Re-configurable hardware programming in a protocol processor unit

SUNIL KALLUR RAMEGOWDA

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

TRITA ICT-EX-2015:206

Acknowledgement

Reconfigurable hardware programming in a protocol processor unit

Sunil Kallur Ramegowda

October 26, 2015

Abstract

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

In general, a protocol processor processes the packets according to the protocol. There are number of protocols like Ethernet and CPRI to define how the data has to be sent and received between the source and destination points. Data packets can be processed using 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 a hardware model and its interconnections in SystemC TLM2.0 based on the protocol description. The hardware model is connected to Universal Verification Methodology (UVM) driver and functionality is verified by simulation. The Ethernet protocol is described using the developed language and the complete framework is verified.

Contents

Abbreviations

APIs Application Programming Interfaces.

ASIC Application Specific Integrated Circuit.

ASIP Application Specific Instruction-set Processors.

BBU Base-Band Unit.

BNF Backus Naur Form.

CLB Configurable Logic Block.

CPRI Common Public Radio Interface.

CRC Cyclic Redundancy Check.

DRRA Dynamic Re-programmable Resource Array.

DSP Digital Signal Processing.

EBNF Extended Backus Naur Form.

EDA Electronic Design Automation.

FCS Frame Check Sequence.

FPGA Field Programmable Gate Arrays.

GPPs General Purpose Processors.

HDLs Hardware Description Languages.

HLS High Level Synthesis.

ICs Integrated Circuits.

IEEE Institute of Electrical and Electronics Engineers.

MAC Media Access Control.

OSI Open Systems Interconnect model.

OVM Open Verification Methodology.

PCIe Peripheral Component Interconnect Express.

PE Processing Element.

PLA Programmable Logic Array.

RBS Radio Base Station.

RTL Register Transfer Level.

SFD Start of Frame Delimiter.

SoCs System on Chips.

SRIO Serial Rapid I/O.

TLM Transaction Level Modeling.

UVM Universal Verification Methodology.

VHDL Very High Speed Integrated Circuit Hardware Description Language.

VMM Verification Methodology Manual.

List of Figures

List of Tables

1 | Introduction

A set of rules define the communication strategy between digital systems. There are many rules which make 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 manufacturers and technologies. The OSI reference model partitions communication systems into 7 abstraction layers. It addresses 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 referred as *protocols* in communication systems. They help in the exchange of information between the digital systems. Most of the higher layers (4 to 7) are processed through generic computer systems and the lower layers with dedicated embedded hardware. General Purpose Processors (GPPs) found in embedded systems cover a broad range from 8 bit low cost microcontrollers, to 32 bit RISC

microprocessors, to today's high performance processors with Digital Signal Processing (DSP) enhancements. They can be considered as hybrid processors which incorporate signal processing features into embedded hardware that unify control and signal processing in a single core. In the communication domain, a network processor is a programmable microprocessor optimized for processing network data packets. Common functions include header parsing, pattern matching, bit manipulation, packet modification, shift and data movement. Software programmability of network processors allows it to be more flexible across a range of applications. Even though all network processors are programmable, the user might not be given access for programming, restricting the programmability to vendors. Other GPPs can be programmed for protocol processing as they are less expensive. Solutions based on GPPs or Application Specific Integrated Circuit (ASIC) exist [5] [6] for protocol processing. GPPs will have more flexibility but are less energy efficient when compared to ASIC which is less flexible but the most energy efficient. Application Specific Instruction-set Processors (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 [7].

Another class of hardware architecture referred as Reconfigurable architecture, are devices that contain programmable function blocks and programmable interconnects between function blocks. The most mature class of reconfigurable architectures is Field Programmable Gate Arrays (FPGA) which are considered fine-grained reconfigurable architectures. Research advances in this field have led to the Coarse grain re-configurable architectures and reconfigurable computing platforms. Hardware resource usage and performance varies depending on the reconfigurable architecture and its level of abstraction [8]. The design of coarse grain reconfigurable hardware architecture requires the compiler to produce the configuration or the hardware compatible code [9]. These files can be produced at 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 reconfigurable hardware is modeled in SystemC language using Transaction Level Modeling (TLM). The files required for the reconfiguration are obtained by parsing the description of protocols which is described in high level language. The 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 data.

1.1   Hardware and programming

Until the late 1960s, Integrated Circuits (ICs) were designed, optimized, and laid out by hand. Gate level simulation appeared in the early 1970s, and cycle based simulation became available by 1979. Place and Route, schematic circuit capture, formal verification and static timing analysis techniques were introduced during the 1980s. Meanwhile Gajski and Kuhn introduced the Y chart for describing the hardware as 3 different domains namely *Behavioural Domain*, the *Structural Domain* and the *Physical Domain* [10]. It represents different levels of hardware abstraction which are indicated by concentric circles as illustrated in Fig. 1. The three axes represents the 3 different domains and the synthesis can be viewed as a process of transformation from one axis to another and/or from higher level to lower level. The behavioural domain describes the functional behavior of the system. The structural domain maps the hardware into subsystems and show they are interconnected. The geometrical domain represents the geometric properties of the system and its subsystems. Each intersection point of domains and concentric circles represents different abstraction levels with respect to their domains. Simulation tools to represent the hardware are widely adapted during 1980s based on hardware description languages such as Verilog(1986) and Very High Speed Integrated Circuit Hardware Description Language (VHDL)(1987).

In the software domain, similar to the hardware domain, progress in design methodologies for programming the hardware has helped with today's sophisticated hardware independent compiler environment. Machine code was once the only means of programming the hardware. In 1950s, assembly level programming was introduced which abstracted the programming one step higher. With the compilers becoming more intelligent, high level languages evolved and associated compilation techniques were developed to improve software productivity. High level languages like Python, Perl, Java, PHP etc are platform independent and provide the user with flexibility and portability by hiding details of hardware architecture. Languages like C, C++ abstracts from assembly level, and together with C compilers, it helps in efficient object code generation. Programmers create interfaces to the C library so the routines can be used from higher level languages. With the new hardware architecture becoming more complex in nature, the software applications are developed to provide good solutions which are in parallel becoming more complex.

1.2   Reconfigurable systems

In 1960, Gerald Estrin proposed the idea of a fixed plus variable structure computer [11]. 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
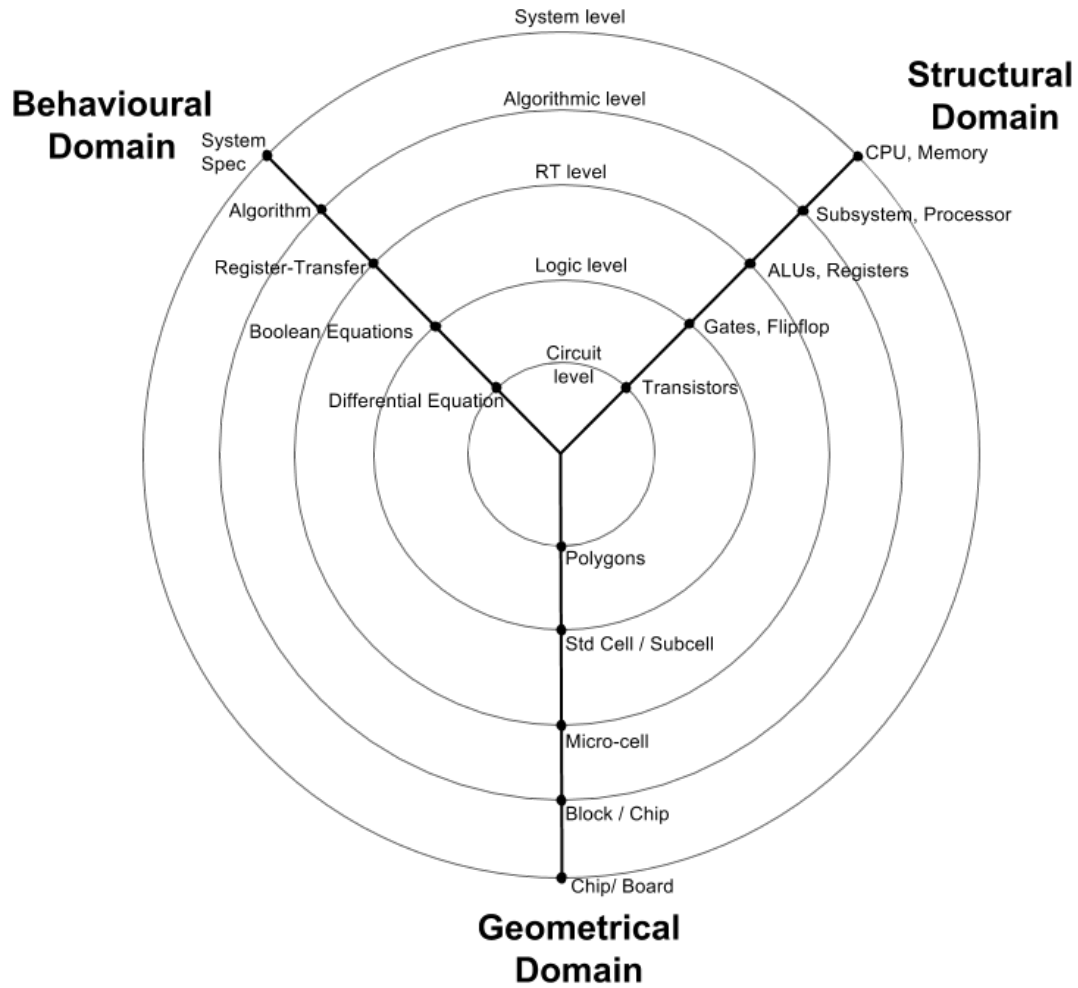
Figure 1: Domains and level of description in Gajski Y-chart

with a proof, the industry did not consider to further innovate in this field and till the 1980's there were no significant developments. In 1985, the reconfigurable Programmable Logic Array (PLA) was patented [12]. Innovation in PLAs further continued with the commercially available FPGA in today's market.

In the field of computer architecture, designers make decisions based on flexibility and performance requirements [13]. ASICs are the least flexible in terms of adapting for any change in the application, and GPPs 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 lie in the extreme corners of the graph between flexibility Vs performance as shown 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 [13].
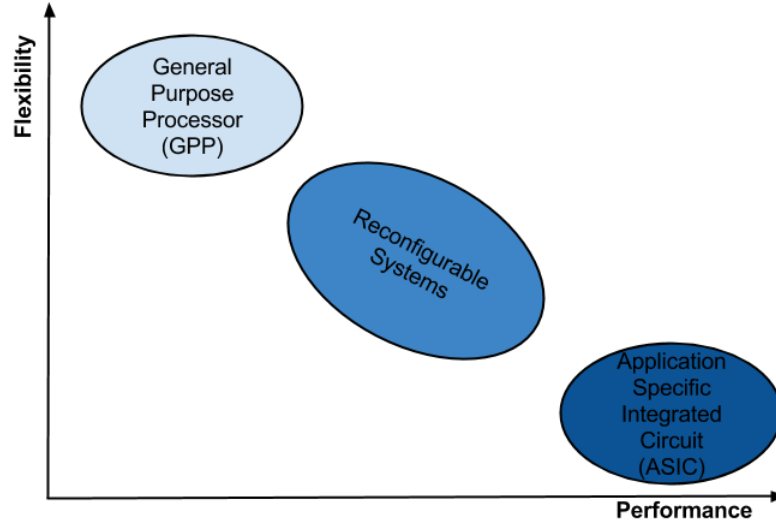
Figure 2: Flexibility Vs performance of hardware classes

1.2.1 Granularity

Reconfigurable devices like FPGA have the Configurable Logic Block (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 have to be considered based on the FPGA architecture. This level of granularity in implementing the functions is called Fine grained Reconfigurable Architecture as it provides the reconfigurable granularity to lowest possible level. These reconfigurable devices are not energy efficient and the execution speed is less than the ASIC counterpart. Another type of reconfigurable devices are the coarse grained reconfigurable architectures. These devices have the granularity at function level. They will configure the function blocks to achieve efficient algorithm implementation. The function blocks can vary from constant blocks to complex functions which are commonly used by the application.

1.2.2 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.
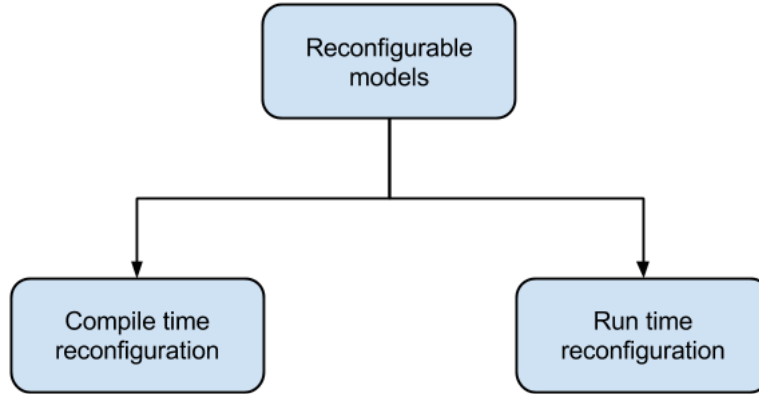
Figure 3: Reconfiguration models

In the 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. An FPGA configured to perform floating point multiplication together with a GPPs will accelerate the performance of the application if the GPPs doesn't have a Floating Point Unit. In the 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 Re-programmable Resource Array (DRRA) fabric developed at KTH Electronic System Dept is an example of this model [14].

1.2.3 Reconfiguration rate

The Fine graine systems will have more reconfiguration data (in FPGAs, it is in term of bit streams), which leads to more configuration 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 reconfigure. This depends on the dynamic reconfiguration architecture whether the complete fabric is reconfigured or partially reconfigured during runtime.

1.3   Purpose of this thesis

Ericsson AB [15] 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 Institute of Electrical and Electronics Engineers (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 between chips at high data rate. Most of these Media Access Control (MAC) layer protocols share common hardware functions. Ericsson design and manufactures custom ASIC chips for Baseband signal processing and Radio signal processing. The data is received and transmitted from these chips by one of the agreed protocols. A reconfigurable hardware can be designed to route the data using different protocols.

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 a 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.

The purpose of this thesis 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 TLM 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.

