Reconfigurable hardware programming in a protocol processor unit

- SUNIL KALLUR RAMEGOWDA

ROYAL INSTITUTE
OF TECHNOLOGY

Master's Degree Project
Dept of ICT/Embedded Systems
KTH, Sweden 2015

IR-EE-Dummy 2000:099

Acknowledgement

Reconfigurable hardware programming in a protocol processor unit

Sunil Kallur Ramegowda

July 10, 2015

Abstract

Reconfigurable hardware architectures are the topic of research for many years.Programming such architectures requires the design of custom compilers to generate the required files for the architecture.

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

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

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

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

Contents

Abbreviations

ASIC Application Specific Integrated Circuit.

ASIP Application Specific Instruction-set Processors.

CPRI Common Public Radio Interface.

EBNF Extended Backus Naur Form.

GPP General Purpose Processor.

OSI Open Systems Interconnect model.

TLM Transaction Level Modeling.

List of Figures

List of Tables

1 Introduction

A set of digital rules define the communication strategy between digital systems.There are many rules which makes the communication possible between systems.Over the decades the rules have evolved into standards.The rules are called as Protocols in communication systems.The Open Systems Interconnect model (OSI) partition the communication system into 7 abstraction layers.There are different protocols for each layers of abstraction. The software and/or hardware changes based on the protocol chosen to process the message and extract the relevant information at each layer of abstraction. The hardware solutions based on a General Purpose Processor (GPP) or an Application Specific Integrated Circuit (ASIC) exists [1][2].GPP will have more flexibility but are less energy efficient when compared to ASIC which are less flexible and most energy efficient. ASIP or domain specific processors are more suitable for the protocol processing task and depending on their architectural characteristics they allow varying degrees of trade-off between flexibility and energy-efficiency[3].Resource and performance varies depending on the reconfigurable architecture and its level of abstraction[4].

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

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

1.1 Background

In 1960 Gerald Estrin, proposed the idea of a fixed plus variable structure computer [5].It consisted of a fixed processor and an array of reconfigurable hardware which was controlled by the fixed processor.Even though the idea was demonstrated with a proof,the industry did not consider to further innovate in these field and till 1980's there were no significant developments. In 1985,the reconfigurable Programmable Logic Array (PLA) was patented[6].Innovation in PLA's further continued with the commercially available Field Programmable Gate Arrays (FPGA) in today's market.

Ericsson AB 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 standard explained by IEEE in 802.3 standard defines the protocol for 10Gbit transfer which is mainly used for communication between the silicon chips. Other protocols include Common Public Radio Interface (CPRI),Serial Rapid IO(SRIO),Xio(Ericsson Specific protocol) for reliable communication between chips at high data rate.Most of these protocols in MAC layer of abstraction have common hardware units. Ericsson design and manufacture Custom ASIC chips for these protocols.The common functions which can be used by different protocols for designing a reconfigurable hardware architecture which will minimize the cost and will provide more flexibility compared to ASIC chips.More details about the protocols and reconfigurable architecture is explained in Chapter 3.

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

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

1.2 Purpose

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

1.3 Goals

The thesis goal is to achieve the below milestones:

- Understand the Reconfigurable hardware architecture designed at Ericsson AB
- Understand Ethernet,Xio and CPRI protocols
- Define a language to describe the protocols in high level words
- 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

1.4 Limits on scope

The thesis focus more on showing the proof of concept considering one to two protocol i.e Ethernet and Xio. The language will be designed such that it is easy with minor modification to extend for other protocols like CPRI,SRIO. Developing and integrating the TLM models for all the protocol's will not be feasible in this time line.

1.5 Structure of the thesis

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

the background about the topic and the literature about different terminologies. The third Chapter will provide the implementation details and the results obtained will be explained in chapter four. The Future work and conclusion are stated in the further chapter five and six.

2 Background

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

2.1 Reconfigurable Systems

In the field of computer architecture, designers make decisions based on flexibility and performance requirement[7].ASIC are the least flexible in terms of adapting for any change in the application and GPP are the most flexible as they are independent of the application and the core can be programmed to make the required algorithm work at the cost of higher power and lower efficiency.ASIC and GPP lies in extreme corners of the graph between Flexibility Vs Performance as in Fig. 2.1.Reconfigurable architectures are intended to fill the gap and provide more flexibility in terms of hardware and potentially higher performance than software[7].

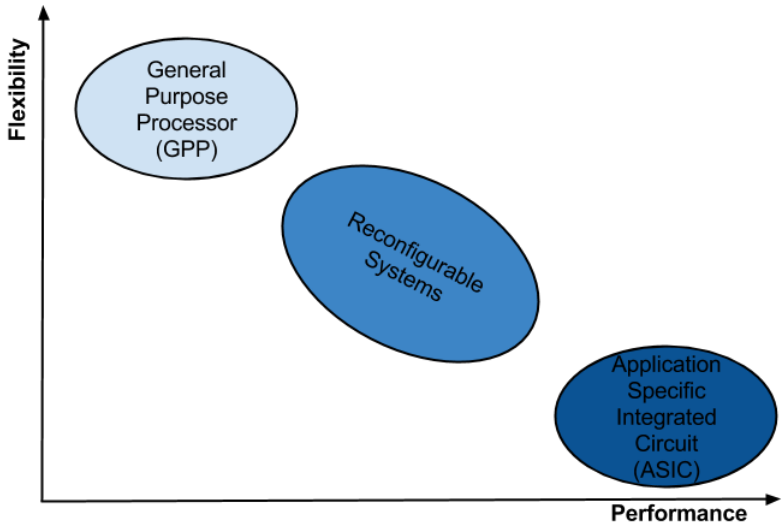

Figure 2.1: Flexibility Vs Performance of Hardware Classes

2.1.1 Granularity

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

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

2.1.2 Reconfiguration Models

The reconfigurable architectures need configuration of hardware.This can be at compile time or at runtime of an application as in Fig. 2.2.

