performs the grammar analysis dynamically at runtime rather statically. The grammar analysis is moved to parse time which helps to handle any non-left recursive context free grammar.

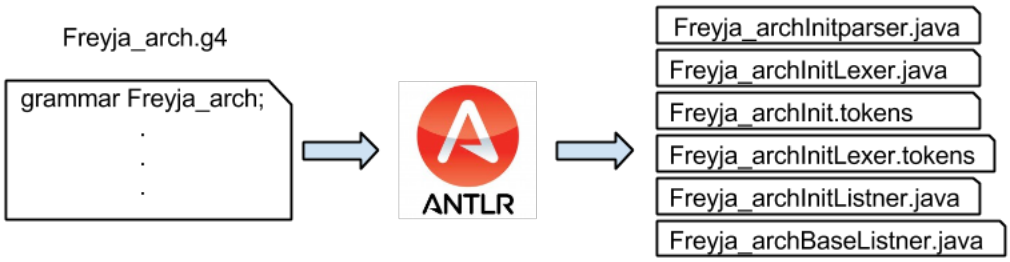

Figure 23: Antlr generated Files

A brief description of the generated file are stated below:

Freyja_archInitparser.java: contains the parser class definition specific to grammar Freyja_arch that recognizes our Freyja protocol processor language syntax.

Freyja_ArchInitLexer.java: This file contains the lexer class definition by analyzing the lexical rules in the grammar.

Freyja_archInit.tokens: Antlr generates a token type number to each token in the grammar and store these values in this file.

Freyja_ArchInitListner.java, Freyja_archBaseListner.java: Antlr parser builds a tree walker that can trigger the callback events to a listener objects. Freyja_archInitListner is the interface that describes the callbacks and Freyja_ArchBaseListner is a set of

empty default implementations.

The backend functions are implemented to produce the Freyja architecture based on the language described.

4.2 Operator Instantiation

As explained in the Figure 24, each of the operator in Freyja architecture consists of a control block to take of the transaction routing and error handling and a memory block for storing the constants and results and the process block to perform the operator function.

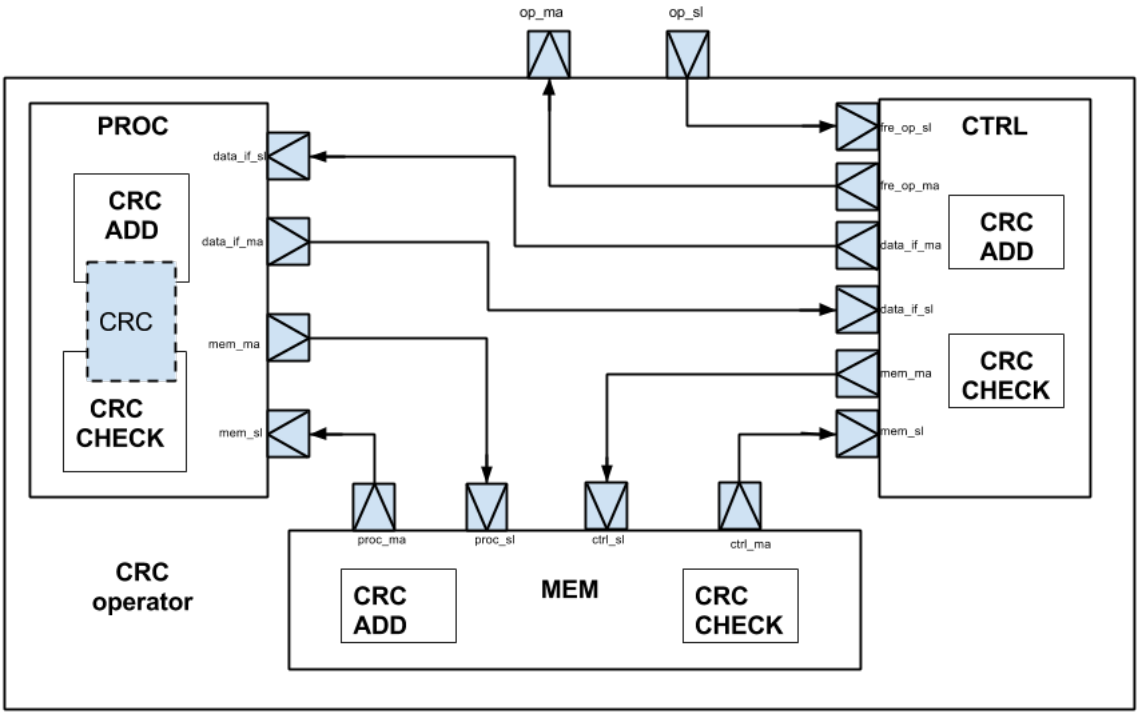

Figure 24: CRC operator in Freyja Architecture

In a protocol processor consisting of operators for transmitting and receiving the data frames, the functions performed during the transmitting stage might need to be performed in the receiver stage. For instance, FCS is appended during the transmitting stage and in the receiver stage the computation of CRC is performed again. In addition, it performs the comparison of the received FCS value and the newly computed value to identify if the received data is erroneous or not.