1.4 Problem description

As explained in the previous section reconfigurable architectures with low 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 [16]. The know-how of the process to build the complete system which solves the practical problems is of great importance. Fig. 4 from the International Technology Roadmap for semiconductors shows how hardware and software design productivity has lagged Moore's law. The need of configuration data for the reconfigurable hardware requires the design of new software tools.
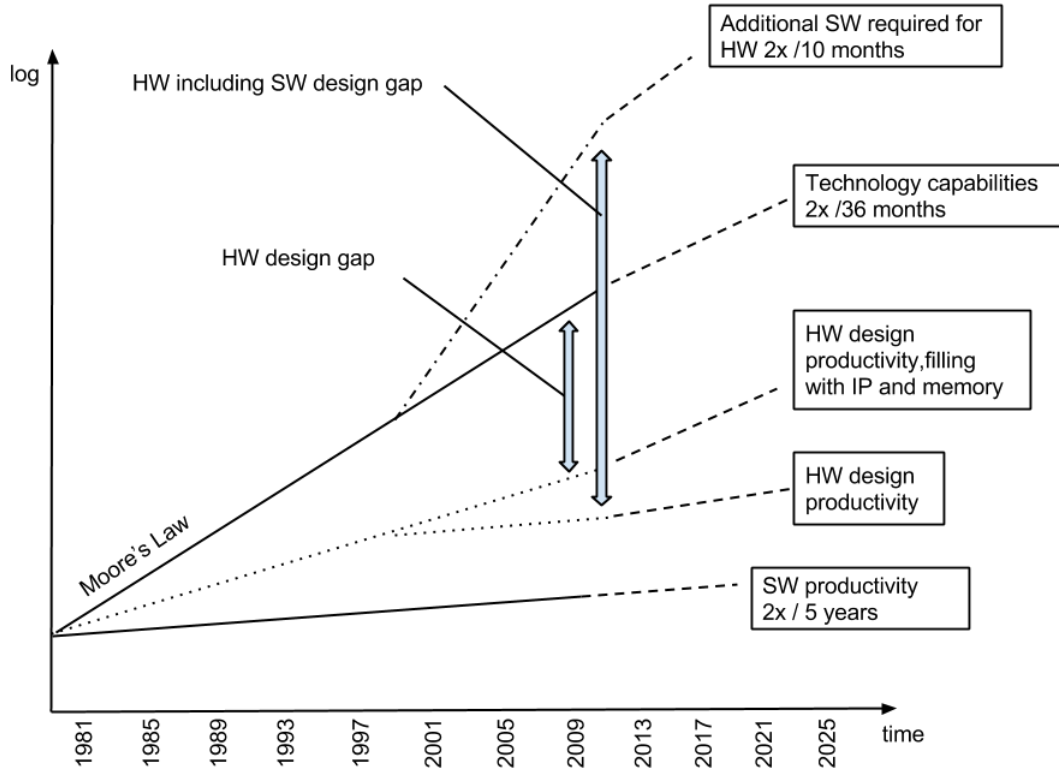
Figure 4: Hardware Software Design Gaps versus Time [1].

The RBS receives and transmit data using different protocols. Identifying errors and providing security are some of the main features for reliable communications. Algorithms used by different protocols for providing such features differ by minor polynomial change. Identifying such functions is required for efficient reconfigurable hardware design. The designer needs to have the overall understanding of hardware and software for such a system. 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. High level language development provides flexibility for the designer to produce reconfiguration data in their own syntax. The development time needs to be evaluated for such an ap-

8

proach. This allows the user with minimal know how about hardware to reconfigure and show results in short time.

1.5 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.

1.6 Limits on scope

The thesis focuses 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 protocols 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.

1.7 Structure of the thesis

The thesis is organized to provide the 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. The three different protocols considered in protocol processors are discussed in detail.

Chapter 3 : This chapter starts with the introduction of system design languages. The complete details of different hardware modeling languages, their comparison and verification methodology is discussed. The later section explains the meaning of a Grammar, the Language and the Parser in computer science world. Then the process of building a language application is explained using Antlr tool.

Chapter 4 : The chapter starts with the comparison of three different protocols discussed in background. The next section explains the complete details of Freyja architecture. It also discusses 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 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 to highlight the control block changes.

Chapter 6 : 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 7 : This chapter concludes the thesis by explaining the outcome of connecting three dots namely reconfigurable hardware, protocols and High level description. The limitations encountered and the future work is discussed further.

2 | Background

To understand the reconfigurable hardware and its terminologies, this chapter explains in detail about different architecture classes and their meaning. Protocols and their common functions are also explained, which helps in designing the reconfigurable protocol processor.

2.1 Fine grain reconfigurable systems

Fine grain reconfigurable logic arrays have evolved into commercially available FPGAs. They are a scalable chip architecture based on a 2D array of simple computational cells with individually configurable processing functions and an electronically configurable interconnect structure allowing complex application circuits to be built from the available cells [17].
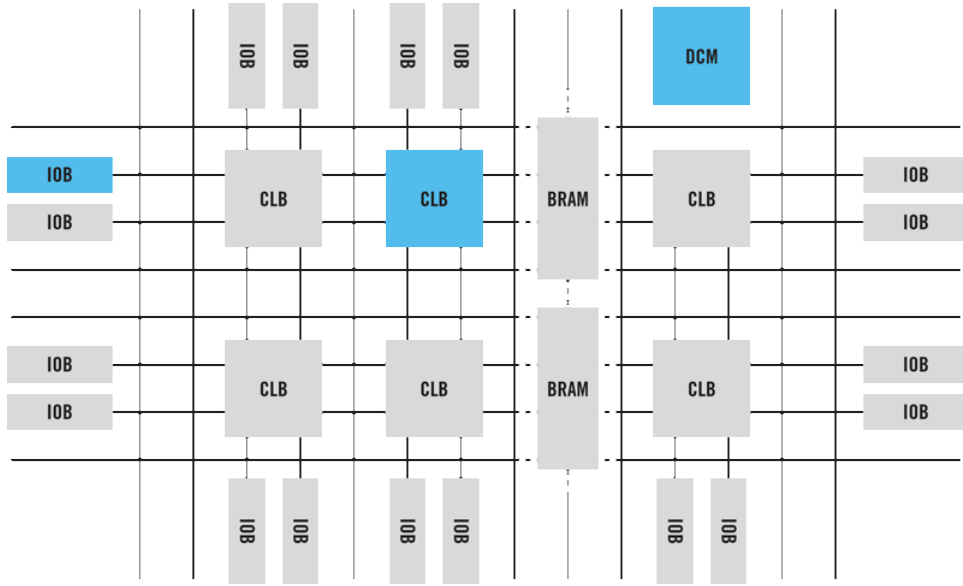
Figure 5: Field Programmable Gate Array [2]

Fig. 5 shows the FPGA architecture with a matrix of CLB connected through programmable interconnects. The CLB is the basic logic unit in a FPGA wherein the number of such units in each device varies depending on cost. Each CLB consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry and flip-flops. The highly flexible switch matrix can be configured to handle combinatorial logic, shift registers or RAM [2]. IOBs refers to the Basic select IO structure.

The interconnect routes the signals between CLBs and to and from IOs. Most of the modern FPGA boards include the Embedded block RAM and Digital Clock Management (DCM).
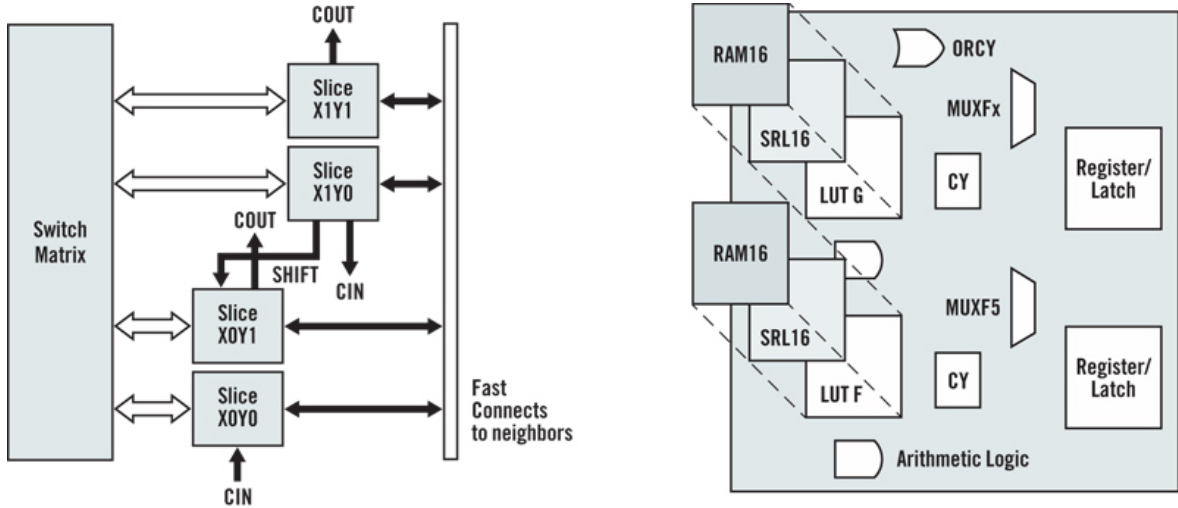
Figure 6: Configurable Logic block [2]

The introduction of FPGA 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. They are attractive compared to the ordinary CPUs because of less power consumption per computation and price per performance [18]. Despite these merits, FPGAs have still achieved limited applicability in industries. The main reason is that most software programmers lack the knowledge of hardware description languages which are used to describe the complex algorithms in FPGAs. FPGA programming requires a more rigorous development process, involving training the programmers beyond application level, resulting in an increase in development cost and time to market.

To overcome the problems stated above, research on mapping high level description to Register Transfer Level (RTL) level started. The growing capabilities of silicon technology and the increasing complexity of applications in recent decades have also forced design methodologies and tools to move to higher abstraction levels. The methodology is called High Level Synthesis (HLS). It enables the automatic synthesis of high level, untimed or partially timed specification (SystemC) to a low-level cycle accurate RTL specifications for efficient implementations in ASICs or FPGAs. During the 1990s, the first generation of commercial HLS tools were avail-

able [19]. HLS tools helped the designer to use High Level Languages (HLLs) for FPGA programming similar to processor programming. The latest generation of HLS tools, in most cases, uses either ANSI C, C++, or languages such as SystemC that add hardware-specific constructs such as timing, hardware hierarchy, interface ports, signals, explicit specification of parallelism, and others [19]. Some of the commercial HLS tools are Mentor's Catapult C, Forte's Cynthesizer, Cadence C-to-Silicon, NEC's CyberWorkbench and a new French company *Synflow* tool has its own language named Cx.

FPGA requires the reconfiguration of programmable fabric either before the program execution or partial reconfiguration on the fly when the program is running. The streaming and multimedia application requires the reconfiguration of fabric on the fly to optimally utilize the fabric. As FPGAs have granularity at the lowest possible bit level, the reconfiguration data is huge and the time for the hardware to reconfigure increases with the complexity. Most of the data handled in real world is either in byte or word width and reconfiguring the interconnect at bit level can be abstracted to higher level. This reduces the overhead of reconfiguring each bit. Similarly, the commonly used combinatorial logics can be defined as standard blocks for the custom design with the standard set of reconfiguration details. These features will reduce the amount of data and time required for reconfiguration and such a hardware class is called Coarse Grain Reconfigurable Logic.

2.2   Coarse grain reconfigurable systems

To overcome the limitations of Fine grained reconfigurable systems, new architectures were explored. Optimally designed processing elements which perform word level data processing are configured with few configuration bits at word level. Due to the word level reconfiguration, a small number of configuration bits is required, resulting into a massive reduction of configuration data, memory needs, and reconfiguration time [20]. Even the interconnections, since they are grouped in buses, they are configured by a single control signal instead of separate control signal for each wire. Also, because few programmable switches are used for configuration purposes and the Processing Element (PE)'s are optimally-designed hardwired units, high performance, small area, and low power consumption are achieved [20].

A generic architecture of a Coarse grain reconfigurable system is illustrated in Fig. 7. It consists of a set of Reconfigurable units (RU), a programmable interconnect, a configuration memory and a controller. The coarse grain reconfigurable part will be designed to take the computationally intensive parts of the application, and in most cases will be coupled to the main processor which takes control of other tasks. Coarse grain architectures are always optimized for the target domain. The
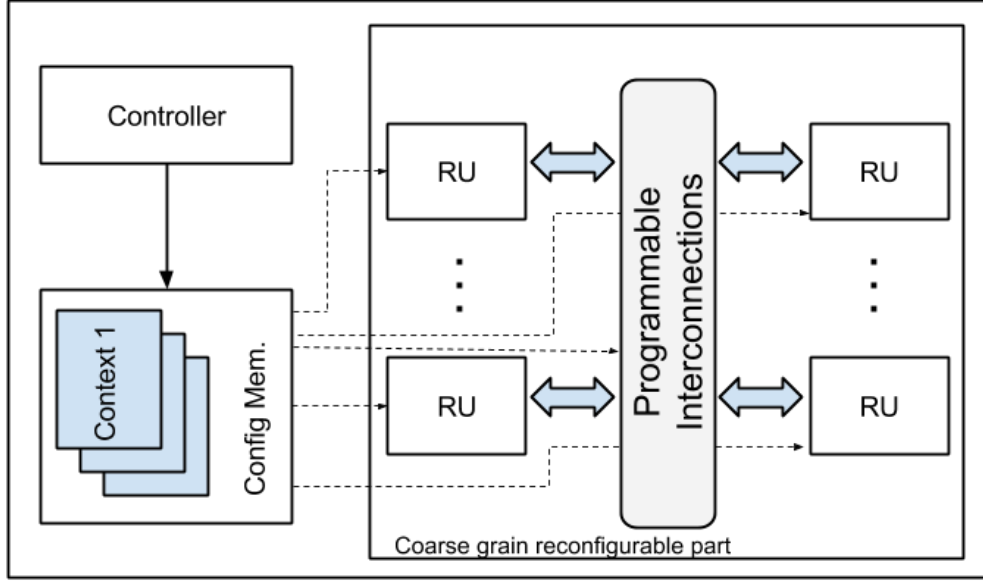
Figure 7: Basic Coarse grain reconfigurable architecture [3]

number of reconfigurable units, their design, the interconnection network, memory and controller are tailored to the domain's needs. The memory will hold the control (configuration information) bits that are used to program the reconfigurable units and interconnection network. The configuration memory may store multiple configuration contexts but only one context is active at a time [20]. The controller is responsible to control the loading of the configuration context from the main memory to the configuration memory, to monitor the execution process of the reconfigurable hardware and to activate reconfiguration contexts.

The programmable interconnection network ensures the communication of data between the computing reconfigurable units. The wires are grouped into buses and are configured by single configuration bits. The interconnection network can be realized by a crossbar, mesh or a mesh variation structure.

The reconfigurable units are the processing units, which are domain specific hardwired units, perform useful operation to accomplish the application requirements. The operations might refer to logic or arithmetic operation. They reconfigure autonomously based on the control information and are therefore different from the CLB's of fine grain architecture. Each unit is configured at word level, configuration bits reconfigure the entire unit and not each slice at bit level. Theoretically, the granularity might vary from bit to any word length. In most of the practical

architectures which are designed, the granularity is either 8bits or above, as the processing of data happens with the word length of 8bits or more.

The reconfigurable units in the architecture are optimized hardware units for the application domain. They can be designed to perform any word level arithmetic or logical operations. As coarse grain reconfigurable systems target a specific domain, the RU's are designed with the operation required by the domain in mind. The reconfigurable units are hardwired units optimized to perform for specific application domain to improve performance, area and power consumption.