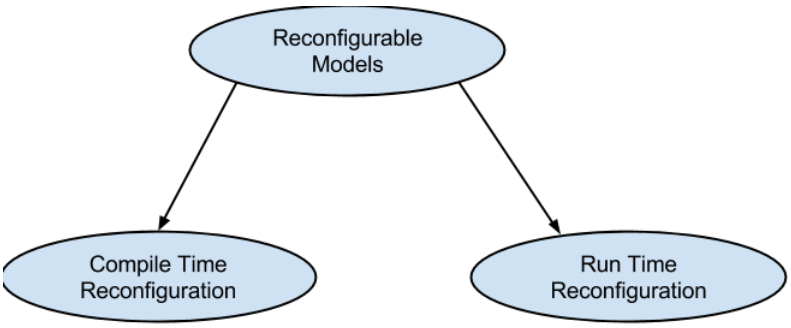

Figure 2.2: Reconfiguration Models

In Compile time reconfiguration model, the reconfigurable hardware system is configured at compile time and will be static during the application run time.In this model the programmable logic can be configured to perform some specific task like hardware accelerators to achieve high performance.FPGA configured to perform floating point multiplication together with a GPP will accelerate the performance of the application if the GPP doesn't have a Floating Point Unit. In Run time reconfiguration model, the reconfiguration hardware is configured at run time and will be dynamically programmed to perform different tasks.The decision for making such dynamic reconfiguration has to be embedded coupled with the application and hence it increase the overhead.The Dynamic Reprogrammable Resource Array(DRRA) fabric developed at KTH Electronic System Dept is an example of this model[8].

2.1.3 Reconfiguration rate

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

2.2 Protocols

The communication between chips in Radio Base station equipments has many protocols to fulfill the requirements of the specification.The protocols differ by standards.The Data link layer protocol's of the OSI model share some common functionalities.The different protocols being used at Ericsson AB and related to this thesis work are described below:

2.2.1 Ethernet

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

Ethernet Transmit

The Ethernet transmit sequence is shown in Fig. 2.3.

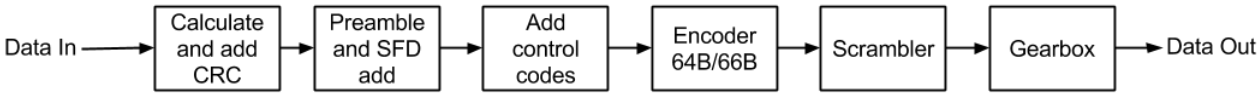

Figure 2.3: Ethernet Transmit

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

Ethernet Receive

The Ethernet Receive sequence is shown in Fig. 2.4. *DataIn* is from the Physical transmission layer and *Dataout* is to the higher protocol layers.

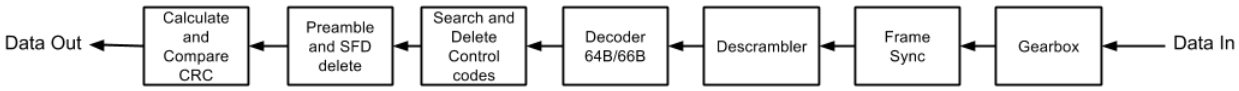

Figure 2.4: Ethernet Receive

Ethernet Raw frame

The Ethernet raw frame format is represented as in Fig. 2.5.Each of these frames enter the transmitter as Datain as in Fig. 2.3 and in the receiver as Data out as shown in Fig. 2.4.

MAC destination address	MAC Source address	802.1Q tag	Ethertype/length	Payload
6 octets	6 octets	4 octets	2 octets	42-1500 octet

Figure 2.5: Ethernet Raw Frame

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

CRC

Cyclic Redundancy check is used to detect errors incurred during the physical transmission. The CRC value is computed by dividing the data to be transmitted with the pre-defined CRC polynomial stored in the memory[9]. The remainder of the division is known as the Frame Check Sequence (FCS).

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

Preamble and SFD

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

Preamble: Total 7bytes, each byte is 10101010.
SFD: 10101011

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

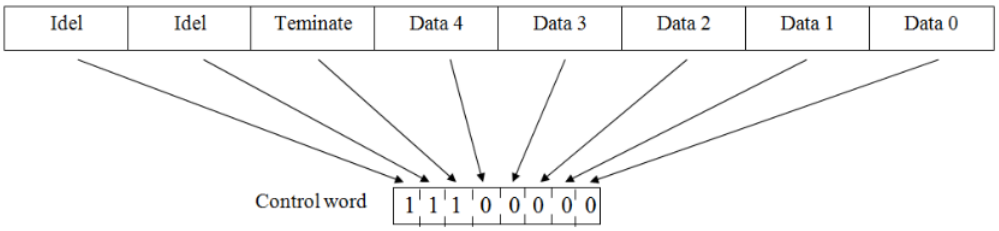

Figure 2.6: Control Word

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

The Encoder block is represented as show in Fig. 2.7. 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 synch header “10” correspond to data and “01” corresponds to control codes

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

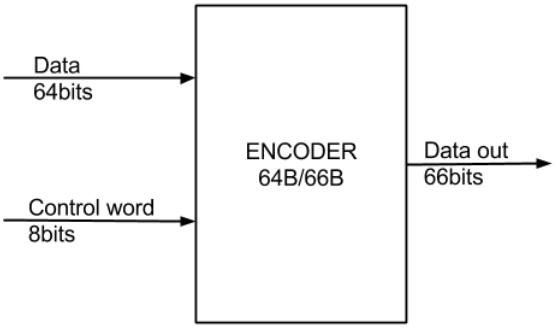

Figure 2.7: Ethernet Encoder block

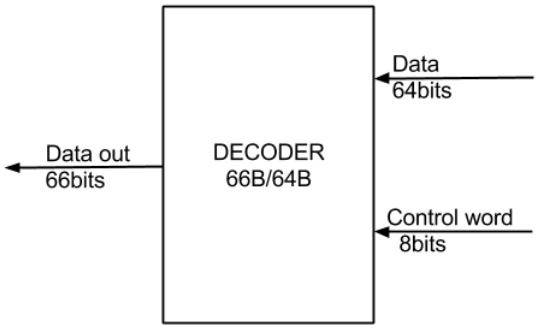

Figure 2.8: Ethernet Decoder block

Scrambler and Descrambler

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

The Descrambler will take the scrambled input and will output the unscrambled data.