The control block of each operator has to identify the destination operator within the process block to forward the payload to exact operator. The language described has to consider this feature when instantiating the operator and defining the control blocks for each operator. An example of CRC block with CRC ADD and CRC check operators are discussed further.

As in Figure 24, the CRC function to compute the CRC is used by both CRC ADD and CRC CHECK operators. The control block has to check if the destination operator is CRC add or CRC check and forwards the data to be processed to the correct operator and also should indicate the memory block to forward the required constants for the process blocks. This is further explained in the next section.

The operator is defined as below:

```
1
Operator CHECKSUM{
3 sc_name : crc;
4 op_type : crc_add,crc_check;
5 op_mem  : 0x01,0x04,0xc1,0x1d,0xb7,0x20,0x00,0x04,0x03,0x04,0x00,
6           0x00,0x01,0x59,0x53,0x10,0x03,0x02,0x04;
errorid : 02,03;
8 }
```

Listing 1: Freyja Operator Instantiation

The operator name CHECKSUM identifies the unique operator in the Freyja architecture. The sc_name field is used to create the systemC file with fre_op_ctrl_crc,fre_op_proc_c files which are the control,process and memory blocks respectively. Each operator definition also takes care of including a switch port and modifying the switch wrapper functions to take care of the interconnections.

4.3 Context switching

The context switching in the process block is performed based on the destination operator for the data packet. The interconnect information is by the control block to determine the protocol and destination operator for the data packet. The control block will then forwards the transaction to the process block where the context switching between the operators are performed. The operator type field will indicate the process block to switch between the operators.

The control block also indicates the memory block to send the required constants for the process block. For the CRC operator, as in Figure 24, the control block will

determine the protocol and the operator type and then informs the memory blocks with the transaction payload. The control block will send the received transaction to the process block. The memory block will decodes the operator and then sends the constants required for the operator through a transaction. The process block will do the context switching of operator between CRC add and CRC check to obtain the result according to the destination operator.

Once the results are obtained, the process block will send it to the control block. The control block will forward the transaction to the next block.

4.4 Memory

The memory block is designed to store the constants required for the operators and also to store the results. But currently the results are not stored in memory, as only one protocol is implemented.

The constants to be stored in the memory are indicated in the op_mem field. Each operator is allowed store all the constants in the memory and based on the operator in use the constants can be forwarded to the process block through a transaction. As in operator checksum definition, the memory contains 19 bytes which are used by the CRC16 and CRC 32 operators.

Memory transaction for each protocol are described in the language as below

```
memory {
crc_add     :00 to 08;
crc_check   :00 to 07,09;
scram       :00 to 07:
descram     :00 to 07;
}
```

Listing 2: Freyja Memory content definition

As in code above the first 8 bytes of contents are sent as a transaction from memory for the CRC ADD operation in CHECKSUM operator. The different forms of notation can be observed in the CRC check wherein the memory bytes can be specified with individual address. Each protocol with the memory code as above can send transaction for different operators. In the above code, the crc add,crc check,scram and descram send the transaction to their respective blocks.

4.5 Transaction handling

The transaction routing for each protocol is described in the input file using the below syntax

CONNECT: <source operator> to <Destination operator>

Based on these all the control blocks will be configured with the destination address for the payload. The switch interconnect will send the transaction to the next operator based on this address. This mapping defines the transaction routing for each protocols and is used to build the destination address automatically using the input description.

4.6 Error handling

Any erroneous packets can be sent to the error handler from each of the operator. The control block will check for the process block result, if there are any errors then it informs the control block through the FBI header and the control block will send the payload to error handler for further processing and will not be sent to the destination operator.

The error id for each operator can be implicitly assumed or even possible to mention as an explicit number.

5 Tests

This chapter describes the test system, testing strategy, input protocol file, discussion about output obtained.

5.1 Complete Test System

As in Figure 25, the first switch port is connected to the UVM test bench. The UVM environment will drive the input signals according to the Freyja architecture. The Freyja interconnect consists of 4 bytes header field and the data payload as in Figure 26. In SystemC implementation the first byte is considered to hold the unique protocol id and the second byte to have the unique operator id, the third byte and fourth byte are for Flow control and Context/error handling.