The GPPs with programmed embedded software is the classical approach which tightly couples data and control flow for many applications. It can be coupled with the general purpose embedded processors in different ways. The design depends on the performance and requirements of the application. Three different scenarios are illustrated in Fig. 8. These additional hardware units coupled to the processor will accelerate the performance or increase the throughput of the overall system. This illustration also helps to understand the different ways of reconfiguring the fabric.
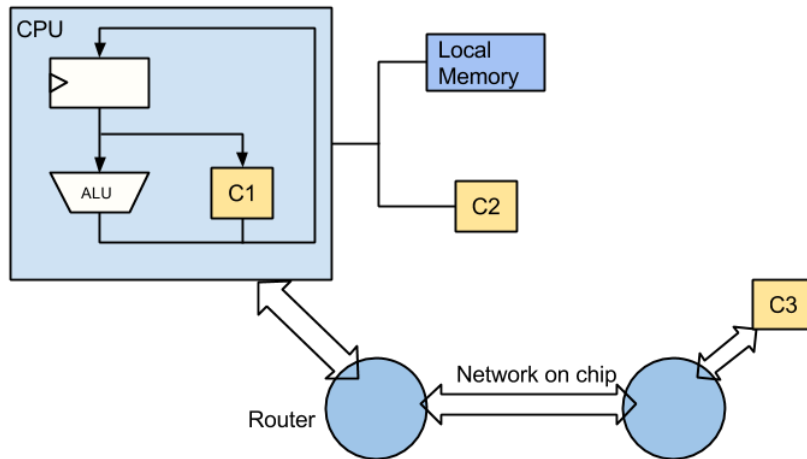
Figure 8: Coarse Grain Reconfigurable blocks [3]

Register mapped reconfigurable blocks give the tightest integration with embedded software. They are created by modifying the micro-architecture of the embedded core. For example as in Fig. 8, C1 is an additional data path next to ALU. For programming such an architecture, it should be visible in the instruction set of the embedded core. Since it requires tight coupling, exploiting the parallelism with the introduction of a new hardware unit becomes cumbersome. The tools need to provide such features for the user to map the application to the new hardware architecture.

Memory mapped reconfiguration is the next variant illustrated as C2 in Fig. 8. This is achieved by providing a memory interface to the reconfigurable block. It results in looser coupling between the software and the reconfigurable block. A set of shared memory locations can convey control and data flow orientated information. As explained in [3], reconfigurable blocks need to share memory address space with other peripherals. The control and data flow information might need to be routed through the CPU which increases the bottleneck.

In Network mapped reconfigurable blocks (C3), the network packets have the control and data flow information. In this case integration of embedded software and reconfigurable blocks can be done using communication primitives. As discussed in [3], Network mapped systems deviate more from the classic sequential model and hence the programming model is more complicated. Table.2 shows the comparison of different coarse grain reconfigurable systems.

Mapping	Architecture Strategy	Reconfiguration Mechanism	Data flow / Control flow Coupling	Energy Efficiency Improvement
Register-Mapped	Custom Datapath	Custom Instructions	Tightly Synchronized	Low
Memory-Mapped	Co processor	Memory mapped instructions	Loosely Synchronized	Medium
Network-Mapped	Peer processor	Configuration Packets	Uncoupled	High

Table 2: Coarse Grain Reconfiguration Mechanism [3]

2.3 High level synthesis

High level synthesis is the process of translating from algorithmic to Register transfer level description as illustrated in Fig. 1. Transforming application level programs which are described in high level languages directly into register transfer level will not require the programmer to know the detailed hardware architecture. High level synthesis tools explore the design space based on information from the high level abstraction with more degrees of freedom. There are many academic and industrial projects showing the concept of HLS [21] [22]. They operate on internal models

known as Control/Data Flow Graphs (CDFG) and produce a RTL model of the hardware implementation [23]. The computationally intensive applications like DSP algorithms were the target in the initial research of HLS tools.

A brief introduction to the academic and industry standard high level synthesis tools is discussed here.

GAUT [24] is an academic High level synthesis tool dedicated to Digital signal processing applications. It is an open source tool which takes a pure C function along with the constraints to extract the potential parallelism and data dependencies. After the allocation, scheduling and binding tasks it will generate the RTL code.

Cynthesizer [19] is a pin and protocol accurate SystemC model used as synthesis input of the HLS tool. It produces the optimized RTL for the specified target technology identified by the user in the form of a .lib file.

A new approach called as *interactive synthesis methodology* is explained in [25], which takes C language as input and allows the user to control and change the synthesis decisions about scheduling, allocation and binding by using the Graphical user interface (GUI) at any time.

DRESC [26] presents a retargetable compiler for a family of coarse grain reconfigurable architectures. It focuses on loop level parallelization for different segments of application code and uses a modulo scheduling algorithm for mapping into hardware blocks.

A generic compilation framework for architectures based on the dataflow execution paradigm is explained in [27]. It describes a method to transform applications described in HLL to Data Flow Graphs (DFGs) and a technique to optimize the same.

All the above tools help the designers in exploring the hardware architectures with more degrees of freedom. In other words, the hardware architecture changes based on the designer input. In most of the coarse grain architecture designs, the synthesis tools are developed for the custom optimized hardware architectures by assuming that designer knows the RTL implementation of the hardware. Few such cases are discussed below:

VESYLA (VEctorizing SYmbolic Language Assembler) [28] is a semiautomatic framework for implementing DSP functions. This tool is developed for the coarse

grain reconfigurable architecture designed at KTH called as DRRA (Dynamically Reconfigurable Resource Array). It takes an untimed C specification of a DSP function with pragmas and generates configware for the DRRA architecture.

The RaPiD (Reconfigurable Pipeline Datapath) [29] architecture is a coarse grain architecture that allows pipelined computational structures to be constructed from an array of arithmetic units, registers and memories. Programming is performed using RaPiD-C, a C-like language with extensions to explicitly specify parallelism, data movement and partitioning [4]. The compilation process produces a structural specification with components specific to the underlying architecture.

PipeRench [30] is a coarse grain reconfigurable system consisting of stages organized in a pipeline structure. It uses a *pipeline reconfiguration* technique to provide fast partial and dynamic reconfiguration, and it also provides runtime scheduling of configuration and data streams. Programming such a complex model is performed using the source language called *dataflow intermediate language* (DIL), which is a single assignment language with C operators. After parsing, the compiler inlines all modules, unrolls all loops, and generates a straight-line, single-assignment code [4].

Pleiades [31] is a coarse grain reconfigurable template with heterogeneous processing elements, optimized for a given domain of algorithms, and which allows runtime reconfiguration. The detailed architecture is discussed in [4], which indicates the mapping concept divided into 2 parts. The first task is to create the template instance, and the other is to map an algorithm onto a processor instance. A comparison of power and performance with respect to a general purpose processor is used to derive the resources for the architecture.

Montium [32] is a coarse grain reconfigurable architecture which resembles a Very Long Instruction Word (VLIW) architecture. It is optimized for the typical operations like correlation, finite impulse response (FIR) filters, matrix and vector multiplication, 8x8 point Discrete Cosine Transform (DCT) and Fast Fourier Transform (FFT). Programming this architecture consists of transforming C to directly architecture dependent code provided kernel written in C code is available. Otherwise C language is translated into CDFG, clustered and finally into Montium C code.

The design of custom hardware architectures require a framework for producing the executable code and/or reconfiguration data. Most of the EDA tools are not mature enough to explore the coarse grain architectures. The projects discussed above develop their own framework for mapping application into their architecture.

2.4 Protocols

The communication between the chips in Radio Base Station (RBS) equipments has many protocols to fulfill the requirements of the specification. The protocols differ by standards. The most commonly used protocols in the wireless base stations are discussed briefly in below section.

Gb Ethernet (GbE) : Is used for low rate signaling paths for which latency is not critical. It supports communication over a wide variety of physical media and transmission ranges, and is therefore well-suited for inter-module communication.

10Gb Ethernet (10GbE) : It is increasingly being used for critical low-latency, fast-path communication. The data rate is consistent with the throughput and latency requirements of typical macrocell base stations. Like all Ethernet variants, 10GbE can support communication over a wide variety of physical media and transmission ranges and is well suited for inter module communication.

Serial Rapid I/O (SRIO) : It is used for low latency, fast path communication, particularly within the Base-Band Unit (BBU). SRIO is a flexible serial interface standard which is well suited to these requirements. It offers the possibility of a very high throughput (5 Gb/s and 6.25 Gb/s) per lane with a low latency. The range of components which support SRIO is more limited than Ethernet, but many components associated with wireless baseband processing have adopted it as their principal high-speed data interface. SRIO is less widely used on GPPs; hence, its utilization is often limited to the BBU applications.

Peripheral Component Interconnect Express (PCIe) : It is sometimes used for both slow and fast-path communication within the base station. It is able to support high data rates (5 Gb/s per lane and 8 Gb/s) and has low latency. It is used less widespread in wireless base stations than either Ethernet or SRIO, and it is normally used to provide connectivity to a GPPs.

Processor local buses : They have been used to provide intra-module communication between a host GPPs and its peripherals. Such interfaces are used to support both slow- and fast-path communication. This type of architecture is now rarely used in new designs and has largely been superseded by serial interfaces and/or Ethernet-based interconnect systems.

Proprietary interfaces like Xio-s : They are used to provide slow- and fast-path communication within the base station when the system designer has control of both ends of the link.

The next section will explain in detail about the 3 protocols which are of interest for the reconfigurable protocol processor. It will explain the different operations performed at each level of protocol processing.

2.4.1   Ethernet

Ethernet is a widely used protocol for data communication. It is typically used in Local Area Network (LAN) applications. The IEEE organization has standardized the protocol and revises it according to the technological advancement. The most recent standard available is from 2012 [33] and it defines the protocol for different applications.

Ethernet transmit

The Ethernet transmit sequence is shown in Fig. 9.
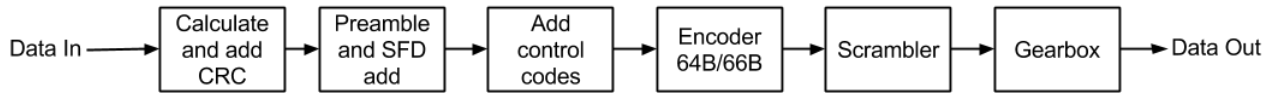
Figure 9: Ethernet Transmit

The *DataIn* is from the higher layers of the protocols which have the data to be transmitted using MAC layer protocol. Physical layer protocols are out of the scope of this thesis.

Ethernet receive

The Ethernet receive sequence is shown in Fig. 10. *DataIn* come from the physical transmission layer and *Dataout* goes to the higher protocol layers.
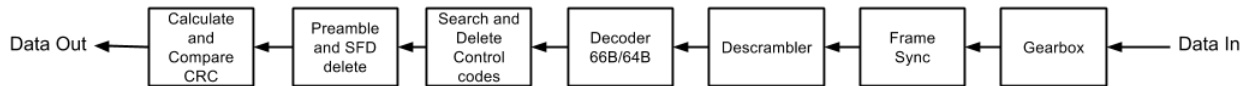
Figure 10: Ethernet Receive

20

Ethernet raw frame

The Ethernet raw frame format is represented as in Table. 3. Each of these frames enters the transmitter *Datain* as in Fig. 9. It exit as *Dataout* in the receiver as shown in Fig. 10.

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 3: Ethernet raw frame

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

CRC

The 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 memory [34]. The remainder of the division is known as the Frame Check Sequence (FCS).

On the transmitter side, the FCS is computed for the incoming data and appended as the 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

The 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).

Preamble	10101010 10101010 10101010 10101010 10101010 10101010 10101010
SFD	10101011

Table 4: Preamble and SFD

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

Control codes

The Add control codes block will add the control codes for the incoming data such that the Encoder block can use the 8 octets to encode the data based on these 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	Idle	Idle	Terminate	Data 4	Data 3	Data 2	Data 1	Data 0
Control word	1	1	1	0	0	0	0	0

Table 5: 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
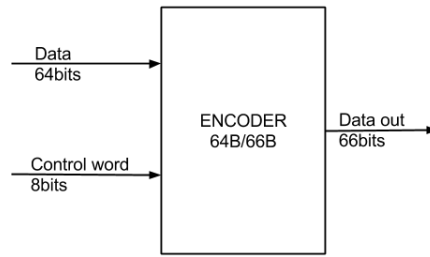
Figure 11: Ethernet Encoder block

The Encoder block is represented as shown in Fig. 11. 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" corresponds to data and "01" corresponds to control codes.

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

Scrambler and De-scrambler

This block is used to randomize the signal so that long sequences 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.
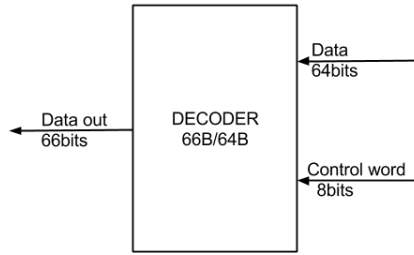
Figure 12: Ethernet Decoder block

Gearbox

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

2.4.2 Xio-s

Xio-s is a Ericsson proprietary protocol used for communication between the chips.
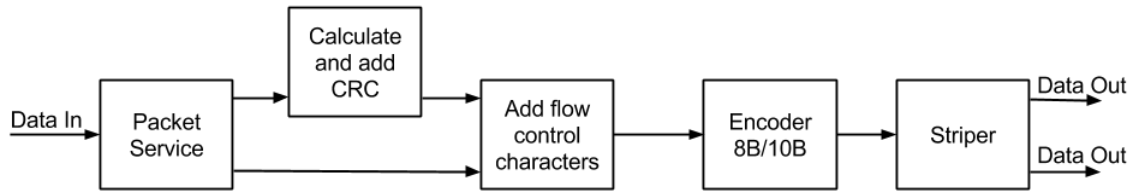
Figure 13: Xio-s Transmitter
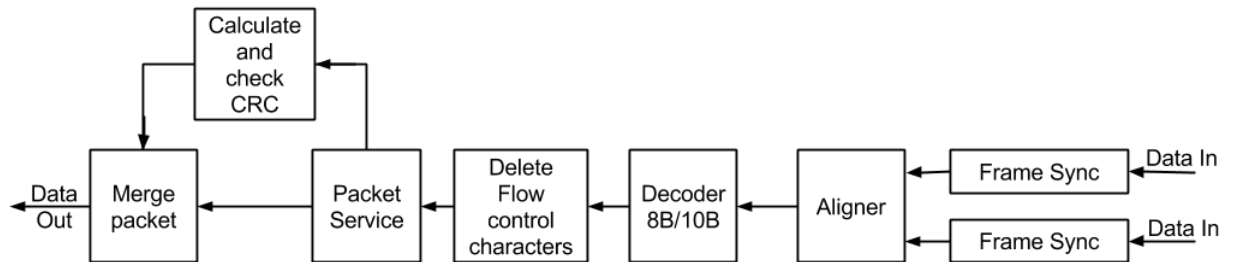
Figure 14: Xio-s Receiver

There are seven different types of packet services for Xio-s protocol. These assign the packet to the proper channel based on the service type. As an example, if the service type is format 1, then it uses only CRC16 channel. Or if the service type is format II, then it uses both CRC16 and CRC32 channels.

23

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

Figure 15: Xio-s Frame format Type 1

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

Figure 16: Xio-s Frame format Type 2

CRC

There are 16bit and 32bit CRC calculations required in the Xio-s protocol. So the polynomials for CRC16 and CRC32 are stored in 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 octet, 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 the encoder and thus the output of decoder will be the same as the input of encoder.

Striper

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

Frame sync

The objective of the frame synchronization is to detect the beginning of new valid data. After the sync header is found the frame sync module outputs 66 bit blocks in the same rate as it gets the incoming blocks, the difference is that the outgoing blocks are skewed according to the placement of the sync header so that the outgoing block starts with the two bit sync header. The frame sync module will not change the incoming data, it will just adjust the boundaries of the data so that its outgoing data comes with the sync header first.

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.