Gearbox

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

2.2.2 Xio-s

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

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

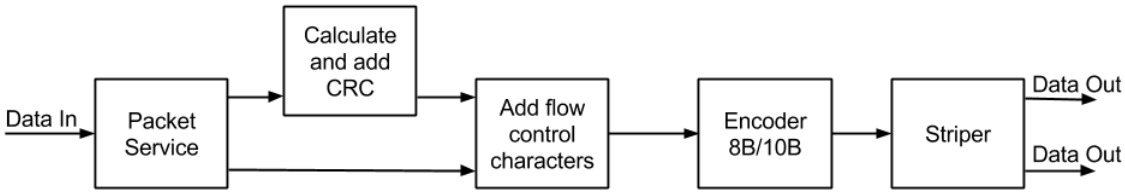

Figure 2.9: Xio-s Transmitter

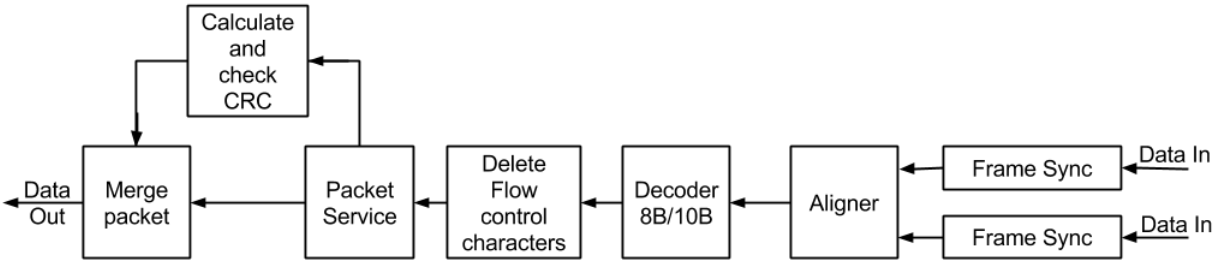

Figure 2.10: Xio-s Receiver

Figure 2.11: Xio-s Frame format Type 1

Figure 2.12: Xio-s Frame format Type 2

type format II then it uses both CRC16 and CRC32 channels.

CRC

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

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

Flow control characters

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

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

Encoder/Decoder

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

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

Striper

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

Frame Sync

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

Aligner

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

2.2.3 CPRI

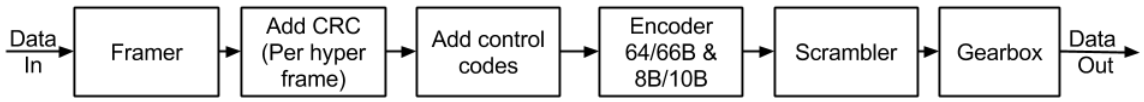

Figure 2.13: CPRI Transmitter

CPRI is an industry cooperation aimed at defining a publicly available specification for the key internal interface of radio base stations between Radio Equipment Control(REC) and the Radio Equipment(RE).It is the co

operating work of Ericsson AB,Huawei Technologies Co.Ltd,NEC corporation,Nortel Networks SA and Siemens AG. All the blocks in this protocol are similar to the blocks in Ethernet protocol. The frame structure is similar to the Xio-s protocol.

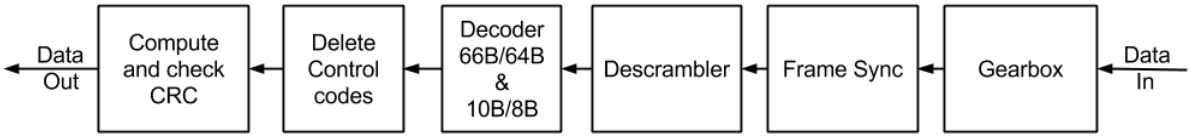

Figure 2.14: CPRI Receiver

2.3 Grammar and Language

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

2.3.1 Parser

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

Figure 2.15: Parser

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

2.3.2 Backus-Naur Form

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

For example,

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

⟨ DIGIT ⟩ ::= [0-9]

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

⟨ int ⟩::= ⟨ DIGIT ⟩ *

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

2.4 Environment and Tools

The Linux based server environment is used for compiling and testing the project files.The Antlr tool is used to define the protocols in a language defined in the grammar using EBNF. The Antlr tool is made to ouput the state table information which contains all the states with transition information.This will be used to reconfigure the hardware to process the input packets according to the protocols.

2.4.1 SystemC

SystemC is an ANSI standard C++ class library for system and hardware design for use by designers and architects who need to address complex systems that are a hybrid between hardware and software [10].It helps in modeling the concurrent processes.This makes it possible to model the hardware which are concurrent systems by nature.

2.4.2 TLM

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

TLM 2.0 standard is used for the implementation of the bus system.

2.4.3 UVM

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

3 Methodology

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

3.1 Research process

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

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

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

3.2 Freyja Architecture

As in Figure 3.1 , Freyja architecture is a reconfigurable protocol processor. It consists of different protocol operators which are connected through the central switch. This switch based network topology can be configured to process the data based on protocols. The Ring bus (RB) interface the Freyja with the higher layers of protocol and it issues the data frames of different protocols as tokens. The Common Memory Interface(CMI) is used to fetch the data from memory. The physical interface is represented by the Serializer/Deserializer block i.e SERDES. The details of each of the Freyja architecture blocks are discussed below.

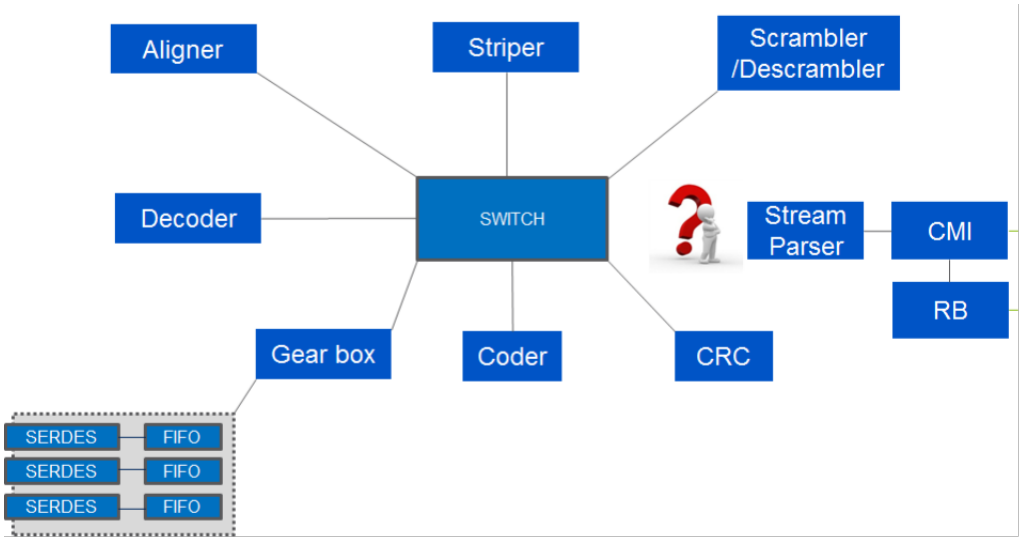

Figure 3.1: Freyja Architecture

Switch port

Freyja Switch port is the smallest unit in the architecture which receives the transactions from different functions and then forwards it to the internally connected next switch port based on the destination of the transaction. It consists of a simple initiator socket,simple target socket,Multipass through initiator socket and Multi pass through target socket.It is represented as in Figure 3.2