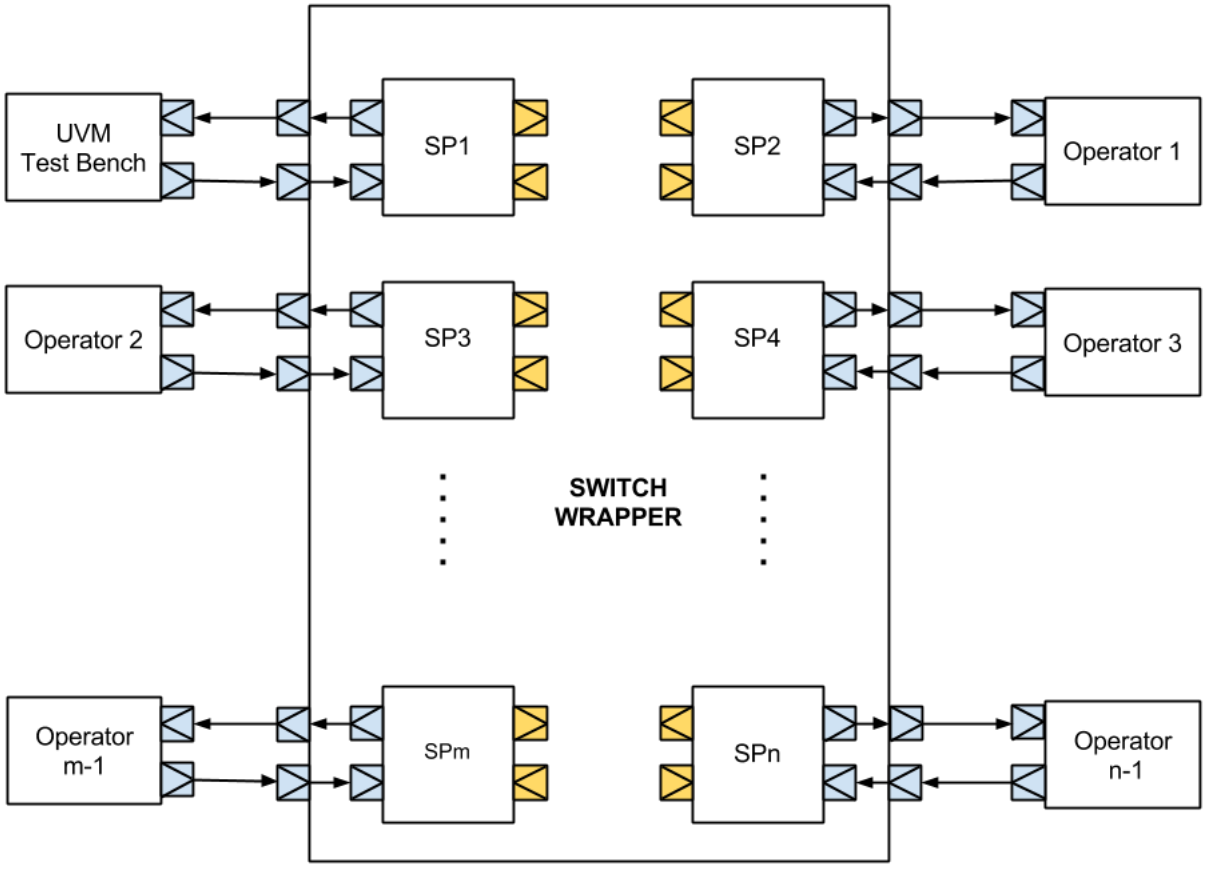

Figure 25: Complete Test System(*Multiport interconnections not shown)

The data bit width is suggested to be 80bits. This complete data is randomly

generated with constraints from UVM sequencer. The UVM driver will drive the input signals to the Freyja architecture. The data is sent as a transaction payload.

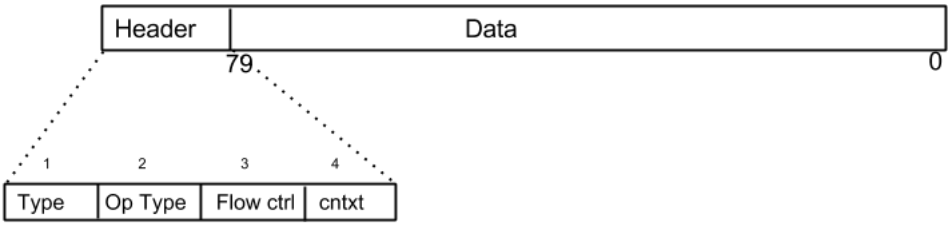

Figure 26: FBI Interconnect

5.2 Payload and blocking transport

Each generic payload transaction has a standard set of bus attributes: Command Address Data Byte enables Streaming width Response status The default values are set for the unused attributes. The address is initially set in the test bench and in the Freyja architecture; each operator control block will modify the address field based on the destination operator address. The data field is set to the result of the operator process blocks.