2.4.3   CPRI
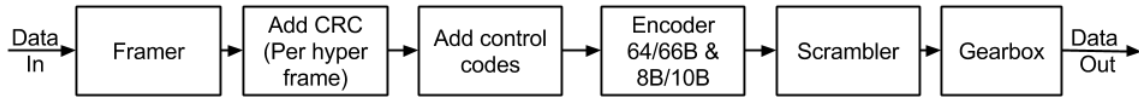
Figure 17: CPRI Transmitter

CPRI is an industry co-operation 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) [35]. 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 the Ethernet protocol. The frame structure is similar to the Xio-s protocol.
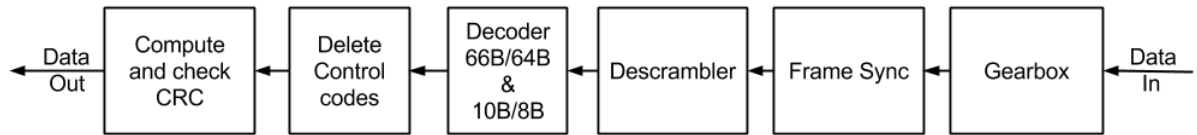
Figure 18: CPRI Receiver

3 | Modeling concepts

This chapter explains in detail about the languages used for system level modeling. It is followed by the methodology used to verify the system. Why and how a new language is developed for programming the architecture at Ericsson AB is explained in the section thereafter. The next section explains the meaning of grammar and language and its terms. It is further followed by the introduction of parser generator tool called Antlr.

3.1 Introduction

The hardware architecture and its interconnections need to be understood to design the protocol processor. The reconfigurable architecture developed at Ericsson AB is called Freyja architecture. The hardware architecture details have to be abstracted in order to be defined in a high level description. Similarly, the Protocols details have to be represented using the same description to reconfigure the hardware for different protocols. Mapping between them requires a custom set of configuration data to define and reconfigure the architecture according to the requirement. To accomplish this, the task 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.

To study the reconfigurable hardware architecture developed at Ericsson AB, one has to understand the modeling language and the reason for its usage. So the next section will provide details about the different system level modeling languages and their abstraction details.

3.2 System-level modeling

A hardware system requires modeling concepts that allow designers to explore different aspects of the system. These concepts include communication, time, structure, hierarchy etc. As depicted in Fig. 1, there are different levels of abstraction and domains in which the development phases vary. For example, in software, designers

carry out performance exploration to identify bottlenecks with respect to time, excluding the lower level hardware details. The Y chart maps the different modeling styles of hardware as 3 different domains. In order to represent the hardware in these domains at different abstraction levels, an executable model is required. The designer should be able to model the different aspects of hardware, it should be scalable, possible to simulate, test and verify the functions. The Register transfer level description of the hardware is synthesized into a gate level netlist by logic synthesis tools. This level of hardware abstraction describes the signal transition between synchronously clocked registers. Traditional modeling languages have emphasized the hardware design flow, whereas the recent languages have incorporated concepts for efficient modeling at system level. With traditional languages like VHDL/Verilog, which are more used for RTL level, it takes too much time to develop for architectural exploration and software development. They also have poor performance i.e takes more time to simulate complex systems and are not available early in the development phase. In this section, Transaction Level Modeling (TLM) is introduced which is an alternative to traditional approaches for efficient abstraction and exploration of architectural details. Transaction level modeling is supported in languages such as SystemC and SpecC by abstract channels that connect communicating modules.

3.2.1   Language comparison

Several languages have emerged to address various aspects of system design. Fig. 19 illustrates the comparison of different languages. Although Ada and Java have proven their value, C/C++ is predominately used today for embedded system software. The Hardware description languages, VHDL and Verilog, are used for simulating and synthesizing digital circuits. With the increase in design complexity, with multi million gates being fabricated, the increase in pressure to get design out faster with first time design success is also higher. Many new languages are used in industries which help in designing the higher abstraction models of hardware to be used for function verification and software validation. Systemverilog helped the designer with more constructs compared to Verilog, which addresses many hardware-oriented system design issues.

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 [36]. SystemC provides an event-driven simulation kernel in C++, together with signals, events, and synchronization primitives, deliberately mimicking the hardware description languages VHDL and Verilog. Matlab and several other tools are widely used for capturing system requirements and developing signal processing algorithms.
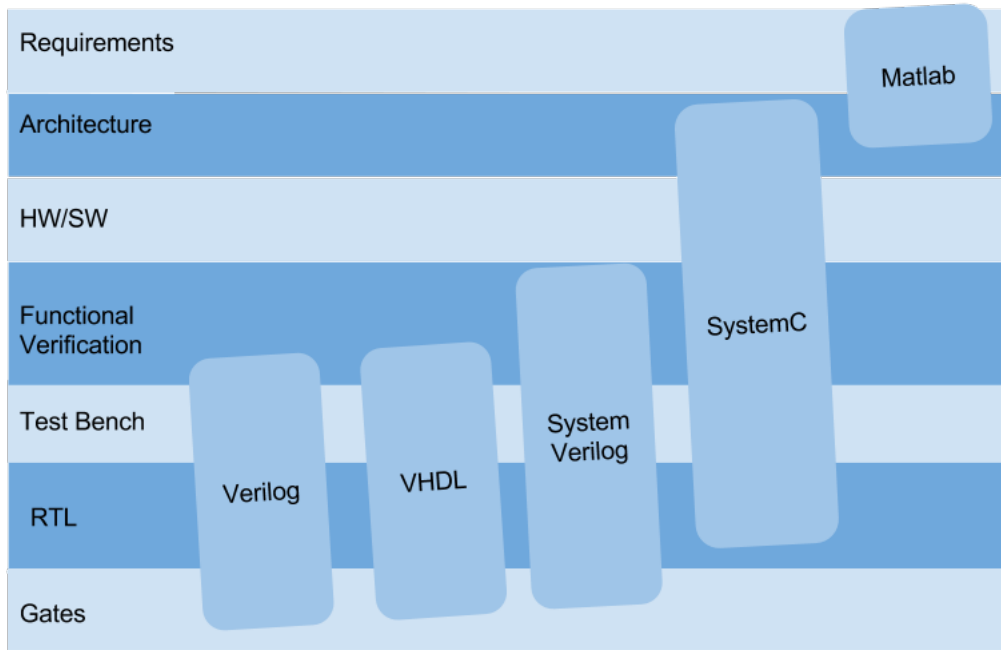
Figure 19: Hardware description languages and abstraction levels [4]

3.2.2 Transaction Level Modeling

Transaction level models use abstract channels to model communication between concurrent processes in the system using function calls [37]. In RT level each bit is instantiated as a port and hence in TLM level it can be abstracted to a byte, word or according to the architecture requirement. Also, the signal assignments of individual bits in RT level are abstracted with function calls which either send or receive the transaction. A transaction can be defined as an exchange of information between the processes in the simulation. It may contain complex data structures with control and data information, burst of data or a simple word of data packet. Time within TLM components are modeled as untimed, approximately or cycle accurate [38].

TLM complements RT-level modeling, in order to perform following activities [4] :

- Hardware micro-architecture exploration and starting point for more detailed hardware modeling.

- System level architectural exploration, such as selecting communication and processing components and Hardware/Software partitioning.

- Virtual platform for software development.

- Reference model for hardware functional verification.

Transaction level modeling is supported by languages like SystemC and SpecC. The communicating modules which are either initiators or targets are connected through channels. An initiator will start sending the transaction while the target will respond to the initiator request. Arbitration and routing algorithms are necessary for multiple initiators and targets to communicate in the system. The arbitration algorithm selects the initiator to be given access to the channel, and the routing algorithm assures that the correct target is addressed by transaction information. All these concepts are used to model the interconnects found in System on Chips (SoCs).

3.2.3   SystemC

SystemC is a system design language that has evolved in response to a pervasive need for a language that improves overall productivity for designers of electronic systems [4]. The open SystemC initiative (OSCI) formed in 1999, maintained a simulation library for SystemC. In December 5 2011, Accellera Systems initiative was formed by the merger of Accellera and the Open SystemC Initiative. SystemC is a C++ library that contains routines and macros to simulate concurrent processes using HDL like semantic. SystemC is now the IEEE Std. 1666 for system-level modeling, design and verification. It offers real productivity gains for the designer by providing hardware and software components design in parallel, but at a high level of abstraction. This higher level of abstraction gives the design team a fundamental understanding of the intricacies and interactions of the entire system, early in the design process and enables better system trade offs, earlier verification, and overall productivity gains through reuse of early system models as executable specifications [4].

Fig. 20 illustrates the layered architecture of SystemC. The shaded blocks are the SystemC core language standard, build upon standard C++. The layers above the standards are additional libraries available for system-level design and verification. A SystemC module encapsulates processes, which describe behavior, and communicates through ports and channels with other SystemC modules. Processes are used to describe concurrency and wait-statements are used to halt process execution for a specific time or until an event occurs. The OSCI SystemC TLM library contains ports, interfaces, channels, and also data structures are used to represent the request and response in an initiator-to-target communication scenario. The C++ notion of header (.h file) is used for the entity and the notion of implementation (.cpp file) for the architecture. Hardware by nature is concurrent and modeling it through

29
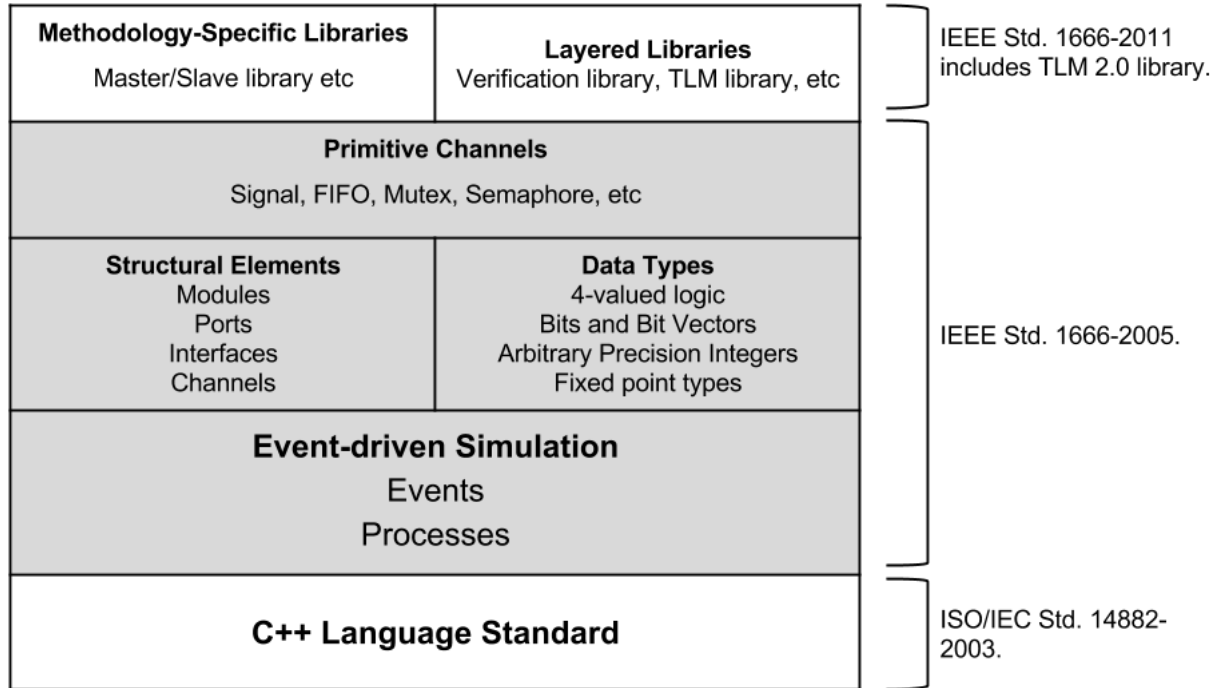
Figure 20: Layered SystemC Architecture

simulation on a host processor is an illusion. It is accomplished by simulating each concurrent unit (defined as SC_METHOD, SC_THREAD or SC_CTHREAD). Each unit is allowed to execute until the simulation of the other unit is required. The simulation of concurrency is the same for SystemC, Verilog, VHDL or any other hardware description languages. The simulator kernel will handle these tasks.

In TLM2.0, an *initiator* is a module that initiates the transaction and a *target* is a module that responds to transaction initiated by other module. The same module can act as an initiator and as a target for example in the case of modeling a bus or a router. TLM2.0 uses sockets to send/receive transaction.

An interconnect is a component which does not modify the transaction, but merely forwards the transaction. Fig. 21 illustrates the different ways of function calls for data exchange through transaction. The TLM2.0 library utils provide the simple initiator and the simple target sockets for user convenience. These are used in modeling the reconfigurable architecture discussed in a further section together with sockets which allow transactions to be sent to multiple destinations.
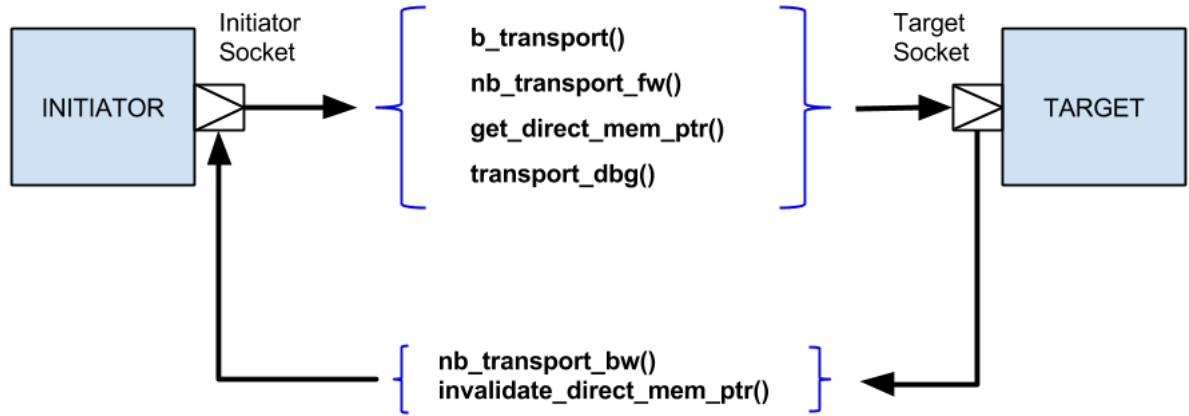
Figure 21: Initiator and Target sockets
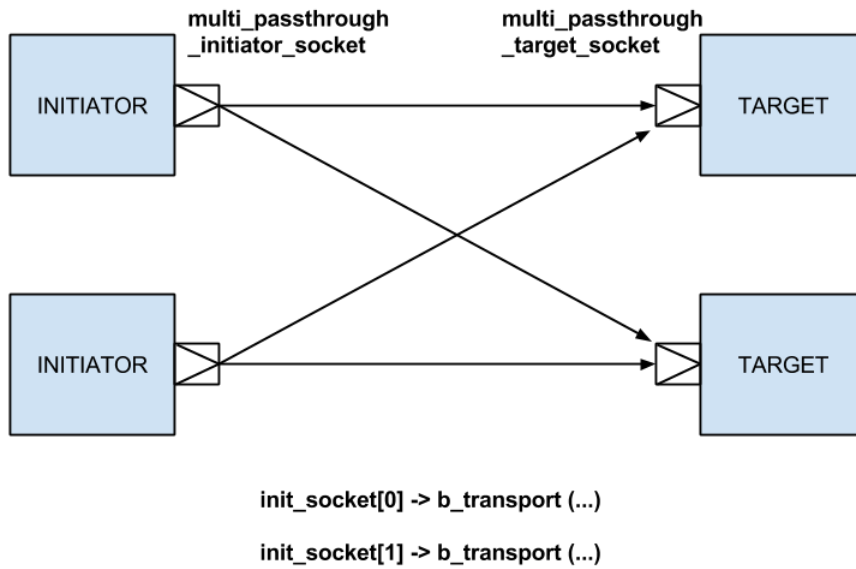
Figure 22: Many to many binding

The transaction being routed can be represented by a simple data structure provided by TLM2.0 utils referred to Generic payload. It has a standard set of bus attributes like command, address, data, byte enables, streaming width, and response status. The command can be either write or read, the data attribute points to a data buffer within the initiator and the data length attribute will give the length of the data array. The streaming width attribute specifies the width of a streaming burst where the address repeats itself. This Generic payload is modeled in each of

the reconfigurable operator blocks for initiating the transaction and the attributes are handled accordingly.