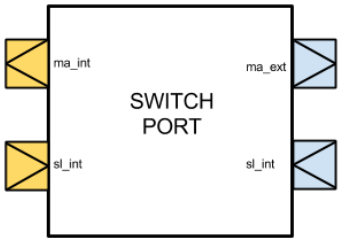

Figure 3.2: Switch port

Operator units

Freyja Operator units are designed to contain a control, process and memory blocks. Each operator function is implemented in the process block and the transaction routing and context switching is performed in the control block. The memory block stores the constants required for the process block. It can also be used to store the result and then the control block can access the results. As in Figure 3.3 , the 3 components of each operator is encapsulated with one simple initiator socket and one target socket which initiates the transaction and receives the transaction from the switch respectively. The transaction received in routed to the control block where it sends the

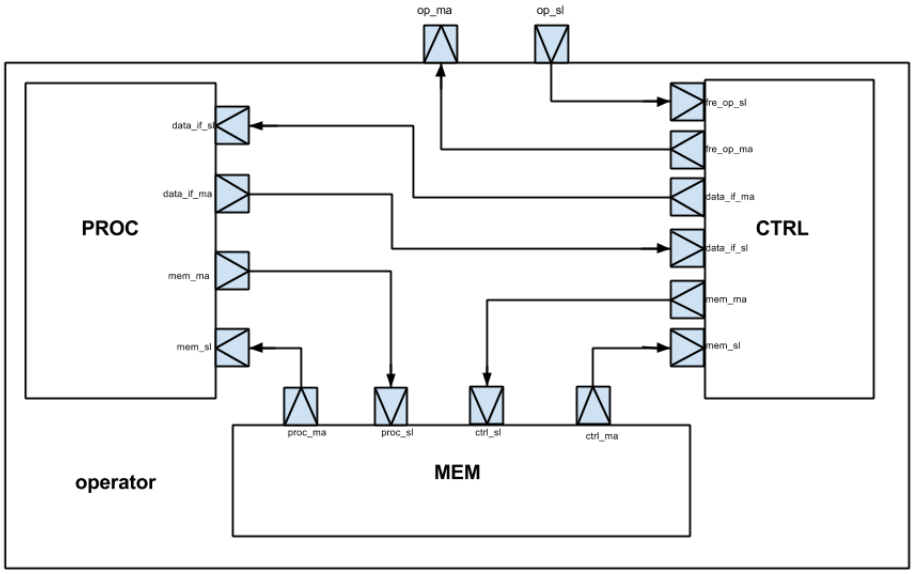

Figure 3.3: Freyja operator

transaction to memory block about the received transaction. Based on the required operator, the memory block will send the transaction to the process block. Meanwhile the received transaction is sent to the process block from the control block to perform the required operation.

Switch wrapper

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

As in Figure 3.4, each switch port can send the transaction to any of the other switch port through the internal multi pass through socket. The other switch ports can receive the transaction using the multi pass through receive socket. A transaction source and destination cannot be the same operator as there is no such interconnection.

Overall architecture

As in Figure 3.5, the overall Frejya architecture consists of switch wrapper instantiating switch ports and the operator blocks for each of the operator functions. A transaction originating from operator 1 as shown in fig with Orange box can be routed to any of the other operators as destination based

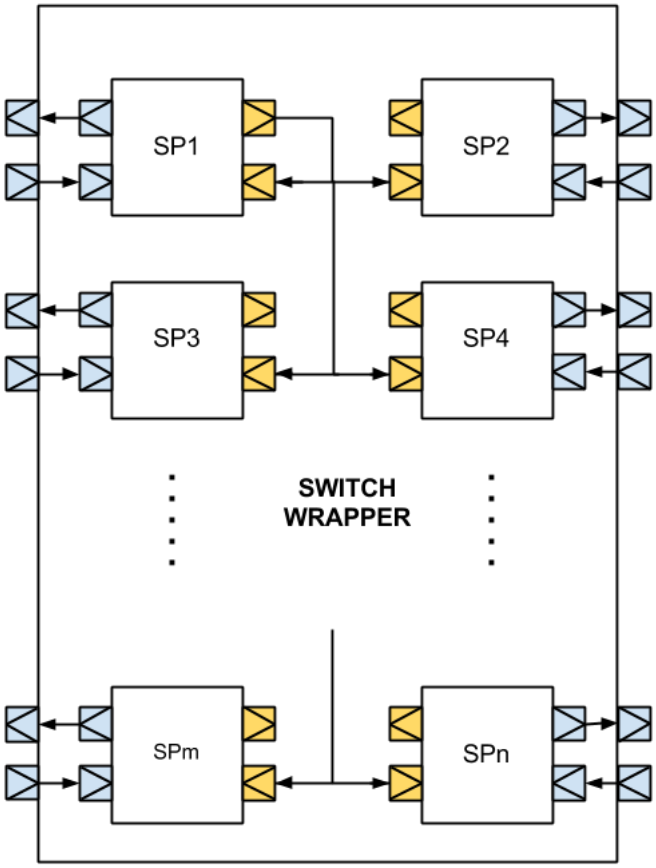

Figure 3.4: Freyja Switch Wrapper

on the address of the payload. This address is assigned in the control block of the operator where the transaction is originated.