5.3 Input and output

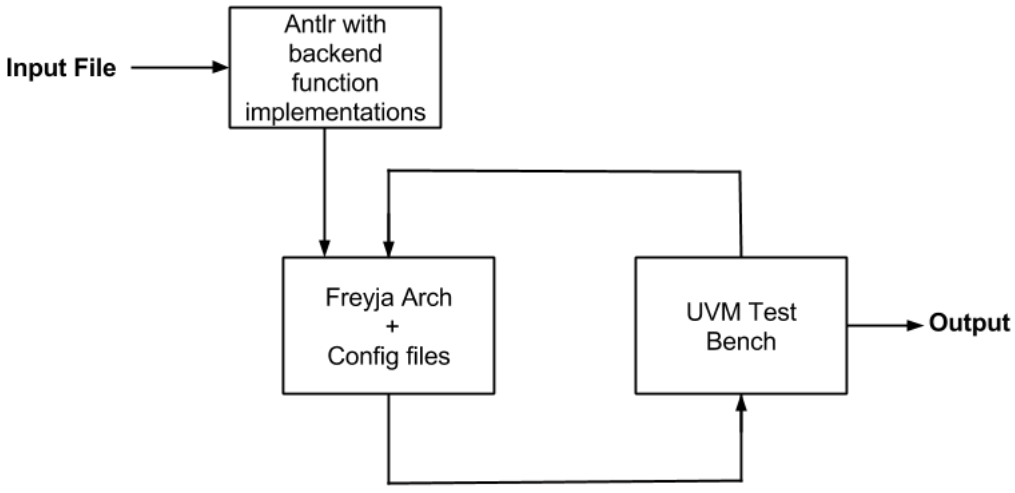

Figure 27: Input and Output of the system

The input file is the description of the protocol using the language defined by the EBNF grammar. The input file is parsed by the Antlr tool; the backend functions are triggered during parsing to output the Freyja architecture and the reconfigurable files. The UVM test bench will send the transaction to the Freyja architecture. The transaction is initiated from the UVM to the Freyja architecture and the final transaction is received back in the UVM. An example of instantiating 4 operator blocks as in Figure 19 is shown below :

```
Operator OPERATOR1{
sc_name : op1;
op_type : op11,op12;
errorid : 01,02;
}

Operator OPERATOR2{
sc_name : op2;
op_type : op21,op22,op23;
errorid : 03,04,05;
}

 Operator OPERATOR3{
 sc_name : op3;
 op_type : op31;
 errorid : 06;
 }

Operator OPERATOR4{
sc_name : op4;
op_type : op41,op42;
errorid :  07;
}
```

Listing 3: Freyja Overall architecture definition

5.4 Blocking transport and timing annotation

The transaction is sent through the socket using the b_transport method of the TLM2.0 blocking transport interface, which passes its transaction arguments by reference and has no return value. The b_transport also carries the timing annotation which is not configured as the main interest as of now is to model the functionality of the target and not modeling any timing detail.

6 Analysis

This chapter explains the analysis of the framework through the integration of Ethernet protocol into the Freyja architecture. Ethernet operators are integrated into the system of files generated from the Antlr tool. To show the context switching between different protocols, Xio-s CRC16 operator is also integrated. The first section analyses the process and results of Ethernet integration and section 6.2 explains the Xio-s protocol integration.

6.1 Ethernet

Ethernet Transmit and Receive operations are explained in Fig. 4 and Fig. 5. The *Datain* in the transmit is passed from the UVM test bench.The UVM sequence is generated with first byte indicating the protocol and the second byte containing the address of the operator. Once the transaction reaches the Switch wrapper,it is decoded and forwarded to the first operator as defined by the input file.

Appendix A shows the complete description of Ethernet protocol. The interconnection between different Ethernet operator is explained in this section.

A part of the Ethernet description is shown in code below:

```
1 Tx_PATH {
2 CONNECT : proc_tx to  crc_add;
3 CONNECT : crc_add to  pre_sfd_add;
4 CONNECT : pre_sfd_add to  cc_a;
5 CONNECT : cc_a to  enc;
6 CONNECT : enc to  scram;
7 CONNECT : scram to  gb_tx;
8 CONNECT : gb_tx to  gb_rx;
9 }
```

Listing 4: Freyja protocol Interconnection

The operators required by the protocols are first instantiated by defining them in the description. The string “protocol” indicates the definition of new protocol in Freyja architecture. The new protocol definition consists of 3 segments : Fields,Tx_path,Rx_Path.

- The Fields section is defined to indicate the details of transaction contents in each protocol
- Tx_Path is used to define the transmitter sequence and
- Rx_Path is used to define the Receiver sequence

The Tx_Path and Rx_Path are for the user to distinguish between the Transmitter and Receiver sequence for each protocol. In Freyja architecture the interconnections have no difference w.r.t Tx_Path and Rx_Path. The interconnect information is used to code the control block of each operator to forward the transaction to the correct destination.