3.3 Verification methodology

In the early days, digital designs were verified by looking at waveforms and performing manual checks. This is a very tedious and time consuming task for the present day complex systems. So automating the verification process using better and efficient frameworks is always preferred. The SystemVerilog language came to aid many verification engineers. Its features like classes, covergroups and constraints have helped verifying complex digital systems. Later Verification Methodologies started to appear. UVM is one of the methodologies that was created for the need to automate verification. UVM is a verification methodology based on the best features of Open Verification Methodology (OVM) and Verification Methodology Manual (VMM) [39].
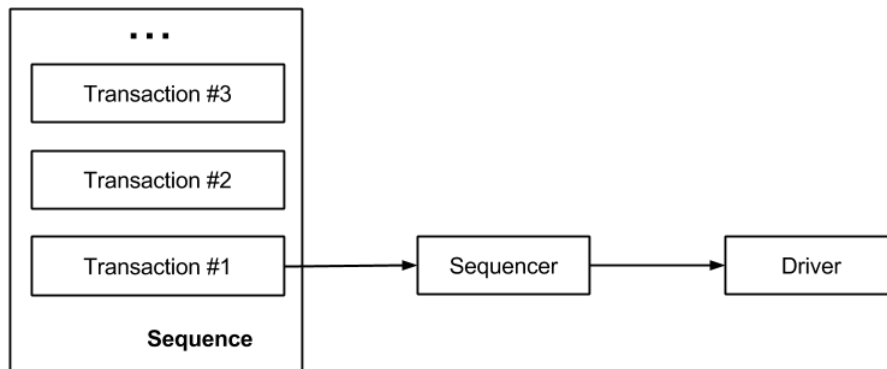
Figure 23: UVM Sequencer

The first step in verifying the designed system is by defining what kind of data is to be sent to the design under test. The detailed architecture of UVM is not discussed as it is out of the thesis scope. Only features relevant to the work are summarized. A transaction can be built from uvm_transaction or uvm_sequence_item classes similar to the transaction in SystemC. The "random" keyword is used to generate a transaction according to the constraints. The sequencer is responsible for sending the transaction to the driver component, which in turn connects to the Design Under Test (DUT).

3.4 High level description of the protocol

In general, how the reconfigurable architectures are programmed and the related work in this field is discussed in Section 2.2, wherein it explained different tools and frameworks developed for computationally intensive DSP architectures. As coarse grain architectures are targeted for specific domains to optimize the architecture design, the synthesis tools designed for protocol processing are discussed in the next section.

3.4.1 Related work

As discussed in the previous section, SystemC enables the specification of hierarchically structured communication protocols using the concepts of interfaces and channels. The SystemC [sv] [40] discusses the way to model the complete protocol environment by extending the SystemC language. It allows specifying the protocol which generates the controller for producer and consumer and sends the data through abstract channels. The tool is called COSYNE (Controller Synthesis Environment).

Clairvoyant [41] describes the protocols using a grammar based specification. It will generate an FSM described in VHDL which can be used by the logic synthesis tools. It was further commercialized and extended as the Synopsys Protocol compiler.

ProGrIL [42] is the Protocol Grammar Interface Language based on the Context free LL(1) grammar. The configuration file will contain the information about the widths and depths of various parts of the grammar processors, such as buses and memories.

In the case of ProGram [43] [44], input specification is similar to the production based specification and the output is the VHDL code of register transfer level. It is distinct in specification description when compared to the other approaches wherein the production rules are independent of port widths. The port widths are specified as design space constraints and the synthesis tool will generate different designs with varying performance for different values of constraints.

All the above discussed methodologies try to develop a generic tool to map the protocol described in high level description directly to register transfer level. This includes the synthesis tools exploring design space for the optimized hardware architecture with respect to area and performance. As discussed in Section. 2.2, for the computationally intensive DSP applications, there are tools which map the high

level description into specific architectures. Many case studies related to custom coarse grain architectures have also been discussed which developed their own set of kernels, software, configuration data for programming such architectures.

The architecture being developed at Ericsson is a custom architecture which requires a programming tool to generate the reconfiguration data and program it according to user definition. The architecture is implemented in SystemC language and it requires a new tool to program the hardware for different configurations. A new high level language is developed to describe the custom architecture.

3.5 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 sequences 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 needs some special constructs. These special constructs make-up the language that can be recognized by regular expressions. For example, the regular expression [0-9] recognizes a single letter in the range from 0 to 9.

3.5.1 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 24: 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.

3.5.2   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 the grammar according to the 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.

A high level language description requires the design of Lexer and parser from scratch. 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. Developing a "bottom-up" parsers is hard to understand and debug, compared to the "top-down" parsers.

There are many tools like Antlr4, APG, AXE, YACC etc which help to build a language application. They provide the user with Lexer and Parser implementations for the grammar defined by the user. This helps in building the language application in a short time. Antlr is one such tool which helps in designing a recursive descent top-down parser with a clear error recovery mechanism.

3.5.3   Antlr

Antlr4 accepts as input any context-free grammar that does not contain indirect or hidden left-recursion [45]. Antlr4 generates a recursive descent top down parser. Antlr4 parser uses a new parsing technology called Adaptive LL(*). ALL(*) prediction mechanism launches sub parsers at decision points and they operate in pseudo

parallel to explore all possibilities of input combinations. It performs the grammar analysis dynamically at runtime rather than statically. Antlr4 currently generates parsers in Java or C#, and the previous version supports even C and C++.
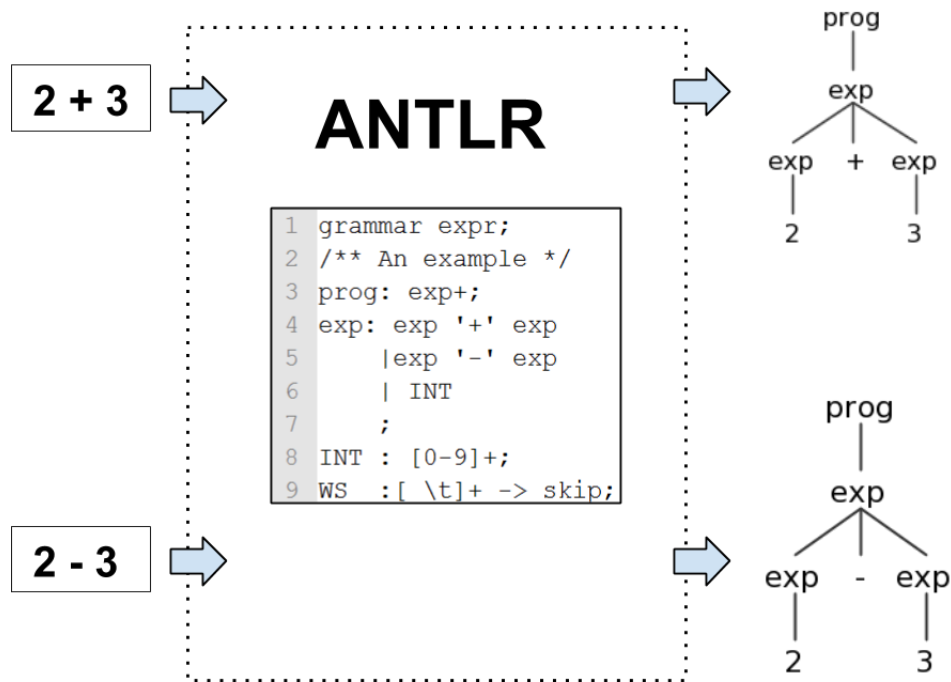
Figure 25: Antlr : Input, Grammar and Parsetree

Antlr4 grammar uses Yacc-like syntax with EBNF operators like the 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.

Fig 25 illustrates Antlr's Yacc-like metalanguage. Antlr4 automatically rewrites the rules to be non left recursive and unambiguous. The grammar analysis is performed dynamically and results are cached in lookahead Deterministic Finite Automaton (DFA) for efficiency [45].

Parse tree listeners and visitors

Antlr provides two tree walking mechanisms in its runtime library. In parse tree Listeners, Antlr generates a ParseTreeListener subclass specific to each grammar with entry and exit methods for each rule. This is suitable for applications wherein the

complete tree needs to be invoked from root until the last leaf node in-order. This is the default method in Antlr. The advantage of using ParsetreeListener mechanism is that it generates automatic Application Programming Interfaces (APIs) for the walker sequence and it is easier to build the language application.

In many circumstances, the designer needs to control the walk sequence. It is not possible to design a language application with visitor method which generate APIs from the left to the right leaf node. The parse tree visitor mechanism is used when the tree walking needs to be controlled. Option -visitor makes Antlr4 to generate a visitor interface from a grammar with a visit method per rule. Then in the application-specific code, a visitor implementation can be called.

4 | Freyja architecture

The three protocols discussed in the background are compared to highlight the common functions shared between them. The reconfigurable architecture called Freyja is explained in detail in the next section, and then 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 instantiation, transaction routing, constants in memory and error handling. Each of these is explained in detail in this chapter.

4.1   Comparison of the 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 2.4.

The common functions in each protocol are highlighted with the same color for the boundary line in Fig 26 & Fig 27. Each of the operators are implemented in SystemC and they receive and process the TLM2.0 transaction to perform the corresponding tasks.

The operators can be multiplexed and a generic operator unit can be designed for the blocks with same color boundary line. For example, the FCS computation is required in all three protocols. They require 16/32 bit polynomial to compute the checksum value. The memory unit stores the polynomials and control data required for the CRC operator to perform a CRC16 or CRC32 operation. Since the CRC check operation performed in the receiver stage, also requires the computation of FCS, a single operator unit can be designed and configured to perform different operation. The CRC operator will be designed in a way, such that it is used to transmit the received transaction by appending the FCS value, and if the transaction is from receiver stage, then the same hardware is used to compute the checksum value and then compared with the received checksum value. The operator switches its context based on the control information received form the memory block. A common hardware architecture can be designed with minimal reconfiguration to perform the protocol processing of different protocols mentioned above.
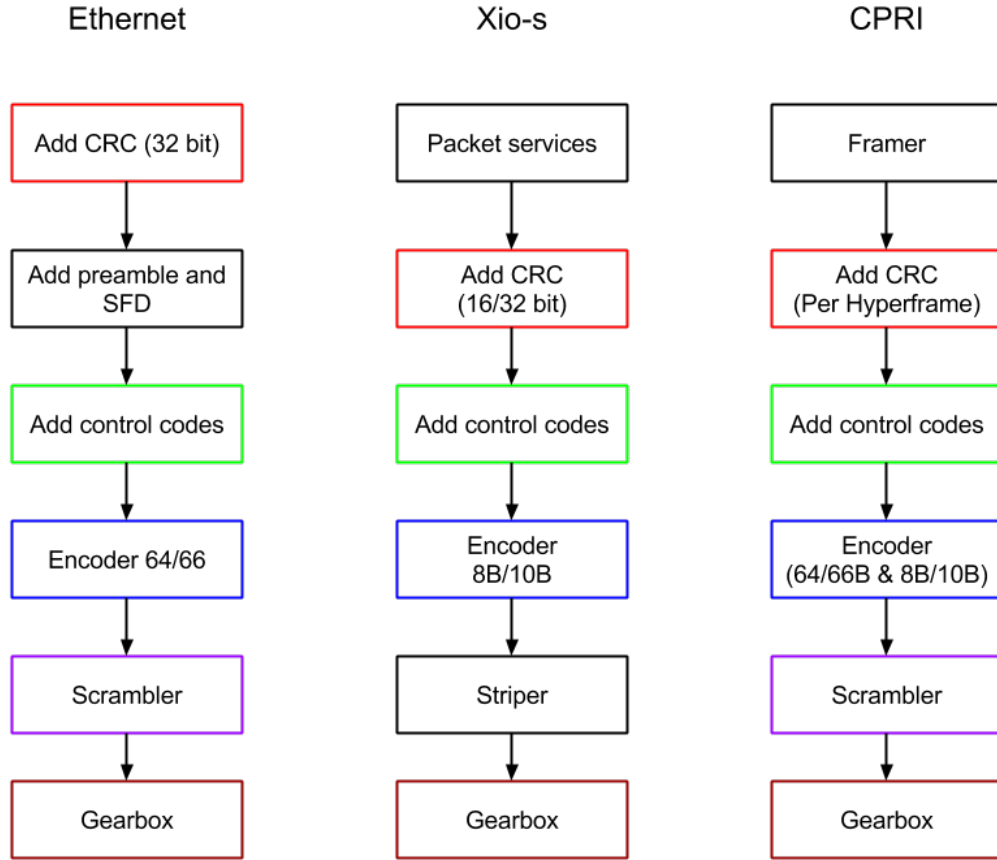
Figure 26: Transmitter of all three protocols

4.2   Architecture details

As depicted in Fig 28, 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) interfaces the Freyja with higher layers. It issues the tokens with data frames from different protocols. 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.
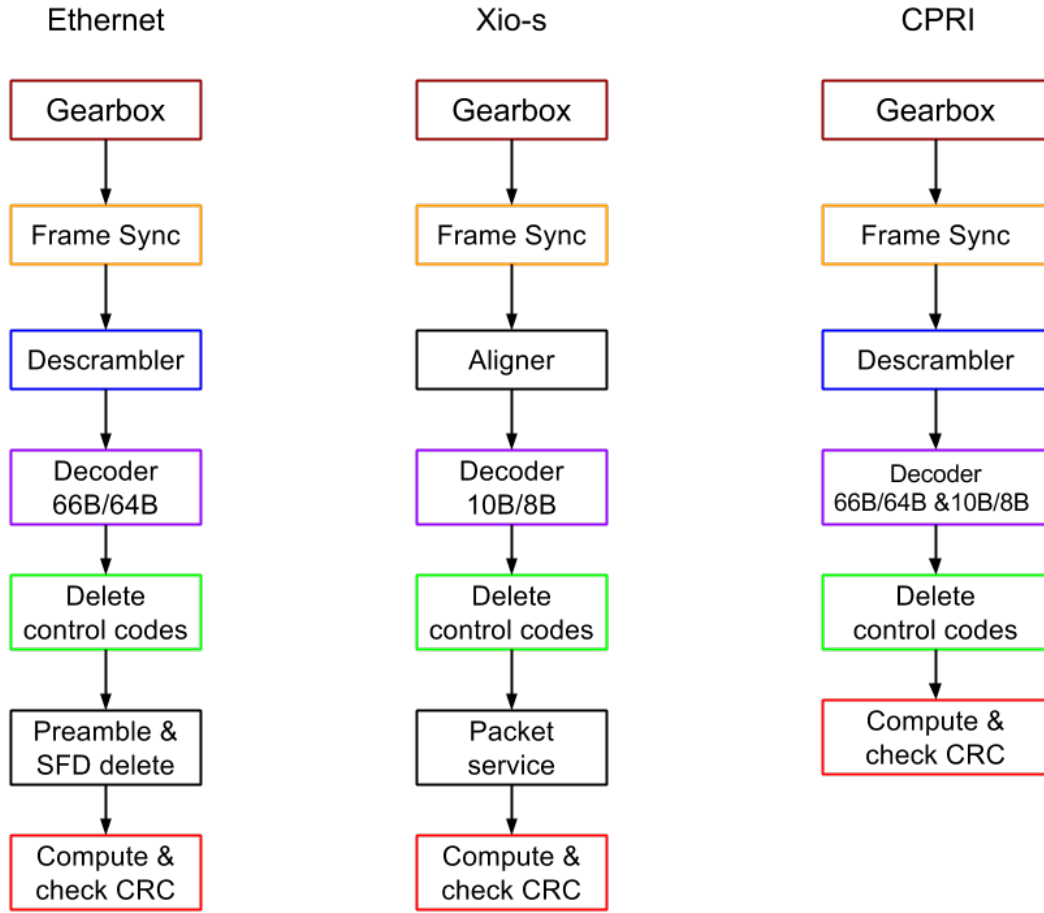
Figure 27: Receiver of all three protocols