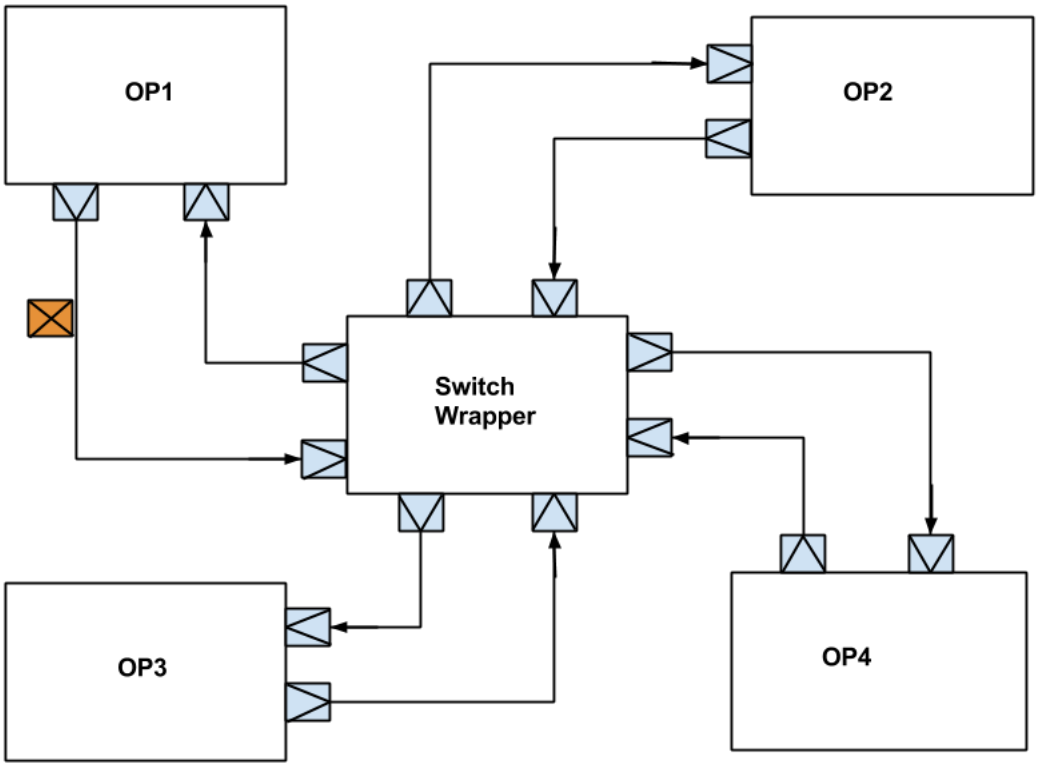

Figure 3.5: Freyja Architecture for 4 operators

3.3 High level description of the protocol

The different operations in each of the protocols are summarized in Section 2.2

3.3.1 Comparison of 3 Protocols

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

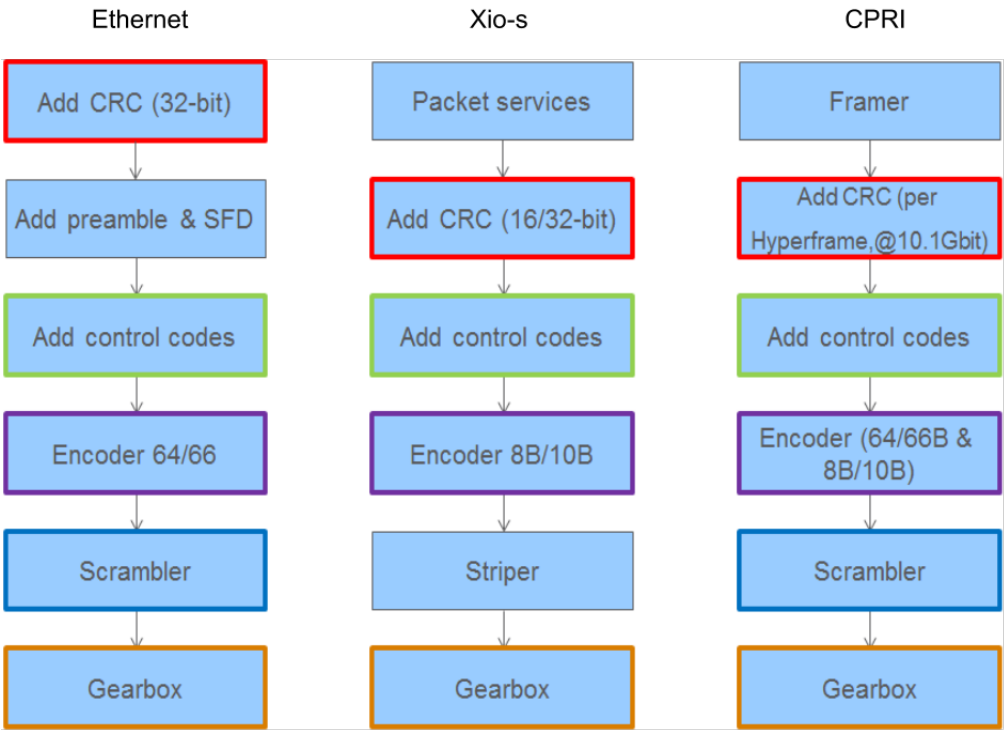

Figure 3.6: Transmitter of all 3 protocols

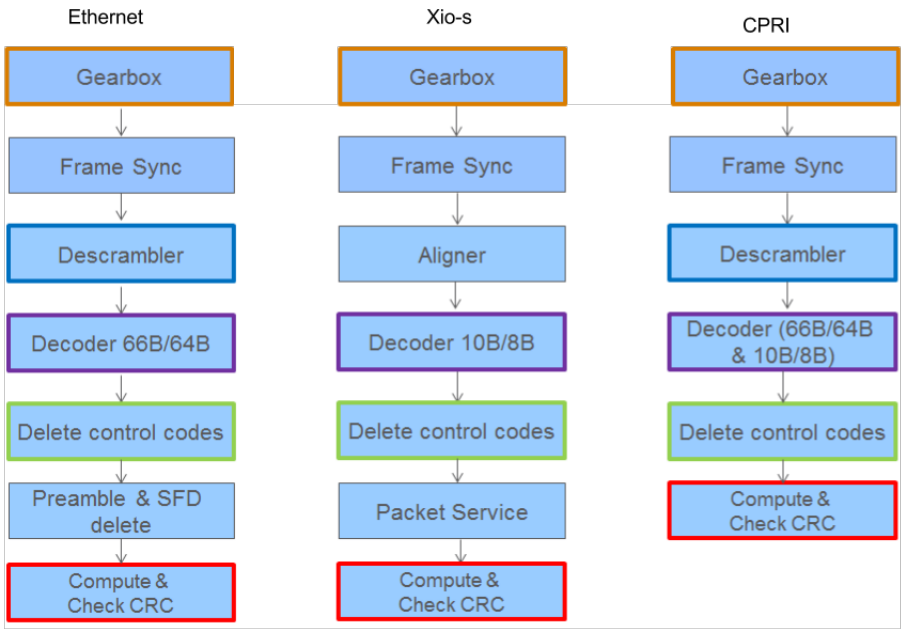

Figure 3.7: Receiver of all 3 protocols

3.3.2 Antlr

ANTLR is a powerful parser generator that can be used to read,process or translate structured text or binary files.It has been adapted in academia and industries for different applications and hence there is a good community help for most common problems.The tool generates the background parser and checker files for the grammar defined in the Bacchus-Naur form(BNF)or the Extended Bacchus-Naur form(EBNF).It makes the framework easy for the user to generate a language application based on the defined grammar.

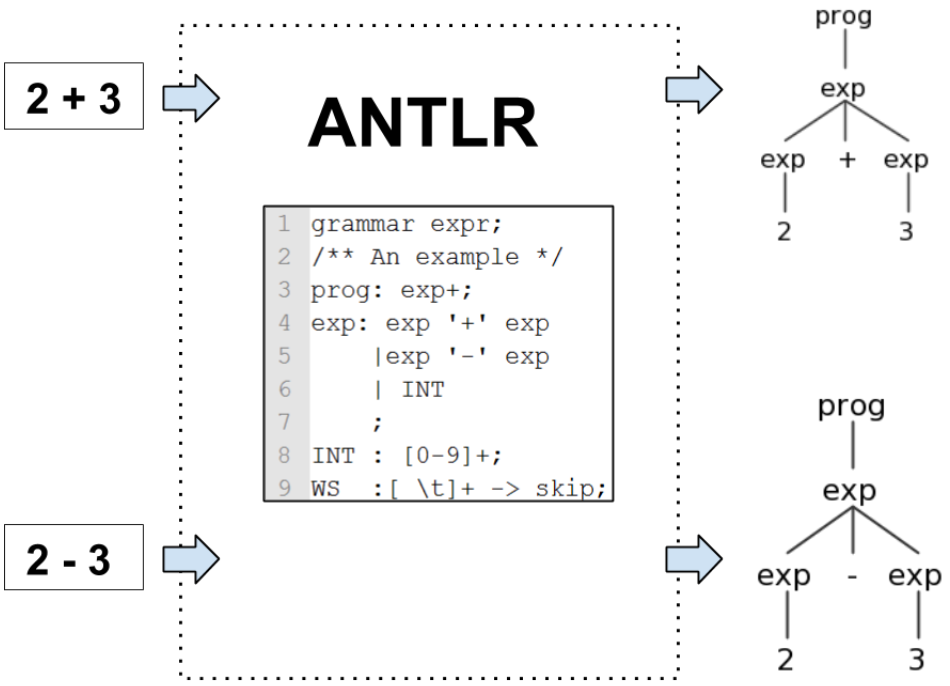

Figure 3.8: Antlr