A part of control block of CRC operator for the description of Ethernet protocol is shown below:

```
//part of fre_op_ctrl_crc.cpp block

if(type == ethernet){
    if(op_type == crc_add){
        data[0] = ethernet;
        sz = sz + 2;
        data[1] = pre_sfd_add;
        gp.set_address(crc_add_2_pre_sfd_add);
    }
    else if(op_type == crc_check){
        data[0] = ethernet;
        sz = sz + 2;
        data[1] = proc;
        gp.set_address(crc_check_2_proc);
    }
}
```

Listing 5: Freyja CRC operator Control Block code segment

In the above code the CRC operator's control block initially check for the protocol and then for the type of the operator. As explained in Fig. 24,the CRC operator multiplex 2 operations. CRC add and CRC check operators are having the same destination operator control block. They share the common memory,hardware for CRC computation and control function. The extra logic required to perform the CRC add and CRC compare are different.

The destination address of the transaction is obtained from the input file. This information is compiled in the form of constants in SystemC environment. The same information can be stored in shared or in the operator memory based on architecture requirement.

The parser implemented will generate these constants which are required for the reconfiguration of hardware. The context here is the possibility to send the transaction to different operators from the same source.

```
//constants for the protocol : ethernet
const unsigned char ethernet = 0;
const  int proc_2_crc_add=0;
const  int crc_add_2_pre_sfd_add=1;
const  int pre_sfd_add_2_cc_a=2;
const  int cc_a_2_enc=3;
const  int enc_2_scram=4;
const  int scram_2_gb=5;
const  int gb_2_descram=5;
const  int descram_2_dec=7;
const  int dec_2_cc_d=8;
const  int cc_d_2_pre_sfd_del=2;
const  int pre_sfd_del_2_crc_check=1;
const  int crc_check_2_proc=0;

```

Listing 6: Freyja Reconfiguration constants

The constants are formed using simple syntax as below

⟨ source operator ⟩ _2_ ⟨ Destination operator ⟩

The constant value assigned is calculated by the parser based on the Multipass through ports of switch port to which the source and destination operators are connected.

The CRC operator also contains 19 bytes of memory elements. These 19 bytes are used by both CRC32 and CRC16 Operators. Ethernet protocol uses only CRC32 and hence only crc_add and crc_check uses first 9 bytes of the memory elements. The parser supports the access of contiguous or individual element. After the control block informs the memory about the protocol and operator, the memory block will send the transaction to the process block which includes the constants required by the operator.

6.2 Xio-s

Xio-s protocol is breifly explained in section 2.2.2. The Xio-s protocol CRC block is integrated into the system. The control block will check for the type of protocol, prioritized according to the description in the input file. Similarly Operators are prioritized in the order they have been defined in the input file. Defining Xio-s after Ethernet protocol description will change the control block code segment as shown below.

```
//part of fre_op_ctrl_crc.cpp block
if(type == ethernet){
    if(op_type == crc_add){
        ...
    }
    else if(op_type == crc_check){
        ...
    }
}
else if(type == xios){
    if(op_type == crc_add){
        ...
    }
    else if(op_type == crc_check){
    ...
    }
}
```

Listing 7: Freyja CRC Ctrl block with 2 protocols

As illustrated in Fig. 8 and Fig. 9, Xio-s protocol requires 2 interfaces to transmit the transaction from one operator to multiple operators. The transactions are sent from *packet service* block to two other operators both in transmission and during receiving stage. This feature is included in the input description. The *Tx_path* and *Rx_Path* in each protocol description will allow the user to define multiple destination operators for the same source operator. The code below depicts a scenario of Xio-s protocol assuming **ps**, **crc_add_16** and **cc_a_xio** are the operators described in the input file.

```
Tx_PATH {
CONNECT : ps to crc_add_16;
CONNECT : ps to cc_a_xio;
}
```

Listing 8: Freyja One source to multiple destination interconnection

Similar logic applies to the receiver section which can be parsed to receive the transaction from different source operators. Assuming **fs1**,**fs2** and **aligner** are Xio-s protocol operators, the below code indicate the way to receive transaction from multiple sources.

```
Rx_PATH {
CONNECT : fs1 to aligner;
CONNECT : fs2 to aligner;
}
```

Listing 9: Freyja multiple source to one destination interconnection

But the Freyja Operator as explained in section 3.2.2 consists of only one interface. This need the change in architecture of operator units and a single interface will not be possible to model the Xio-s protocol. The operators can be customized by defining more strings in the operator instantiation stage. Once the architecture is modeled, the parsed input description can be used to output the architecture to match the interface requirement.

context switching at process block

7 Conclusion and Future work

This chapter concludes the thesis by describing the learning outcomes of reconfigurable architectures used for protocol processing and the framework design by grammar based language application. It also describes the limitations encountered during the thesis work and further section describes the future work specific to the thesis work and in general to the research topic.

7.1 Reconfigurable architecture

Ericsson AB investigated Switch and Mesh based network topology for efficient protocol processing. Based on performance,cost,scalability and other internal factors switch based network topology named as Freyja is considered for further implementation and research. The reconfigurable hardware implemented using systemC consisted of a central switch with all the operators connected to it as explained in section 3.2.