4.2.1   Switch port

Freyja switch port is the smallest unit in the architecture. It receives the transaction from different functions and then forwards it to the internally connected next switch port based on the destination address of the transaction. It consists of a *simple initiator socket*, a *simple target socket*, a *multi pass through initiator socket* and a *multi pass through target socket*. It is illustrated in Fig 29.
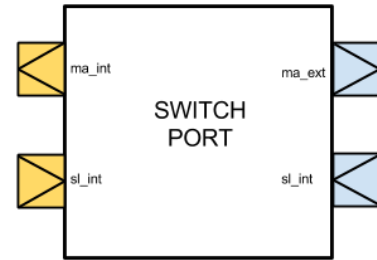
Figure 29: Switch port
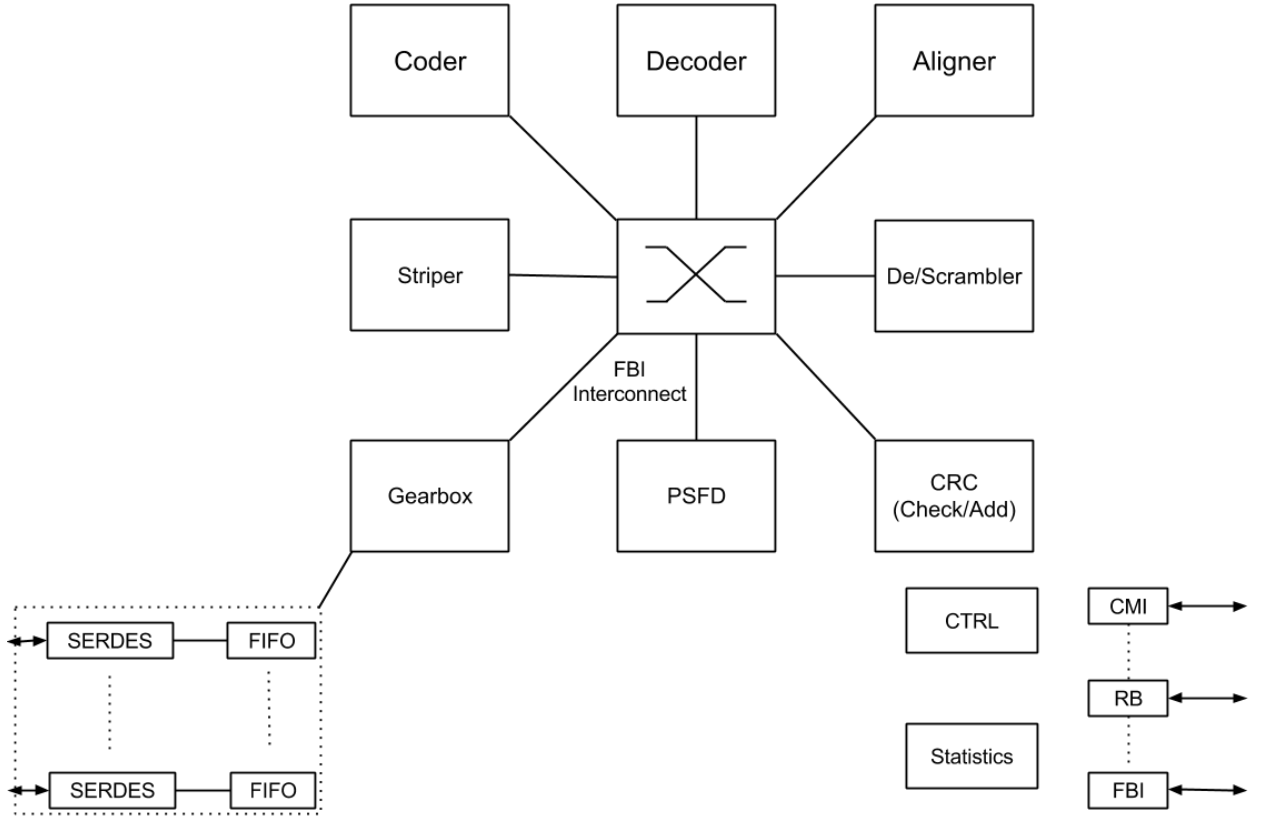
Figure 28: Switch based Freyja network topology

4.2.2   Operator units

Freyja Operator units are designed to contain a control, a process and a memory block. 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 process block results. The control block receives the result directly from process block transaction or from the memory contents.

As in Fig 30, the three components of each operator are encapsulated with one simple initiator socket and one target socket which initiates and receives the transaction from the switch respectively. The transaction received is routed to the control block. The control block will inform the memory block to encapsulate the required data for the particular operator and to send it to process block. Meanwhile the received transaction is sent to the process block from the control block to perform the required operation.
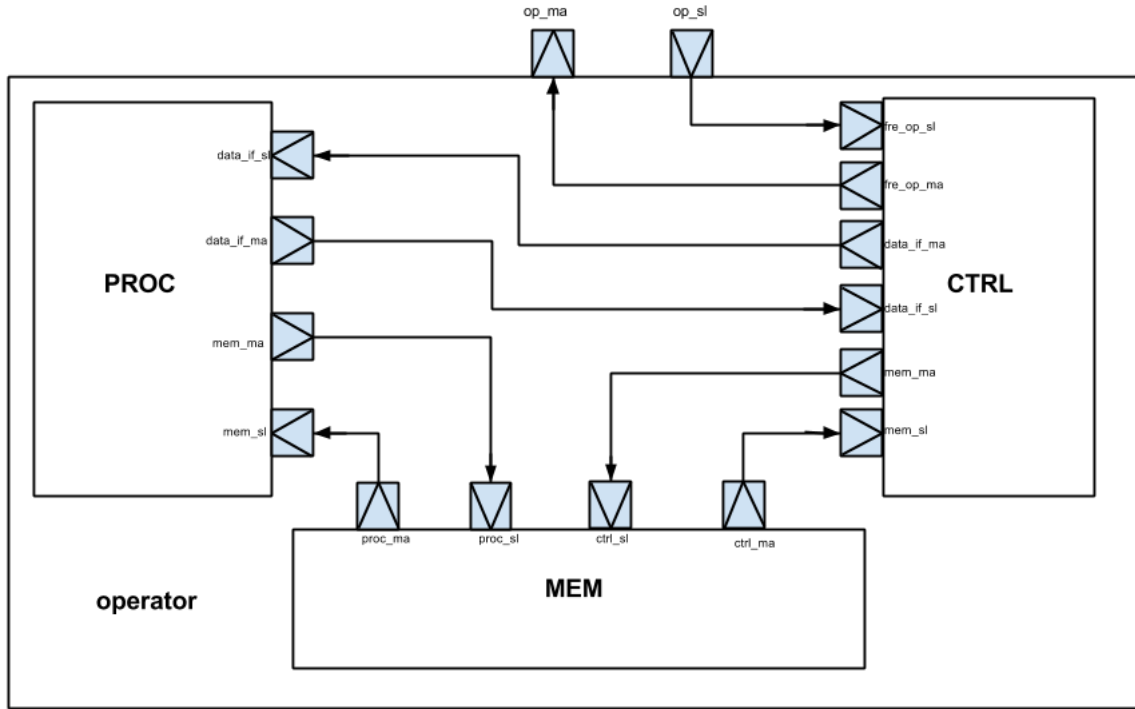
Figure 30: Freyja operator

4.2.3   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 can be seen in Fig 31, each switch port can send the transaction to any of the other switch port through the internal multipassthrough 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.

4.2.4   Overall architecture

An illustration of Freyja architecture with four operator units is shown in Fig 32.

The overall Frejya architecture consists of switch wrapper instantiating switch ports, and the operator blocks. A transaction originating from operator 1 as shown in Fig 32 with orange box can be routed to any of the other operators as destination based on the address of the payload header data. This will be assigned in the control block of the operator.
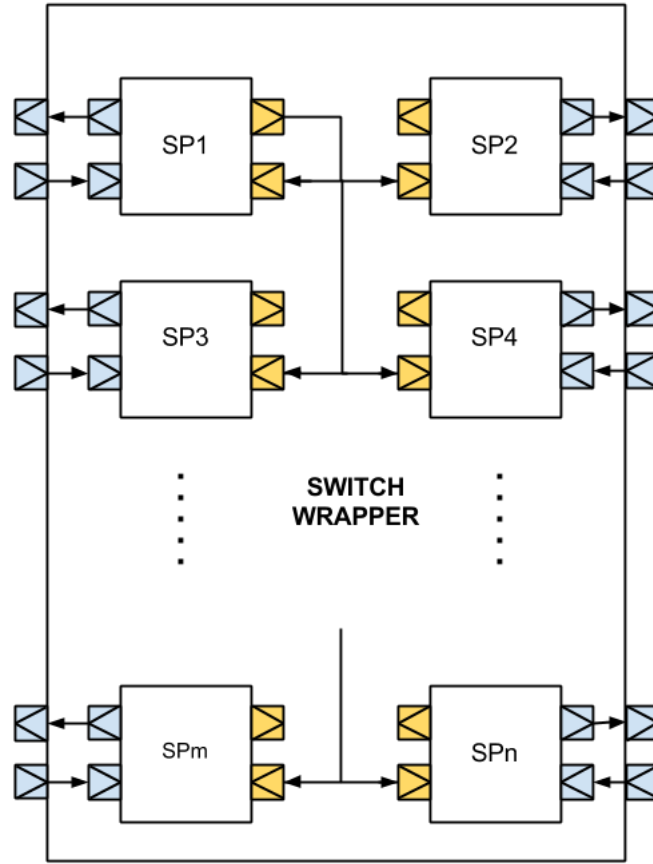
Figure 31: Freyja Switch Wrapper

4.3 Parser implementation

The Antlr tool is used as a parser generator to develop the language. The grammar is defined using EBNF.

A brief description of the generated file is stated below:

Freyja_archInitparser.java: This file contains the parser class definition specific to the grammar in freyja_arch.g4 (Appendix B), that recognizes the syntax of Freyja protocol processor language.

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
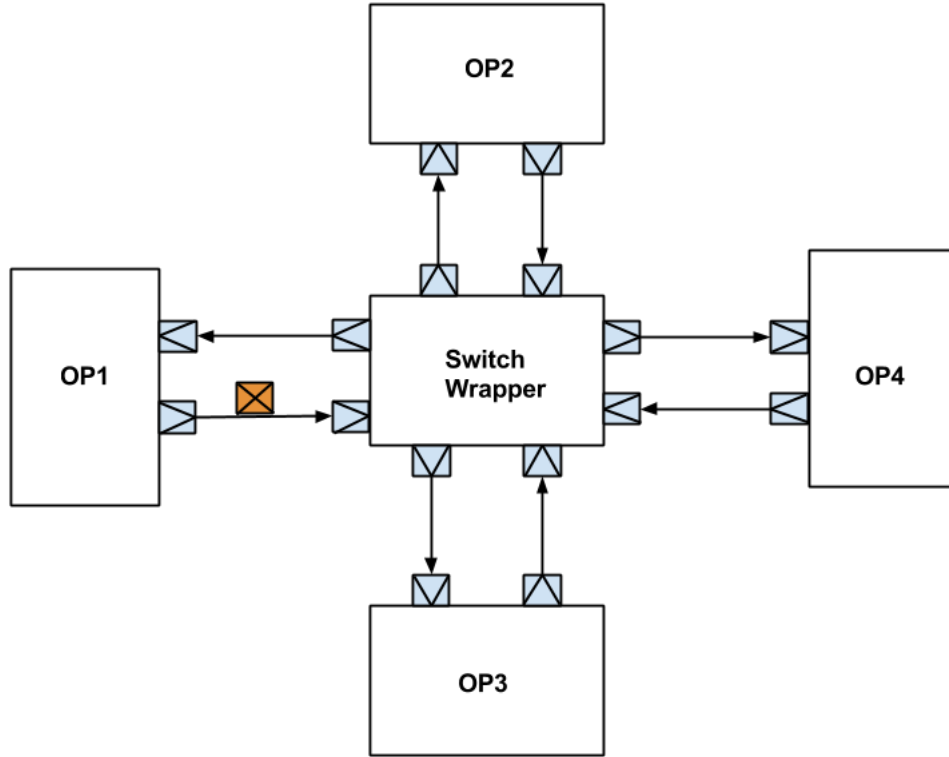
Figure 32: Freyja Architecture for 4 operators
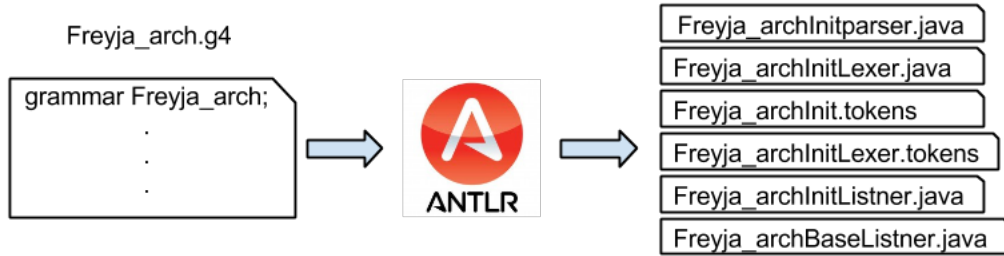
Figure 33: Antlr generated Files

in the grammar and store these values in this file.

Freyja_archInitListner.java, Freyja_archBaseListner.java: The Antlr
parser builds a tree walker that can trigger the callback events to a listener object.
Freyja_archInitListner is the interface that describes the callbacks and Freyja_archBaseListner
is a set of empty default implementations.

The back-end functions are implemented to produce the Freyja architecture based on the described language.