Antlr is a parser generator tool.It allows the user to express the grammar to match syntactic structures like programming language arithmetic expressions.As in Fig below, the grammar can parse the expression and produce the parse tree and the backend functions to implement the interpreter, compiler or translator .

Parse tree listners and Visitors

Antlr provides 2 tree walking mechanisms in its runtime library. In parse tree Listners, ANTLR generates a parsetreeListner subclass specific to each grammar with entry and exit methods for each rule. This is suitable for

applications wherin the complete tree need to be invoked from root till the last leaf node. The parse tree Visitor mechanism is used when the tree walking needs to be controlled.

3.4 Verification and Testing

4 Developed language

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

4.1 Parser implementation

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

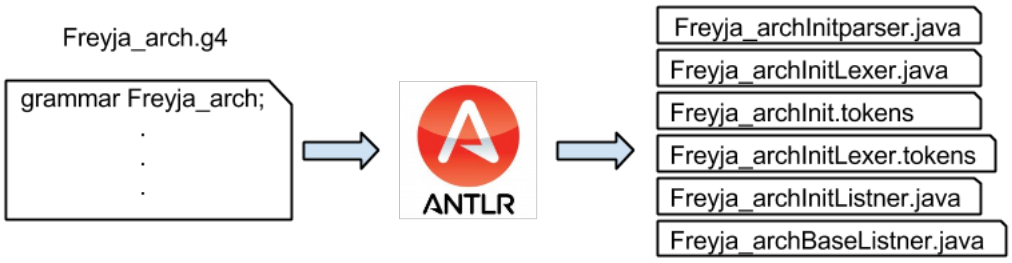

Figure 4.1: Antlr generated Files

A brief description of the generated file are stated below:

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

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

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

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

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

4.2 Operator Instantiation

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

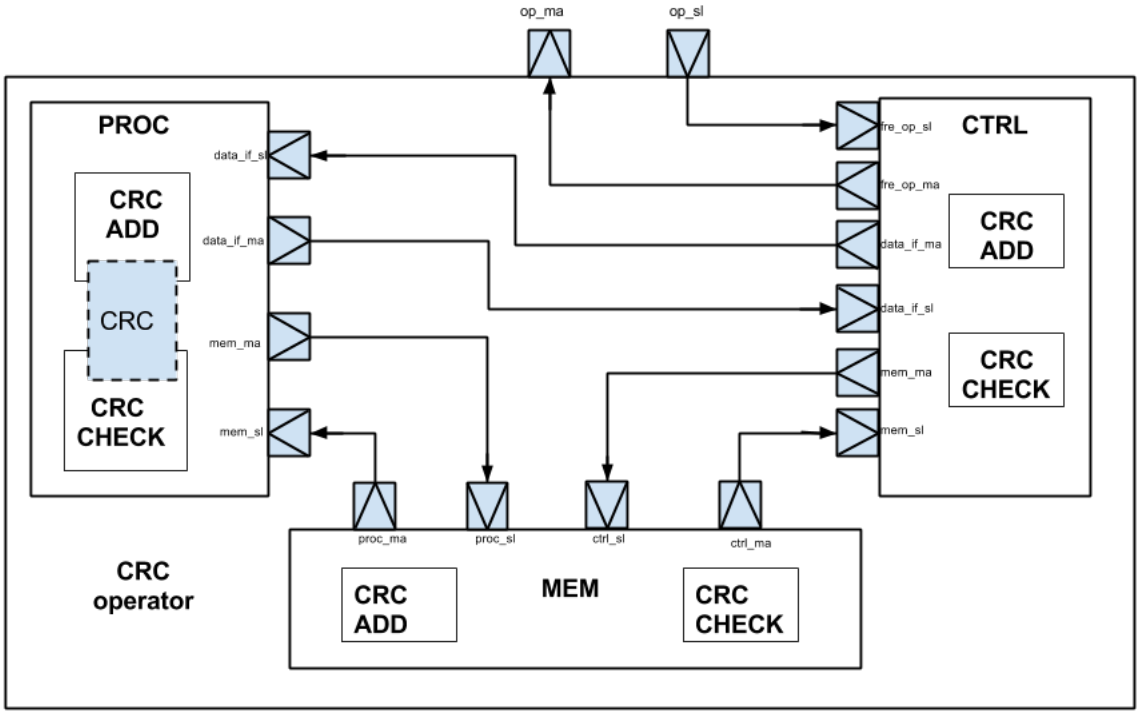

Figure 4.2: CRC operator in Freyja Architecture

In a protocol processor consisting of operators for transmitting and receiving the data frames, the functions performed during the transmitting stage might need to be performed in the receiver stage. For instance, FCS is

appended during the transmitting stage and in the receiver stage the computation of CRC is performed again. In addition, it performs the comparison of the received FCS value and the newly computed value to identify if the received data is erroneous or not.

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

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

The operator is defined as below:

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

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

4.3 Context switching

The context switching in the process block is performed based on the destination operator for the data packet. The interconnect information is by the control block to determine the protocol and destination operator for the data packet. The control block will then forwards the transaction to the pro-

cess block where the context switching between the operators are performed. The operator type field will indicate the process block to switch between the operators.

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

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

4.4 Memory

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

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

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

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

As in code above the first 8 bytes of memory contents are sent as a transaction from memory for the CRC ADD operation in CHECKSUM operator.

The different forms of notation can be observed in the CRC check wherein the memory bytes can be specified with individual address. Each protocol with the memory code as above can send transaction for different operators. In the above code, the crc add,crc check,scram and descram send the transaction to their respective blocks.

4.5 Transaction handling

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

CONNECT: <source operator> to <Destination operator>

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

4.6 Error handling

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

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

5 Tests

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

5.1 Complete Test System

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

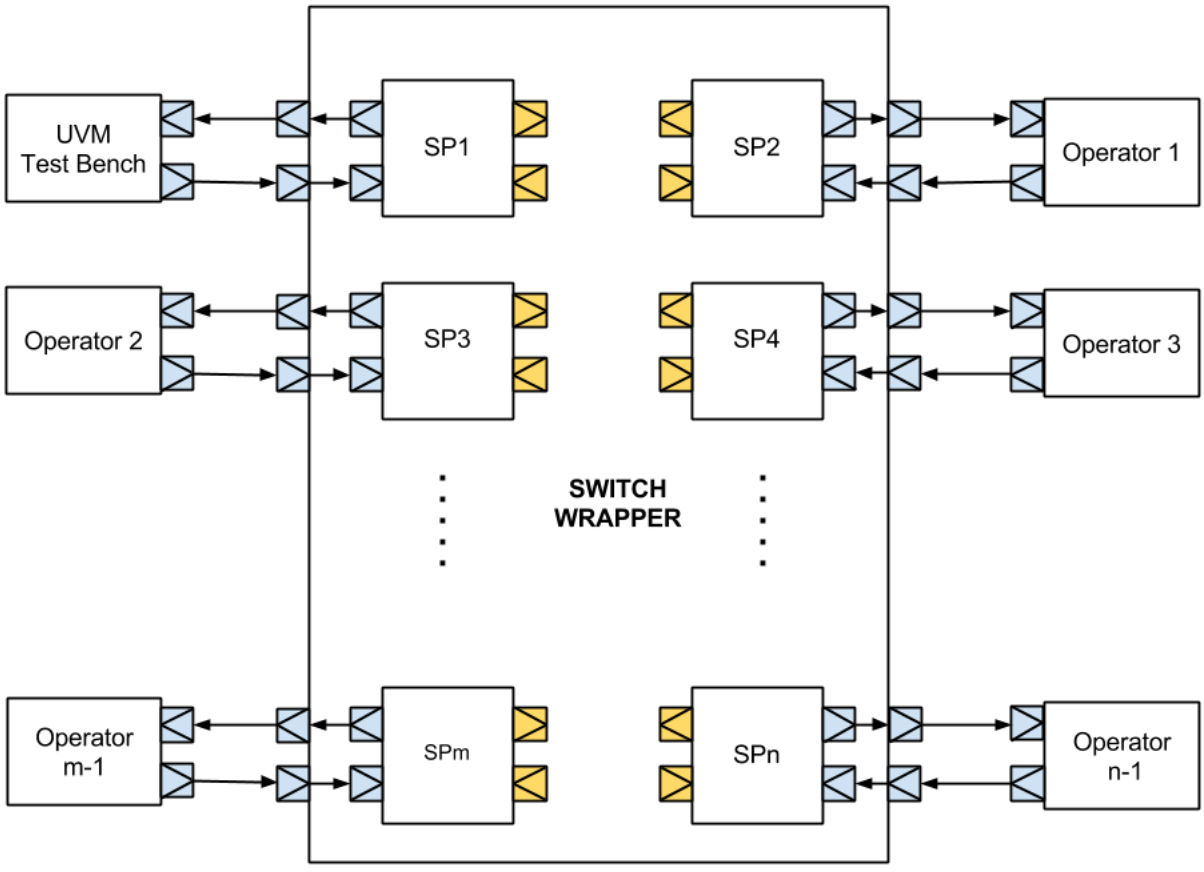

Figure 5.1: Complete Test System(*Multiport interconnections not shown)