The design of a new architecture require manually setting up simulations,estimation of resources and synthesizing the hardware which consists of system,logic and physical synthesis. This is a time consuming process and decreases the productivity of the research. Research project called TACO(Tools for Application-specific Hardware/Software Codesign) at university of Turku, explains the functions,features and capabilities required by such a tool [23]. One such requirement is importing architecture details in the tool from systemC top level files. The same approach is carried in Freyja architecture by abstracting the systemC details into the high level description.

The main objective of this work is to show the complete process of mapping from high level description to the system level implementation. The features mapped consisted of different operator instantiation,interconnections,memory,error handling and routing to multiple destinations from each operators for different protocols. In general protocol processors are more complex including features like Flow control,FIFOs,arbitration mechanism,latency of each computation,dead lock avoidance etc.

The mapping of such finer details in the high level description will be straight forward with the approach presented in this work. For example the FIFOs can be instantiated similar to the Operator instantiation by extending the grammar. The FIFO block can be further configured with the variable depth of the buffer and signaling the overloading because of back pressure. How these details are represented in high level description depends on the architecture implementation. Another example of extension is to extend the grammar to hold the latency of each operator.

The same framework can be used at different levels of synthesis to output the required files for the architecture. This reduces the time to reconfigure and setup the environment. For all these to accomplish, the reconfigurable architecture should be defined. The framework can be adapted based on the architecture. The approach in this work shows the concept of mapping few core features which are easy to extend for more refined details.

Apart from abstracting high level architecture details, the core functions of each operator can also be represented using high level description. More research in academia and industry have already proved the concept. These tools input from Algorithms to Register Transfer level (ALU,Reg,Mux). This transformation is called as High level synthesis [24] [25].

7.2 Protocol sharing

As illustrated in section 3.4 the protocols share common functions. The features that can be shared are discussed at system level based on algorithms. The gain of sharing and reducing the hardware cost comes with the need of reconfiguration and back pressure in the system. The buffer length required to handle this and further details can be abstracted to the higher level to produce the reconfiguration content for the architecture. Automating the calculation of these resource utilization through simulation and importing the result to configure the hardware will reduce the architecture design time.

The implemented work shows the concept of mapping one resource to different protocols. Sharing requires arbitration algorithms to decide who can access the operator at any instant of time. It can be based on priority. However mathematical tools like Matlab can be used to implement the equations and calculate the resource usage. More utilized resources like CRC computation can be replicated for avoiding the back pressure. TACO project [23] provides a similar research conclusion for the protocol processor which exploits the resource sharing at logic levels. The tool identifies the common hardware blocks and creates multiple cores of them to meet the constraints. Similar approach can be extended with Freyja architecture for frequently used operators.

Some bottlenecks would be the memory interface and token access from higher layers. The system needs to be configured with different feasible combinations for higher throughput. In future,these details can be abstracted to high level description for configure the overall system.

7.3 Language framework

Programmers are building domain-specific languages, configuration file formats, network protocols and numerous data file formats as well as traditional programming language compilers and interpreters. The development of such systems can be made faster by using the language building tools.

Programmers tend to avoid using language tools, resorting to adhoc methods, partly because of the raw and low level interface to these tools [26]. Using grammar based approach to build parsers will offer a more natural, high fidelity, robust and maintainable means of encoding a language-related problem. Most grammar development is done today with a simple text editor. The ANTLR4 parser generator [27] attempts to make grammars more accessible to the programmer by generating recursive descent parsers that are very similar to what programmer would build by hand.

Antlr4 supports rapid grammar developemnt by using ANTLR's built in interpreter, thus, providing immediate feedback during development [28]. The parse tree associated with matching input (as in Fig. 14), helps in debugging for error in grammar definition. If the input sequence is not in the language recognized by specified start rule, Antlr4 inserts an error node into the parse tree to indicate where the problem occurred.

All the above features of Antlr4 tool helps to build the language application in a more structured way. Providing the framework in multiple languages makes the programmer comfortable to build with their skilled language. The most encountered problems are resolved with the community support and the book written from the author Terrence Parr [27].

7.4 Limitations

The thesis started with the focus of developing the complete framework and to show the concept of using grammar based language application in hardware/software co design. The reconfigurable architecture modeled in systemC abstracted more hardware details. Though language is developed to include all future requirements, the implementations in the backend has to be extended for the new features of the hardware. Even though 3 protocols and their functions are discussed, it is not targeted to integrate all the protocols. As stated in section 7.1 the high level description is targeted with respect to current Freyja architecture. The Antlr4 tool supports java language to output the framework and support for other languages are yet to be released.

7.5 Future work

Freyja architecture is still under implementation and in future, the complete details of the architecture can be abstracted into high level description by extending the grammar. Features like multiple core instantiations and transaction from different sources to same destination (required for Xio-s protocol) are possible to represent but still requires the backend implementations to modify according to the interconnection defined in future. Once the reconfigurable file format required by RTL hardware implementations are defined, the same framework can output such files to speedup the design process. As explained in section 2.1 the granularity of Freyja architecture is at algorithmic level. Common hardware blocks at logic levels and exploiting the reconfigurability for freyja architecture can be considered further and then the grammar can be extended similar to TACO protocol processor [23]. The header as explained in Fig. 26, can be described in the high level description which provides control in manipulating the length of each field.