4.4   Operator Instantiation

As illustrated in Fig 34, each of the operators in the Freyja architecture consists of a control block to take care 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.
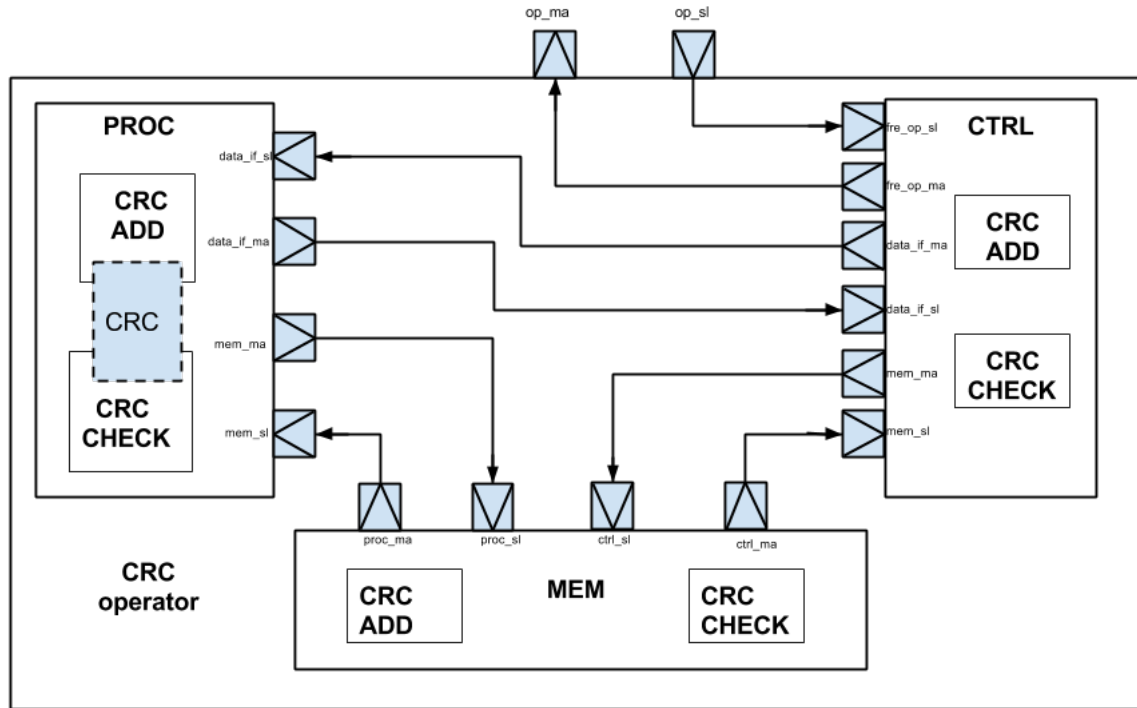
Figure 34: CRC operator in Freyja Architecture

The functions performed during the transmitting stage might need to be performed in the receiver stage of a protocol processor. 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 the respective 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 Fig 34, the CRC computation function is used by both 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. 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 }
```

Figure 35: Freyja Operator Instantiation

- **Operator:** This is the keyword used to define the operator. "CHECKSUM"
  should be the unique operator name in Freyja architecture.

- **sc_name:** This keyword is used to define the operator file name. It create
  the systemC files fre_op_ctrl_crc, fre_op_proc_crc and fre_op_mem_crc
  which are the control, process and memory blocks, respectively.

- **op_type:** This keyword is used to define different operators like "crc_add"
  and "crc_check" inside one *Operator* block. Each operator definition also takes
  care of inserting a switch port and modifying the interconnections of switch
  wrapper function.

- **op_mem:** This keyword is used to define the constants to be stored inside
  the local memory of Operator "CHECKSUM". The constant 0x04C11DB7
  stored in 1 to 4 bytes is the CRC32 polynomial, byte 5, 6 and 7 are reserved
  for indicating the crc control information like crc_type, crc_len_conf and
  length_conf. Similarly the polynomial for CRC16 and control information is
  stored from byte 9 to 18.

- **errorid:** This keyword is used to indicate the error id for each operator. A
  unique number is assigned for each of the op_type string of all operators.

In "CHECKSUM", crc_add and crc_check is assigned with error id 2 and 3 respectively.

4.5   Context switching

The context switching in the process block is performed based on the destination operator for the data packet. The control block will 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.

As depicted in Fig 34, the control block determines the protocol and the operator type for the CRC operator. It then informs the memory blocks with the transaction payload to send the required information to process block. Simultaneously, the control block will send the received transaction to the process block. The memory block will send 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 them to the control block. The control block will forward the transaction to the destination operator.

4.6   Memory

The memory block is designed to store the constants required for the operators. The constants to be stored in the memory are indicated in the *op_mem* field as explained in section 4.4. Each operator is allowed to 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 the operator checksum definition, the memory contains 19 bytes which are used by the CRC16 and CRC32 operators.

```
1  memory {
2  crc_add     :00 to 08;
3  crc_check   :00 to 07, 09;
4  scram       :00 to 07:
5  descram     :00 to 07;
6  }
```

Figure 36: Freyja Memory content definition

Memory transaction for each protocol are described in the language shown in Fig 36. The first 8 bytes of memory contents are sent as a transaction for *crc_add* operation in Ethernet protocol. The memory bytes can be written as individual bytes (for example, 01, 02, ...), or as contiguous location (for example 01 to 04, 08 to 10 ...), or a combination of both. Each protocol with the memory code as above can send transactions for different operators. In the above code, the *crc add*, *crc check*, *scram* and *descram* send the transaction to their respective blocks.

4.7   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.

4.8   Error handling

Any erroneous packet can be sent to the error handler from each of the operator. The control block will verify the result and if the error flag is raised, the transaction is encoded with the corresponding error id of the operator and forwarded to the error handler. The error id for each operator is indicated with the keyword "errorid" as explained in section 4.4.

5 | Illustrative example

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 5.2 explains the Xio-s protocol integration.

5.1 Ethernet

Appendix A shows the complete description of Ethernet protocol. The description starts with the keyword "FunctionUnits" followed by the architecture name, followed by the description of all the operators in the Freyja protocol processor shown in Fig. 28. Each of the "Operator" corresponds to the protocol functions discussed in the section 2.4.1. It is followed by the "protocol" keyword and information specific to Ethernet protocol. In the "fields" section, the raw ethernet frame is represented. This information helps in decoding the corresponding data for debugging purpose only. The "memory" section described the constants in the local memory blocks. For Ethernet protocol, the *crc_add*, *crc_check*, *scram* and *descram* process block will receive the constants from the memory block located in each of their Operator block.

The operators involved in Ethernet transmit and receive sequence were shown in Fig. 9 and Fig. 10, but are reproduced below in Fig. 38. The *Datain* in the transmit is passed from the UVM test bench. A UVM sequence is generated with the first byte indicating the protocol and the second byte containing the address of the "PROCESSOR" operator. Once the transaction reaches the *switch wrapper*, it is decoded and forwarded to the *proc_tx* operator. The *proc_tx* operator is defined to extract the data from transaction payload.

Implemented in our high level language, the interconnection between different Ethernet operators is explained in this section. A part of the Ethernet transmitter description is shown in Fig. 37 and the receiver description is shown in in Fig. 39. The corresponding interconnection in Freyja (switch wrapper interconnection is not shown) for the Ethernet protocol is shown in Fig. 38.

```
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 }
```

Figure 37: Ethernet transmitter description
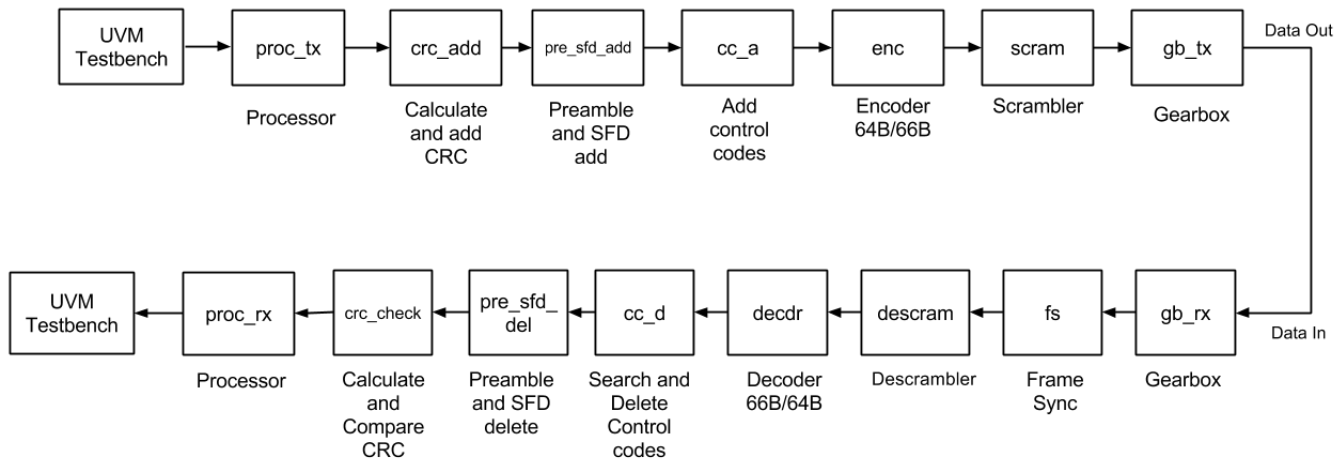
Figure 38: Ethernet protocol interconnection in Freyja

```
1 RX_PATH {
2 CONNECT : gb_rx to  fs;
3 CONNECT : fs  to   descram;
4 CONNECT : descram to  decdr;
5 CONNECT : decdr to   cc_d;
6 CONNECT : cc_d to  pre_sfd_del;
7 CONNECT : pre_sfd_del to  crc_check;
8 CONNECT : crc_check to  proc_rx;
9 }
```

Figure 39: Ethernet receive description

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. It consists of 3 segments : *Fields, Tx_path, Rx_Path.*

- The *Fields* section is defined to indicate the details of the transaction contents in each protocol.

- *Tx_Path* is used to define the transmitter sequence.

- *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 the Freyja architecture, the interconnections have no difference with respect to the *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. The Antlr output for Ethernet protocol with *crc_add* and *crc_check* operator in "CHECKSUM" operator is shown in Fig 40.

```
1  //part of fre_op_ctrl_crc.cpp block

   if(type == ethernet){
       if(op_type == crc_add){
5          data[0] = ethernet;
           sz = sz + 2;
           data[1] = pre_sfd_add;
           gp.set_address(crc_add_2_pre_sfd_add);
       }
10     else  if(op_type == crc_check){
           data[0] = ethernet;
           sz = sz + 2;
           data[1] = proc;
           gp.set_address(crc_check_2_proc);
15     }
   }
```

Figure 40: Freyja CRC operator Control Block code segment.

In the above code, the CRC operator's control block initially checks for the protocol and then for the type of the operator. As illustrated in Fig. 34, the CRC operator multiplexes two operations. The *crc_add* and *crc_check* operators share the control block, the common memory and, hardware for CRC computation. The *crc_add* need extra logic to append FCS value to the data payload and *crc_check* requires extra logic to compare the computed and received FCS value.

Based on the input description, Antlr will generate the reconfiguration data. The destination address of the transaction is configured using this data. This information is compiled in the form of constants in the SystemC environment. The language supports the possibility to send the transaction to different operators from the same source.

```
1  // output file : fre_constants.h
   //constants for the protocol : ethernet
   const unsigned char ethernet = 0;
   const   int  proc_2_crc_add=0;
5  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;
10 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;
15 const   int  crc_check_2_proc=0;
```

Figure 41: Freyja Reconfiguration constants.

The constants are formed using a simple syntax as below

⟨ source operator ⟩ _2_ ⟨ Destination operator ⟩

The constant value assigned is calculated by the parser based on the switch port interconnections. The CRC operator also contains 19 bytes of memory elements. These 19 bytes are used by both CRC32 and CRC16 Operators. The Ethernet protocol uses only CRC32 and hence only *crc_add* and *crc_check* uses the first 9 bytes of the memory elements. 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.

5.2   Xio-s

The Xio-s protocol is briefly explained in section 2.4.2. The CRC block of Xio-s protocol 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, the

Operators are prioritized in the order they have been defined in the input file. Defining Xio-s after the Ethernet protocol description will change the control block code segment as shown in Fig. 42 .

```
1  //output file
   //part of fre_op_ctrl_crc.cpp block
   if(type == ethernet){
       if(op_type == crc_add){
5          //modifies generic payload header
           //for ethernet CRC add operation
           ...
       }
       else if(op_type == crc_check){
10         //modifies generic payload header
           //for Ethernet CRC check operation
           ...
       }
   }
15 else if(type == xios){
       if(op_type == crc_add){
           //modifies generic payload header
           //for Xio−s CRC add operation
           ...
20     }
       else if(op_type == crc_check){
           //modifies generic payload header
           //for Xio−s CRC check operation
           ...
25     }
   }
```

Figure 42: Freyja CRC control block with two protocols

As illustrated in Fig. 13 and Fig. 14, the Xio-s protocol requires 2 interfaces to transmit the transaction from one operator to multiple operators. The transactions are sent from the *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 packet service (**ps**), CRC16 addition (**crc_add_16**) and control codes addition (**cc_a_xio**) are the operators described in the input file.

Similar logic applies to the receiver section which can be parsed to receive the transaction from different source operators. Assuming frame sync1 (**fs1**), frame

```
1 Tx_PATH {
2 CONNECT : ps to  crc_add_16;
3 CONNECT : ps to  cc_a_xio;
4 }
```

Figure 43: Freyja One source to multiple destination interconnection

sync (**fs2**) and aligner (**aligner**) are the Xio-s protocol operators, the below code indicates the way to receive transaction from multiple sources.