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

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

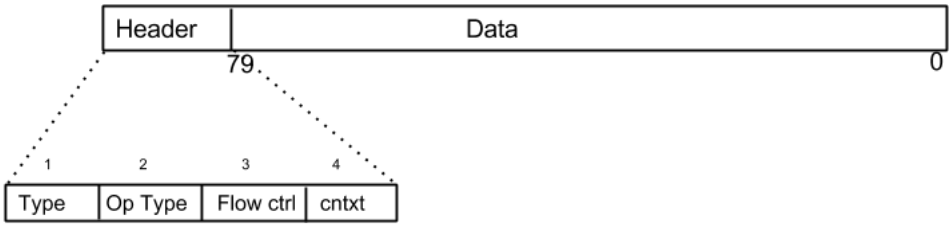

Figure 5.2: FBI Interconnect

5.2 Payload and blocking transport

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

5.3 Input and output

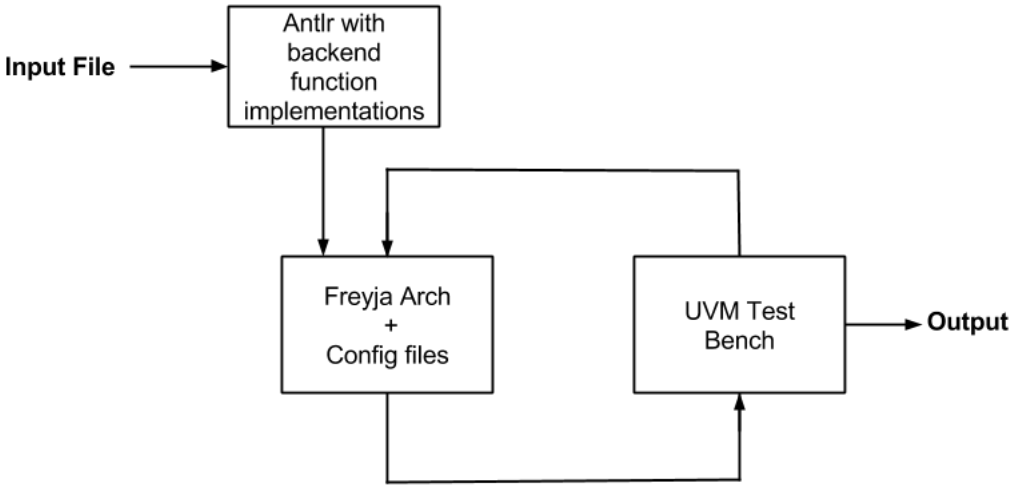

Figure 5.3: Input and Output of the system

The input file is the description of the protocol using the language defined by the EBNF grammar. The input file is parsed by the Antlr tool; the back-end functions are triggered during parsing to output the Freyja architecture and the reconfigurable files. The UVM test bench will send the transaction to the Freyja architecture. The simulation is performed using Quartus tool chain. The transaction is initiated from the UVM to the Freyja architecture and the final transaction is received back in the UVM. An example of instantiatig 4 operator blocks and interconnecting them using the input file and the corresponding output is shown below:

5.4 Blocking transport and timing annotation

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

6 Analysis

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

6.1 Ethernet

Ethernet Transmit and Receive operations are explained in Fig. 2.3 and Fig. 2.4.The *Datain* in the transmit is passed from the UVM test bench.The UVM sequence is generated with first byte indicating the protocol and the second byte containing the address of the operator.Once the transaction reaches the Switch wrapper,it is decoded and forwarded to the first operator as defined by the input file.

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

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

```
Tx_PATH {
CONNECT : proc_tx to  crc_add;
CONNECT : crc_add to  pre_sfd_add;
CONNECT : pre_sfd_add to  cc_a;
CONNECT : cc_a to  enc;
CONNECT : enc to  scram;
CONNECT : scram to  gb_tx;
CONNECT : gb_tx to  gb_rx;
}
```

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

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

- Rx_Path is used to define the Receiver sequence

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

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

```
//part of fre_op_ctrl_crc.cpp block

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

In the above code the CRC operator's control block initially check for the Protocol and then for the type of the operator.As explained in Fig. 4.2,the CRC operator multiplex 2 operations. CRC ADD and CRC check operators are having the same destination operator control block. They share the common memory,hardware for CRC computation and control function. The extra logic required to perform the CRC add and CRC compare are different.

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

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

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

The constants are formed using simple syntax as below

⟨ source operator ⟩ _2_ ⟨ Destination operator ⟩

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

The CRC operator also contains 19 bytes of memory elements.These 19 bytes are used by both CRC32 and CRC16 Operators.Ethernet protocol uses only CRC32 and hence only crc_add and crc_check uses first 9 bytes of the memory elements.The parser supports the access of contiguous or individual 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.

6.2 Xio

The Xio-s protocol is defined using the input file and only CRC operator was integrated.

change in control block! include one source to many destination operators ! context switching at process block

7 Conclusion

7.1 Limitations

7.2 Future work

References

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

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

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

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

[5] 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.

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

[7] 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.

[8] 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.

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

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

Appendices

A Ethernet Protocol Description

```
FunctionUnits FREYJA:

Operator PROCESSOR{
sc_name : proc;
op_type : proc_tx,proc_rx;
errorid : 01,13;
}

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

Operator PRE_SFD{
sc_name : pre_sfd_add;
op_type : pre_sfd_add,pre_sfd_del;
errorid : 04,05;
}

Operator CORRECTING_CODES_ADD{
sc_name : cc_a;
op_type : cc_a;
errorid : 06;
}

Operator ENCODER{
sc_name : enc_64_66;
op_type : enc;
errorid : 07;
}

Operator SCRAMBLER{
sc_name : scram;
```

```
op_type : scram,descram;
op_mem  : 0x01,0x00,0x00,0x00,0x80,0x00,0x00,0x04;
errorid : 08;
}

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

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

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

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

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


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

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


Tx_PATH {
CONNECT : proc_tx to  crc_add;
CONNECT : crc_add to  pre_sfd_add;
CONNECT : pre_sfd_add to  cc_a;
CONNECT : cc_a to  enc;
CONNECT : enc to  scram;
CONNECT : scram to  gb_tx;
CONNECT : gb_tx to  gb_rx;
}


RX_PATH {
CONNECT : gb_rx to  descram;
CONNECT : descram to  decdr;
CONNECT : decdr to  cc_d;
CONNECT : cc_d to  pre_sfd_del;
CONNECT : pre_sfd_del to  crc_check;
CONNECT : crc_check to  proc_rx;
}
```