The integration of Xio-s and CPRI protocols into the framework is straight forward. The process block i.e core functions of each operator needs to be modified to integrate these protocols into the system. Ethernet integration is set as an example. The grammar supports definition of multiple protocols in the language. The implementation of protocol functions doesn't require any modifications in the grammar.

Antlr4 is made to output the framework in Java language and the tool (in future!) will support other languages like c,c++. Based on the project requirement the language can be selected. Java has good commands for File IO operations and handling the objects. Major backend implementation deals with storing the parsed data in the form of Hashmap and Linkedlist. These facts can be considered when building similar concept for other reconfigurable architectures.

In general, the framework can be extended for all levels of synthesis while designing the hardware. The development environment will be heterogeneous at different levels and a common framework for producing the top level configurable files will reduce the errors. Once the architecture is specified, the tool can be made to generate SystemC model for simulation, a Matlab model for estimation of resources and VHDL model for synthesis of the architecture. The resource estimation can also be integrated within the tool by developing an interpreter using the parsed input.

References

[1] D. Szczesny, A. Showk, S. Hessel, A. Bilgic, U. Hildebrand, and V. Frascolla, "Performance analysis of lte protocol processing on an arm based mobile platform," in *System-on-Chip, 2009. SOC 2009. International Symposium on*, pp. 056–063, Oct 2009.

[2] A. Abnous and J. Rabaey, "Ultra-low-power domain-specific multimedia processors," in *VLSI Signal Processing, IX, 1996., [Workshop on]*, pp. 461–470, Oct 1996.

[3] K. Keutzer, S. Malik, and A. Newton, "From asic to asip: the next design discontinuity," in *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 84–90, 2002.

[4] M. Badawi, A. Hemani, and Z. Lu, "Customizable coarse-grained energy-efficient reconfigurable packet processing architecture," in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pp. 30–35, June 2014.

[5] Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga, "Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching," in *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 21–28, Dec 2010.

[6] G. Estrin, "Reconfigurable computer origins: the ucla fixed-plus-variable (f+v) structure computer," *Annals of the History of Computing, IEEE*, vol. 24, pp. 3–9, Oct 2002.

[7] D. Page and L. Peterson, "Re-programmable pla," Apr. 2 1985. US Patent 4,508,977.

[8] G. Genest, R. Chamberlain, and R. Bruce, "Programming an fpga-based super computer using a c-to-vhdl compiler: Dime-c," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pp. 280–286, Aug 2007.

[9] J. Mena, R. Deken, J. Coker, M. Johnstone, S. Ramirez, and P. Frey, "High level synthesis of a front end filter and dsp engine for analog to digital conversion x2013; a case study," in *VLSI Test Symposium (VTS), 2010 28th*, pp. 252–252, April 2010.

[10] R. Bergamaschi, "Bridging the domains of high-level and logic synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 582–596, May 2002.

[11] "Ericsson ab." http://www.ericsson.com/, 2015. [Online; accessed 1-March-2015].

[12] E. Panainte, K. Bertels, and S. Vassiliadis, "Instruction scheduling for dynamic hardware configurations [m-jpeg encoder case study]," in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 100–105 Vol. 1, March 2005.

[13] M. Keating, "The future of design," in *The Simple Art of SoC Design*, pp. 171–180, Springer New York, 2011.

[14] A. Waza, R. N. Mir, and H. N. ud din, "Reconfigurable architectures," *Journal of Advanced Computer Science & Technology*, vol. 1, no. 4, pp. 337–346, 2012.

[15] M. Shami and A. Hemani, "Partially reconfigurable interconnection network for dynamically reprogrammable resource array," in *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, pp. 122–125, Oct 2009.

[16] "Ieee standard for ethernet." https://standards.ieee.org/about/get/802/802.3.html, 2012. [Online; accessed 1-Feb-2015].

[17] W. Peterson and D. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, pp. 228–235, Jan 1961.

[18] "Cpri specification overview." http://www.cpri.info/spec.html, 2014. [Online; accessed 1-Feb-2015].

[19] "Ieee standard system c language reference manual," *IEEE Std 1666-2005*, pp. 0_1–423, 2006.

[20] "Tlm 2.0 transaction level modeling library and whitepaper." http://accellera.org/downloads/standards/systemc, 2009. [Online; accessed 1-Feb-2015].

[21] "Standard universal verification methodology class reference." http://accellera.org/downloads/standards/uvm, 2014. [Online; accessed 1-Feb-2015].

[22] S. H. Terence Parr and K. Fisher, "Adaptive ll(*) parsing: The power of dynamic analysis." http://www.antlr.org/papers/allstar-techreport.pdf, 2014. [Online; accessed 1-March-2015].