```
1 Rx_PATH {
2 CONNECT : fs1 to  aligner;
3 CONNECT : fs2 to  aligner;
4 }
```

Figure 44: Freyja multiple source to one destination interconnection

The Freyja Operator unit as explained in section 4.2.2 consists of only one interface. But for receiving transaction simultaneously from 2 different sources, it requires the operator unit to have 2 interfaces. The operator units 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.

6 | Test system

This chapter describes the complete test system, the testing strategy, the input protocol file and the output obtained.

6.1 Complete test system

Verification is an important step to make sure the correct working of newly developed architecture. The UVM test bench is connected to the Design Under Test (DUT) that is the Freyja architecture. The UVM ports are connected to the *switchport 0* in the Freyja architecture as illustrated in Fig 45.
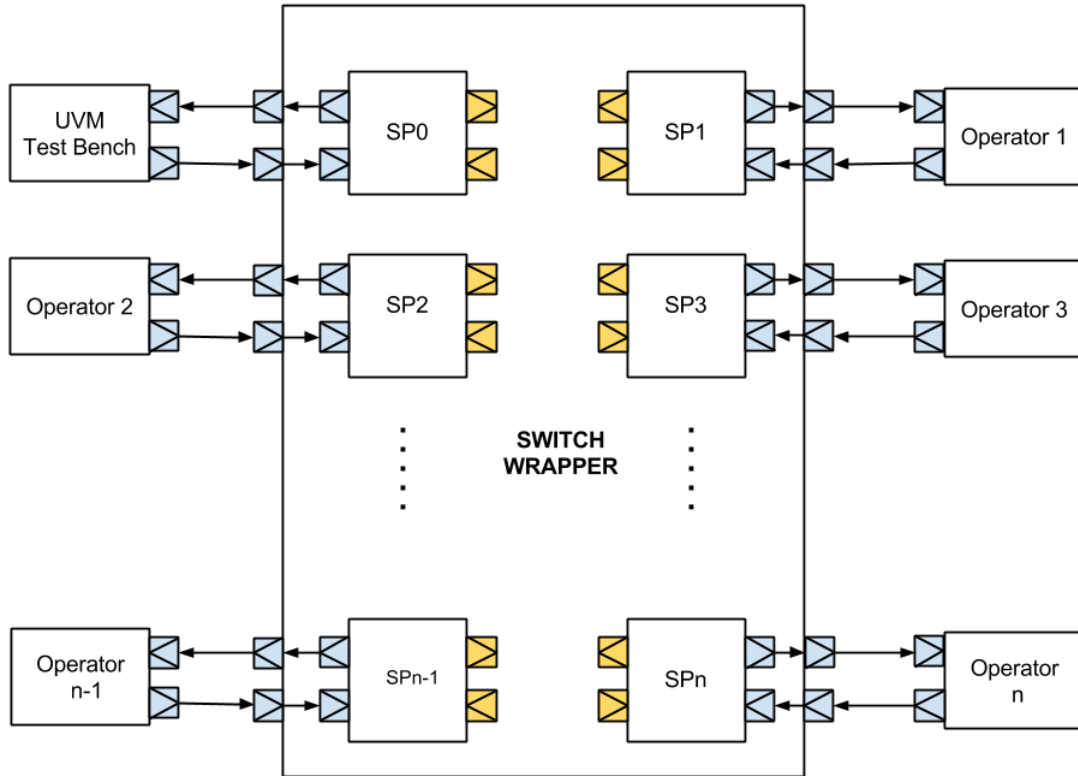

Figure 45: Complete Test System(*Multiport interconnections not shown)

The total number of *switch ports* instantiation inside *switch wrapper* depends on the total *Operator* units defined in the input description. The UVM environment will drive the input signals to the Freyja architecture. The Freyja interconnect

consists of a 4 bytes header field and the data payload as in Fig 46. In the SystemC implementation, the first byte is considered to hold the unique protocol id and the second byte to have the unique operator id, while the third byte and fourth byte are for Flow control and Context/error handling. The data bit width is suggested to be 80bits. This complete data is randomly generated with constraints from the UVM sequencer. The UVM driver will drive the input signals to the Freyja architecture. The data is sent as a transaction payload.
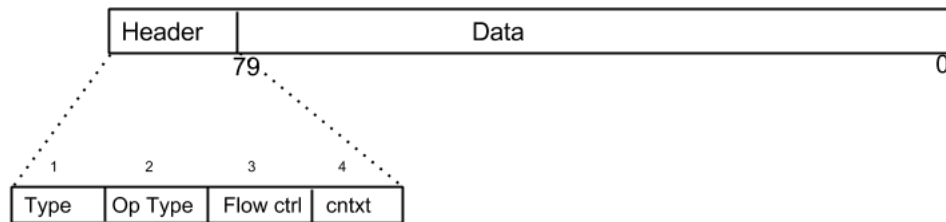
Figure 46: FBI Interconnect

6.2   Generic payload

Each generic payload transaction has a standard set of bus attributes:

- **Address:** This field informs the control block of the operator unit about the source of transaction. The control block will change the address field to the destination operator address, based on the configuration data generated by Antlr.

- **Data:** The target shall copy the data to or from the data array, with respect to the semantics of other attributes. This field contains the data according to the protocol *Field* section. For example, In Ethernet MAC address, payload length and data forms the Data packet.

- **Byte enables:** This is used to create burst data transfers.

- **Streaming width:** This field indicates the number of data bytes transferred on each beat in burst transfer.

- **Response status:** This field indicates the status of bus operation. The target may set the response attribute as received without errors or to one of the error type.
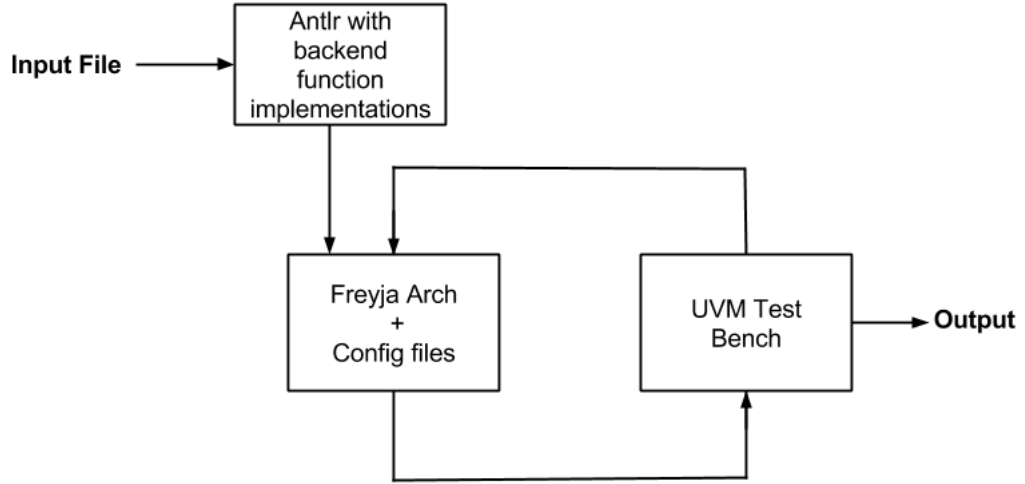
Figure 47: Input and Output of the system

6.3   Input and Output

The input file is the description of the protocol using the developed language. The input file is parsed by the Antlr tool. The back-end functions are triggered during parsing to output the Freyja architecture and the reconfiguration files. A constrained random transaction is initiated from the UVM to the Freyja architecture and the final transaction is received back by the UVM.

6.4   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 advanced, as the main interest is to model the functionality of the target and not modeling any timing detail.

7 | Conclusion and future work

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

7.1 Reconfigurable architecture

Ericsson AB investigated Switch and Mesh based network topologies 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 4.2.

The design of a new architecture requires 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. The research project called TACO (Tools for Application-specific Hardware/Software Codesign) at university of Turku explains the functions, features and capabilities required by a tool to solve these challenges [46]. One such requirement is importing architecture details in the tool from SystemC top level files. The same approach is applied 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 operator 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. 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 language. 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. The language can be extended to have 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 be accomplish, the reconfigurable architecture should be known. 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.

7.2   Protocol sharing

As illustrated in section 4.1, 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 this 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. The TACO project [46] 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 the Freyja architecture for frequently used operators.

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 [47]. Using a gram-

mar 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 [48] attempts to make grammars more accessible to the programmer by generating recursive descent parsers that are very similar to what the programmer would build by hand.

Antlr4 supports rapid grammar developemnt by using Antlr's built in interpreter, thus, providing immediate feedback during development [49]. The parse tree associated with the matching input (as in Fig. 24) 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 the 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 preferred language. The most encountered problems are resolved with the community support and the book written by the author Terrence Parr [48].

7.4   Limitations

The thesis started with the focus of developing the complete framework and to show the concept of using a language application in hardware/software co design. The reconfigurable architecture modeled in systemC abstracted more hardware details. Though a language is developed to include all future requirements, the implementations in the backend have to be extended for the new features of the hardware. Even though three protocols and their functions are discussed, integration of only Ethernet and part of Xio-s protocol is considered. As stated in section 7.1, the high level description is targeted with respect to the current Freyja architecture. The Antlr4 tool supports Java language and other languages are yet to be released.

7.5   Future work

The Freyja architecture is still under implementation and in the future, the complete details of the architecture can be abstracted into high level description by extending the language. Features like multiple core instantiations and transactions from different sources to same destination (required for Xio-s protocol) are possible to represent but still require the backend implementations to modify according to the interconnection defined in future. Once the reconfigurable file format required by RTL hardware implementations is defined, the same framework can output such

files to speedup the design process. As explained in section 1.2, the granularity of Freyja architecture is at algorithmic level. Exploiting the reconfigurability for the Freyja architecture can be considered further with respect to common hardware blocks at logic levels and then the language can be extended similar to the TACO protocol processor [46]. The header, as depicted in Fig. 46, 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 that is the core functions of each operator needs to be modified to integrate these protocols into the system. Antlr4 is made to output the framework in Java language and the tool (in future!) will support other languages like C and C++. Based on the project requirements the language can be selected. Java has good commands for file IO operations and object handling. The major back-end implementation deals with storing the parsed data in the form of hash-map and linked-list. These facts can be considered when building similar concepts 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 the SystemC model for simulation, a Matlab model for estimation of resources and a 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] M. Keating, "The future of design," in *The Simple Art of SoC Design*, pp. 171–180, Springer New York, 2011.

[2] "Xilinx, all programmable." `http://www.xilinx.com/fpga/`, Aug. 2015. [Online; accessed 1-Sept-2015].

[3] P. Schaumont, K. Sakiyama, A. Hodjat, and I. Verbauwhede, "Embedded software integration for coarse-grain reconfigurable systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pp. 137–, April 2004.

[4] J. D. David C. Black, *SystemC: From the Ground Up.* Springer US, 2004.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] D. Gajski and R. Kuhn, "Guest editors' introduction: New vlsi tools," *Computer*, vol. 16, pp. 11–14, Dec 1983.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] F. Mayer-Lindenberg, "High-level fpga programming through mapping process networks to fpga resources," in *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on*, pp. 302–307, Dec 2009.

[18] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, pp. 41–49, Apr 2000.

[19] 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.

[20] G. Theodoridis, D. Soudris, and S. Vassiliadis, "A survey of coarse-grain reconfigurable architectures and cad tools," in *Fine- and Coarse-Grain Reconfigurable Computing* (S. Vassiliadis and D. Soudris, eds.), pp. 89–149, Springer Netherlands, 2007.

[21] 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.

[22] 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.

[23] 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.

[24] `http://hls-labsticc.univ-ubs.fr/`, 2015. [Online; accessed 20-Sept-2015].

[25] D. Shin, A. Gerstlauer, R. Domer, and D. Gajski, "An interactive design environment for c-based high-level synthesis of rtl processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, pp. 466–475, April 2008.

[26] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: a retargetable compiler for coarse-grained reconfigurable architectures," in *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pp. 166–173, Dec 2002.

[27] M. Alle, K. Varadarajan, A. Fell, S. K. Nandy, and R. Narayan, "Compiling techniques for coarse grained runtime reconfigurable architectures," in *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ARC '09, (Berlin, Heidelberg), pp. 204–215, Springer-Verlag, 2009.

[28] O. Malik, A. Hemani, and M. Shami, "High level synthesis framework for a coarse grain reconfigurable architecture," in *NORCHIP, 2010*, pp. 1–6, Nov 2010.

[29] D. Cronquist, P. Franklin, S. Berg, and C. Ebeling, "Specifying and compiling applications for rapid," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pp. 116–125, Apr 1998.

[30] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "Piperench: a reconfigurable architecture and compiler," *Computer*, vol. 33, pp. 70–77, Apr 2000.

[31] A. Abnous, K. Senoy, M. Wan, and J. Rabaey, "Evaluation of a low-power reconfigurable dsp architecture," in *proceedings 5 th Reconfigurable Architectures workshop (RAW98), March 30*, pp. 55–60, Springer, 1998.

[32] P. Heysters and G. Smit, "Mapping of dsp algorithms on the montium architecture," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pp. 6 pp.–, April 2003.

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

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

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

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

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

[38] C. Shin, P. Grun, N. Romdhane, C. Lennard, G. Madl, S. Pasricha, N. Dutt, and M. Noll, "Enabling heterogeneous cycle-based and event-driven simulation in a design flow integrated using the SPIRIT consortium specifications," vol. 11, September 2007.

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

[40] R. Siegmund and D. Muller, "A novel synthesis technique for communication controller hardware from declarative data communication protocol specifications," in *Design Automation Conference, 2002. Proceedings. 39th*, pp. 602–607, 2002.

[41] A. Seawright and F. Brewer, "Clairvoyant: a synthesis system for production-based specification," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 172–185, June 1994.

[42] R. Bloks, "A grammar based approach towards the automatic implementation of data communication protocols in hardware," in *Ph. D thesis, Eindhoven University of Technology, Sept 1993*, 1993.

[43] J. Oberg, A. Kumar, and A. Hemani, "Grammar-based hardware synthesis from port-size independent specifications," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, pp. 184–194, April 2000.

[44] J. Oberg, A. Kumar, and A. Hemani, "Grammar-based hardware synthesis of data communication protocols," in *System Synthesis, 1996. Proceedings., 9th International Symposium on*, pp. 14–19, Nov 1996.

[45] 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].

[46] 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.*

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

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

[49] 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{
44 sc_name : gb_tx;
45 op_type : gb_tx;
46 errorid : 09;
47 }

   Operator GEARBOX_RX{
50 sc_name : gb_rx;
51 op_type : gb_rx;
52 errorid : 13;
53 }

55 Operator FRAMESYNC{
56 sc_name : fs ;
57 op_type : fs ;
58 errorid : 10;
59 }
60
   Operator DECODER{
62 sc_name : dec_66_64;
63 op_type : decdr;
64 errorid : 11;
65 }

   Operator CORRECTING_CODES_DEL{
68 sc_name : cc_d;
69 op_type : cc_d;
70 errorid : 12;
71 }


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


   memory {
84 crc_add    : 00 to  08;
85 crc_check  : 00 to  07, 09;
86 scram      : 00 to  07:
87 descram    : 00 to  07;
88 }


90
```

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


    RX_PATH {
104 CONNECT : gb_rx to  fs;
105 CONNECT : fs  to   descram;
106 CONNECT : descram to  decdr;
107 CONNECT : decdr to  cc_d;
108 CONNECT : cc_d to  pre_sfd_del;
109 CONNECT : pre_sfd_del to  crc_check;
110 CONNECT : crc_check to  proc_rx;
111 }
```

B | Grammar file : freyja_arch.g4

```
1  grammar freyja_arch;
   freyja : fus protocols;
   fus:'FunctionUnits ' ID':' operators;
   operators:operator*;
5  operator:'operator ' ID'{' operator_fields '}';
   operator_fields: filename op_types opmem* memgrp* errorid;
   filename:'sc_name : ' ID';';
   op_types:'op_type : ' ID ids;

10 opmem:'op_mem  : ' HEX (',' HEX)* ';';
   memgrp:'mem_grp : ' N ns;
   errorid :'errorid  : ' N ns;
   ns: ';' | ','N ns;
   ids : ';' | ','ID ids;
15


   protocols: protocol*;
   protocol: 'protocol ' P_NAME p_details;
   p_details: packet memory* tx_packet rx_packet;
20 packet: ' fields : ' (p_field)*;
   p_field : hex_def | o_def;
   o_def: name position;
   position : '['N']';
   hex_def: name':' hex_val;
25 hex_val:HEX;
   name: ID;

   memory:'memory' '{' memcontents '}';
   memcontents: (opername ':' mem_bytes ';')*;
30 mem_bytes: range (',' range)*;
   range: (n1 'to' n2) | n1;
   opername: ID;
   n1: N;
   n2: N;
35


   tx_packet: 'Tx_PATH' '{' path* '}';
   rx_packet: 'Rx_PATH' '{' path* '}';
   path: 'CONNECT' ':' id1 'to' id2 ';';
40 id1 : ID;
   id2 : ID;


   P_NAME: 'ETHERNET'| 'Xio' | 'CPRI' | 'SRIO' ;
45 HEX:'0x' HEXNUM+;
```

70

```
HEXNUM: ('0'..'9'|'a'..'f'|'A'..'F');
N: [0−9][0−9];
ID: [a−zA−Z_0−9]+;
WS : (' '|[\t\r\n]+) −> skip;
```

C | 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 converting 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 the 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.

Look-ahead Parser

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

Left recursion

A grammar is left recursive if there a non terminal symbol which is derived in such a way that it exists as the left most symbol.

Context free grammar

A formal grammar is context free when its production rules can be applied regardless of the context of a non terminal.