[23] S. Virtanen, T. Lundstrĺom, and J. Lilius, "A design tool for the taco protocol processor development framework," in *n Proceedings of the 18 IEEE NorChip conference, 6-7 November 2000, Turku, Finland.*

[24] A. M. Philippe Coussy, *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Netherlands, 2008.

[25] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *Design Test of Computers, IEEE*, vol. 26, pp. 8–17, July 2009.

[26] J. Bovet and T. Parr, "Antlrworks: An antlr grammar development environment," *Softw. Pract. Exper.*, vol. 38, pp. 1305–1332, Oct. 2008.

[27] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[28] M. Daly, Y. Mandelbaum, D. Walker, M. Fernández, K. Fisher, R. Gruber, and X. Zheng, "Pads: An end-to-end system for processing ad hoc data," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, (New York, NY, USA), pp. 727–729, ACM, 2006.

Appendices

A Ethernet Protocol Description

```
1
FunctionUnits FREYJA:
3
Operator PROCESSOR{
5 sc_name : proc;
6 op_type : proc_tx,proc_rx;
7 errorid : 01,13;
8 }
10 Operator CHECKSUM{
11 sc_name : crc;
12 op_type : crc_add,crc_check;
13 op_mem : 0x01,0x04,0xc1,0x1d,0xb7,0x20,0x00,0x04,0x03,
14 0x04,0x00,0x00,0x01,0x59,0x53,0x10,0x03,0x02,0x04;
15 errorid : 02,03;
16 }
Operator PRE_SFD{
19 sc_name : pre_sfd_add;
20 op_type : pre_sfd_add,pre_sfd_del;
21 errorid : 04,05;
22 }
Operator CORRECTING_CODES_ADD{
25 sc_name : cc_a;
26 op_type : cc_a;
27 errorid : 06;
28 }
30 Operator ENCODER{
31 sc_name : enc_64_66;
32 op_type : enc;
33 errorid : 07;
34 }
35
Operator SCRAMBLER{
37 sc_name : scram;
38 op_type : scram,descram;
39 op_mem : 0x01,0x00,0x00,0x00,0x80,0x00,0x00,0x04;
40 errorid : 08;
41 }
```

```
Operator GEARBOX_TX{
sc_name : gb_tx;
op_type : gb_tx;
errorid : 09;
}

Operator GEARBOX_RX{
sc_name : gb_rx;
op_type : gb_rx;
errorid : 13;
}

Operator FRAMESYNC{
sc_name : fs;
op_type : fs;
errorid : 10;
}

Operator DECODER{
sc_name : dec_66_64;
op_type : decdr;
errorid : 11;
}

Operator CORRECTING_CODES_DEL{
sc_name : cc_d;
op_type : cc_d;
errorid : 12;
}


protocol ETHERNET
fields : DA[06]
SA[06]
VLAN:0x8100
VLAN[02]
Len[02]
PL[20]


memory {
crc_add     :00 to 08;
crc_check   :00 to 07,09;
scram       :00 to 07:
descram     :00 to 07;
}


Tx_PATH {
```

```
92 CONNECT : proc_tx to  crc_add;
93 CONNECT : crc_add to  pre_sfd_add;
94 CONNECT : pre_sfd_add to  cc_a;
95 CONNECT : cc_a to  enc;
96 CONNECT : enc to  scram;
97 CONNECT : scram to  gb_tx;
98 CONNECT : gb_tx to  gb_rx;
99 }
100


   RX_PATH {
103 CONNECT : gb_rx to  descram;
104 CONNECT : descram to  decdr;
105 CONNECT : decdr to  cc_d;
106 CONNECT : cc_d to  pre_sfd_del;
107 CONNECT : pre_sfd_del to  crc_check;
108 CONNECT : crc_check to  proc_rx;
109 }
```

B Language Recognition terms

Language

A Language is a set of valid sentences which are composed of phrases,sub-phrases and so on.

Grammar

A Grammar formally defines the syntax rules of a language.

Syntax tree or Parse tree

This is a tree structure representation of a sentence.The leaves of the tree are symbols or tokens of the sentence

Token

A token is a symbol in a language like Identifier, keyword or an operator symbol.

Lexer

It performs lexical analysis by conversting input character streams into Tokens.

Parser

A Parser checks the sentence structure against the rules of a grammar.

Top down parser

It is a type of parsing strategy where one first looks at the highest level OF Parse tree (root) and works down the parse tree by reaching the leaf nodes.

Bottom up parser

In this parsing strategy the input text is processed from the lowest level to highest level (root).

Recursive Descent Parser

It is a kind of top down parser built from a set of mutually recursive procedures where each such procedure usually implements one of the productions of the grammar.

LookAhead parser

It defines the number of tokens accessible to the parser in making decisions at each point.

Left recursion

context free grammar

Regular grammar